

Outline

1 Scientific Computing

2 Approximations

3 Computer Arithmetic



Scientific Computing

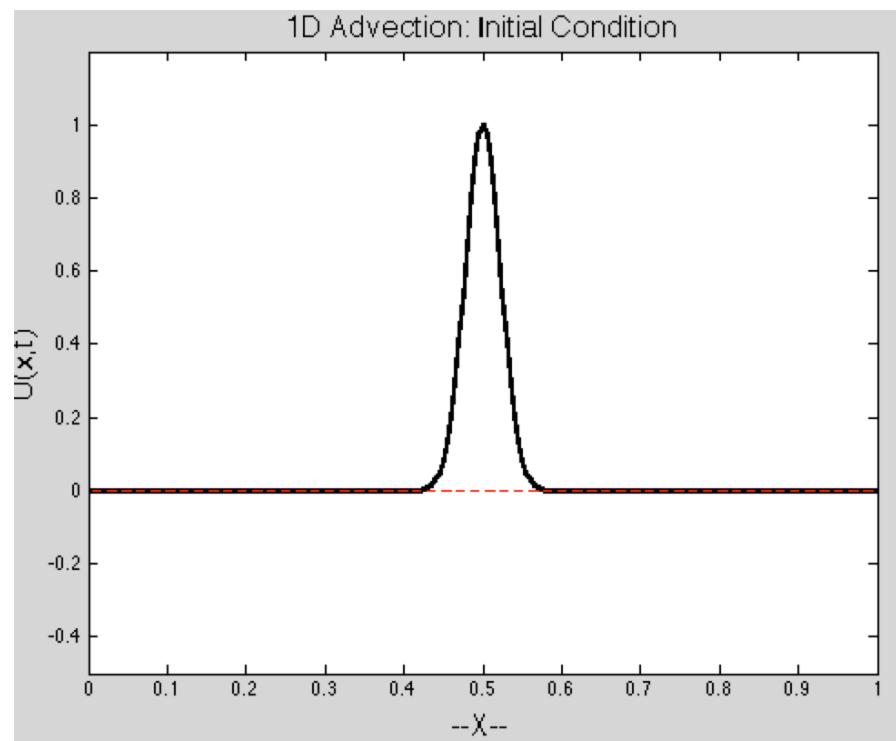
- What is *scientific computing*?
 - Design and analysis of algorithms for numerically solving mathematical problems in science and engineering
 - Traditionally called *numerical analysis*
- Distinguishing features of *scientific computing*
 - Deals with *continuous* quantities
 - Considers effects of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



Example: Convective Transport

$$\frac{\partial u}{\partial t} = -c u(x, t) + \begin{cases} \circ & \text{initial conditions} \\ \circ & \text{boundary conditions} \end{cases}$$

(See Fig. 11.1 in text.)



Example: Convective Transport

$$\frac{\partial u}{\partial t} = -c u(x, t) + \begin{cases} \circ & \text{initial conditions} \\ \circ & \text{boundary conditions} \end{cases}$$

- Examples:
 - Ocean currents:
 - Pollution
 - Saline
 - Thermal transport
 - Atmosphere
 - Climate
 - Weather
 - Industrial processes
 - Combustion
 - Automotive engines
 - Gas turbines
- Problem Characteristics:
 - Large (sparse) linear systems
 - millions to billions of degrees of freedom
- Demands
 - Speed
 - Accuracy
 - Stability (ease of use)



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 → finite
 - differential → algebraic
 - nonlinear → linear
 - complicated → 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*: $\frac{\text{absolute error}}{\text{true value}}$
- 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 $f: \mathbb{R} \rightarrow \mathbb{R}$ 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: $\hat{f}(\hat{x}) - f(x) =$

 $\hat{f}(\hat{x}) - f(\hat{x}) + f(\hat{x}) - f(x)$
computational error + **propagated data error**
- Algorithm has no effect on propagated data error



Data Error and Computational Error

- Typical problem: compute value of function $f: \mathbb{R} \rightarrow \mathbb{R}$ 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: $\|\hat{f}(\hat{x}) - f(x)\| \leq$
$$\|\hat{f}(\hat{x}) - f(\hat{x})\| + \|f(\hat{x}) - f(x)\|$$

computational error

+ 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



Taylor Series (Very important for SciComp!)

