Harvard Applied Mathematics 205

Unit 0: Overview of Scientific Computing

Instructor: Chris H. Rycroft

Scientific Computing

Computation is now recognized as the "third pillar" of science (along with theory and experiment)

Why?

- Computation allows us to explore theoretical/mathematical models when those models can't be solved analytically. This is usually the case for real-world problems
- Computation allows us to process and analyze data on large scale
- Advances in algorithms and hardware over the past 50 years have steadily increased the prominence of scientific computing

What is Scientific Computing?

Scientific computing (SC) is closely related to numerical analysis (NA)

"Numerical analysis is the study of algorithms for the problems of continuous mathematics"
Nick Trefethen, SIAM News, 1992.

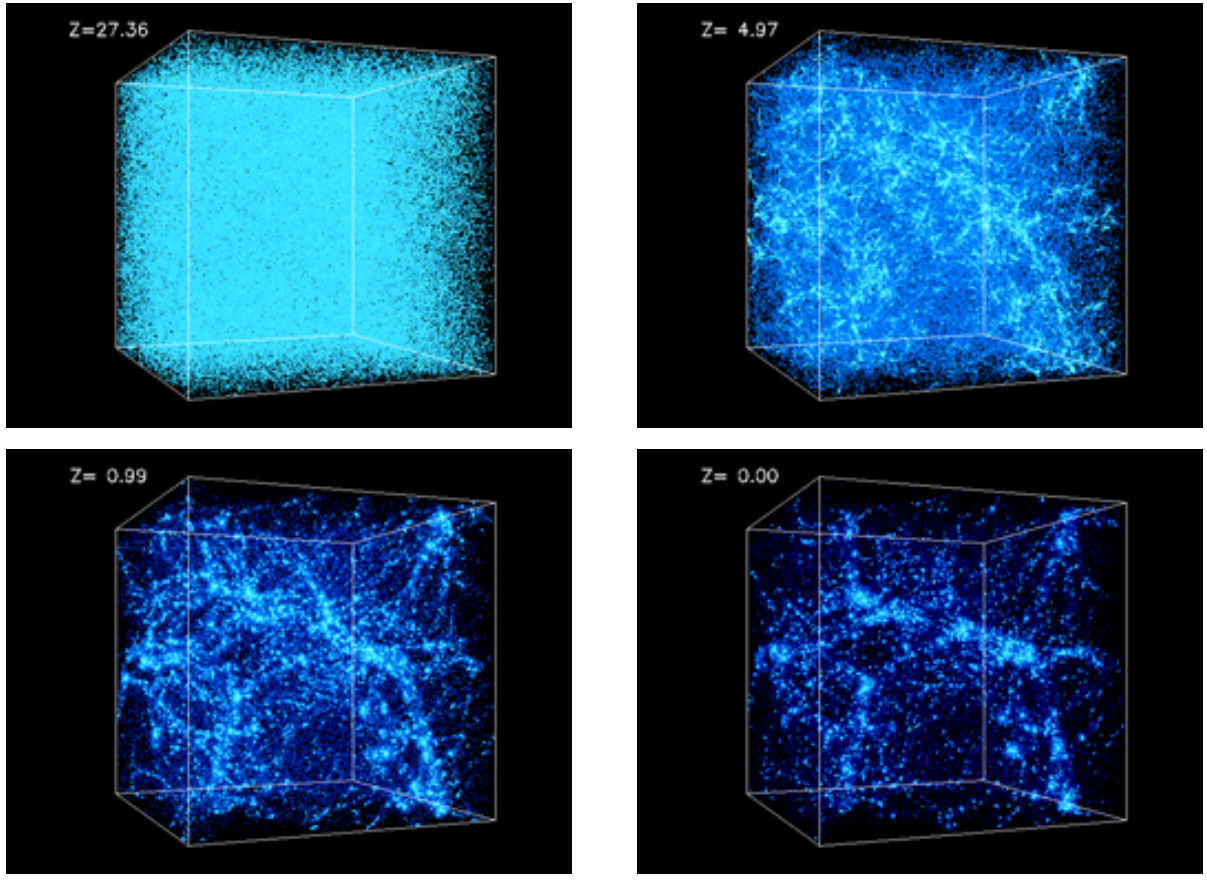
NA is the study of these algorithms, while SC emphasizes their application to practical problems

Continuous mathematics: algorithms involving real (or complex) numbers, as opposed to integers

