CS450 / ECE 491: Introduction to Scientific Computing

- Course webpage:
  https://relate.cs.illinois.edu/course/cs450-s25/
- Quizzes, homework, lecture notes, links to recorded lectures will be found on the Relate page.
- Homework and quiz submissions will also be submitted on the Relate page.
  - Quizzes due before each lecture
  - Homeworks are weekly, except for Exam weeks
  - 4-cr hour will have two projects, counting for ~3 HWs
- Midterm exams (best 3 of 4) and final exam will be at CBTF.

CS 450 Staff

- Paul Fischer: 4320 Siebel
  - fischerp@illinois.edu
  - Off. Hours: Tue 12:30 - 1:30
  - Thu 12:30 - 1:30

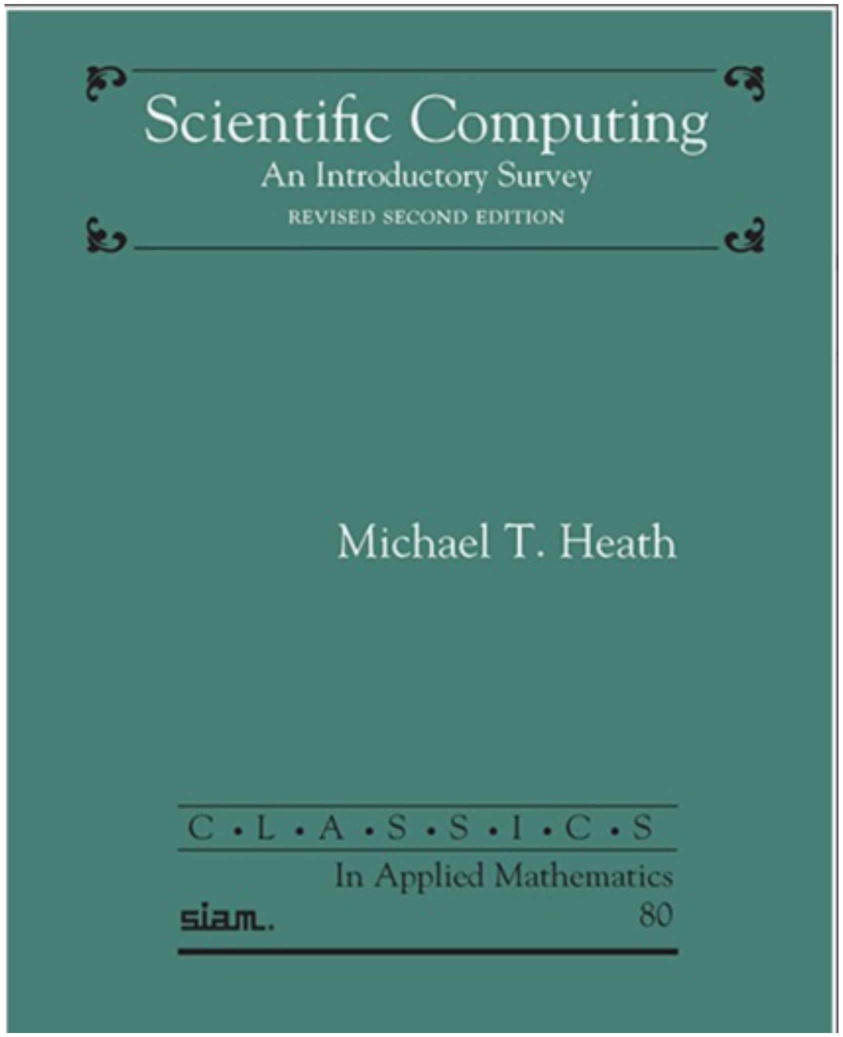
- Hansheng Liu (TA)
  - 2-3 Thu & TBD

- Jonathan Wang (TA)
  - TBD

- Ruining Zhao
  - Monday 4-5 PM
  - Friday 2-3 PM

- Office hours will be in the CS tutorial space in the basement of Siebel, save for my office hours on Thursday, which will be in my office.
- These are tentative hours
- Up-to-date schedule will be posted on the course Relate page.

Text Book: *Scientific Computing*, M. Heath


Scientific Computing
An Introductory Survey
REVISED SECOND EDITION
Michael T. Heath
C • L • A • S • S • I • C • S
In Applied Mathematics
siam.
80

- The lectures will be designed to expand on and complement the text.
- HWs, quizzes, exams are mostly drawn from the text material.
- Slides / demos will be posted on the Relate page.
- Lectures will be recorded.

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

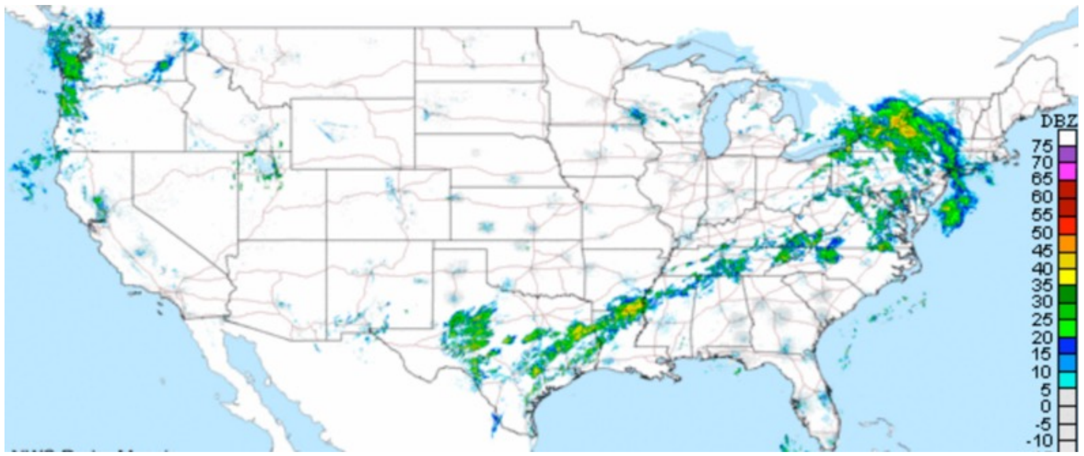
Simulation Example: Convective Transport

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

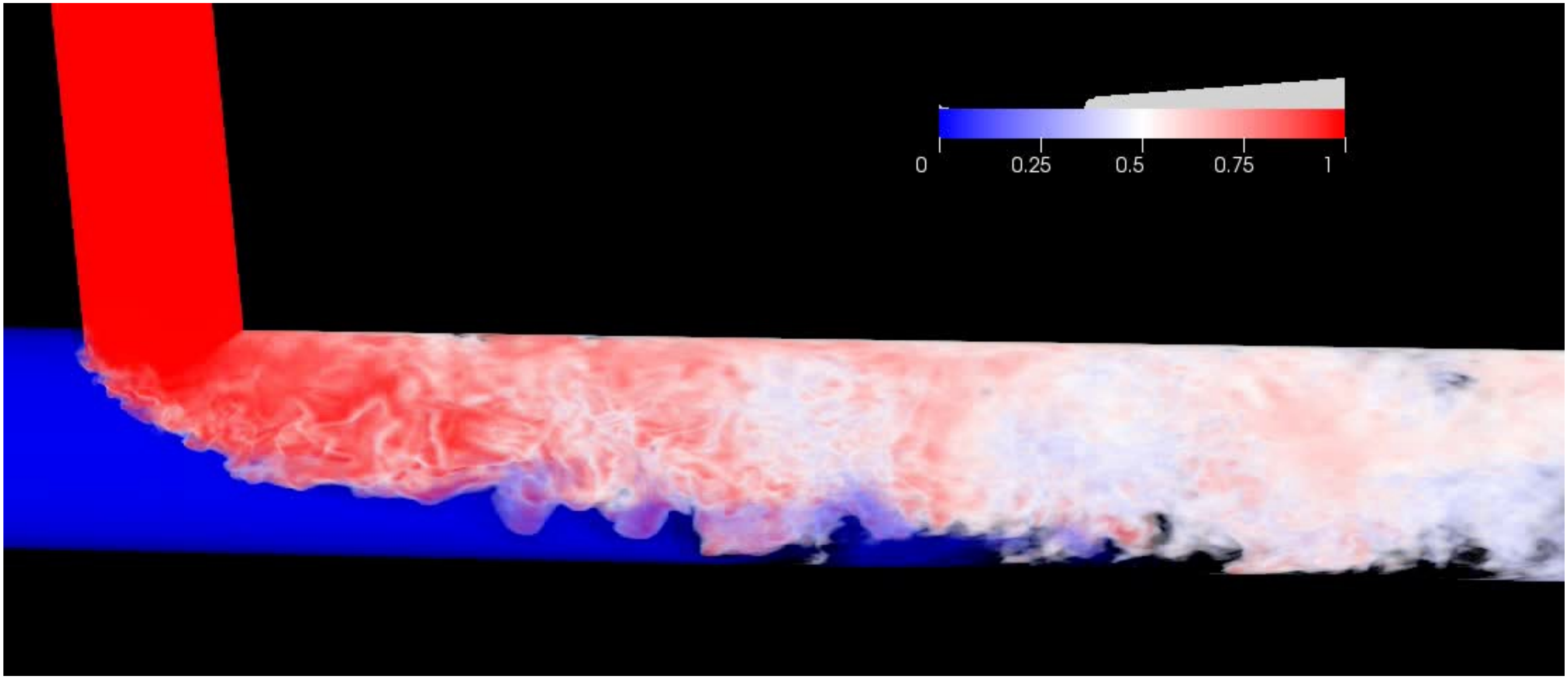
- Examples:
  - Ocean currents:
    - Pollution
    - Saline
    - Thermal transport
  - Atmosphere
    - Climate
    - Weather
  - Industrial processes
  - Combustion
    - Automotive engines
    - Gas turbines

- Problem Characteristics:
  - Large (sparse) linear systems: $n=10^6 - 10^{12}$

- Demands:
  - *Accurate approximations*
  - *Fast (low-cost) algorithms*
  - *Stable algorithms*



Simulation Example: Convective Transport



- Temperature distribution in hot + cold mixing at T-junction
  - 100 million dofs: 20 hours runtime on 16384 cores
  - Solve several $10^8 \times 10^8$ linear systems every second

Numerical Simulation

- Related course material

Linear systems	- chapter 2
Eigenvalues / eigenvectors	- chapter 4
Interpolation	- chapter 7
Numerical integration/differentiation	- chapter 8
Initial value problems	- chapter 9
Bounary value problems	- chapter 10
Numerical PDEs (simplified)	- chapter 11

Data Fitting / Optimization

- Examples
  - Weather prediction (data assimilation)
  - Image recognition
  - Etc.

- Related course material

Linear / Nonlinear Least Squares	- chapter 3
Singular Value Decomposition	- chapter 3/4
Nonlinear systems	- chapter 5
Optimization	- chapter 6
Interpolation	- chapter 7

Main Topics / Take-Aways for Chapter 1 (1/2)

- Conditioning of a problem
- Condition number
- Stability of an algorithm
- Errors
  - Relative / absolute error
  - Total error = computational error + propagated-data error
  - Truncation errors
  - Rounding errors

Main Topics / Take-Aways for Chapter 1 (2/2)

- Floating point numbers: IEEE 64
- Floating point arithmetic
  - Rounding errors
  - The Standard Model: $fl(a \circ b) = (a \circ b)(1 + \epsilon)$
  - Commutativity and associativity
  - Cancellation

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 perturbations in input data

- Computational algorithm should not make sensitivity worse.

General Strategy

- Replace difficult problem by easier one having same or closely related solution
  - infinite dimensional problem $\longrightarrow$ finite dimensional one
  - differential equation $\longrightarrow$ algebraic equation
  - nonlinear problem $\longrightarrow$ (sequence of) linear problems
  - complicated $\longrightarrow$ simple
- Solution obtained may only *approximate* that of the original problem

General Strategy, Examples

- If $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ is a polynomial $\approx f(x)$, then we can readily evaluate $\int p\,dx \approx \int f\,dx$, where the original integral may be inaccessible.

- Here, we have a finite-dimensional problem of finding $n+1$ coefficients, a_j, such that $p(x) \approx f(x)$.

- For nonlinear systems of the form $\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{f}$, we can set up an iteration of the form $\mathbf{A}(\mathbf{x}_k)\mathbf{x}_{k+1} = \mathbf{f}$, starting with an initial guess $\mathbf{x}_0$.

- This requires repeated solutions of *linear* systems, which is generally easier.

- More sophisticated iteration strategies can be used for rapid convergence and robustness.