- If $f^{(k)}$ exists (is bounded) on $[x, x + h]$, $k = 0, \dots, m$, then there exists a $\xi \in [x, x + h]$ such that

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \dots + \frac{h^m}{m!}f(m)(\xi).$$

- Take $m = 2$:

$$\frac{f(x + h) - f(x)}{h} = f'(x) + \frac{h}{2}f''(\xi).$$

Taylor Series (Very important for SciComp!)

- If $f^{(k)}$ exists (is bounded) on $[x, x + h]$, $k = 0, \dots, m$, then there exists a $\xi \in [x, x + h]$ such that

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \dots + \frac{h^m}{m!}f(m)(\xi).$$

- Take $m = 2$:

$$\underbrace{\frac{f(x + h) - f(x)}{h}}_{\text{computable}} = \underbrace{f'(x)}_{\text{desired result}} + \frac{h}{2}f''(\xi). \quad \textcolor{red}{\text{Truncation error}}$$

Taylor Series (Very important for SciComp!)

- If $f^{(k)}$ exists (is bounded) on $[x, x + h]$, $k = 0, \dots, m$, then there exists a $\xi \in [x, x + h]$ such that

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \cdots + \frac{h^m}{m!}f(m)(\xi).$$

- Take $m = 2$:

$$\underbrace{\frac{f(x + h) - f(x)}{h}}_{\text{computable}} = \underbrace{f'(x)}_{\text{desired result}} + \underbrace{\frac{h}{2}f''(\xi)}_{\text{truncation error}}$$

- **Truncation error:** $|\frac{h}{2}f''(\xi)| \approx \frac{h}{2}f''(x)$ as $h \rightarrow 0$.

Q: Suppose $|f''(x)| \approx 1$.

Can we take $h = 10^{-30}$ and expect

$$\left| \frac{f(x + h) - f(x)}{h} - f'(x) \right| \leq \frac{10^{-30}}{2} ?$$

$$\underbrace{\frac{f(x+h) - f(x)}{h}}_{\text{computable}} = \underbrace{f'(x)}_{\text{desired result}} + \underbrace{\frac{h}{2}f''(\xi)}_{\text{truncation error}}$$

- **Truncation error:** $|\frac{h}{2}f''(\xi)| \approx \frac{h}{2}f''(x)$ as $h \rightarrow 0$.