NA/SC are quite distinct from Computer Science, which usually focus on discrete mathematics (*e.g.* graph theory or cryptography)

Scientific Computing: Cosmology

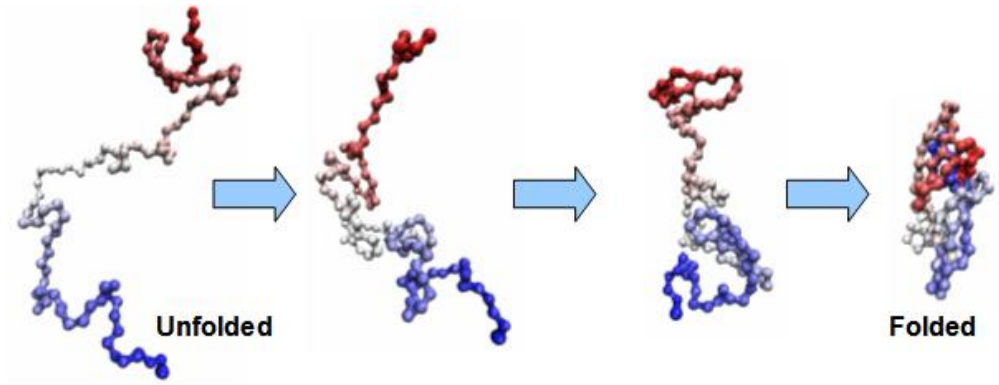
Cosmological simulations allow researchers to test theories of galaxy formation



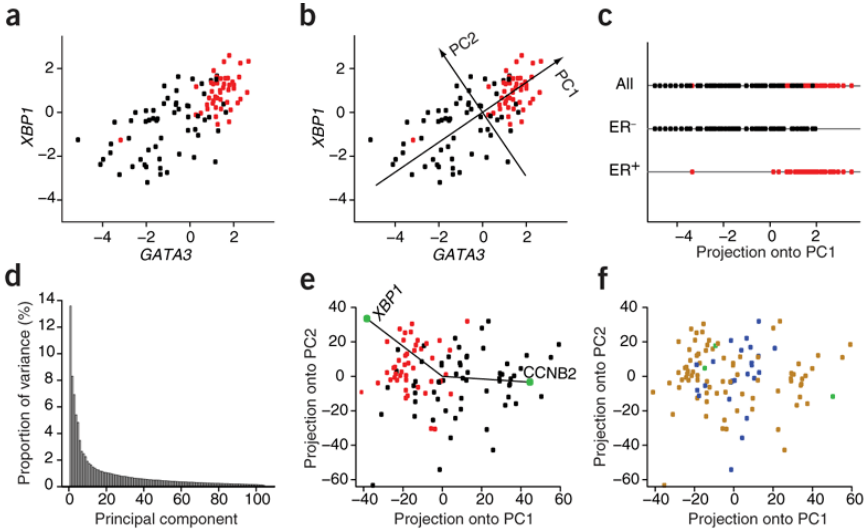
(cosmicweb.uchicago.edu)

Scientific Computing: Biology

Scientific computing is now crucial in molecular biology, *e.g.* protein folding (cnx.org)