Sources of Approximation

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

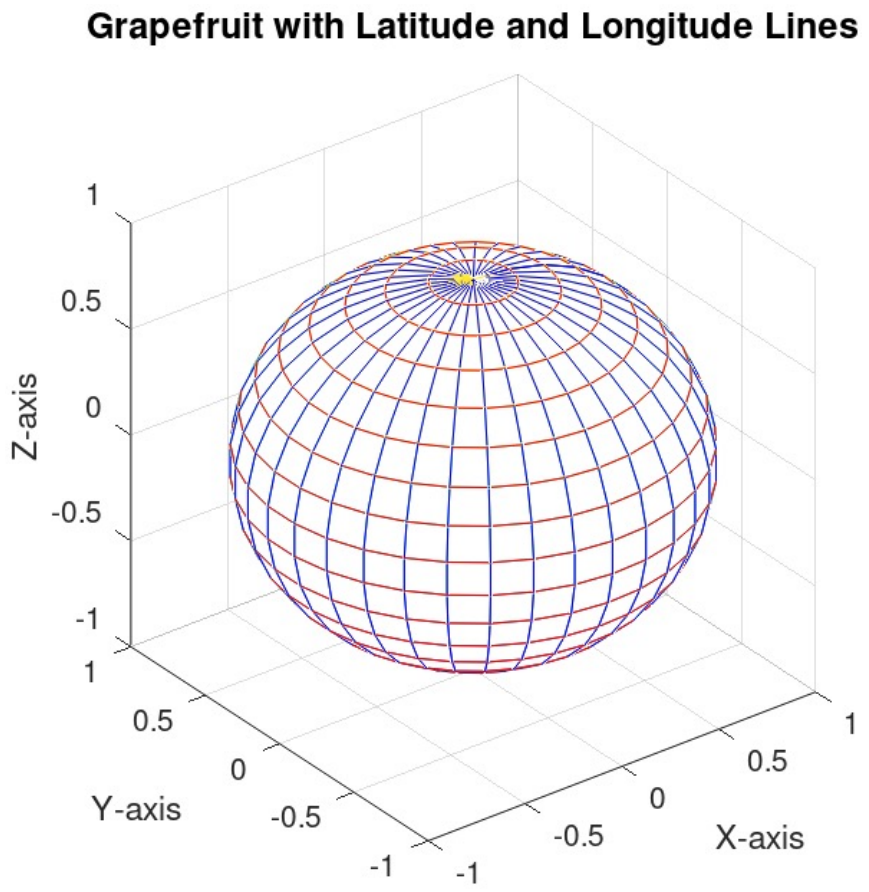
Example: Approximation

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

Aside: About Shape of the Earth and Precision

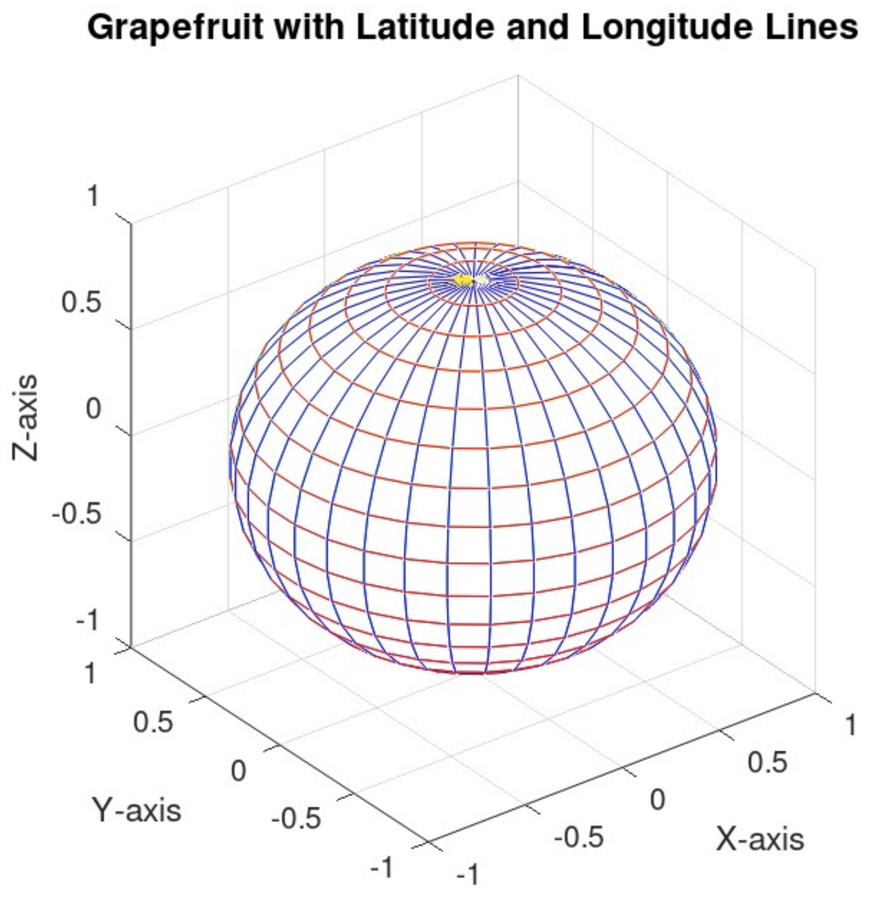
- *Degree of latitude at the north/south pole ~ 111.7 km*
- *Degree of latitude at the equator ~ 110.5 km*

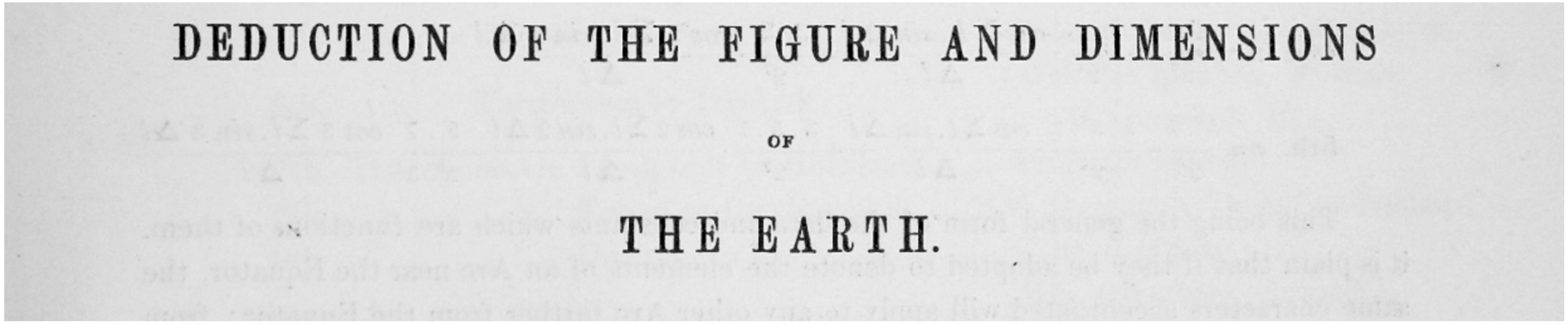
- *Measurements were carried out by French expeditions to Lapland and Ecuador in the **1730**s to determine shape of the Earth, and to define the **meter**.*
- *Original definition of the meter was one 10millionth of the distance from the north pole to the equator.*



Aside: About Shape of the Earth and Precision

- ***Q:*** *What is the polar circumference of the Earth?*
- ***A:*** *40 million meters = 40,000 km*
- *More modern measurements put this value at 40,008 km.*
- *So, the original measurements were accurate to 0.02% -* ***in the final result!***
- *Many of the techniques discussed in this course, (least squares, Gauss-Seidel relaxation, normal distributions, etc.) were developed for accurate measurement of the Earth.* ***How accurate****?*



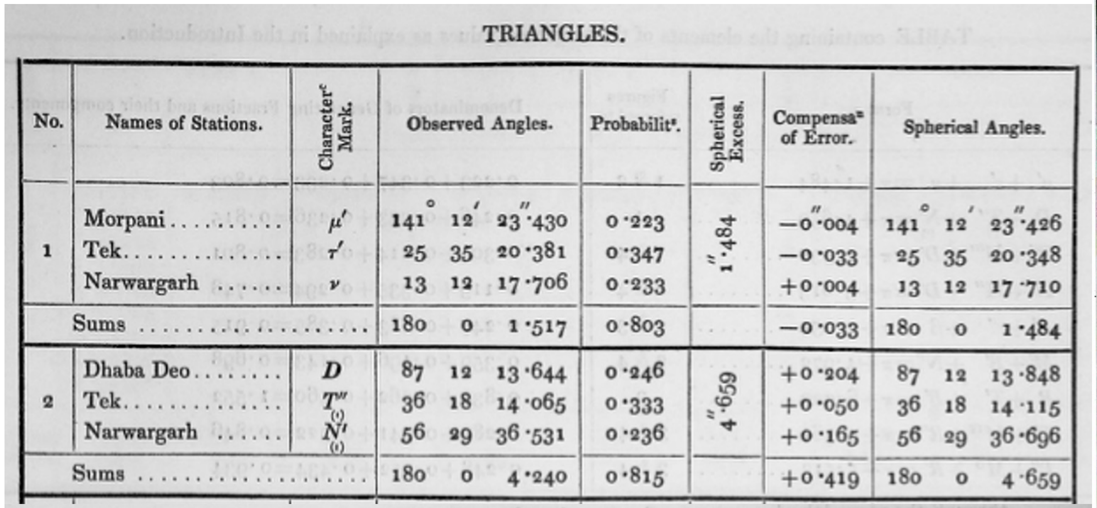


DEDUCTION OF THE FIGURE AND DIMENSIONS

OF

THE EARTH.

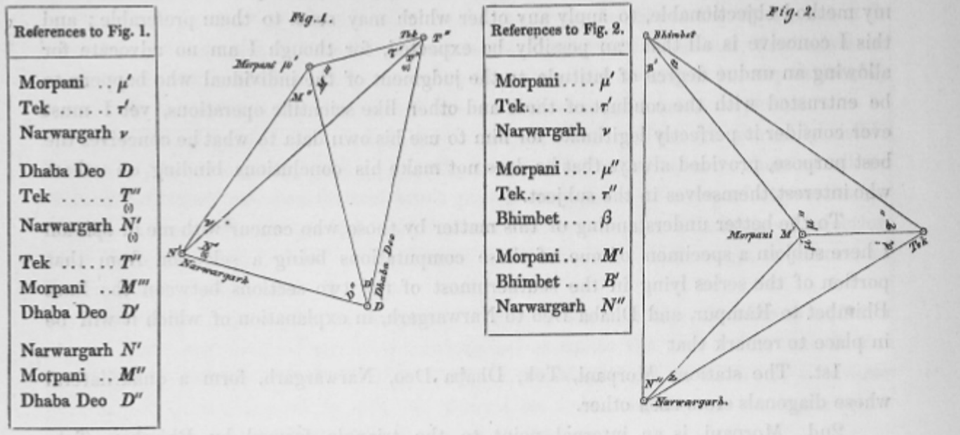
*An Account of the Measurement of Two Sections of the Meridional Arc of India, Bounded by the Parallels of 18° 3' 15" and 29° 30' 48". **Everest, Lieut.-Colonel George**. 1847*



TRIANGLES.

