Scientific Computing: An Introductory Survey

Chapter 1 – Scientific Computing

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign

Outline

Scientific Computing

- What is *scientific computing*?
  - Design and analysis of algorithms for numerically solving mathematical problems in science and engineering
  - Traditionally called *numerical analysis*
- Distinguishing features of *scientific* computing
  - Deals with *continuous* quantities
  - Considers effects of approximations
- Why *scientific computing*?
  - Simulation of natural phenomena
  - Virtual prototyping of engineering designs

Well-Posed Problems

- 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 input data
- Computational algorithm should not make sensitivity worse

General Strategy

- Replace difficult problem by easier one having same or closely related solution
  - infinite $\rightarrow$ finite
  - differential $\rightarrow$ algebraic
  - nonlinear $\rightarrow$ linear
  - complicated $\rightarrow$ simple
- Solution obtained may only *approximate* that of original problem

Sources of Approximation

- Before computation
  - modeling
  - empirical measurements
  - previous computations
- During computation
  - truncation or discretization
  - rounding
- Accuracy of final result reflects all 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 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 in 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) $\times$ (1 $+$ rel error)
- True value 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: compute value of function $f\colon \mathbb{R} \to \mathbb{R}$ for given argument
  - $x =$ true value of input
  - $f(x) =$ desired result
  - $\hat{x} =$ approximate (inexact) input
  - $\hat{f} =$ approximate function actually computed
- Total error: $\hat{f}(\hat{x}) - f(x) =$

$$\underbrace{\hat{f}(\hat{x}) - f(\hat{x})}_{\text{computational error}} + \underbrace{f(\hat{x}) - f(x)}_{\text{propagated data error}}$$

- Algorithm has no effect on propagated data error

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 approximations such as truncating infinite series 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, but one of these usually dominates

< interactive example >

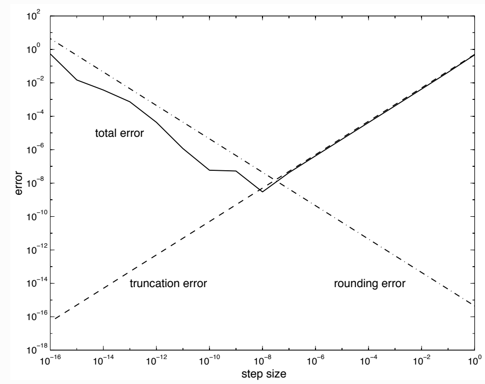
Example: Finite Difference Approximation

- 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

- Suppose we want to compute $y = f(x)$, where $f\colon \mathbb{R} \to \mathbb{R}$, but obtain approximate value $\hat{y}$
- *Forward error*: $\Delta y = \hat{y} - y$
- *Backward error*: $\Delta x = \hat{x} - x$, where $f(\hat{x}) = \hat{y}$

$$\begin{tikzcd}
x \arrow[rr, "f"] \arrow[rrd, "\hat{f}"] \arrow[d, leftrightarrow, "\text{backward error}"'] & & y = f(x) \arrow[d, leftrightarrow, "\text{forward error}"] \\
\hat{x} \arrow[rr, "f"] & & \hat{y} = \hat{f}(x) = f(\hat{x})
\end{tikzcd}$$

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

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
- 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: $\Delta x = \hat{x} - x \approx 1.0472 - 1 = 0.0472$

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 $\text{cond} \gg 1$

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 Function

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

Absolute forward error: $f(x + \Delta x) - f(x) \approx f'(x)\Delta x$

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

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

- Relative error in function value can be much larger or smaller than that in input, depending on particular f and x

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 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 of problem
- 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 or unstable algorithm to well-conditioned problem
- Applying stable algorithm to well-conditioned problem yields accurate solution

Floating-Point Numbers

- Floating-point number system is characterized by four integers

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

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

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 SP	2	24	-126	127
IEEE DP	2	53	-1022	1023
Cray	2	48	-16383	16384
HP calculator	10	12	-499	499
IBM mainframe	16	6	-64	63

- Most 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 systems, mantissa m of nonzero floating-point number always satisfies $1 \le m < \beta$
- Reasons for normalization
  - representation of each number unique
  - no digits wasted on leading zeros
  - leading bit need not be stored (in binary system)

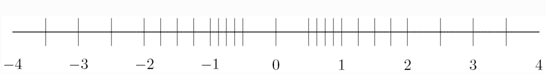
Properties of Floating-Point Systems

- 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 $\mathrm{fl}(1 + \epsilon) > 1$
- Maximum relative error in representing real number x within range of floating-point system is given by

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

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
- So IEEE single and double precision systems have about 7 and 16 decimal digits of precision, respectively

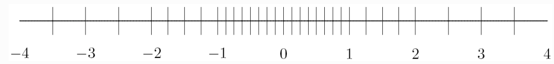
Machine Precision, continued

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

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

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

- *Addition or subtraction*: Shifting of 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 Series

- Infinite series

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

  has finite sum in floating-point arithmetic even though real series is divergent
- 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, $x \ \texttt{flop}\ y = \text{fl}(x \ \texttt{op}\ y)$, i.e., floating-point arithmetic operations produce correctly rounded results
- Computers satisfying IEEE floating-point standard achieve this ideal as long as $x \ \texttt{op}\ y$ is within range of floating-point system
- But some familiar laws of real arithmetic are 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 idea 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, $i = 1, \ldots, n$, 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