Or statistical analysis of gene expression

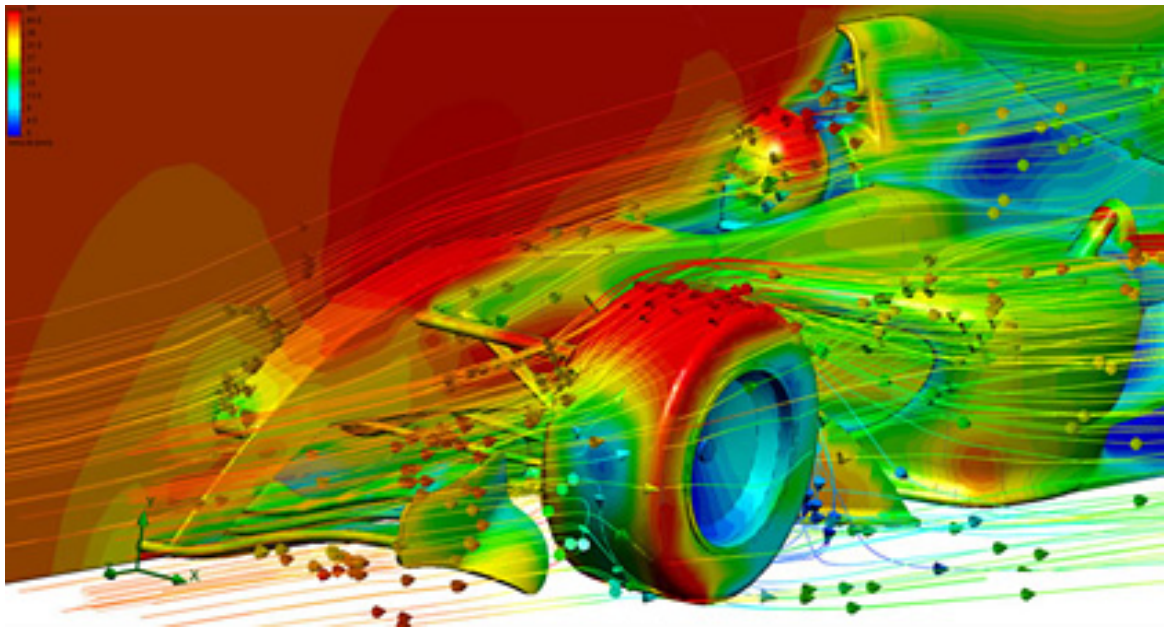


(Markus Ringner, Nature Biotechnology, 2008)

Scientific Computing: Computational Fluid Dynamics

Wind-tunnel studies are being replaced and/or complemented by CFD simulations

- Faster/easier/cheaper to tweak a computational design than a physical model
- Can visualize the entire flow-field to inform designers

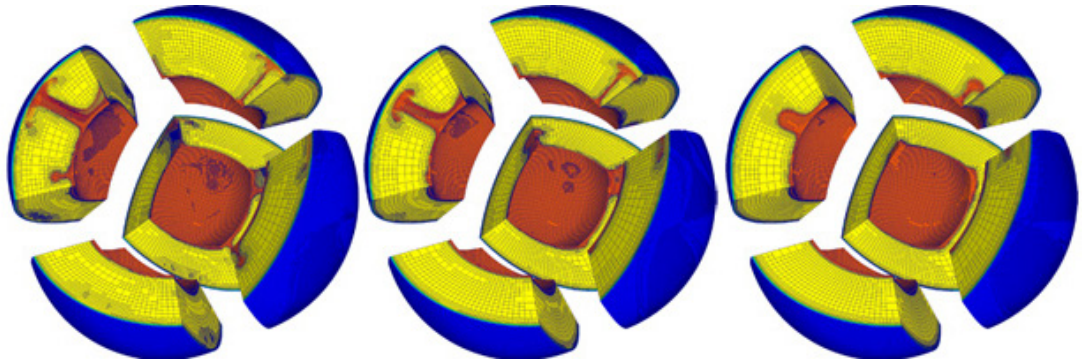


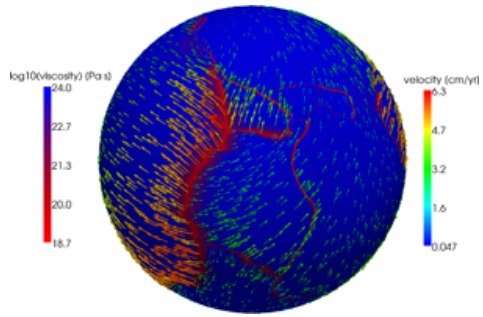
(www.mentor.com)

Scientific Computing: Geophysics

In geophysics we only have data on the Earth's surface

Computational simulations allow us to test models of the interior





(www.tacc.utexas.edu)

What is Scientific Computing?

