Secure https://relate.cs.illinois.edu/course/cs450-f17/

Bookmarks BookmarksBar BookmarksMenu com.apple.ReadingList

CS450 Fall 2017 Jump to Student Grading Content Staff Signed in as fischerp@illinois.edu

CS 450 Numerical Analysis

course description	time	location
Course catalog entry	3:30-4:45pm MW	Digital Computer Laboratory

Course staff

Instructor

name	email	office hours	office
Prof. Paul Fischer	fischerp@illinois.edu	Thursday 9:30 - 11:00	4320 Siebel Center

Teaching assistants

name	email	office hours	location
Josh Bevan	jjbevan2@illinois.edu	TBD	SC 0207
Nick Christensen	njchris2@illinois.edu	Wednesday 1:00 - 3:00	SC 0207
Setare Hajarolasvadi	hajarol2@illinois.edu	Monday 9:00 - 11:00	SC 0207
Thilina Rathnayake	rbr2@illinois.edu	Tuesday 1:00 - 3:00	SC 0207

Textbook

Scientific Computing: An Introductory Survey by Michael T. Heath, McGraw-Hill, 2nd edition, 2002

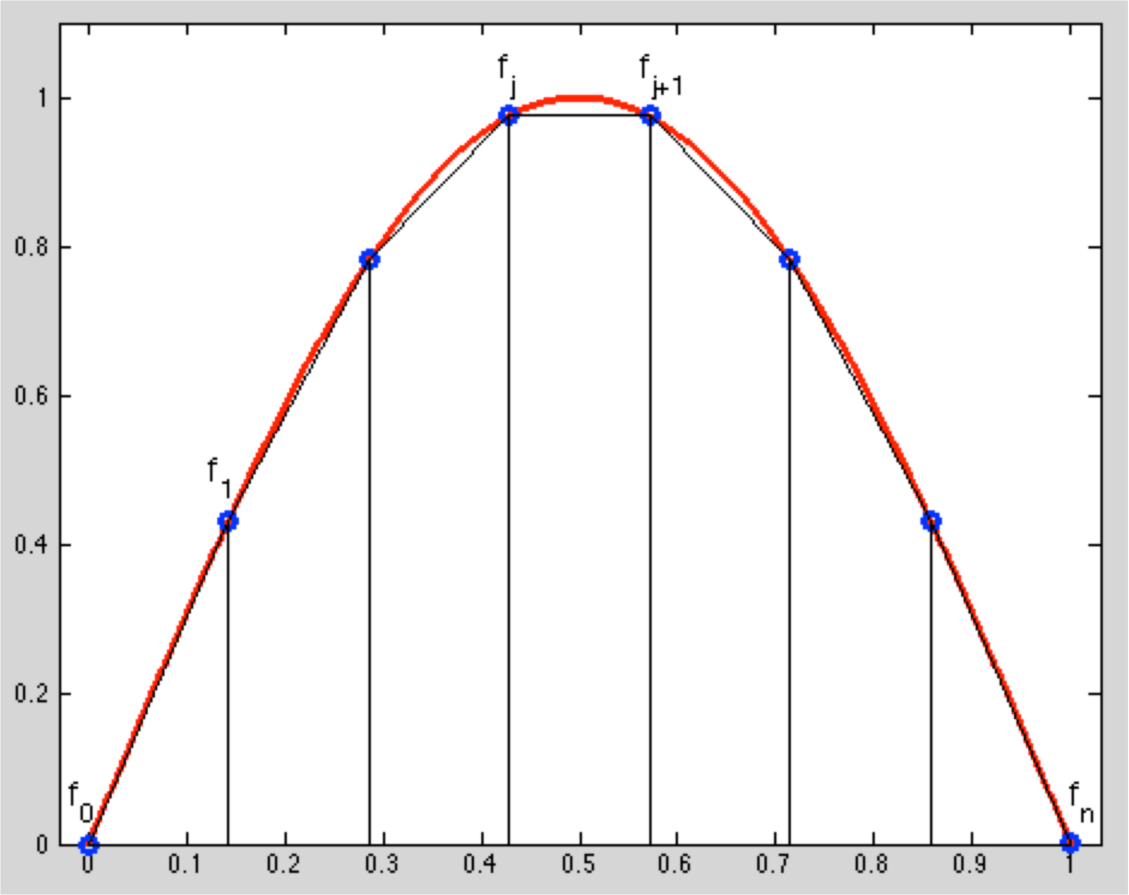
Outline

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

Example: Numerical Integration with Trapezoid Rule

- Evaluate f(x) at n+1 points, $x_j = a+jh$, $h:=(b-a)/n$
- Sum the areas under the n trapezoidal panels; denote result as T_n.
- *Q: How large must n be for "suitably small" error, E_n ?*