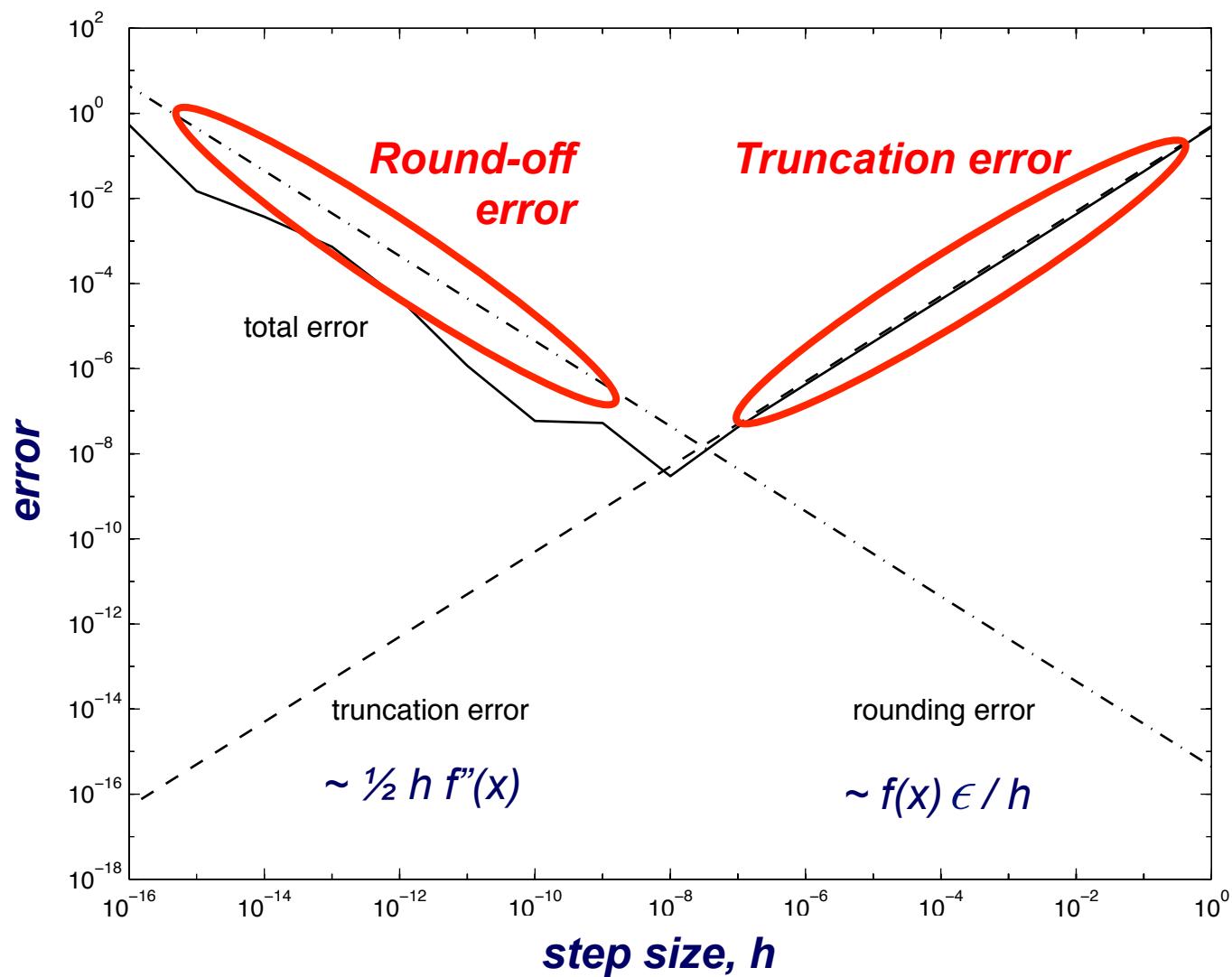
Q: Suppose $|f''(x)| \approx 1$.

Can we take $h = 10^{-30}$ and expect

$$\left| \frac{f(x+h) - f(x)}{h} - f'(x) \right| \leq \frac{10^{-30}}{2} ?$$

A: Only if we can compute every term in finite-difference formula (**our algorithm**) with sufficient accuracy.