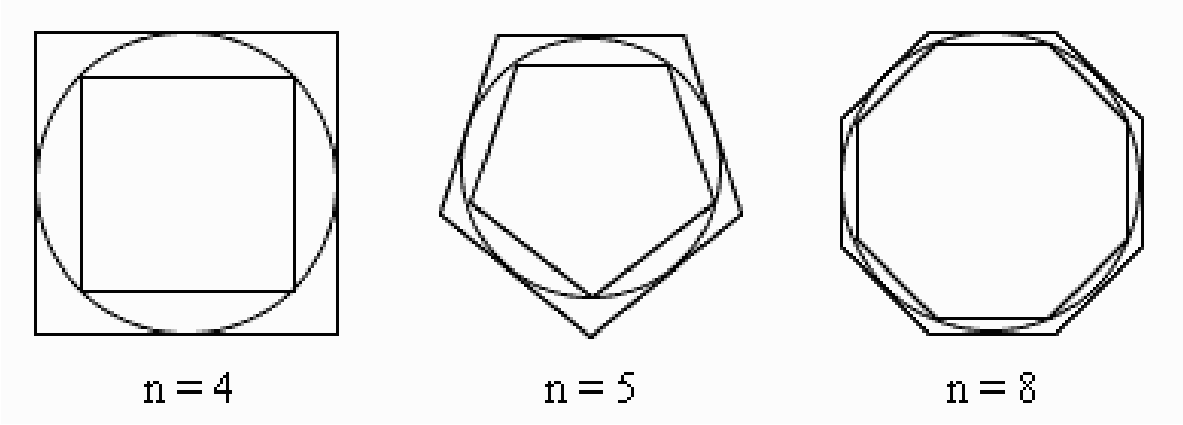
NA and SC have been important subjects for centuries, even though the names we use today are relatively recent.

One of the earliest examples: calculation of π. Early values:

- Babylonians: $3^1/_8$
- Quote from the Old Testament: *"And he made the molten sea of ten cubits from brim to brim, round in compass, and the height thereof was five cubits; and a line of thirty cubits did compass it round about"* – 1 Kings 7:23. Implies $\pi \approx 3$.
- Egyptians: $4(^8/_9)^2 \approx 3.16049$

What is Scientific Computing?

Archimedes' (287–212 BC) approximation of π used a recursion relation for the area of a polygon



Archimedes calculated that $3\frac{10}{71} < \pi < 3\frac{1}{7}$, an interval of 0.00201

What is Scientific Computing?

Key numerical analysis ideas captured by Archimedes:

- Approximate an infinite/continuous process (area integration) by a finite/discrete process (polygon perimeter)
- Error estimate ($3\frac{10}{71} < \pi < 3\frac{1}{7}$) is just as important as the approximation itself

What is Scientific Computing?

We will encounter algorithms from many great mathematicians: Newton, Gauss, Euler, Lagrange, Fourier, Legendre, Chebyshev, ...

They were practitioners of scientific computing (using "hand calculations"), *e.g.* for astronomy, optics, mechanics, ...

Very interested in accurate and efficient methods since hand calculations are so laborious

Calculating π more accurately

James Gregory (1638–1675) discovers the arctangent series

$$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots.$$

Putting $x = 1$ gives

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots,$$

but this formula converges very slowly.

Formula of John Machin (1680–1752)

If $\tan\alpha = 1/5$, then

$$\tan 2\alpha = \frac{2\tan\alpha}{1-\tan^2\alpha} = \frac{5}{12} \implies \tan 4\alpha = \frac{2\tan 2\alpha}{1-\tan^2 2\alpha} = \frac{120}{119}.$$

This very close to one, and hence

$$\tan\left(4\alpha - \frac{\pi}{4}\right) = \frac{\tan 4\alpha - 1}{1+\tan 4\alpha} = \frac{1}{239}.$$

Taking the arctangent of both sides gives the Machin formula

$$\frac{\pi}{4} = 4\tan^{-1}\frac{1}{5} - \tan^{-1}\frac{1}{239},$$

which gives much faster convergence.

The arctangent digit hunters

1706 John Machin, *100 digits*
1719 Thomas de Lagny, *112 digits*
1739 Matsunaga Ryohitsu, *50 digits*
1794 Georg von Vega, *140 digits*
1844 Zacharias Dase, *200 digits*
1847 Thomas Clausen, *248 digits*
1853 William Rutherford, *440 digits*
1876 William Shanks, *707 digits*

A short poem to Shanks[1]

Seven hundred seven
Shanks did state
Digits of π he would calculate
And none can deny
It was a good try
But he erred in five twenty eight!

[1] If you would like more poems and facts about π, see slides from *The Wonder of Pi*, a public lecture Chris gave at Newton Free Library on 3/14/19.