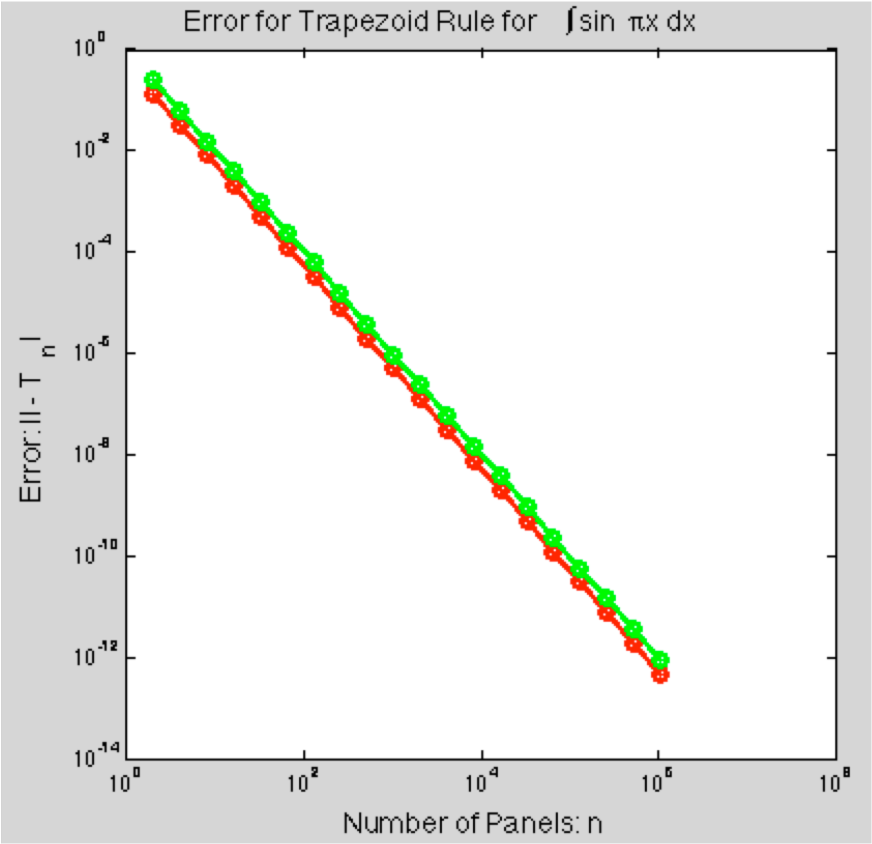
$$\mathcal{I} = \int_0^1 \sin \pi x \, dx$$

$$T_n = h \left[\frac{f_0}{2} + f_1 + f_2 \cdots + f_{n-1} + \frac{f_n}{2} \right]$$

$$E_n := I - T_n$$

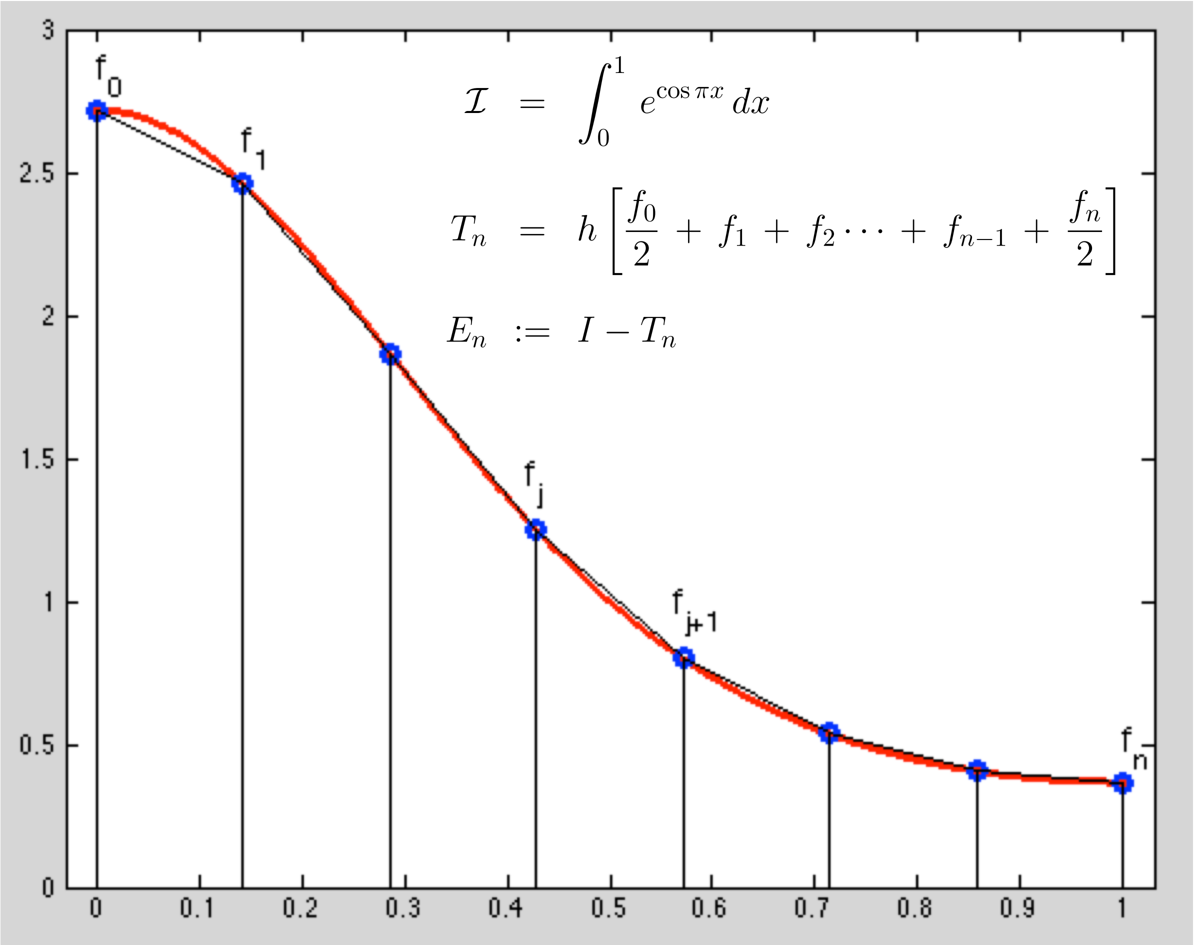
Example: Numerical Integration Using Trapezoid Rule