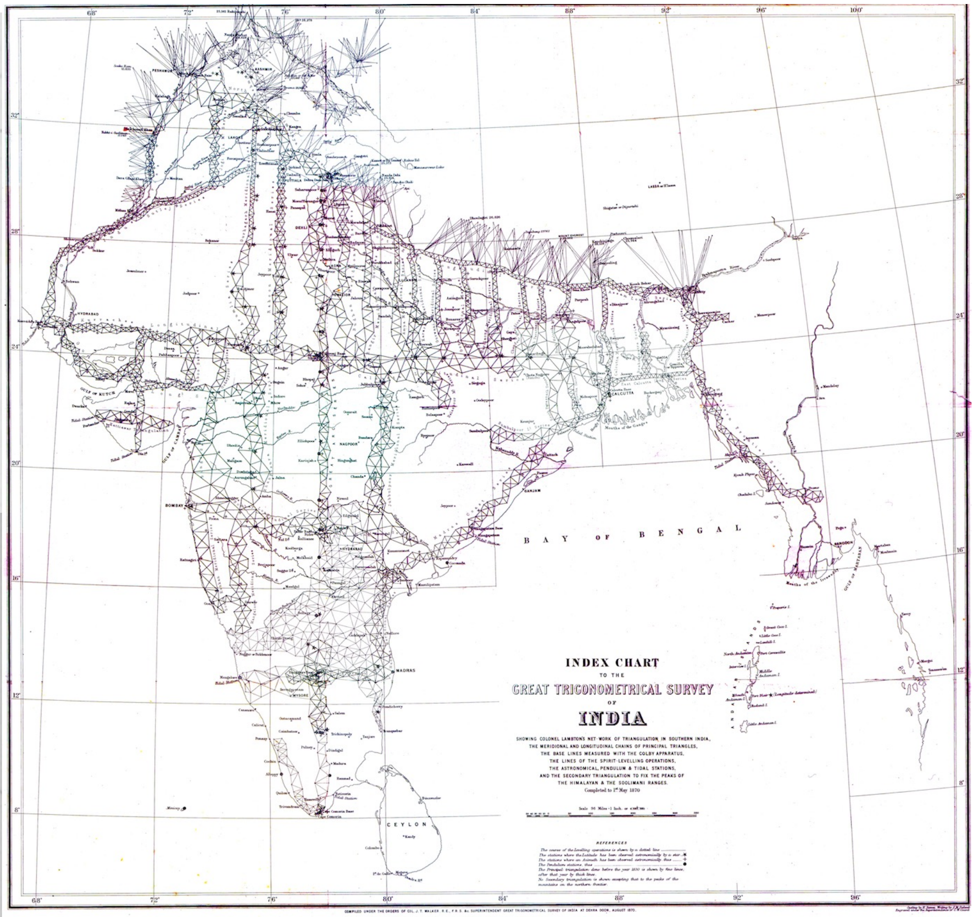
No.	Names of Stations.	Character Mark.	Observed Angles.			Probabilit^y.	Spherical Excess.	Compensa^n of Error.	Spherical Angles.		
			°	′	″			″	°	′	″
1	Morpani	μ'	141	12	23·430	0·223	1″·484	−0·004	141	12	23·426
	Tek..............	τ'	25	35	20·381	0·347		−0·033	25	35	20·348
	Narwargarh	ν	13	12	17·706	0·233		+0·004	13	12	17·710
	Sums		180	0	1·517	0·803		−0·033	180	0	1·484
2	Dhaba Deo........	D	87	12	13·644	0·246	4″·659	+0·204	87	12	13·848
	Tek..............	$\underset{(1)}{T'''}$	36	18	14·065	0·333		+0·050	36	18	14·115
	Narwargarh	$\underset{(1)}{N'}$	56	29	36·531	0·236		+0·165	56	29	36·696
	Sums		180	0	4·240	0·815		+0·419	180	0	4·659

DIAGRAMS of the MORPANI POLYGON.



References to Fig. 1.	
Morpani ..	μ'
Tek	τ'
Narwargarh	ν
Dhaba Deo	D
Tek	$\underset{(1)}{T'''}$
Narwargarh	$\underset{(1)}{N'}$
Tek	T''
Morpani ..	M'''
Dhaba Deo	D'
Narwargarh	N'
Morpani ..	M''
Dhaba Deo	D''

References to Fig. 2.	
Morpani....	μ'
Tek........	τ'
Narwargarh	ν
Morpani....	μ''
Tek........	τ''
Bhimbet	β
Morpani	M'
Bhimbet	B'
Narwargarh	N''



BAY OF BENGAL
CEYLON
INDEX CHART
TO THE
GREAT TRIGONOMETRICAL SURVEY
OF
INDIA
SHOWING COLONEL LAMBTON'S NET-WORK OF TRIANGULATION IN SOUTHERN INDIA,
THE MERIDIONAL AND LONGITUDINAL CHAINS OF PRINCIPAL TRIANGLES,
THE BASE LINES MEASURED WITH THE COLBY APPARATUS,
THE LINES OF THE SPIRIT-LEVELLING OPERATIONS,
THE ASTRONOMICAL, PENDULUM & TIDAL STATIONS,
AND THE SECONDARY TRIANGULATION TO FIX THE PEAKS OF
THE HIMALAYAN & THE SOOLIMANI RANGES.
Completed to 1st May 1870
REFERENCES

Absolute Error and Relative Error

- ***Absolute error:*** **approximate value − true value**

- ***Relative error:***
$$\frac{\textbf{approximate value} - \textbf{true value}}{\textbf{true value}}$$

- Equivalently,
$$\textbf{approx value} = (\textbf{true value}) \times (\textbf{1+rel error})$$

- 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

Approximate Relative Error

- If $y = f(x)$ is true value and $\hat{y} = \hat{f}(\hat{x}) \approx y$ is approximate value, then

absolute error: $\Delta y := \hat{y} - y$

relative error:
$$\begin{aligned}\frac{\Delta y}{y} &= \frac{\Delta y}{\hat{y}\,(1 - \Delta y/\hat{y})} \\ &\approx \frac{\Delta y}{\hat{y}}\,(1 + \Delta y/\hat{y}) \quad \text{(by Taylor series expansion)} \\ &= \frac{\Delta y}{\hat{y}}\left(1 + O\left(\frac{\Delta y}{y}\right)\right)\end{aligned}$$

Measuring Error

- Suppose $\mathbf{x}$ and $\mathbf{y}$ are *vectors*, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$

- **Q**: How do we measure error?

- **A**: Vector norms!

Vector Norm

- Recall, a ***vector norm*** is a scalar function $f(\mathbf{x})$ that returns a "magnitude" of the input vector $\mathbf{x}$.

- In symbols, often written $\|\mathbf{x}\|$, potentially with a subscript such as $\|\mathbf{x}\|_2$ or $\|\mathbf{x}\|_W$ to indicate a particular chosen measure or weight (W).

- **Define *norm:***

  A function $\|\mathbf{x}\| : \mathbb{R}^n \longrightarrow \mathbb{R}$ is called a norm if and only if

  1. $\|\mathbf{x}\| > 0 \iff \mathbf{x} \neq 0.$
  2. $\|\gamma\mathbf{x}\| = |\gamma|\,\|\mathbf{x}\|$ for all scalars γ
  3. Obeys triangle inequality, $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

Example: p-norms

- For integers $p \geq 1$ we define the *p-norms*,

$$\|\mathbf{x}\|_p = \left\| \left[\, x_1 \; \cdots x_n \, \right] \right\| = \sqrt[p]{|x_1|^p + \cdots + |x_n|^p}$$

- $p = 1, \; 2 \, \infty$ are particularly important

- **Demo:** Vector Norms

Which Norm to Use?

- For *finite-dimensional* systems, with n ***fixed***, we have the ***equivalence of norms*** property.

- Given any two vector norms, $\|\cdot\|$ and $\|\cdot\|_*$, there exist positive constants, c and C, possibly dependent on n, such that for all $\mathbf{x} \in \mathbb{R}^n$,

$$c\|\mathbf{x}\|_* \;\leq\; \|\mathbf{x}\| \;\leq\; C\|\mathbf{x}\|_*$$

- The implication is that if $\|\Delta\mathbf{y}\|_* \longrightarrow 0$ in one norm, it does so for all norms.

- Consequently, we can bound the error in the norm of our choosing, i.e., whatever is most convenient given the information available.

Norms and Errors

- If we are computing a vector result, $\hat{\mathbf{y}}$, the error is a vector, $\Delta \mathbf{y} \;=\; \hat{\mathbf{y}} - \mathbf{y}$.

- To answer the question, *How large is the error?*, we need to apply a norm.

Attempt 1:

$$\text{Magnitude of error} \;\neq\; \|\text{true value}\| \;-\; \|\text{approximate value}\|$$

Wrong! How does this fail?

Attempt 2:

$$\text{Magnitude of error} \;=\; \|\text{true value} \;-\; \text{approximate value}\|$$

Data Error and Computational Error

- Typical problem: compute value of function $f(x) : \mathbb{R} \longrightarrow \mathbb{R}$ 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: $\hat{f}(\hat{x}) - f(x) =$

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

$$\text{computational error} \;+\; \text{propagated data error}$$

- ***Algorithm as no effect on propagated data error.***

Bounds on Total Error

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

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

$$\text{computational error} + \text{propagated data error}$$

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

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

$$|\text{computational error}| + |\text{propagated data error}|$$

- This is a standard trick in which we add and subtract the same quantity to isolate separate effects.

- We then bound each effect independently, using appropriate analysis.

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 on them terminating iterative sequence before convergence

- Computational error is sum of truncation and rounding error

- Truncation error dominates in practice. **(WHY?)**

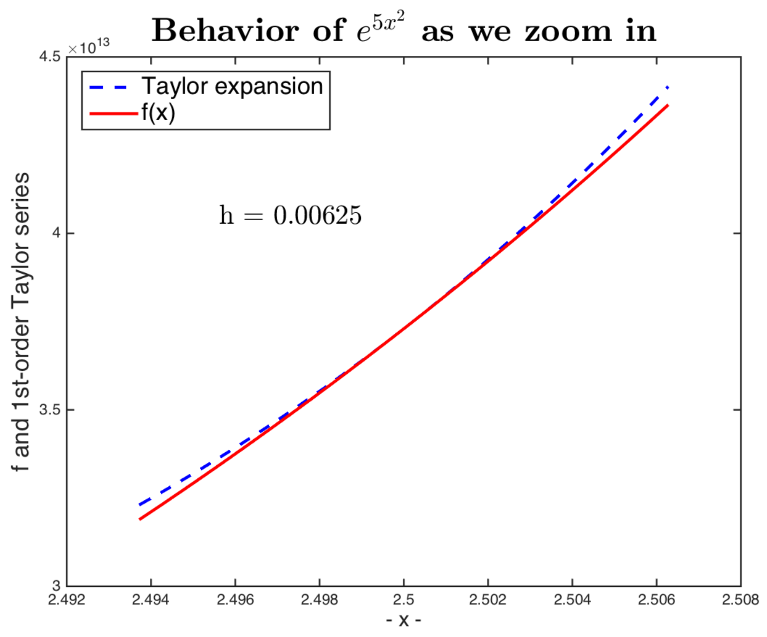
Truncation Error Example

- Recall Taylor series:
  - 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

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

This simply says that, as we zoom in ($h \longrightarrow 0$), $f(x)$ looks like a line.

taylor_demo.m



Truncation Error Example

❑ Recall Taylor series:

- 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

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

- Taylor series are fundamental to numerical methods and analysis.

- Newton's method, optimization algorithms, and numerical solution of differential equations all rely on understanding the behavior of functions in the neighborhood of a specific point or set of points.

- In essence, numerical methods convert calculus from the continuous back to the discrete.

- (A way of avoiding caculus. :))

Truncation Error Example

- Recall Taylor series:
  - 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

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

- Can use the Taylor series to generate approximations to $f'(x)$, $f''(x)$, etc., by evaluating f at x, $x \pm h$, $x \pm 2h$.
- We then solve for the desired derivative and consider $\lim h \longrightarrow 0$.

Truncation Error Example

- Recall Taylor series:
  - 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

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

  - Take $m = 2$:

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

Truncation Error Example

- Recall Taylor series:
  - 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 $m = 2$:

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

Truncation error

  - **Truncation error:** $\frac{h}{2}f''(\xi) \approx \frac{h}{2}f''(x)$ as $h \longrightarrow 0$.
    - To be precise, $\frac{h}{2}f''(\xi) = \frac{h}{2}f''(x) + O(h^2)$

Truncation Error Example

- **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: How Can We Use this Analysis

- Denote the finite-difference formula as

$$\frac{\delta_h f}{\delta x} \;:=\; \frac{f(x+h)-f(x)}{h} \;=\; f'(x) \;+\; \frac{h}{2}f''(x) \;+\; O(h^2)$$

  for h fixed, *"sufficiently small"*

- Also evaluate:

$$\frac{\delta_{2h} f}{\delta x} \;:=\; \frac{f(x+2h)-f(x)}{2h} \;=\; f'(x) \;+\; \frac{2h}{2}f''(x) \;+\; O(h^2)$$