Example: Finite Difference Approximation



Example: Finite Difference Approximation

- Error in finite difference approximation

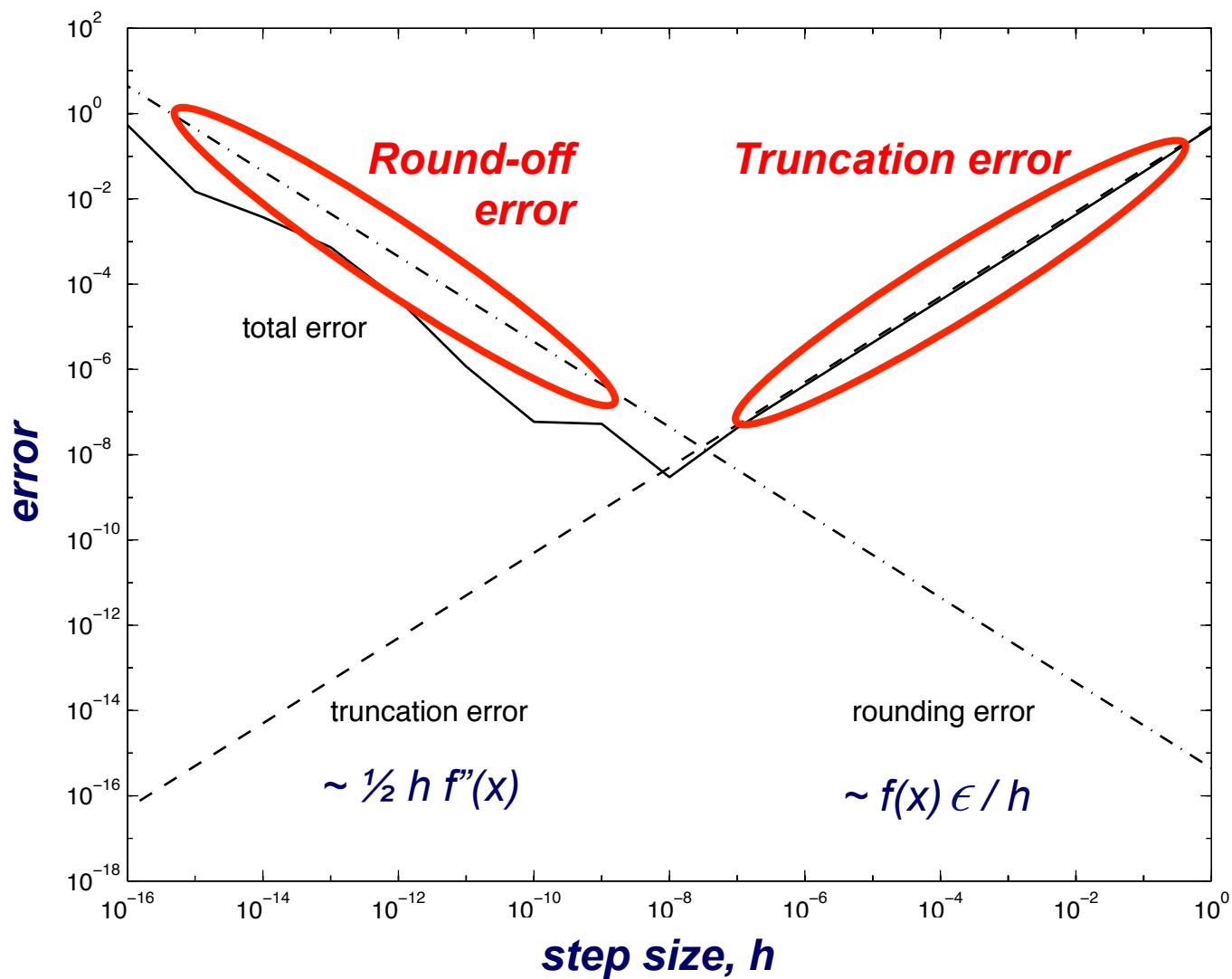
$$f'(x) \approx \frac{f(x + h) - f(x)}{h}$$