- For f(x)=sin(π x), we see that the error scales like $1/n^2$.
- This is generally the expected behavior for the trapezoid rule, but not always.



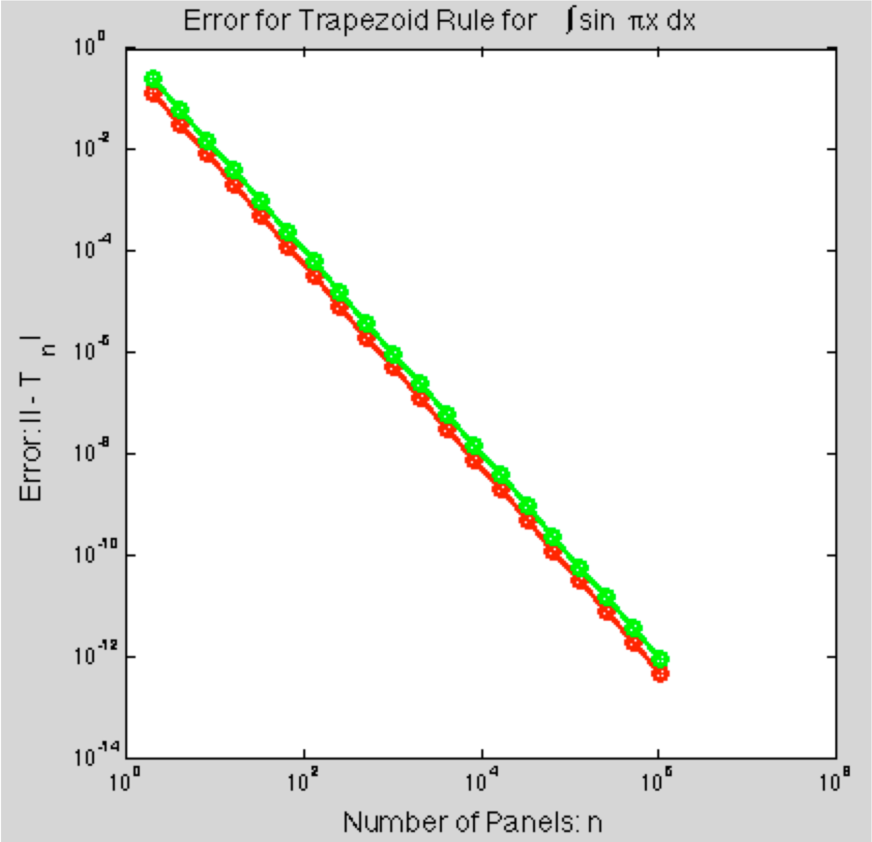
Example: Numerical Integration with Trapezoid Rule

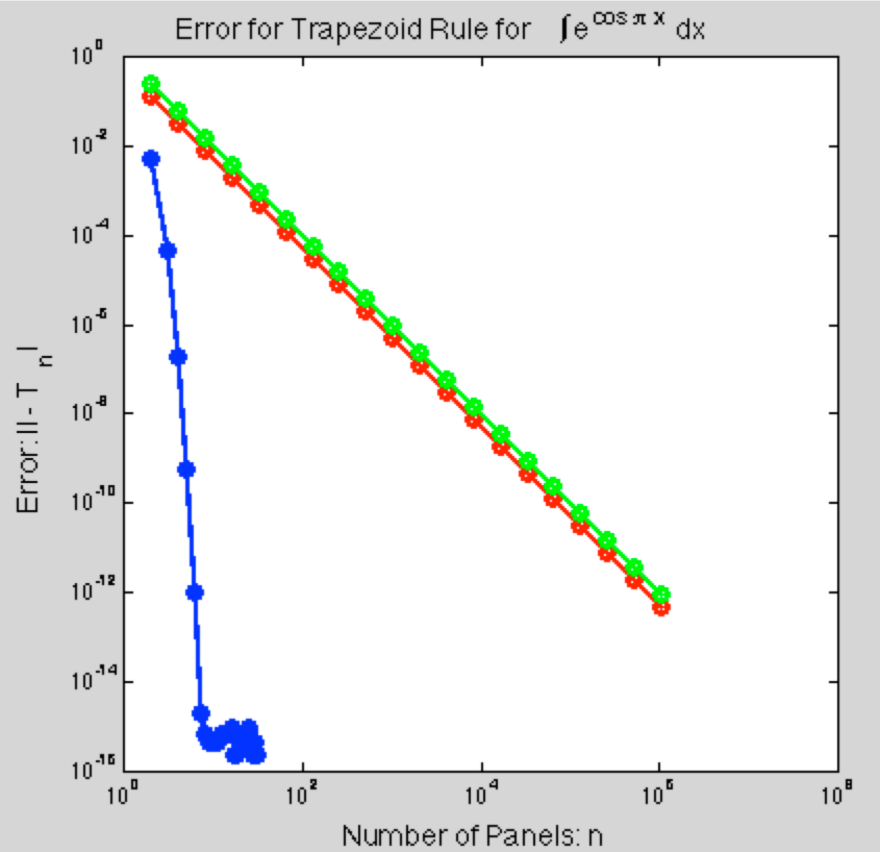
- Evaluate f(x) at n+1 points, x_j = a+jh, h:=(b-a)/n
- Sum the areas under the n trapezoidal panels, denote result as T_n.
- *Q: How large must n be for "suitably small" error ?*