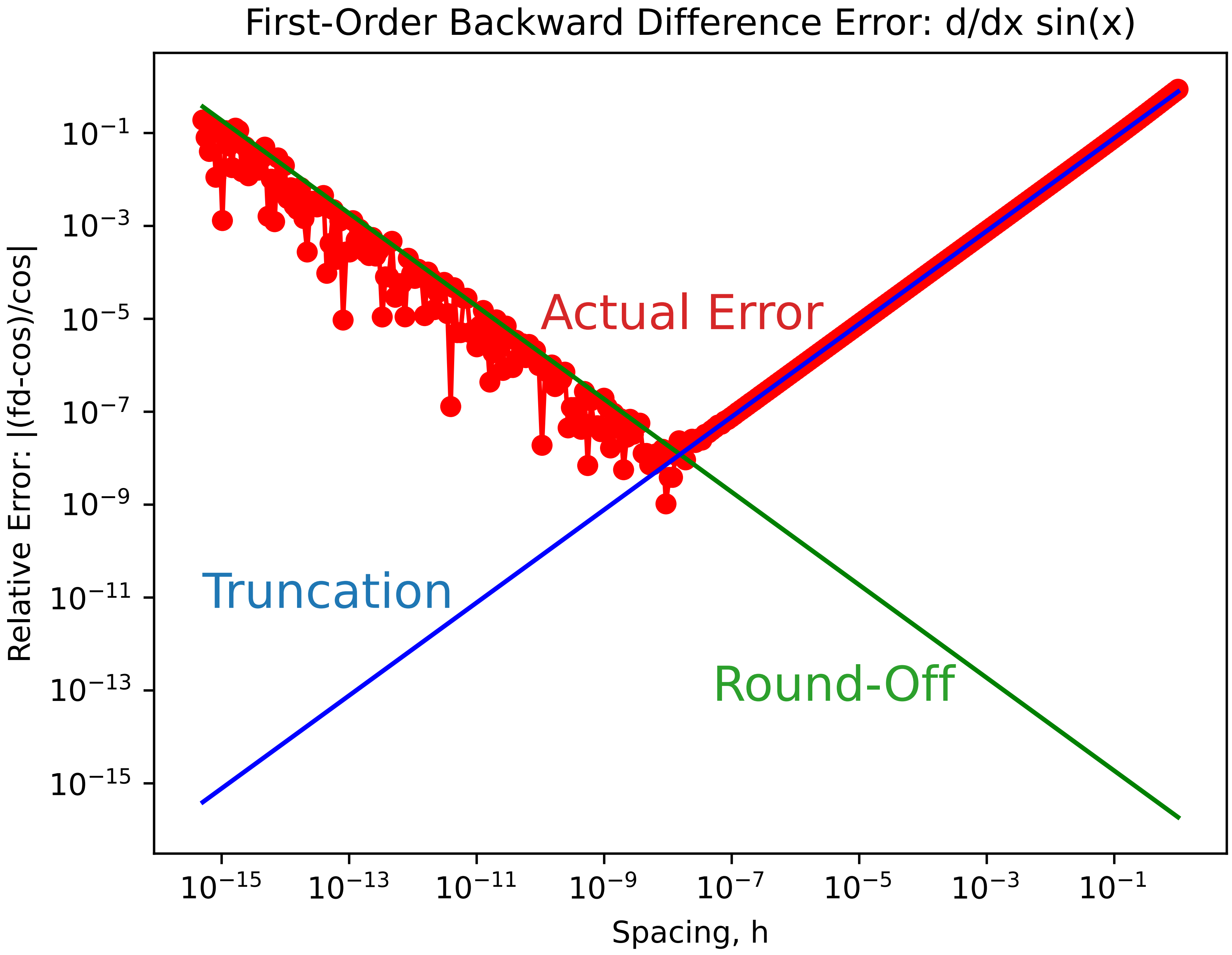
- *Leading order error term is 2× larger.*

- Subtract second result from 2× first to arrive at

$$2\frac{\delta_h f}{\delta x} - \frac{\delta_{2h} f}{\delta x} \;=\; f'(x) \;+\; O(h^2)$$

- Demo: fdiff1.m

Finite Differences and Truncation/Round-Off Error



Round-Off Error

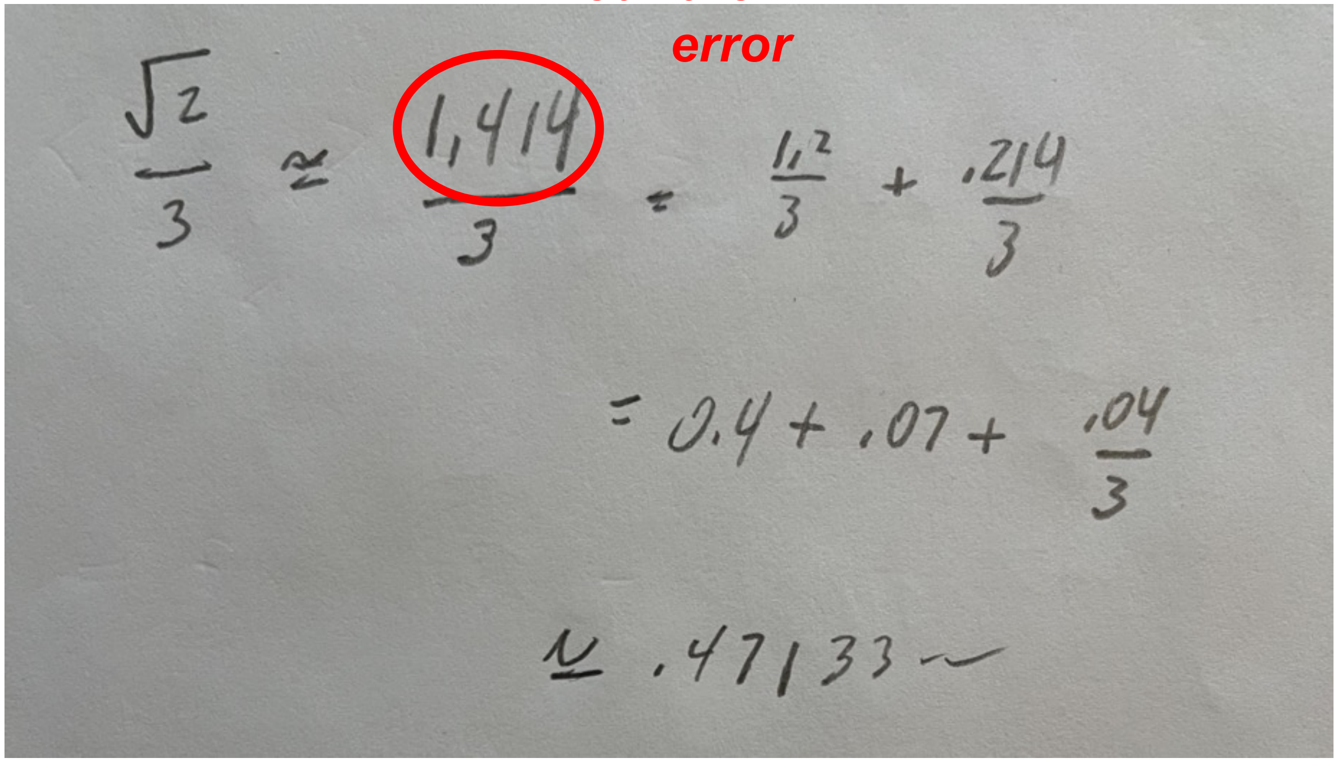
- In general, round-off error will prevent us from representing f(x) and f(x+h) with sufficient accuracy to reach such a result.
- Round-off is a principal concern in scientific computing. (Though once you're aware of it, you generally know how to avoid it as an issue.)
- Round-off results from having finite-precision arithmetic and finite-precision storage in the computer. (e.g., how would you store 1/3 in a computer?)
- Most scientific computing is done either with 32-bit or 64-bit arithmetic, with 64-bit being predominant. (IEEE 754-2008)
- Because of implications for machine learning, vendors are now building hardware support for fast 16-bit operations and some scientific computing applications are trying to leverage this hard-ware and also gain through reduced memory-bandwidth demands.

Round-Off Error

- We are of course all very familiar with round-off error.

- When we perform computation with pencil and paper, we inevitably truncate infinite-decimal results, and propagate these forward.

Round-off error



$$\frac{\sqrt{2}}{3} \approx \frac{\boxed{1{,}414}}{3} = \frac{1{,}2}{3} + \frac{{,}214}{3}$$

$$= 0.4 + {,}07 + \frac{{,}04}{3}$$

$$\approx {,}47133\sim$$

Round-Off Example: Matrix Iteration

- Round-off error is effectively a source of noise (i.e., perturbation) at every step of a calculation

- To illustrate, we consider the matrix iteration

$$\mathbf{x}_k \;=\; \mathbf{A}\,\mathbf{x}_{k-1} \;=\; \mathbf{A}^2\,\mathbf{x}_{k-2} \;=\; \cdots \;=\; \mathbf{A}^k\,\mathbf{x},$$

  for a 2×2 matrix $\mathbf{A}$ with an initial vector $\mathbf{x}$.

- Suppose (per chapter 4) that there are two distinct *eigenvectors*, $\mathbf{s}_1$ and $\mathbf{s}_2$, and scalars (*eigenvalues*), λ_1 and λ_2, satisfying

$$\mathbf{A}\mathbf{s}_1 \;=\; \lambda_1\mathbf{s}_1 \;\text{ and }\; \mathbf{A}\mathbf{s}_2 \;=\; \lambda_2\mathbf{s}_2.$$

Round-Off Example: Matrix Iteration, cont'd

- If $\mathbf{x} = a_1\mathbf{s}_1 + a_2\mathbf{s}_2$, then

$$\begin{aligned}
\mathbf{A}\,\mathbf{x} &= \mathbf{A}\left(a_1\mathbf{s}_1 + a_2\mathbf{s}_2\right) \\
&= a_1\mathbf{A}\mathbf{s}_1 + a_2\mathbf{A}\mathbf{s}_2 \\
&= a_1\lambda_1\mathbf{s}_1 + a_2\lambda_2\mathbf{s}_2 \\
\\
\mathbf{A}^2\,\mathbf{x} &= \mathbf{A}\left(\mathbf{A}\mathbf{x}\right) \\
&= a_1\lambda_1^2\mathbf{s}_1 + a_2\lambda_2^2\mathbf{s}_2 \\
\\
\mathbf{A}^k\,\mathbf{x} &= a_1\lambda_1^k\mathbf{s}_1 + a_2\lambda_2^k\mathbf{s}_2
\end{aligned}$$

Round-Off Example: Matrix Iteration, cont'd

$$\mathbf{A}^k \mathbf{x} \;=\; a_1 \lambda_1^k \mathbf{s}_1 \;+\; a_2 \lambda_1^k \mathbf{s}_2$$

- Let's consider and example where $a_1 = 1$ and $a_2 = 0$, so $\mathbf{x} = \mathbf{s}_1$.
- Then, in exact arithmetic, $\mathbf{A}^k \mathbf{x} \;=\; \lambda_1^k \mathbf{s}_1$.
- If the arithmetic is *inexact*, however, then we can expect

$$\mathbf{A}\,\mathbf{x} \;=\; \lambda_1^k \,\mathbf{s}_1 \;+\; \epsilon\, \lambda_2^k\, \mathbf{s}_2,$$

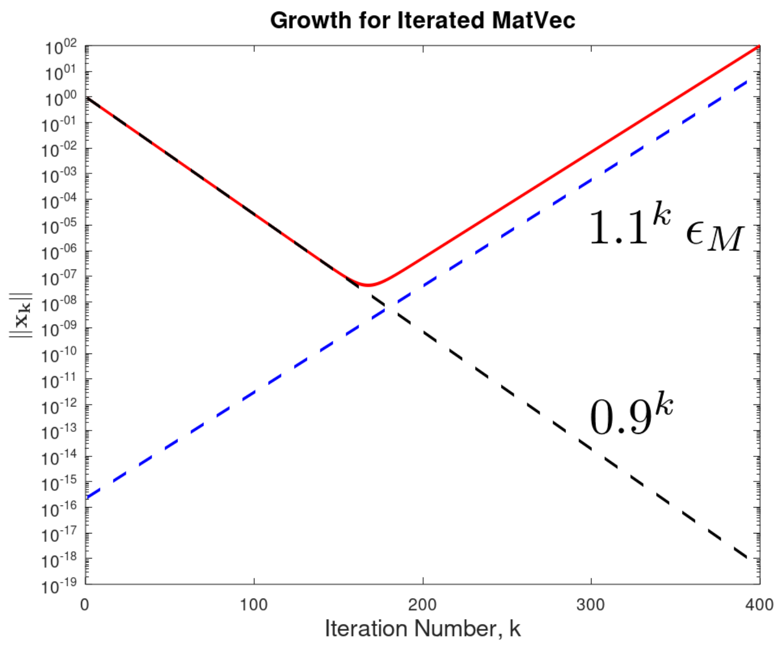
  where ϵ is related to round-off error in the associated arithmetic (or in the representation of $\mathbf{s}_1$, perhaps).
- If $|\lambda_2| \;\leq\; |\lambda_1|$, then we will get something close to the expected result.
- However, if $|\lambda_2| \;>\; |\lambda_1|$, this small $O(\epsilon)$ perturbation will eventually be amplified.