Scientific Computing vs. Numerical Analysis

SC and NA are closely related, each field informs the other

Emphasis of AM205 is Scientific Computing

> We focus on knowledge required for you to be a responsible user of numerical methods for practical problems

Sources of Error in Scientific Computing

There are several sources of error in solving real-world Scientific Computing problems

Some are beyond our control, *e.g.* uncertainty in modeling parameters or initial conditions

Some are introduced by our numerical approximations:

- Truncation/discretization: We need to make approximations in order to compute (finite differences, truncate infinite series...)
- Rounding: Computers work with *finite precision arithmetic*, which introduces rounding error

Sources of Error in Scientific Computing

It is crucial to understand and control the error introduced by numerical approximation, otherwise our results might be garbage

This is a major part of Scientific Computing, called error analysis

Error analysis became crucial with advent of modern computers: larger scale problems $\Longrightarrow$ more accumulation of numerical error

Most people are more familiar with rounding error, but discretization error is usually far more important in practice

Discretization Error vs. Rounding Error

Consider finite difference approximation to $f'(x)$:

$$f_{\text{diff}}(x; h) \equiv \frac{f(x+h) - f(x)}{h}.$$

From Taylor series

$$f(x+h) = f(x) + hf'(x) + f''(\theta)h^2/2, \text{ where } \theta \in [x, x+h]$$

we see that

$$f_{\text{diff}}(x; h) = \frac{f(x+h) - f(x)}{h} = f'(x) + f''(\theta)h/2.$$

Suppose $|f''(\theta)| \leq M$, then bound on discretization error is

$$|f'(x) - f_{\text{diff}}(x; h)| \leq Mh/2.$$

Discretization Error vs. Rounding Error

But we can't compute $f_{\text{diff}}(x; h)$ in exact arithmetic

Let $\tilde{f}_{\text{diff}}(x; h)$ denote finite precision approximation of $f_{\text{diff}}(x; h)$

Numerator of $\tilde{f}_{\text{diff}}$ introduces rounding error $\lesssim \epsilon |f(x)|$
(on modern computers $\epsilon \approx 10^{-16}$, will discuss this shortly)

Hence we have the rounding error

$$\begin{aligned} |f_{\text{diff}}(x; h) - \tilde{f}_{\text{diff}}(x; h)| &\lesssim \left| \frac{f(x+h) - f(x)}{h} - \frac{f(x+h) - f(x) + \epsilon f(x)}{h} \right| \\ &\leq \epsilon |f(x)|/h \end{aligned}$$

Discretization Error vs. Rounding Error

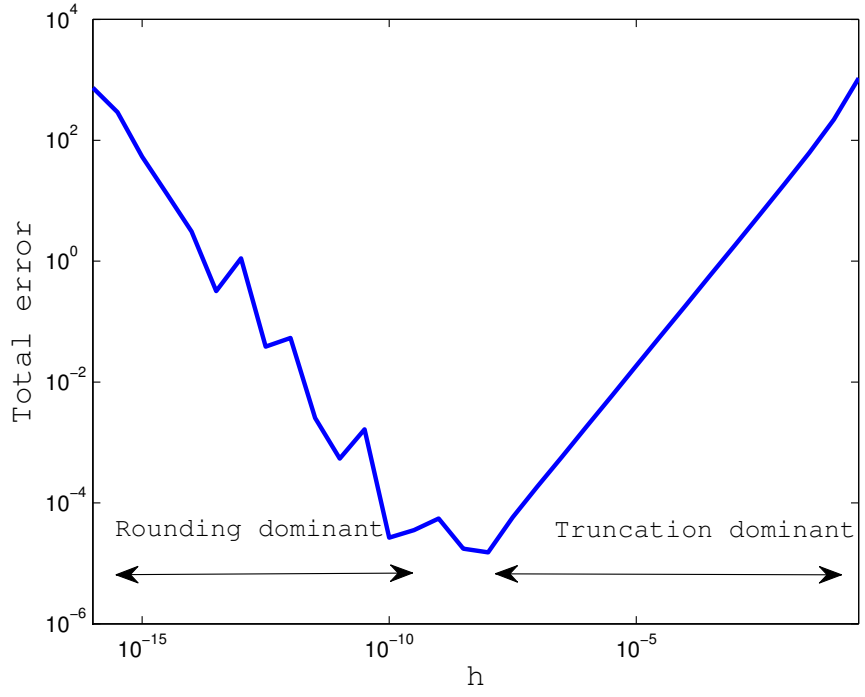
We can then use the triangle inequality ($|a+b| \leq |a| + |b|$) to bound the total error (discretization and rounding)

$$\begin{aligned} |f'(x) - \tilde{f}_{\text{diff}}(x;h)| &= |f'(x) - f_{\text{diff}}(x;h) + f_{\text{diff}}(x;h) - \tilde{f}_{\text{diff}}(x;h)| \\ &\leq |f'(x) - f_{\text{diff}}(x;h)| + |f_{\text{diff}}(x;h) - \tilde{f}_{\text{diff}}(x;h)| \\ &\leq Mh/2 + \epsilon|f(x)|/h \end{aligned}$$