Example: Numerical Integration Using Trapezoid Rule

- For $f(x)=\sin(\pi x)$, we see that the error scales like $1/n^2$
- For $f(x) = e^{\cos \pi x}$, the error scales like e^{-cn}, for some positive constant, c.





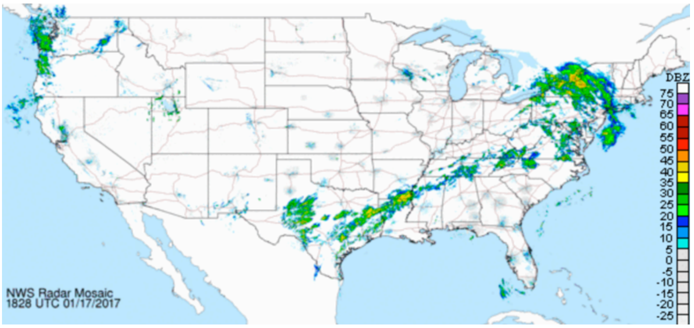
Example: *Convective Transport*

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

- 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)*

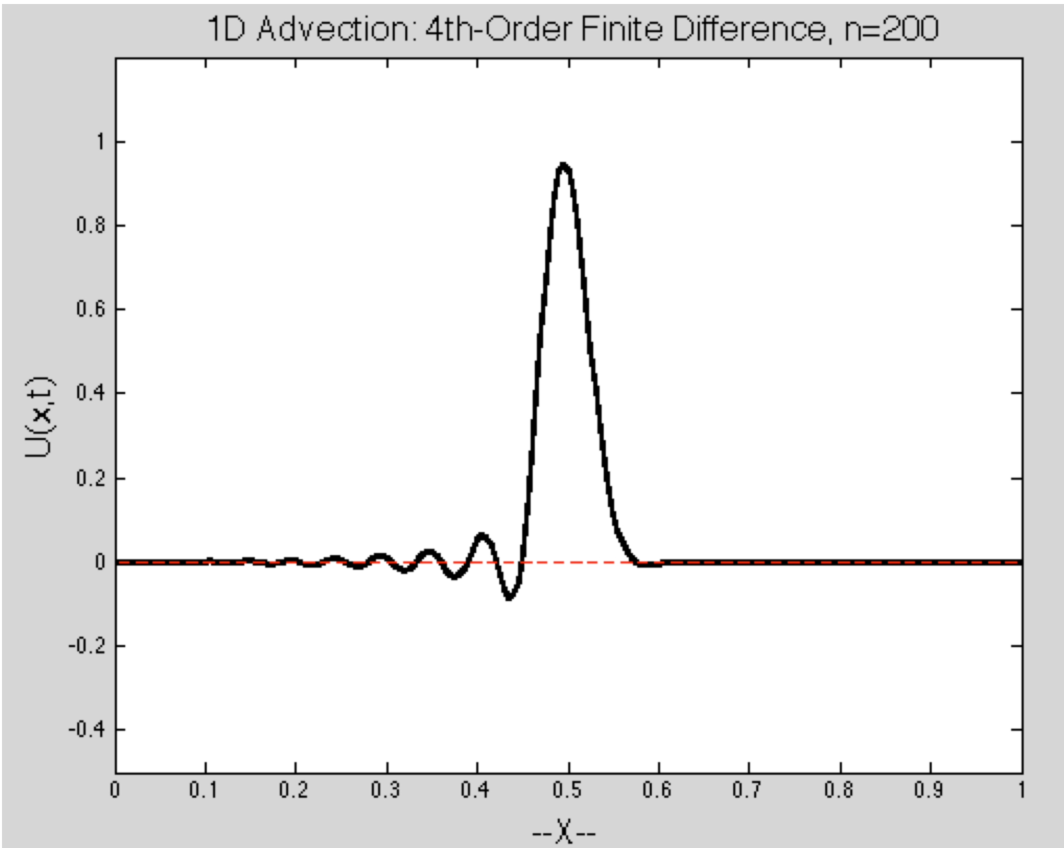


Example: *Convective Transport*

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

(See Fig. 11.1 in text.)