Round-Off Example: Matrix Iteration, cont'd

- Consider a case with

$$\lambda_1 = 0.9 \quad \lambda_2 = 1.1$$
$$a_1 = 1 \quad a_2 = 0$$

- We expect
  - $\|\mathbf{x}_k\| \sim 0.9^k$ in the early iterations and
  - $\|\mathbf{x}_k\| \sim 1.1^k$ for larger iteration counts, k.



Forward and Backward Error

- Suppose we want to compute $y = f(x)$, where $f : \mathbb{R} \longrightarrow \mathbb{R}$, but obtain an approximate value, $\hat{y}$.

- *Forward error*: $\Delta y \;=\; \hat{y} \;-\; y$

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

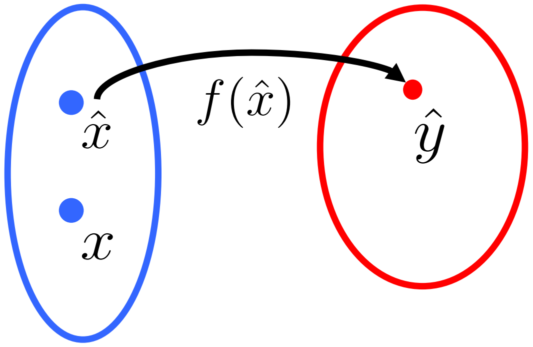
$$\begin{tikzcd}
x \arrow[r, "f"] \arrow[dr, "\hat{f}"] \arrow[d, leftrightarrow, "\text{backward error}"'] & y = f(x) \arrow[d, leftrightarrow, "\text{forward error}"] \\
\hat{x} \arrow[r, "f"] & \hat{y} = \hat{f}(x) = f(\hat{x})
\end{tikzcd}$$

Backward Error Analysis

- Idea: approximate solution is exact solution to modified problem
- How much must the original problem change to give the obtained result?
- How much data error in the input would explain *all* error in computed result?
- Approximate solution is good if it an exact solution to a *nearby* problem
- Backward error is often easier to estimate than forward error.

Backward Error Analysis

- User wants: $y = f(x)$.
- Algorithm produces: $\hat{y} = \hat{f}(x)$.
- Backward error analysis asks
  - *Is there an $\hat{x}$ near x such that $f(\hat{x}) = \hat{y}$?*



- Uses for backward error
  - Original data might be known only to tolerance $\epsilon > |x - \hat{x}|$. In other words, *We can't distinguish small errors induced by the algorithm from acceptably small errors in the input.*
  - We will see later that we can often bound the forward error in terms of the *condition number* times the backward error.

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 $1.4^2 = 1.96$, absolute backward error is

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

or relative backward error of 2 percent

Example: Backward Error Analysis

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

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

- Forward error is given by

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

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

Example: Backward Error Analysis, continued

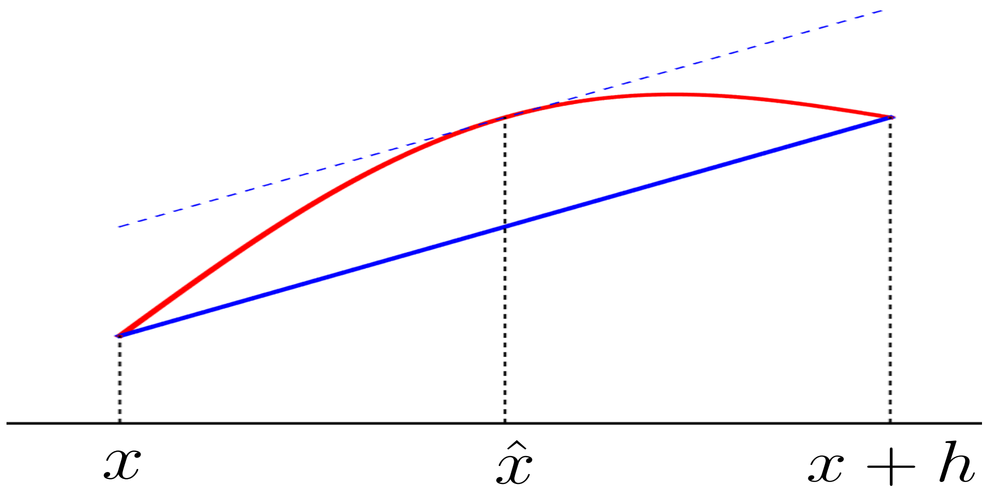
- For $x = 1$,

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

- Forward error: $\Delta y = \hat{y} - y \approx 0.5 - 0.5403 = -.0403$
- Backward error: $\Delta x = \hat{x} - x \approx 1.0472 - 1 = 0.0472$

Relative forward error ~ 8%
Relative backward error ~ 5%

Backward Error Analysis for Finite Difference Example



- By *Mean Value Theorem*, there exists an $\hat{x} \in [x, x+h]$ such that

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

  (Assuming f is differentiable on $[x, x+h]$.)

- Backward error for the finite difference approximation is bounded by $|h|$, assuming no round-off error.

Sensitivity and Conditioning

- Problem is *insensitive*, or *well-conditioned*, if relative change in input causes similar relative change in solution (output)

- Problem is *sensitive*, or *ill-conditioned*, if relative change in solution can be much larger than the relative change in the input data

Condition Number

- ***Condition number*** is a measure of the sensitivity of the *problem*

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

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

$$= \frac{|\textit{ relative forward error}|}{|\textit{ relative backward error}|}$$

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:

$$\text{cond} \;=\; \frac{|\text{ relative change in output }|}{|\text{ relative change in input }|} \;=\; \frac{|(\hat{y}-y)/y|}{|(\hat{x}-x)/x|} \;=\; \frac{|\Delta y/y|}{|\Delta x/x|}$$

- We, can, however also define an ***absolute condition number***, which is useful when x or y are zero.

- This is the perturbation in y that is generated by a perturbation in x,

$$\text{cond}_{\text{abs}} \;=\; \frac{|(\hat{y}-y)|}{|(\hat{x}-x)|} \;=\; \frac{|\Delta y|}{|\Delta x|}$$

Condition Number, continued

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

$$|\text{ relative forward error }| = \text{cond} \times |\text{ relative backward error }|$$

- In general, condition number varies with input and in practice we don't know it exactly.

- We might be able find a rough estimate or upper bound over the domain of inputs, so the relationship becomes

$$|\text{ relative forward error }| \lesssim \text{cond} \times |\text{ relative backward error }|$$

- Thus, the condition number allows us to bound the *forward error* in terms of the *backward error*, which might be easier to estimate.

Example: Evaluating a Function

- Evaluating $f(x)$ with an approximate input $\hat{x} = x + \Delta x$ instead of the true input x yields

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

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

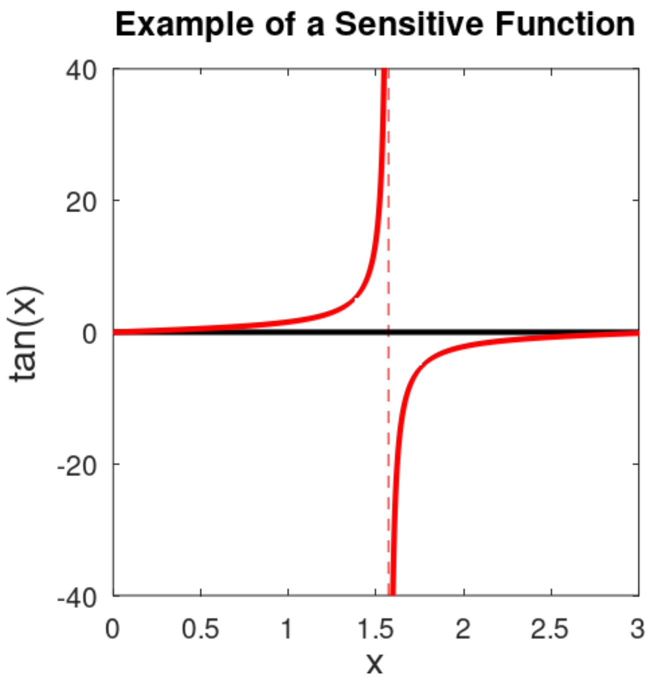
  Condition number: $\mathrm{cond} \;\approx\; \left| \frac{f'(x)\Delta x / f(x)}{\Delta x / x} \right| \;=\; \boxed{\left| \frac{x f'(x)}{f(x)} \right|}$

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

Example: Sensitivity

- $\tan(x)$ is sensitive for arguments near $\pi/2$.
  - $\tan(1.57078) \approx 6.12490 \times 10^4$
  - $\tan(1.57079) \approx 1.58058 \times 10^5$

- Relative change in output is a quarter million times larger than relative change in input
  - For $x = 1.57079$, cond $\approx 2.48275 \times 10^5$



- Recall,

$$y = \tan(x) = \frac{\sin(x)}{\cos(x)} = \frac{f}{g}$$

$$\frac{dy}{dx} = \frac{f'g - g'f}{g^2} = \frac{\cos^2 + \sin^2}{\cos^2} = \sec^2(x)$$

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

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 the are of opposite sign but close to the same magnitude.

  This ill-conditioning is often referred to as *cancellation.*

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.

$$\frac{\delta_h x}{\delta x} := \frac{f(x+h) \; - \; f(x)}{h}$$

- If $f(x)$ is *also* ill-conditioned, then perturbations (i.e., round-off) in $x+h$ and x can also contribute to (and possibly dominate) round-off error.

Stability

- Algorithm is *stable* if returned result is relatively insensitive to perturbations *during* the computation
- Stability of *algorithms* is analogous to conditioning of problems
- From point of view of backward error analysis, algorithm is stable if returned result is exact solution to a nearby problem
- For a stable algorithm, effect of computational error is no worse than the effect of a 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 a stable algorithm to an ill-conditioned problem or unstable algorithm to well-conditioned problem

- Applying stable algorithm to well-conditioned problem will yield an 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., monomials or Lagrange polynomials on uniformly spaced nodes)

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 and our actual input is thus

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

- The next topic discusses the set of representable numbers.
- We'll sometimes refer to this set of "floating point numbers" as *F.*
- We'll primarily be concerned with two things –
  - the relative precision,
  - the maximum absolute value representable.

Floating-Point Numbers

- Floating-point number system is characterized by four integers

β	base or radix
p	precision
(L, U)	exponent range

- Number is represented as

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

  - Here, we have p digits, d_0, $d_1, \ldots, d_{p-1}$.
  - Each digit is in the range $0 \leq d_i \leq \beta - 1$.
  - Representations are typically *normalized*, meaning that $d_0 \neq 0$.
  - The *exponent* is in the interval $L \leq E \leq U$.

- Modern computers use base $\beta = 2$ (i.e., binary representation).

Floating-Point Number Examples

- Base 10 $\quad x = \pm 1.2345678901234567 \times 10^{\pm 9}$
- Base 2 $\quad x = \pm 1.0101010101010101 \times 2^{\pm 1001}$ $\quad (\times 2^{\pm 9})$
- Base 16 $\quad x = \pm 1.23456789abcdef01 \times 16^{\pm 9}$

Floating-Point Numbers, continued

- Portions of floating-point numbers are designated as
  - *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 floating-point systems

System	β	p	L	U
IEEE DP	2	53	-1022	1023
IEEE SP	2	24	-126	127
IEEE HP	2	24	-126	127
Cray	2	48	-16383	16384

- Most modern computers use binary ($\beta = 2$) arithmetic
- IEEE floating-point systems are now almost universal
- Jim Cody lecture on “pre-IEEE” days

Normalization

- Floating-point systems are *normalized* if leading digit, d_0 is always nonzero for any entry $\neq 0$.