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

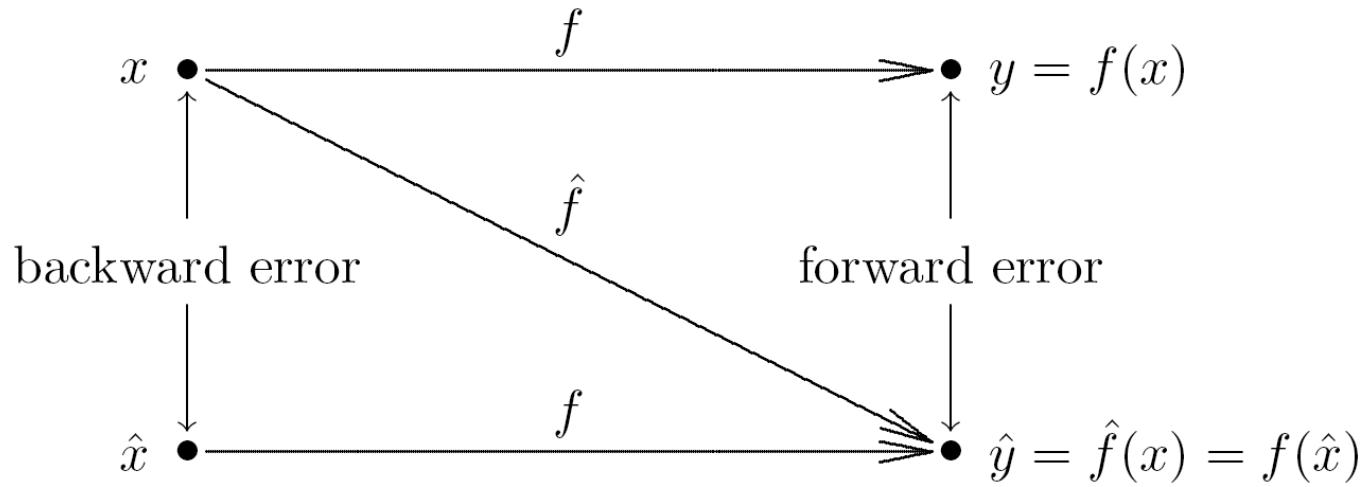


Round-Off Error

- In general, round-off error will prevent us from representing $f(x)$ and $f(x+h)$ with sufficient accuracy to reach such a result.
- Round-off is a principal concern in scientific computing.
(Though once you're aware of it, you generally know how to avoid it as an issue.)
- Round-off results from having finite-precision arithmetic and finite-precision storage in the computer. (e.g., how would you ever store π in a computer?)
- Most scientific computing is done either with 32-bit or 64-bit arithmetic, which 64-bit being predominant.

Forward and Backward Error

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



Example: Forward and Backward Error

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

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

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 $x = 1$,

$$y = f(1) = \cos(1) \approx 0.5403$$

$$\hat{y} = \hat{f}(1) = 1 - 1^2/2 = 0.5$$

$$\hat{x} = \arccos(\hat{y}) = \arccos(0.5) \approx 1.0472$$

- 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 $\text{cond} \gg 1$



Note About Condition Number

- ❑ It's tempting to say that a large condition number indicates that a small change in the input implies a large change in the output.
- ❑ However, to be dimensionally correct, we need to be more precise.
- ❑ A large condition number indicates that a small ***relative*** change in input implies a large ***relative*** change in the output:

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

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

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



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

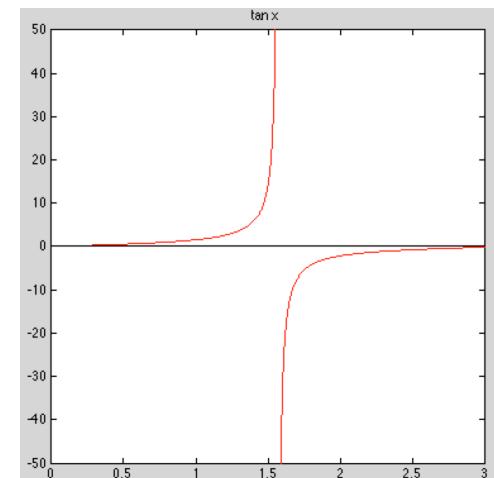
Condition number: $\text{cond} \approx \left| \frac{f'(x)\Delta x/f(x)}{\Delta x/x} \right| = \left| \frac{xf'(x)}{f(x)} \right|$

- 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 $x = 1.57079$, $\text{cond} \approx 2.48275 \times 10^5$



Condition Number Examples

- Q: *In our finite difference example, where did things go wrong?*

Using the formula, $\text{cond} = \left| \frac{x f'(x)}{f(x)} \right|$, what is the condition number of the following?

$$f(x) = a x$$

$$f(x) = \frac{a}{x}$$

$$f(x) = a + x$$

Condition Number Examples

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

For $f(x) = ax$, $f' = a$, $\text{cond} = \left| \frac{x a}{ax} \right| = 1$.

For $f(x) = \frac{a}{x}$, $f' = -ax^{-2}$, $\text{cond} = \left| \frac{x \frac{-a}{x^2}}{\frac{a}{x}} \right| = 1$.