<example: convect_demo>



Characteristics of Numerical Computations

- **Problem** (given):
  - Easy (well-posed)
  - Hard (ill-posed)

- **Method**:
  - Good (stable & accurate)
  - Bad (unstable or inaccurate)

- **Outcome**:
  - Good
  - Garbage
  - Partially Garbage

- NOTES:
  - Trust, but ***verify !***
  - If initial problem is ill-posed, it may be possible to reformulate to an easier problem to solve.

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) × (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: $\hat{f}(\hat{x}) - f(x) =$

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

computational error + propagated data error

- Algorithm has no effect on propagated data error

Data Error and Computational Error

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

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

computational error + propagated data error

- Algorithm has no effect on propagated data error

Taylor Series (Very important for SciComp!)

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

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

- Specifically, this implies

  $$\left| f(x+h) \;-\; \left(f(x) \;+\; hf'(x) \;+\; \frac{h^2}{2} f''(x) \;+\; \cdots \;+\; \frac{h^{m-1}}{(m-1)!} f^{(m-1)}(x) \right) \right| \;\leq\; \left| \frac{h^m}{m!} f^{(m)}(\xi) \right|$$

  with the net result that the Taylor series converges as $h \longrightarrow 0$ for m and *x fixed.*

Taylor Series

- Basically, assuming that $f'(x) \neq 0$, this implies that $f(x)$ looks like a *line* as you zoom in near x.
- Moreover, we can use this result to derive approximations to derivatives of $f(x)$.
- Take $m = 2$:

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

Truncation error

- Take $m = 2$:

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

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

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

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

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

- Take $m = 2$:

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

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

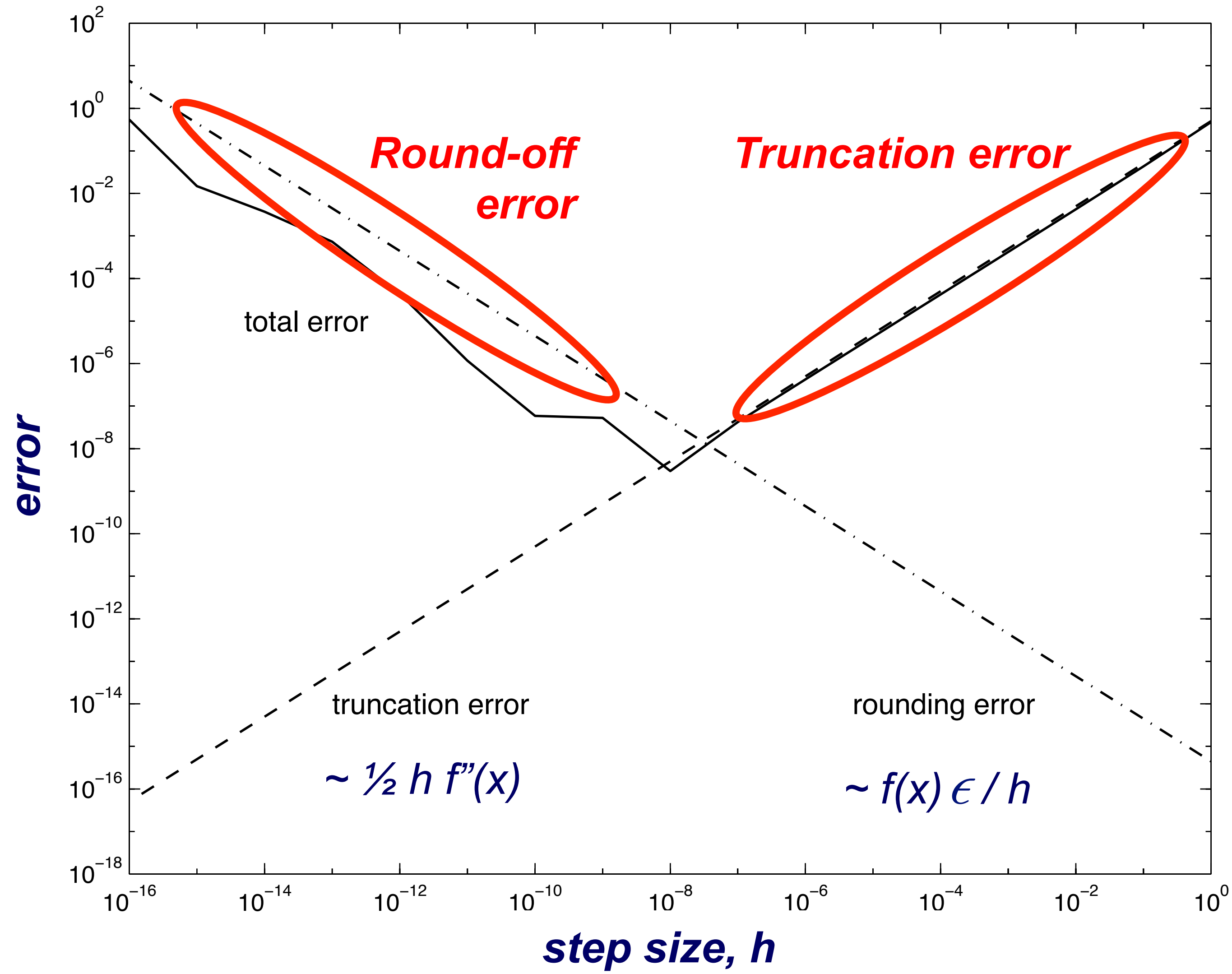
- **Q:** Suppose $|f''(x)| \approx 1$.