- In normalized systems, mantissa m of nonzero floating-point number always satisfies $1 \leq m < \beta$

- Reasons for normalization:
  - representation of each number is unique
  - no digits wasted on leading zeros
  - leading bit need not be stored in binary systems

`https://bartaz.github.io/ieee754-visualization/`

Normalized Mantissa Examples

- Decimal:

  1.2345

  9.814×10^{-2}

- Binary:

  1.010×2^{-3}

  1.111×2^{4}

Not normalized:

.00101

Binary Representation of π

- In 64-bit floating point,

$$\pi_{64} \approx 1.1001001000011111101101010100010001000010110100011 \times 2^1$$

- In reality,

$$\pi = 1.100100100001111110110101010001000100001011010001100001000110 10 \cdots \times 2^1$$

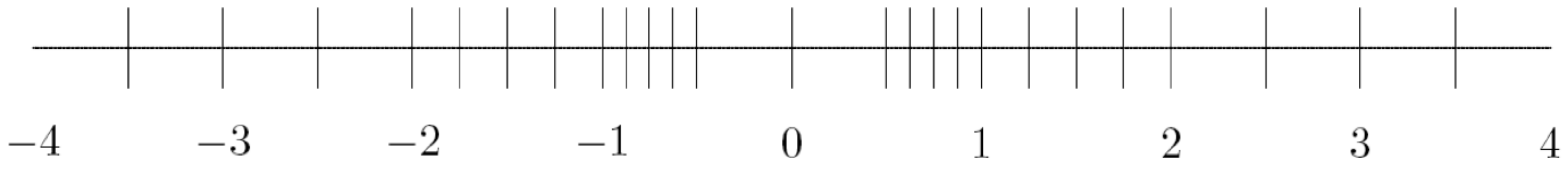
- They will (potentially) differ in the 53rd bit...

$$\pi - \pi_{64} = 0.000000000000000000000000000000000000000000000000000000\underbrace{100011010\cdots}_{\boxed{\textit{Rounding Error}}} \times 2^1$$

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

Example Floating-Point System



- Tick marks indicate all 25 numbers in floating point system having $\beta = 2$, $p = 3$, $L = -1$, and $U = 1$
  - UFL $= (1.00)_2 \times 2^{-1} = (0.5)_{10}$
  - OFL $= (1.11)_2 \times 2^{1} = (3.5)_{10}$
- At sufficiently high magnification, normalized floating-point systems look grainy and unevenly spaced

Example: numbers.m

Rounding Rules

- If real number x is not exactly representable, then it is approximated by "nearby" number in $\hat{F}$, $fl(x)$

- 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 call *round toward zero*
  - *round to nearest*: $fl(x)$ is nearest element of $\hat{F}$ 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 standard

Machine Precision

- Probably the most important number in a floating point system is machine precision (or machine epsilon or unit roundoff, denoted by ϵ_{mach} or ϵ_M.
  - With rounding by chopping, $\epsilon_M = \beta^{1-p}$
  - With rounding to nearest, $\epsilon_M = \frac{1}{2}\beta^{1-p}$

- Alternative definition is smallest number ϵ such that $fl(1+\epsilon) > 1$, where "$fl()$ is the result of assigning the argument to the element of $\hat{F}$ according to the rounding rule

- Maximum relative error in representing real number x in $\hat{F}$ is given by

$$\left| \frac{fl(x) - x}{x} \right| \leq \epsilon_M$$

Machine Precision, continued

- The relationship

$$\left|\frac{fl(x) - x}{x}\right| \le \epsilon_M$$

  can be re-expressed as

$$fl(x) = x\,(1 + \epsilon_x)$$

  with $|\epsilon_x| \le \epsilon_M$

- The advantage of this expression is that the approximation is expressed as an equality, which is easier to work with

Machine Precision, continued

- For IEEE floating point systems
  - $\epsilon_M = 2^{-24} \approx 10^{-7}$ in single (32-bit) precision
  - $\epsilon_M = 2^{-53} \approx 10^{-16}$ in single (64-bit) precision

- So IEEE single (FP32) and double (FP64) 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
                                                            .0000003141592653589793
                                                            .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.

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

$$\begin{aligned}
x - \hat{x} &= 238462643383279502884197169399375105820974944.9230781\ldots = 2.3846\ldots \times 10^{44} \\
&\approx .7590501687441757 \times 10^{-16} \times x \\
&< 1.110223024625157e-16 \times x \\
&\approx \epsilon_{\mathrm{mach}} \times x.
\end{aligned}$$

- 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

- Unit roundoff, ϵ_M, should not be confused with the underflow level, UFL

- Unit roundoff, ϵ_M, is determined by the number of digits (or bits) in the *mantissa*, whereas underflow level UFL is determined by the number digits or bits in the *exponent* field

- In all *practical* floating-point systems,

$$0 < UFL \ll \epsilon_M \ll OFL$$

Machine Precision, continued

$$0 < UFL \ll \epsilon_M \ll OFL$$

- Despite the remarkably small magnitude of UFL, there is (potentially) a mechanism to represent numbers between UFL and 0 by using *denormalized numbers* (leading bit $\neq 1$).

- Most (fast) floating-point hardware does not support them and operations with them can be thousands of times slower than working with normalized elements of $\hat{F}$

- Practically, any denormalized number ≈ 0, so flushing these values to 0 is not harmful

- The principal reason to support "d-norms" is that it guarantees $fl(x - y) = 0$ *iff* $fl(x) = fl(y)$.

- This property, and the extended precision afforded by d-norms, can be important in low-precision (FP32, FP16) arithmetic.

Summary of Ranges for IEEE Double Precision

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

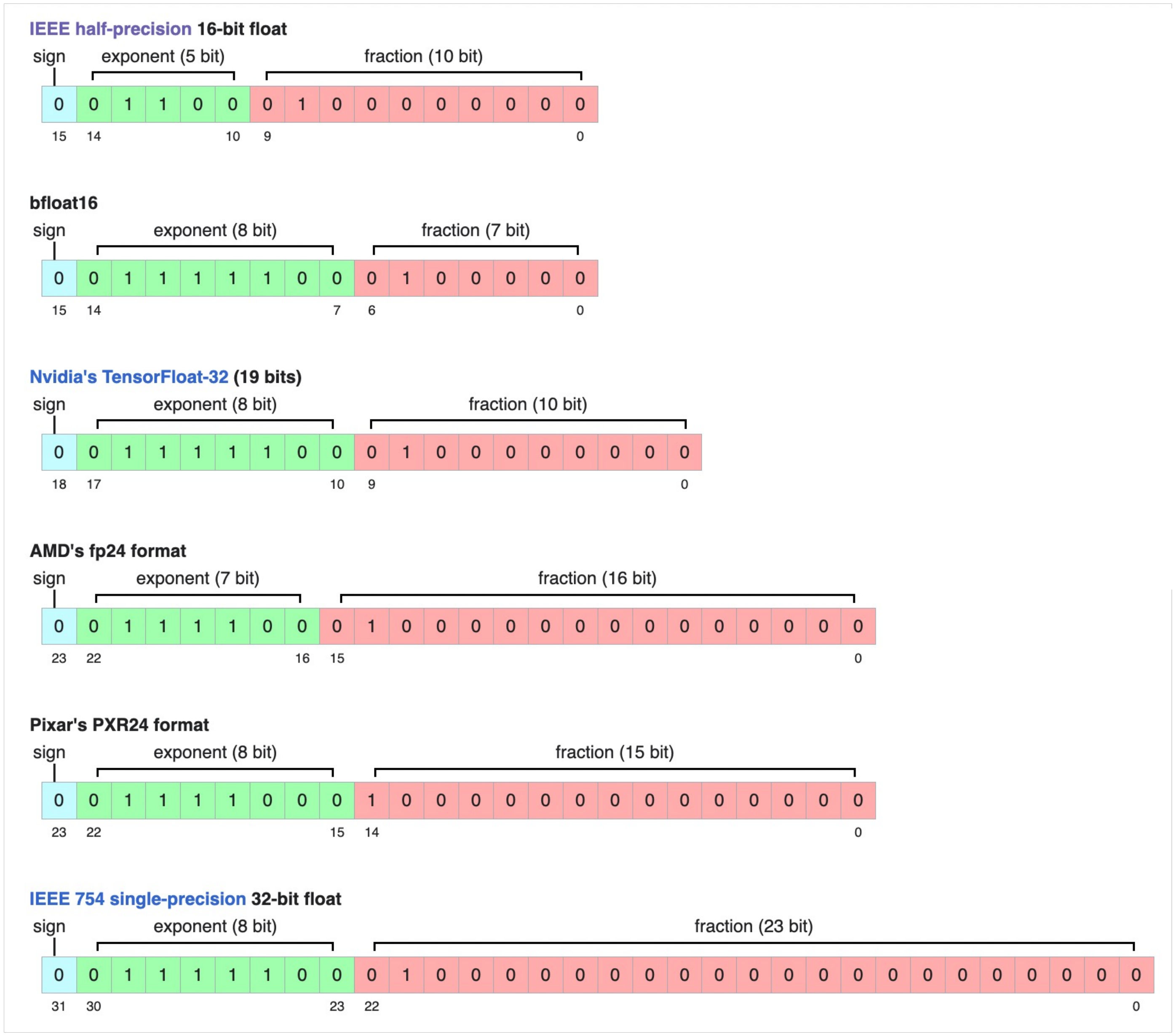
Q: How many atoms in the Universe?

Q: How many positive FP64 numbers < 1?

Exceptional Values

- IEEE floating-point standard provides special values to indicate exceptional situations
  - `Inf`, which stands for "infinity," results from dividing a finite number by 0, (e.g., 1/0)
  - `NaN`, which stands for "not a number," results from undefined or indeterminate operations such as 0/0, 0*`Inf`, or `Inf/Inf`
- `Inf` and `NaN` are implemented in IEEE arithmetic through special reserved values of the exponent field
- Note that 0 is also a special number as it is not normalized

Several low-precision formats *(Wikipedia bfloat16)*



Sources of Error in Floating-Point Arithmetic

- *Addition or subtraction*: Shifting of mantissa to make exponents match may cause loss of some (possibly all) digits of smaller number
- *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 representation of $1/3$ in base 10 or of $1/10$ in binary
- Result of floating-point arithmetic operation my differ from result of corresponding real arithmetic operation on same operands

Example: Floating-Point Arithmetic

- Consider the following statements, executed in FP64 with matlab

```
format longe  % Set output format to sci. notation, all digits
x=1000000/3
y=333333
d=x-y
```

  which produces the following results

$$\begin{aligned} x &= 3.333333333333333e+05 \\ y &= 3.333330000000000e+05 \\ d &= 3.333333333139308e-01, \end{aligned}$$

- We color the output of the difference in blue and red, where blue indicates the *correct* digits of x that remain after subtraction, and red indicates so-called garbage digits.
- Note that flushing the red digits to 0 would not be more accurate than the garbage bits that are shown above, as the values of interest would be all "3"s in this case
- This is an example of *cancellation* than the garbage bits that are shown above, as the values of interest would be all "3"s in this case

Floating-Point Arithmetic, continued

- Real result may also fail to be representable because exponent is beyond available range

- Overflow is more serious than underflow because there is *no* good approximation to arbitrarily large magnitudes, whereas 0 is often a good approximation for arbitrarily small magnitudes

- Overflow is generally fatal, whereas underflow may be silently set to 0

Resume Here for Lec. 3

Standard Model for Floating Point Arithmetic

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

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

  which is more conveniently analyzed by backward error analysis.
- For example, with op $= +$,

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

Floating-Point Arithmetic, continued

- Ideally, x flop $y \;=\; fl(x \text{ op } y)$, i.e., floating-point arithmetic operations produce correctly rounded results

- Computers satisfying IEEE floating-point standard achieve this ideal as long as x op y is within range of $\hat{F}$

- Some familiar laws of real arithmetic, however, are not necessarily valid in floating-point system

- Floating-point addition and multiplication are commutative, but *not* associative

- Example:
$$\begin{aligned} \epsilon &:= 0.9\epsilon_M \\ (1+\epsilon)+\epsilon &= 1 \\ 1+(\epsilon+\epsilon) &> 1 \end{aligned}$$

Example: Summing a Series

- Infinite series

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

  is divergent, yet evaluates to a *finite* number in floating-point arithmetic

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

$$\frac{1}{n} < \epsilon_M \sum_{k=1}^{n-1} \frac{1}{k}$$

- ***Jupyter demo***

Example: Summing a Series

- Infinite series

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

  is divergent, yet evaluates to a *finite* number in floating-point arithmetic

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

$$\frac{1}{n} < \epsilon_M \sum_{k=1}^{n-1} \frac{1}{k}$$

- Q: How long would it take to realize failure in FP32? FP64?

Time to Sum

- If

$$\begin{aligned} S_n &:= \sum_{k=1}^{n} \frac{1}{k} \\ S_{n+1} &= S_n + \frac{1}{n+1} \\ &= S_n \left(1 + \frac{1}{S_n}\frac{1}{n+1}\right) \\ &= S_n \quad \text{if} \left(\frac{1}{S_n}\frac{1}{n+1}\right) < \epsilon_M \end{aligned}$$

- (Guess) Estimate $S_n \approx 100$.
- With $\epsilon_M \approx 10^{-16}$, summation stops changing when

$$100 \cdot (n+1) \;\approx\; 10^{16} \quad \Longrightarrow \quad n \;\approx\; 10^{14},$$

  so need about 10^{14} operations.