For $f(x) = a + x$, $f' = 1$, $\text{cond} = \left| \frac{x \cdot 1}{a+x} \right| = \frac{1}{|a+x|}$.

- The condition number for $(a + x)$ is < 1 if a and x are of the same sign, but it is > 1 if they are of opposite sign, and potentially $\gg 1$ if they are of opposite sign but close to the same magnitude.

Condition Number Examples

- Subtraction of two positive (or negative) values of nearly the same magnitude is ill-conditioned.
- Multiplication and division are benign.
- Addition of two positive (or negative) values is also OK.
- In our finite difference example, the culprit is the subtraction, more than the division by a small number.

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



Examples of Potentially Unstable Algorithms

- Examples of potentially unstable algorithms include
 - Gaussian elimination without pivoting
 - Using the normal equations to solve linear least squares problems
 - High-order polynomial interpolation with unstable bases (e.g., uniformly distributed sample points or monomials)

Unavoidable Source of Noise in the Input

- Numbers in the computer are represented in finite precision.
- Therefore, unless our set of input numbers, x , are perfectly representable in the given mantissa, we already have an error, Δx , and our actual input is thus

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

- The next topic discusses the set of representable numbers.
- We'll primarily be concerned with two things –
 - the relative precision,
 - the maximum absolute value representable.

Relative Precision Example

$$x = 3141592653589793238462643383279502884197169399375105820974944.9230781\dots = \pi \times 10^{60}$$

$$x - \hat{x} = 238462643383279502884197169399375105820974944.9230781... = 2.3846... \times 10^{44}$$

$$\approx .7590501687441757 \times 10^{-16} \times x$$

$$< 1.110223024625157e - 16 \times x$$

$$\approx \epsilon_{\text{mach}} \times x.$$

- The difference between $x := \pi \times 10^{60}$ and $\hat{x} := \text{fl}(\pi \times 10^{60})$ is large:

$$x - \hat{x} \approx 2.4 \times 10^{44}.$$

- The *relative* error, however, is

$$\frac{x - \hat{x}}{x} \approx \frac{2.4 \times 10^{44}}{\pi \times 10^{60}} \approx 0.8 \times 10^{-16} < \epsilon_{\text{mach}}$$

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



Floating-Point Numbers, continued

- Portions of floating-point 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 $1 \leq m < \beta$
- Reasons for normalization
 - representation of each number unique
 - no digits wasted on leading zeros
 - leading bit need not be stored (in binary system)

Example



Binary Representation of π

- In 64-bit floating point,

$$\pi \approx 1.100100100001111101101010100010001000010110100011 \times 2^1$$

- In reality,

$$\pi = 1.10010010000111110110101010001000101101000110000100011010 \dots \times 2^1$$

- They will (potentially) differ in the 53rd bit...

$$\pi = 0.00100011010 \dots \times 2^1$$



Properties of Floating-Point Systems

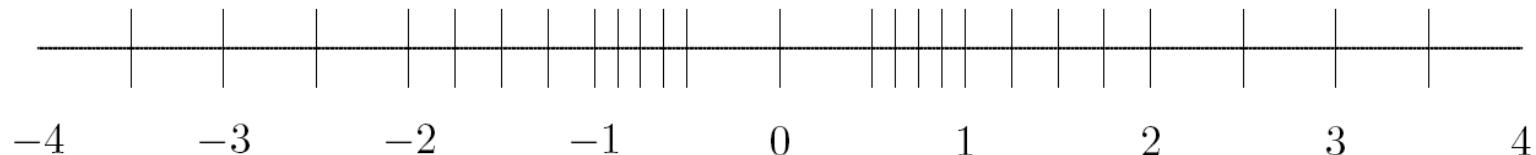
- Floating-point number system is finite and discrete
- Total number of normalized floating-point numbers is

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

- Smallest positive normalized number: $\text{UFL} = \beta^L$
- Largest floating-point number: $\text{OFL} = \beta^{U+1}(1 - \beta^{-p}) \approx \beta^U$
- 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 $U = 1$
 - OFL = $(1.11)_2 \times 2^1 = (3.5)_{10}$
 - UFL = $(1.00)_2 \times 2^{-1} = (0.5)_{10}$
- At sufficiently high magnification, all normalized floating-point systems look grainy and unequally spaced