  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

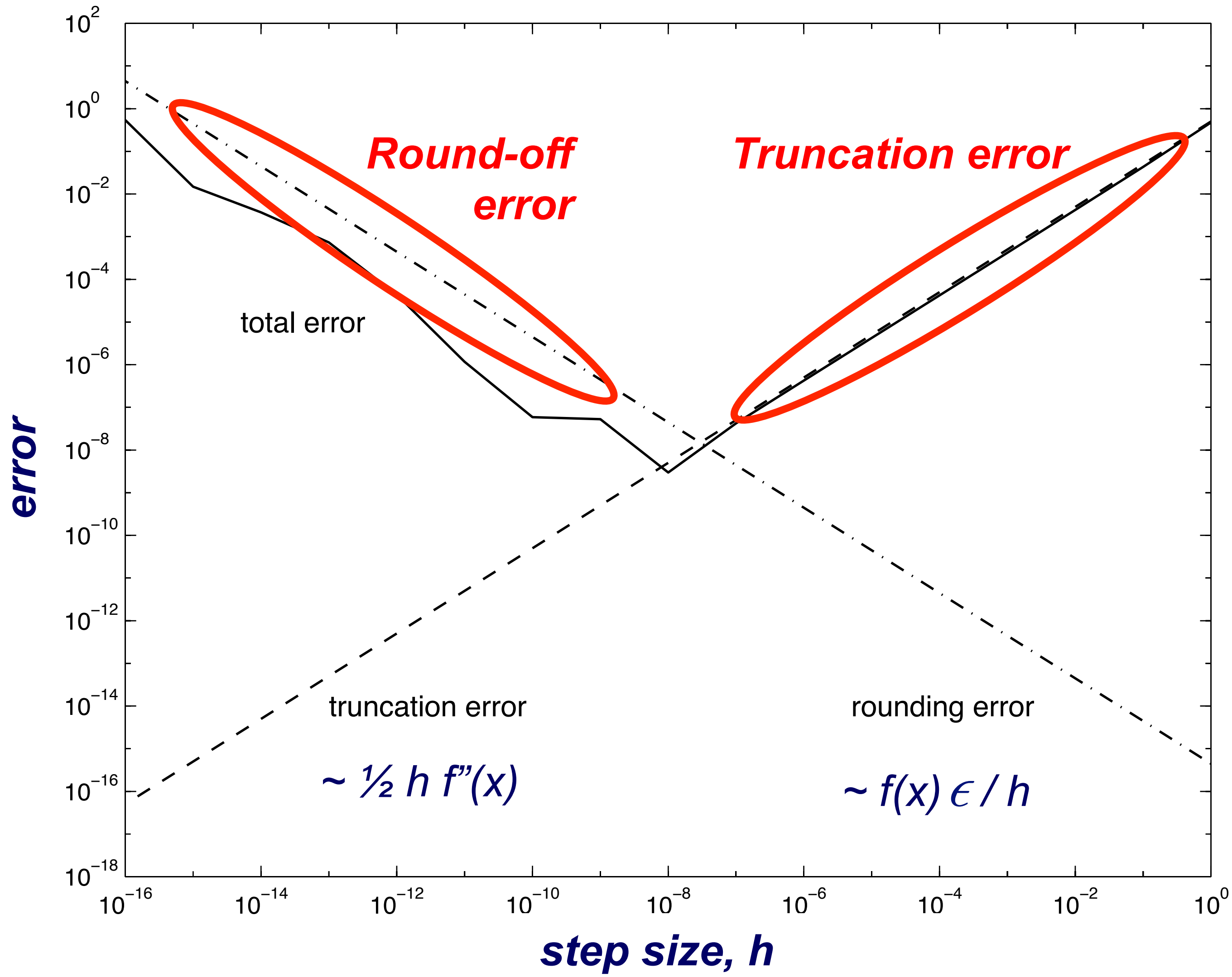
- Error in finite difference approximation

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

  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 machine precision.

- 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.