- Suppose we can sustain 1 GFLOPS = 10^9 op/sec.
- Time $= \dfrac{\text{number of ops}}{\text{rate}} \;=\; \dfrac{10^{14}\text{ops}}{10^{9}\text{ops/sec}} \;=\; 10^5$ seconds $\;\approx\;$ 30 hours

Cancellation

- Subtraction between two p-digit numbers having same sign and similar magnitudes yields a result with fewer than p digits, so it is in principle exactly representable

- The problem, as we saw in the earlier example, is that we can be left with just a small number of significant digits in the result

- It is often helpful to *make a small correction*, when possible, to avoid cancellation between a pair of numbers that are converging to the same value

Cancellation Example

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

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

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:

$$\begin{aligned} \epsilon &:= 0.9\epsilon_M \\ (1+\epsilon)-(1-\epsilon) &= 0 \text{ in floating point} \end{aligned}$$

  True result, however, is 2ϵ

- The issue is that subraction of two quantities of equal sign and similar magnitude is *ill-conditioned*

Cancellation, continued

- The *most* significant (leading) digits are lost to cancellation, whereas digits lost in rounding are *least* significant

- Because of this effect, it's generally a bad idea to compute small quantities as the difference of relatively large quantities

- For example, summing the series
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$
for $x \ll -1$ may fail due to cancellation

  fp_test.m

fp_test.m

```
hdr; format shorte

arg = 30;
exact = exp(-arg)

s1 = 1;  n1=1;
s2 = 1;  n2=1;  denom=1;

display('      j          Exact        Sum Bad       Sum Good      R. Err Bad    R. Err Good')

for j=1:80;

    denom = denom*j;
    n1 = n1*(-arg);
    n2 = n2*(arg);

    s1 = s1 + n1/denom;   %  BAD WAY
    s2 = s2 + n2/denom;   %  GOOD WAY
    e2 = 1./s2;

    err1 = abs(exact-s1)/exact;    %  BAD
    err2 = abs(exact-e2)/exact;    %  GOOD

    if mod(j,10)==0;
       display([j exact s1 e2 err1 err2]), pause(.15)
    end;
end;
```

Example: Quadratic Formula

- Two solutions of $ax^2 + bx + c = 0$ are

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

- Suppose $b < 0$ and $b^2 \gg 4ac$.
  - Evaluation of $-b + \sqrt{b^2 - 4ac}$ is OK
  - Possible cancellation for $-b - \sqrt{b^2 - 4ac}$ in second solution

- Rewrite second solution as

$$\begin{aligned}
x &= \frac{-b-\sqrt{b^2-4ac}}{2a} \cdot \frac{(-b+\sqrt{b^2-4ac})}{(-b+\sqrt{b^2-4ac})} \\
&= \frac{b^2-(b^2-4ac)}{2a\cdot(-b+\sqrt{b^2-4ac})} \\
&= \frac{2c}{-b+\sqrt{b^2-4ac}}
\end{aligned}$$

which avoids the $-b-\sqrt{b^2\cdots}$ cancellation

Example: Standard Deviation

- Mean and standard deviation of a sequence:

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

- Mathematically equivalent formula

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

allows "on-the-fly" evaluation of σ without two passes through the data

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

Example: Finite Differences and Truncation/Round-Off Error

- Taylor Series:

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

- So, expect $e_{abs} := \left|\frac{\delta f}{\delta x} - \frac{df}{dx}\right| = O(h) \longrightarrow 0$ as $h \longrightarrow 0$.

- Assuming $f(x)$ is *well-condtioned*, computed value of $\frac{\delta f}{\delta x}$ using ***standard model*** is

$$\begin{aligned}\left\langle\frac{\delta f}{\delta x}\right\rangle &= \frac{\langle f(x+h)\rangle - \langle f(x)\rangle}{h} \\ &= \frac{f(x+h)(1+\epsilon_+) - f(x)(1+\epsilon_0)}{h} \\ &= \frac{f(x+h) - f(x)}{h} + \frac{f(x+h)\epsilon_+ - f(x)\epsilon_0}{h},\end{aligned}$$

  - ϵ_+, ϵ_0 random variables of arbitrary sign with magnitude $\leq \epsilon_M$.
  - This computed expression ignores round-off in x, h, $x+h$ and division.
  - Those errors are benign if f is well-conditioned (i.e., $\epsilon_M \left|xf'(x)/f\right| \ll 1$)

- Use $f(x+h) = f(x) + hf'(\xi)$ to write the first round-off term as:

$$\frac{f(x+h)\epsilon_+}{h} \;=\; \frac{(f(x)+hf'(\xi))\epsilon_+}{h} \;=\; \frac{f(x)\epsilon_+}{h} \;+\; f'(\xi)\epsilon_+ \;\approx\; \frac{f(x)\epsilon_+}{h}.$$

- Round-off error is thus

$$\frac{f(x+h)\epsilon_+ \;-\; f(x)\epsilon_0}{h} \;=\; f(x)\frac{\epsilon_+ \;-\; \epsilon_0}{h} \;+\; \epsilon_+ f'(\xi) \;\approx\; f(x)\frac{\epsilon_+ \;-\; \epsilon_0}{h} \;=\; f(x)\frac{\epsilon_M}{h}R,$$

where R is a random variable with $|R| < 2$.

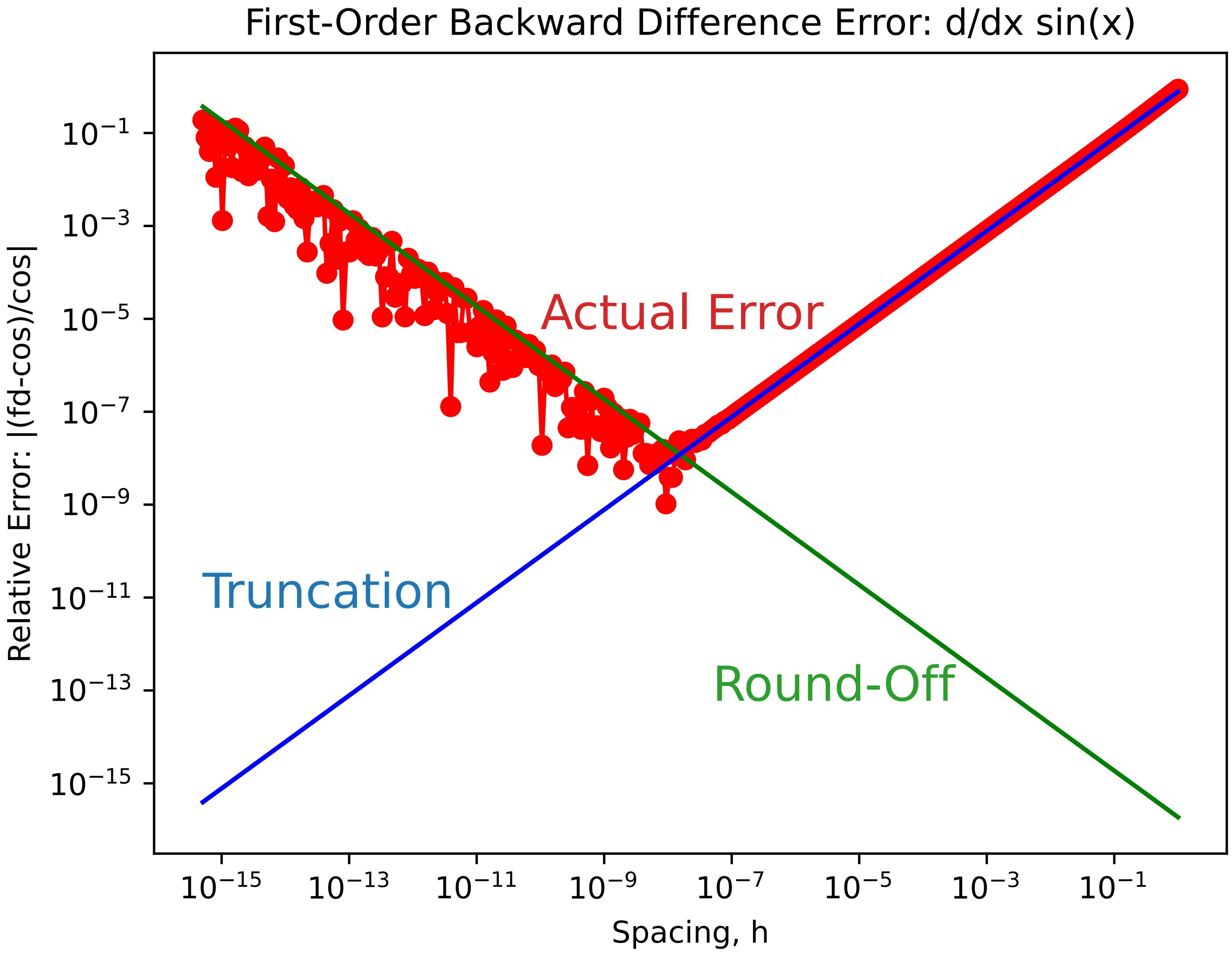
- So, computed approximation is

$$\left\langle \frac{\delta f}{\delta x} \right\rangle \;=\; \frac{\delta f}{\delta x} \;+\; \frac{\epsilon_+ - \epsilon_0}{h} f(x) \;+\; O(\epsilon_+)$$

$$\approx\; f'(x) \;+\; \underbrace{\frac{h}{2} f''(x)}_{TE} \;+\; \underbrace{R\frac{\epsilon_M}{h} f(x)}_{RE}.$$

- Notice, *crucially*, that the units of each term on the right match.
- This is a good way to check your work.

Finite Differences and Truncation/Round-Off Error



Review/Summary of a Couple of Topics

- Floating-Point representation summary
- Equivalence of vector norms

Summary: Binary Representations

Representations (normalized, unless otherwise indicated):

bfloat16	
8 bit exponent	
7 bit fraction	$\epsilon_M = 2^{-7} \approx .008$
OFL	$\approx 10^{38}$
UFL	$\approx 10^{-38}$
minval denorm	$\approx 10^{-40}$

IEEE 754 single-precision 32-bit float	
8 bit exponent	
23 bit fraction	$\epsilon_M = 2^{-23} \approx 10^{-7}$
OFL	$\approx 10^{38}$
UFL	$\approx 10^{-38}$
minval denorm	$\approx 10^{-45}$

IEEE half-precision 16-bit float	
5 bit exponent	
10 bit fraction	$\epsilon_M = 2^{-10} \approx 10^{-3}$
OFL	≈ 65504
UFL	$\approx 10^{-4}$
minval denorm	$\approx 10^{-7}$

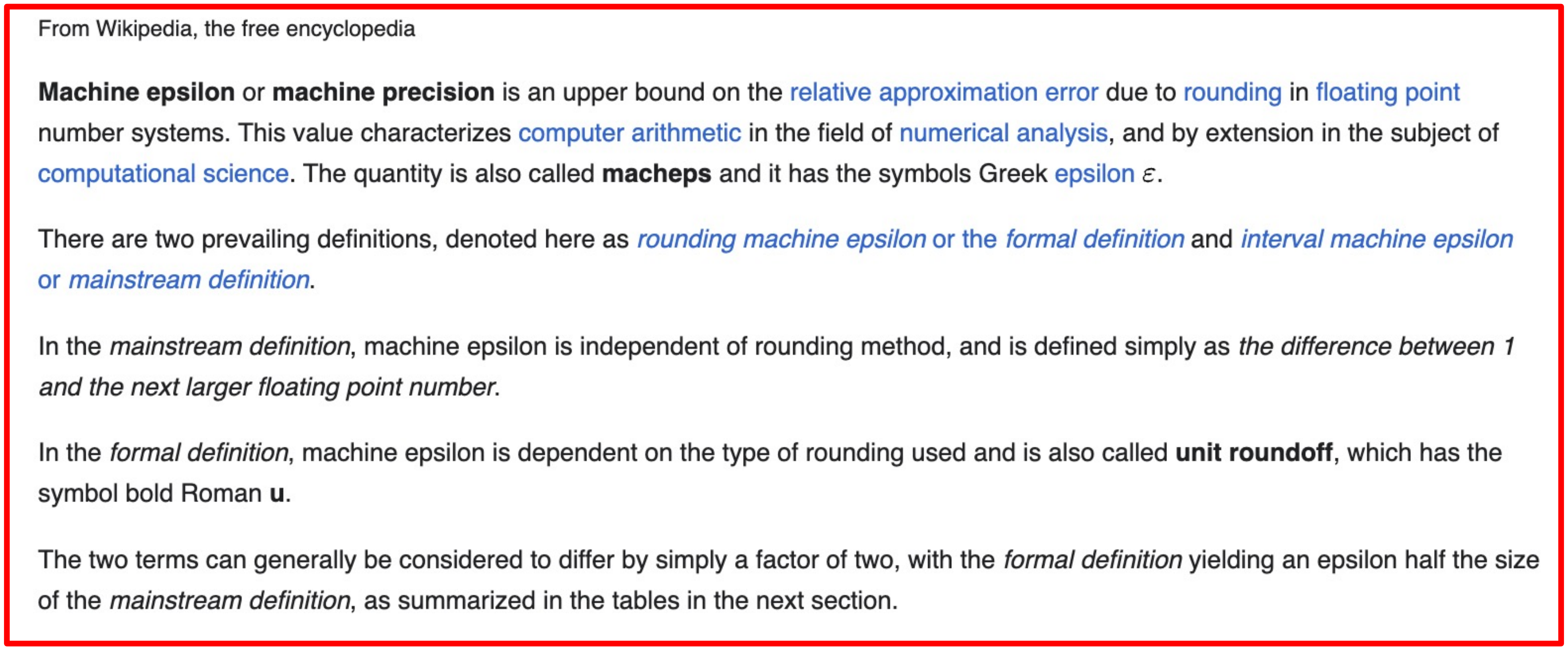
IEEE 754 double-precision 64-bit float	
11 bit exponent	
52 bit fraction	$\epsilon_M = 2^{-52} \approx 10^{-16}$
OFL	$\approx 10^{308}$
UFL	$\approx 10^{-308}$
minval denorm	$\approx 10^{-324}$