Rounding Rules

- If real number x is not exactly representable, then it is approximated by “nearby” floating-point number $\text{fl}(x)$
- 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*: $\text{fl}(x)$ 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



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

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



Rounded Numbers in Floating Point Representation

- The relationship on the preceding slide,

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

can be conveniently thought of as:

$$\text{fl}(x) = x (1 + \epsilon_x)$$

$$|\epsilon_x| \leq \epsilon_{\text{mach}}$$

- The nice thing is the expression above has an equality, which is easier to work with.

Machine Precision, continued

- 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



Advantage of Floating Point

- By sacrificing a few bits to the exponent, floating point allows us to represent a Huge range of numbers....

- ❑ All numbers have same *relative* precision.
 - ❑ The numbers are not uniformly spaced.
 - ❑ Many more between 0 and 10 than between 10 and 100!

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,

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

$$\beta^L$$

$$\beta^U$$

$$\beta=2$$



Summary of Ranges for IEEE Double Precision

$$p = 53 \quad \epsilon_{\text{mach}} = 2^{-p} = 2^{-53} \approx 10^{-16}$$

$$L = -1022 \quad UFL = 2^L = 2^{-1022} \approx 10^{-308}$$

$$U = 1023 \quad OFL \approx 2^U = 2^{1023} \approx 10^{308}$$

Q: How many atoms in the Universe?

Quiz

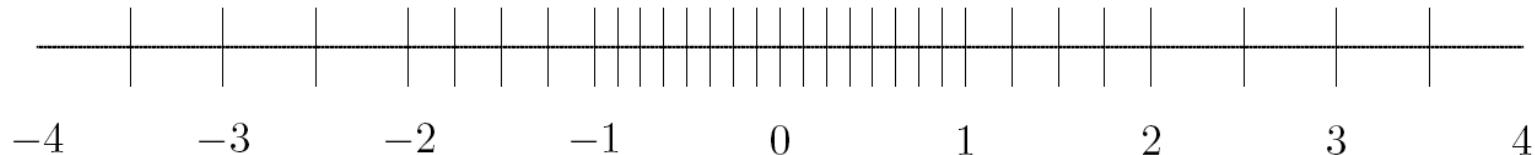
- Which of the following are exactly represented in IEEE floating point?
- a) 1
 - b) 3
 - c) 5
 - d) 0.1
 - e) 0.3
 - f) 0.5
 - g) 1/1024

Impact of Finite Precision on Approximation and Iteration

- The presence of round-off error implies that there is no point to approximating a function to a relative error that is $< \epsilon_M$.
- Similarly, with iterative methods that we'll see later in the course there is no point to driving the relative residual to $< \epsilon_M$.
- QUIZ:
 - An iteration scheme reduces the relative error by $\frac{1}{2}$ with each iteration. What is the maximum number of iterations you would need if you are working in 64-bit IEEE floating point (i.e., IEEE double precision)?
 - Another scheme reduces the relative error by a factor of 10 with each iteration. What is the maximum number of iterations required.
 - Another scheme starts with an initial relative error of 0.1, and its error is squared with each iteration. How many iterations are required to get to machine precision?

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*



Denormalizing: *normal(ized) and subnormal numbers*

- ❑ With normalization, the smallest (positive) number you can represent is:
 - ❑ $\text{UFL} = 1.00000\dots \times 2^L = 1. \times 2^{-1022} \approx 10^{-308}$
- ❑ With subnormal numbers you can represent:
 - ❑ $x = 0.00000\dots 01 \times 2^L = 1. \times 2^{-1022-53} \approx 10^{-324}$
- ❑ Q: Would you want to denormalize??
 - ❑ Cost: Often, subnormal arithmetic handled in software – sloooooow.
 - ❑ Number of atoms in universe: $\sim 10^{80}$
 - ❑ Probably, UFL is small enough.
- ❑ Similarly, for IEEE DP, OFL $\sim 10^{308} \gg$ number of atoms in universe.
→ Overflow will never be an issue (unless your solution goes unstable).