  (Machine Learning is moving towards 16-bit precision…)

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*: $\Delta y = \hat{y} - y$
- *Backward error*: $\Delta x = \hat{x} - x$, where $f(\hat{x}) = \hat{y}$

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

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*:

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

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

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

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:

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

Condition Number

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

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

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

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

Example: Evaluating Function

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

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

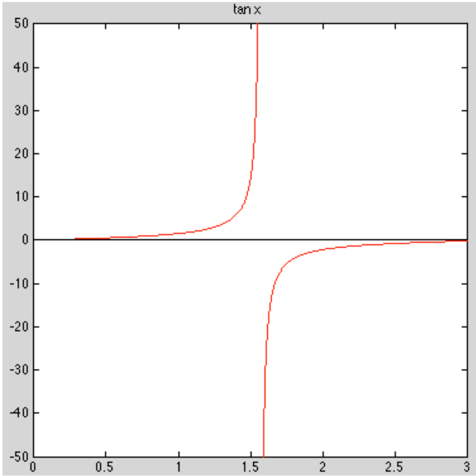
  Relative forward error: $$\frac{f(x + \Delta x) - f(x)}{f(x)} \approx \frac{f'(x)\Delta x}{f(x)}$$

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

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

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 $\approx 2.48275 \times 10^5$



Condition Number Examples

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

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

$$
\begin{aligned}
f(x) &= a\,x \\
f(x) &= \frac{a}{x} \\
f(x) &= a + x
\end{aligned}
$$

Condition Number Examples

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

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

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

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

- 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 in 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

$$
\begin{array}{rcrcl}
x & = & 3141592653589793238462643383279502884197169399375105820974944.9230781... & = & \pi \times 10^{60} \\
\hat{x} & = & 3141592653589793000000000000000000000000000000000000000000000.0000000... & \approx & \pi \times 10^{60} \\
\hline
x - \hat{x} & = & 238462643383279502884197169399375105820974944.9230781... & = & 2.3846... \times 10^{44} \\
& \approx & .7590501687441757 \times 10^{-16} \times x & & \\
& < & 1.110223024625157e-16 \times x & & \\
& \approx & \epsilon_{\text{mach}} \times x. & &
\end{array}
$$

- 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 $0 \leq d_i \leq \beta - 1, \; i = 0, \ldots, p-1, \;$ 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 SP	2	24	-126	127
IEEE DP	2	$\boxed{53}$	-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,

$$\pi \approx 1.1001001000011111101101010100010001000010110100011 \times 2^1$$

- In reality,

$$\pi = 1.10010010000111111011010101000100010000101101000110000100011010 \cdots \times 2^1$$

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

$$\pi = 0.00000000000000000000000000000000000000000000000000000100011010 \cdots \times 2^1$$

In this case, we get lucky and we have more than 53 bits correct because of the trail of 0 bits after the 53rd…

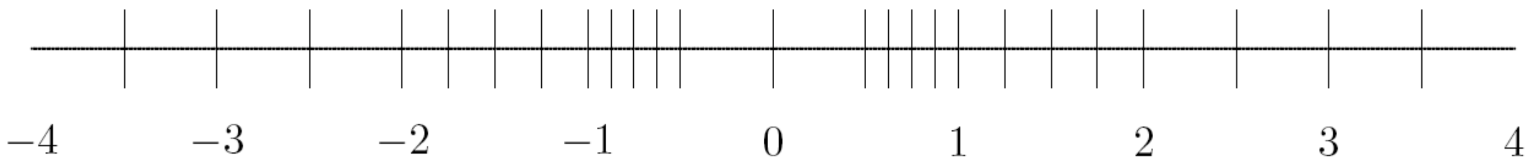
Properties of Floating-Point Systems

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

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

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,

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

can be conveniently thought of as:

$$\mathrm{fl}(x) \;=\; x\,(1+\epsilon_x)$$

$$|\epsilon_x| \;\le\; \epsilon_{\mathrm{mach}}$$

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

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

Advantage of Floating Point

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

```
3.141592653589793 e 307
         :
         :
         :
314159265358979300000000000000000000000000000000000000000000.
       31415926535897930000000000000000000000000000000000000.
              3141592653589793000000000000000000000000000000.
                       3141592653589793000000000000000000000.
                              314159265358979300000000000000.
                                     31415926535897930000000.
                                            3141592653589793.
                                                    31415926.53589793
                                                           3.141592653589793
                                                            .000000314159265358979 3
                                                            .0000000000003141592653589793
                                                            .0000000000000000003141592653589793
                                                            .0000000000000000000000003141592653589793
                                                            .0000000000000000000000000000003141592653589793
                                                            .0000000000000000000000000000000000003141592653589793
                                                                   :
                                                                   :
                                                                   :
                                                           3.141592653589793 e-307
```

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

Relative Precision Example

Let's look at the highlighted entry from the preceding slide.

$$
\begin{array}{rcl}
x &=& 3141592653589793238462643383279502884197169399375105820974944.9230781\ldots \;\;=\;\; \pi \times 10^{60} \\
\hat{x} &=& 3141592653589793000000000000000000000000000000000000000000000.0000000\ldots \;\;\approx\;\; \pi \times 10^{60}
\end{array}
$$

$$
\begin{array}{rcl}
x - \hat{x} &=& 2384626433832795028841971693993751058209749 44.9230781\ldots \;\;=\;\; 2.3846\ldots \times 10^{44} \\
&\approx& .7590501687441757 \times 10^{-16} \;\times\; x \\
&<& 1.110223024625157e - 16 \;\times\; x \\
&\approx& \epsilon_{\mathrm{mach}} \;\times\; x.
\end{array}
$$