- *dnorms are rather superfluous in 64-bit and can slow arithmetic operations by 1000x if implemented in software*

- *In lower precision, however, they can significantly extend the dynamic and potentially valuable if implement in hardware*

Much Ado about (almost) Nothing

- Remarkably, there is conflicting information about what value, exactly, is recognized as machine precision, ϵ_M.
- With round-to-nearest, ϵ_M is technically $2^{-53} \approx 10^{-16}$ in IEEE FP64.
- Matlab and numpy, however, they indicate $\epsilon_M = 2^{-52} \approx 2 \times 10^{-16}$
- The following from Wikipedia clarifies the ambiguity



From Wikipedia, the free encyclopedia

Machine epsilon or **machine precision** is an upper bound on the relative approximation error due to rounding in floating point number systems. This value characterizes computer arithmetic in the field of numerical analysis, and by extension in the subject of computational science. The quantity is also called **macheps** and it has the symbols Greek epsilon ε.

There are two prevailing definitions, denoted here as *rounding machine epsilon* or the *formal definition* and *interval machine epsilon* or *mainstream definition*.

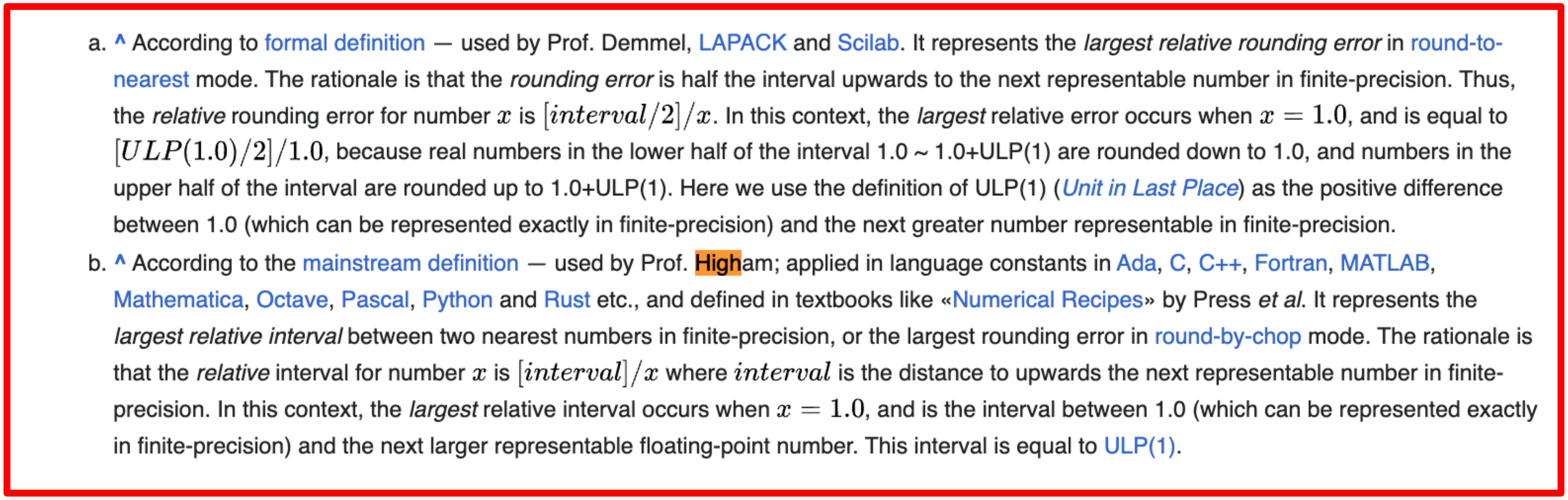
In the *mainstream definition*, machine epsilon is independent of rounding method, and is defined simply as *the difference between 1 and the next larger floating point number*.

In the *formal definition*, machine epsilon is dependent on the type of rounding used and is also called **unit roundoff**, which has the symbol bold Roman **u**.

The two terms can generally be considered to differ by simply a factor of two, with the *formal definition* yielding an epsilon half the size of the *mainstream definition*, as summarized in the tables in the next section.

Much Ado about (almost) Nothing

- Remarkably, there is conflicting information about what value, exactly, is recognized as machine precision, ϵ_M.
- With round-to-nearest, ϵ_M is technically $2^{-53} \approx 10^{-16}$ in IEEE FP64.
- Matlab and numpy, however, they indicate $\epsilon_M = 2^{-52} \approx 2 \times 10^{-16}$
- The following from Wikipedia clarifies the ambiguity

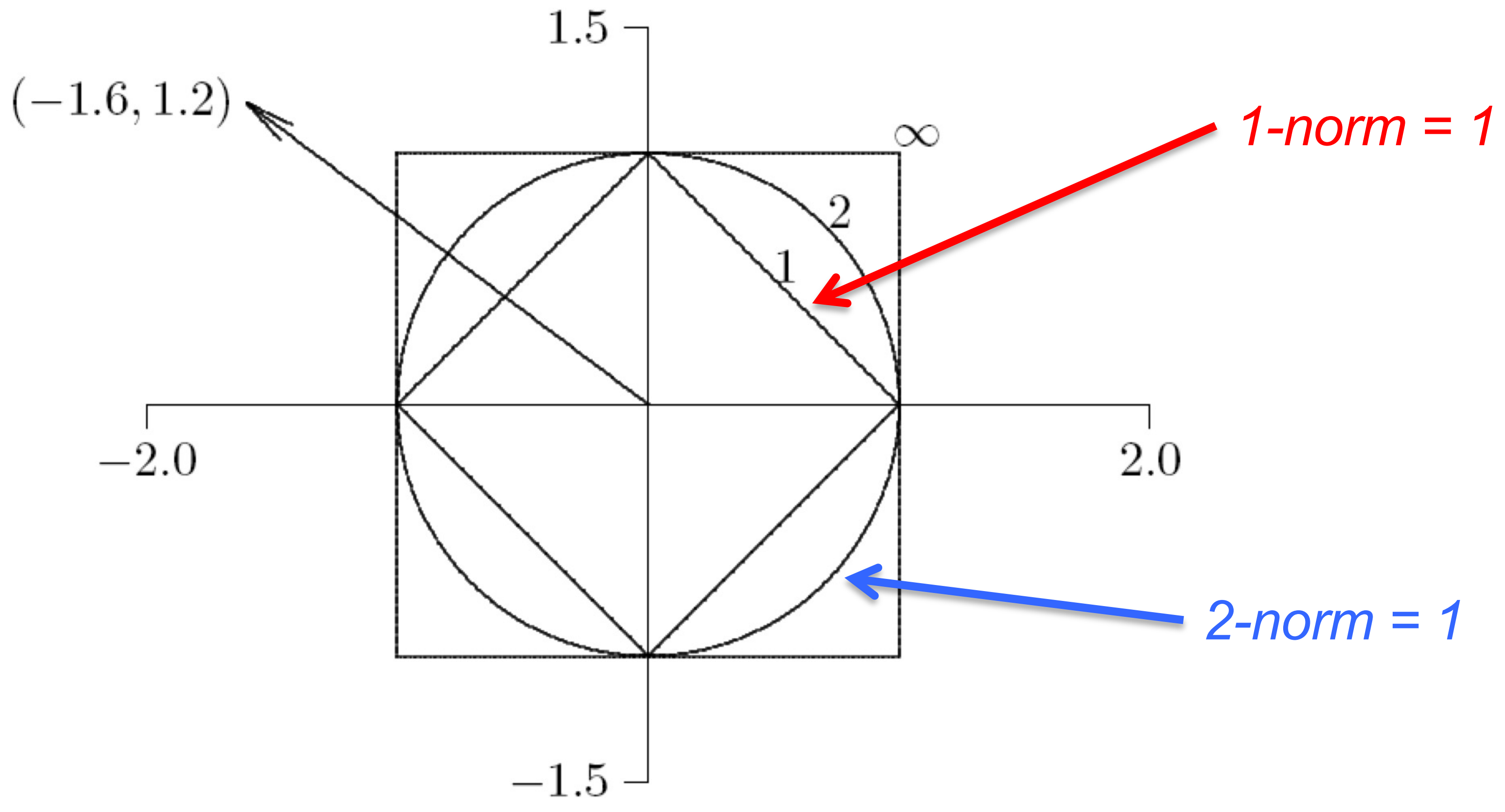


a. ^ According to formal definition — used by Prof. Demmel, LAPACK and Scilab. It represents the *largest relative rounding error* in round-to-nearest mode. The rationale is that the *rounding error* is half the interval upwards to the next representable number in finite-precision. Thus, the *relative* rounding error for number x is $[interval/2]/x$. In this context, the *largest* relative error occurs when $x = 1.0$, and is equal to $[ULP(1.0)/2]/1.0$, because real numbers in the lower half of the interval 1.0 ~ 1.0+ULP(1) are rounded down to 1.0, and numbers in the upper half of the interval are rounded up to 1.0+ULP(1). Here we use the definition of ULP(1) (*Unit in Last Place*) as the positive difference between 1.0 (which can be represented exactly in finite-precision) and the next greater number representable in finite-precision.

b. ^ According to the mainstream definition — used by Prof. Higham; applied in language constants in Ada, C, C++, Fortran, MATLAB, Mathematica, Octave, Pascal, Python and Rust etc., and defined in textbooks like «Numerical Recipes» by Press *et al*. It represents the *largest relative interval* between two nearest numbers in finite-precision, or the largest rounding error in round-by-chop mode. The rationale is that the *relative* interval for number x is $[interval]/x$ where $interval$ is the distance to upwards the next representable number in finite-precision. In this context, the *largest* relative interval occurs when $x = 1.0$, and is the interval between 1.0 (which can be represented exactly in finite-precision) and the next larger representable floating-point number. This interval is equal to ULP(1).

Review: Vector Norms Example

- Unit sphere in two dimensions for $\|\cdot\|_1$, $\|\cdot\|_2$, and $\|\cdot\|_\infty$ norms



- For indicated vector $\mathbf{x} = [-1.6\ 1.2]^T$, norms are

$$\|\mathbf{x}\|_1 = 2.8, \quad \|\mathbf{x}\|_2 = 2.0, \quad \|\mathbf{x}\|_\infty = 1.6$$

Norm Equivalence Review

- For any $\mathbf{x} \in \mathbb{R}^n$, $\|\mathbf{x}\|_1 \geq \|\mathbf{x}\|_2 \geq \|\mathbf{x}\|_\infty$

- *Hint: Consider x = [1 1 … 1]^T*

- We also have

$$\|\mathbf{x}\|_1 \leq \sqrt{n}\|\mathbf{x}\|_2, \quad \|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty, \quad \|\mathbf{x}\|_1 \leq n\|\mathbf{x}\|_\infty$$

- Thus norms differ by at most a constant and are hence equivalent.

- If one is small, they all must be proportionally small.

- More generally, *for any pair of vector norms,* $\|\cdot\|_\circ$ and $\|\cdot\|_*$, with n *fixed*, we have positive constants c and C (which depend on n), such that

$$c\,\|\mathbf{x}\|_\circ \;\leq\; \|\mathbf{x}\|_* \;\leq\; C\,\|\mathbf{x}\|_\circ$$