Since ϵ is so small, here we expect discretization error to dominate until h gets sufficiently small

Discretization Error vs. Rounding Error

For example, consider $f(x) = \exp(5x)$, f.d. error at $x = 1$ as function of h:



Exercise: Use calculus to find local minimum of error bound as a function of h to see why minimum occurs at $h \approx 10^{-8}$

Discretization Error vs. Rounding Error

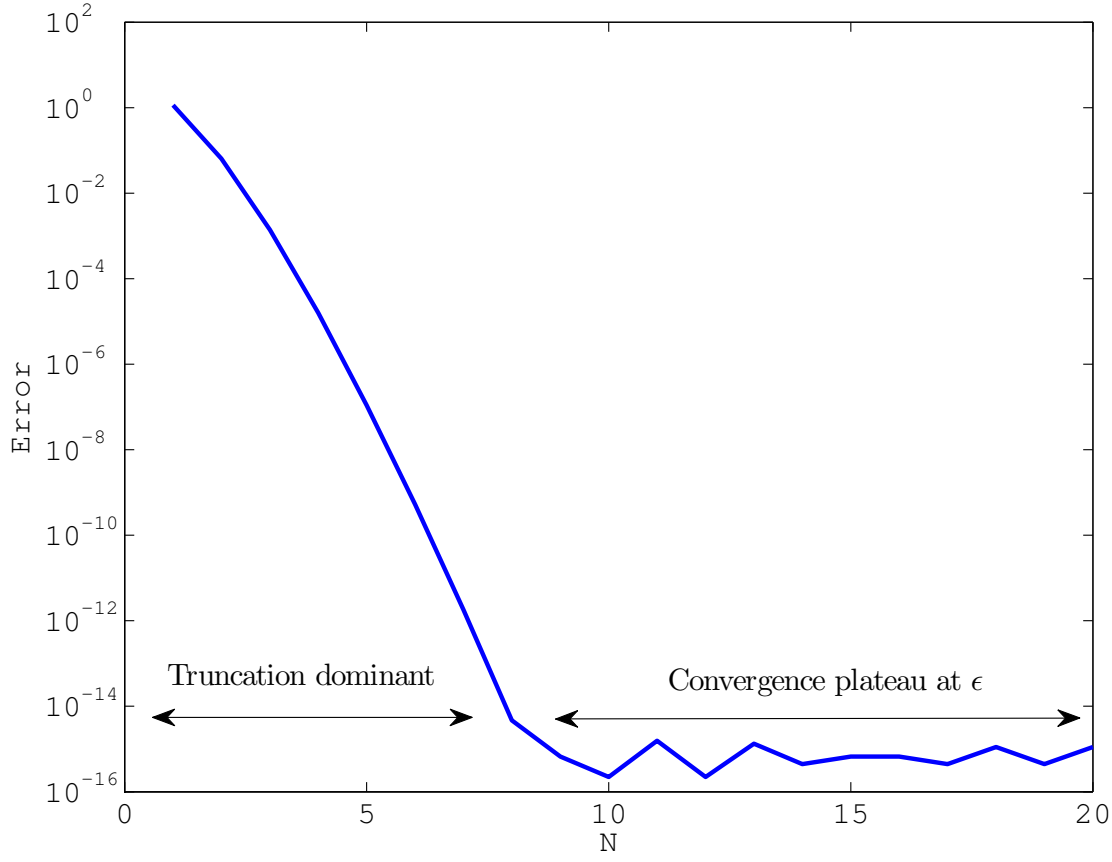
Note that in this finite difference example, we observe error growth due to rounding as $h \to 0$

This is a nasty situation, due to factor of h on the denominator in the error bound

A more common situation (that we'll see in Unit 1, for example) is that the error plateaus at around ϵ due to rounding error

Discretization Error vs. Rounding Error

Error plateau:



Absolute vs. Relative Error

Recall our bound $|f'(x) - \tilde{f}_{\text{diff}}(x;h)| \leq Mh/2 + \epsilon|f(x)|/h$