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 $0/0$, $0 * \text{Inf}$, or Inf/Inf
- `Inf` and `NaN` are implemented in IEEE arithmetic through special reserved values of exponent field
- ***Note: 0 is also a special number --- it is not normalized.***



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

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

Q: How long would it take to realize failure?



Floating-Point Arithmetic, continued

- Ideally, $x \text{ flop } y = \text{fl}(x \text{ op } y)$, i.e., floating-point arithmetic operations produce correctly rounded results
- Computers satisfying IEEE floating-point standard achieve this ideal as long as $x \text{ op } y$ 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 $1 + (\epsilon + \epsilon) > 1$



Standard Model for Floating Point Arithmetic

- Ideally, $x \text{ flop } y = \text{fl}(x \text{ op } y)$, with $\text{op} = +, -, /, ^*$.
- This standard met by IEEE.
- Analysis is streamlined using the *Standard Model*:

$$\text{fl}(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| \leq \epsilon_{\text{mach}},$$

which is more conveniently analyzed by backward error analysis.

- For example, with $\text{op} = +$,

$$\text{fl}(x + y) = (x + y)(1 + \delta) = x(1 + \delta) + y(1 + \delta).$$

- With this type of analysis, we can examine, say, floating-point multiplication.

$$x(1 + \delta_x) \cdot y(1 + \delta_y) = x \cdot y(1 + \delta_x + \delta_y + \delta_x \cdot \delta_y) \approx x \cdot y(1 + \delta_x + \delta_y),$$

which says that our relative error in multiplication is approximately $(\delta_x + \delta_y)$.

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 Example

- Cancellation leads to promotion of garbage into “significant” digits

$$\begin{array}{rcl} x & = & 1 \ . \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ b \ b \ g \ g \ g \ g \ e \\ y & = & 1 \ . \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ b' \ b' \ g \ g \ g \ g \ e \\ x - y & = & 0 \ . \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ b'' \ b'' \ g \ g \ g \ g \ e \\ & = & b'' \ . \ b'' \ g \ g \ g \ ? \ ? \ ? \ ? \ ? \ ? \ ? \ ? \ ? \ ? \ e - 9 \end{array}$$

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
- *Of the basic operations, + - * / , with arguments of the same sign, only subtraction has cond. number significantly different from unity. Division, multiplication, addition (same sign) are OK.*



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

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

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



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



Finite Difference Example

- What happens when we use first-order finite differences to approximate $f'(x)$.

$$\text{fl} \left(\frac{\delta f}{\delta x} \right) = \frac{\hat{f}(\hat{x} + \hat{h}) - \hat{f}(\hat{x})}{\hat{h}} =: \frac{\hat{f}_1 - \hat{f}_0}{\hat{h}}.$$

- We know that $f(x)$ will be represented only to within relative tolerance of ϵ_{mach} .

$$\begin{aligned}\hat{f}_1 &= f(\hat{x} + \hat{h})(1 + \epsilon_1) \\ \hat{f}_0 &= f(\hat{x})(1 + \epsilon_0)\end{aligned}$$

with $|\epsilon_0| \leq \epsilon_{\text{mach}}$ and $|\epsilon_1| \leq \epsilon_{\text{mach}}$.

- The other error terms are smaller in magnitude (i.e., higher powers in h and/or ϵ_{mach}), and we have

$$\begin{aligned}\text{fl} \left(\frac{\delta f}{\delta x} \right) &\approx \frac{f_1 - f_0}{h} + \frac{f_1 \epsilon_1 - f_0 \epsilon_0}{h} \\ &\approx \frac{f_1 - f_0}{h} + \frac{\epsilon_1 - \epsilon_0}{h} f(x).\end{aligned}$$

- The last term is bounded by

$$\left| \frac{\epsilon_1 - \epsilon_0}{h} f(x) \right| \leq 2 \frac{\epsilon_{\text{mach}}}{h} |f(x)|.$$