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

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

- The *relative* error, however, is

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

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,

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

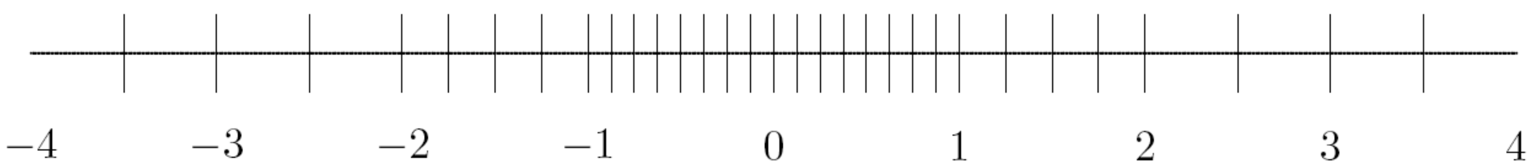
Summary of Ranges for IEEE Double Precision

$$
\begin{array}{rclcl}
p & = & 53 & \epsilon_{\text{mach}} = 2^{-p} = 2^{-53} \approx 10^{-16} \\
L & = & -1022 & UFL = 2^{L} = 2^{-1022} \approx 10^{-308} \\
U & = & 1023 & OFL \approx 2^{U} = 2^{1023} \approx 10^{308}
\end{array}
$$

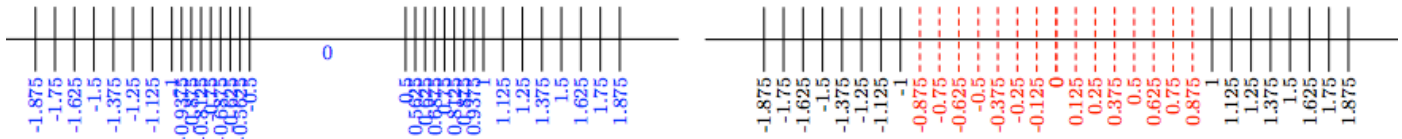
Q: How many atoms in the Universe?

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*



(a) Without subnormal numbers. (b) With subnormal numbers (dashe red).

ig. 2. The *tiny* floating-point format with and without subnormals (focus c

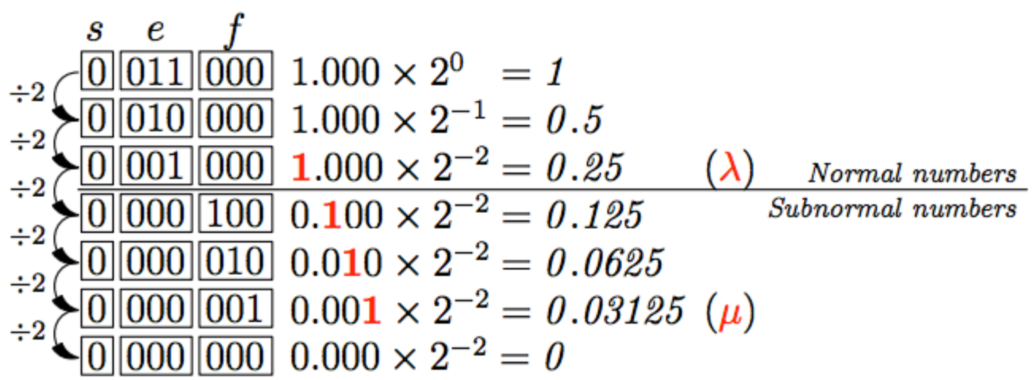


Fig. 3. Repeated division by two from 1.0 to 0.0 in the *tiny* format

Denormalizing: *normal(ized) and subnormal numbers*

- With normalization, the smallest (positive) number you can represent is:
  - UFL = 1.00000… x 2^{L} = 1. x 2^{-1022} ~= 10^{-308}
- With subnormal numbers you can represent:
  - x = 0.00000…01 x 2^{L} = 1. x $2^{-1022-53}$ ~= 10^{-324}
- Q: Would you want to denormalize??
  - Cost: Often, subnormal arithmetic handled in software – slooooow.
  - Number of atoms in universe: ~ 10^{80}
  - Probably, UFL is small enough.
  - Similarly, for IEEE DP, OFL ~ 10^{308} >> 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 $0/0$, $0 * \texttt{Inf}$, or $\texttt{Inf}/\texttt{Inf}$
- `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, $x \texttt{ flop } y = \mathrm{fl}(x \texttt{ op } y)$, i.e., floating-point arithmetic operations produce correctly rounded results
- Computers satisfying IEEE floating-point standard achieve this ideal 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 $1 + (\epsilon + \epsilon) > 1$

Standard Model for Floating Point Arithmetic

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

$$\mathrm{fl}(x \,\mathrm{op}\, y) \;=\; (x \,\mathrm{op}\, y)(1+\delta), \qquad |\delta| \le \epsilon_{\mathrm{mach}},$$

  which is more conveniently analyzed by backward error analysis.

- For example, with op = +,

$$\mathrm{fl}(x+y) \;=\; (x+y)(1+\delta) \;=\; x(1+\delta) \,+\, y(1+\delta).$$

- 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

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

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

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

  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

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

Finite Difference Example

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

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

- 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 $|\epsilon_0| \leq \epsilon_{\mathrm{mach}}$ and $|\epsilon_1| \leq \epsilon_{\mathrm{mach}}$.

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

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

- The last term is bounded by

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

fdiff_demo.m