This is a bound on Absolute Error[2]:

$$\text{Absolute Error} \equiv \text{true value} - \text{approximate value}$$

Generally more interesting to consider Relative Error:

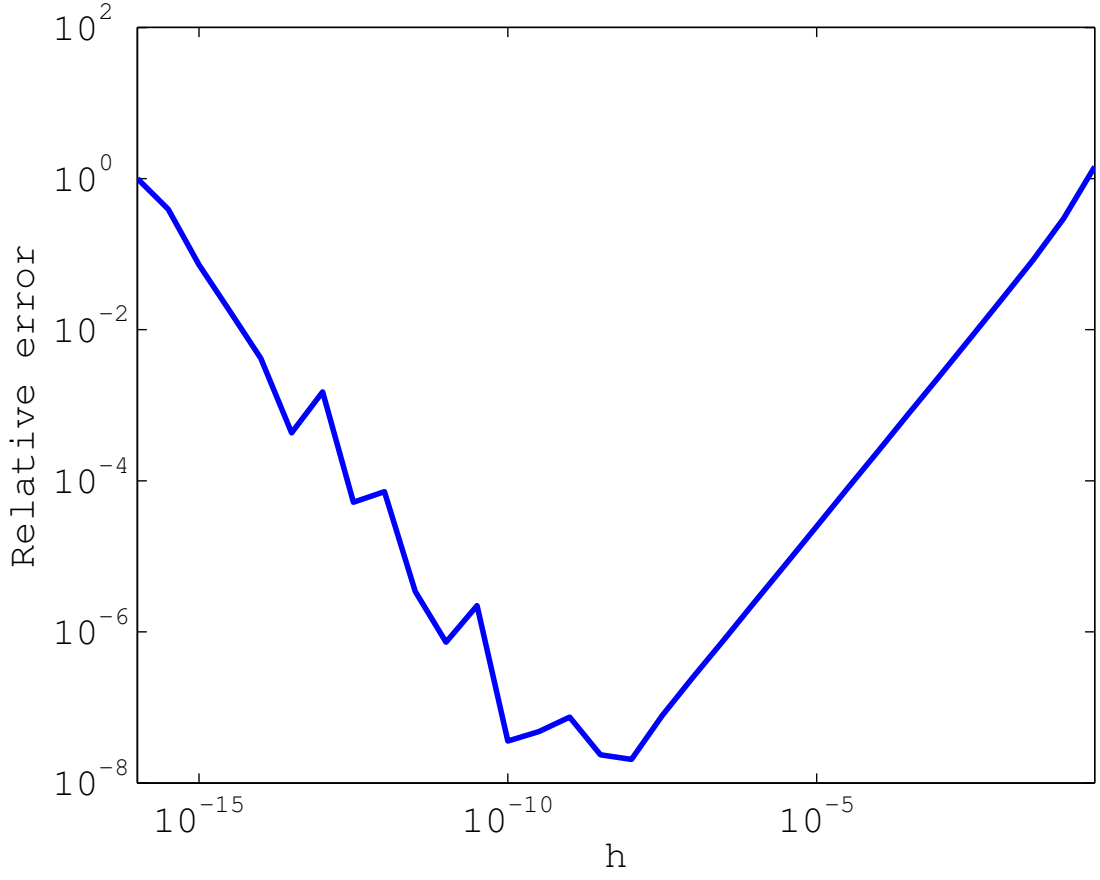
$$\text{Relative Error} \equiv \frac{\text{Absolute Error}}{\text{true value}}$$

Relative error takes the scaling of the problem into account

[2] We generally don't know the true value, we often have to use a surrogate for the true value, *e.g.* an accurate approximation using a different method

Absolute vs. Relative Error

For our finite difference example, plotting relative error just rescales the error values



Sidenote: Convergence plots

We have shown several plots of error as a function of a discretization parameter

In general, these plots are very important in scientific computing to demonstrate that a numerical method is behaving as expected

To display convergence data in a clear way, it is important to use appropriate axes for our plots

Sidenote: Convergence plots

Most often we will encounter algebraic convergence, where error decreases as αh^β for some $\alpha, \beta \in \mathbb{R}$

Algebraic convergence: If $y = \alpha h^\beta$, then

$$\log(y) = \log \alpha + \beta \log h$$

Plotting algebraic convergence on log–log axes asymptotically yields a straight line with gradient β

Hence a good way to deduce the algebraic convergence rate is by comparing error to αh^β on log–log axes

Sidenote: Convergence plots

Sometimes we will encounter exponential convergence, where error decays as $\alpha e^{-\beta N}$ as $N \to \infty$

If $y = \alpha e^{-\beta N}$ then $\log y = \log \alpha - \beta N$

Hence for exponential convergence, better to use semilog-y axes (like the previous "error plateau" plot)

Numerical sensitivity

In practical problems we will always have input perturbations (modeling uncertainty, rounding error)

Let $y = f(x)$, and denote perturbed input $\hat{x} = x + \Delta x$

Also, denote perturbed output by $\hat{y} = f(\hat{x})$, and $\hat{y} = y + \Delta y$

The function f is *sensitive* to input perturbations if $\Delta y \gg \Delta x$

This is sensitivity inherent in f, independent of any approximation (though approximation $\hat{f} \approx f$ can exacerbate sensitivity)

Sensitivity and Conditioning

For a sensitive problem, small input perturbation $\Longrightarrow$ large output perturbation

Can be made quantitative with concept of condition number[3]

$$\text{Condition number} \equiv \frac{|\Delta y/y|}{|\Delta x/x|}$$

$$\begin{aligned} \text{Condition number} \gg 1 \quad &\Longleftrightarrow \quad \text{small perturbations} \\ &\qquad\quad\ \ \text{are amplified} \\ &\Longleftrightarrow \quad \text{ill-conditioned problem} \end{aligned}$$

[3] Here we introduce the *relative* condition number, generally more informative than the absolute condition number

Sensitivity and Conditioning

Condition number can be analyzed for all sorts of different problem types (independent of algorithm used to solve the problem), *e.g.*

- Function evaluation, $y = f(x)$
- Matrix multiplication, $Ax = b$ (solve for b given x)
- Matrix equation, $Ax = b$ (solve for x given b)

See notes: Numerical conditioning examples

Stability of an algorithm

In practice, we solve problems by applying a numerical method to a mathematical problem, *e.g.* apply Gaussian elimination to $Ax = b$

To obtain an accurate answer, we need to apply a stable numerical method to a well-conditioned mathematical problem

Question: What do we mean by a stable numerical method?

Answer: Roughly speaking, the numerical method doesn't accumulate error (*e.g.* rounding error) and produce garbage

We will make this definition more precise shortly, but first, we discuss rounding error and finite-precision arithmetic

Code examples

From here on, a number of code examples will be provided.

They will all be available via the `am205_examples` Git repository.

Git is one example of version control software, which tracks the development of files in a software project. It has many desirable features, such as allowing files to be compared to any previous version,[4] and allowing multiple people to collaborate.

In the slides, notation like [*code_example.py*] will be used to indicate an associated example in the repository.

[4] This is *extremely* useful for debugging.

Code examples

You can simply browse files on the Github website, or download a current snapshot as a ZIP file.

Git can be installed as a command-line utility on all major systems. To get a copy of the repository, type

```
git clone git@github.com:chr1shr/am205_examples.git
```

Then, at later times, you can type

```
git pull
```

to obtain any updated files. Graphical interfaces for Git are also available.

Finite-precision arithmetic

Key point: we can only represent a finite and discrete subset of the real numbers on a computer.

The standard approach in modern hardware is to use binary floating point numbers (basically "scientific notation" in base 2),

$$\begin{aligned} x &= \pm(1 + d_1 2^{-1} + d_2 2^{-2} + \ldots + d_p 2^{-p}) \times 2^E \\ &= \pm(1.d_1 d_2 \ldots d_p)_2 \times 2^E \end{aligned}$$

Finite-precision arithmetic

We store

$$\underbrace{\pm}_{\text{1 sign bit}} \qquad \underbrace{d_1, d_2, \ldots, d_p}_{p \text{ mantissa bits}} \qquad \underbrace{E}_{\text{exponent bits}}$$

Note that the term bit is a contraction of "binary digit"[5].

This format assumes that $d_0 = 1$ to save a mantissa bit, but sometimes $d_0 = 0$ is required, such as to represent zero.

The exponent resides in an interval $L \leq E \leq U$.

[5] This terminology was first used in Claude Shannon's seminal 1948 paper, *A Mathematical Theory of Communication*.

IEEE floating point arithmetic

Universal standard on modern hardware is IEEE floating point arithmetic (IEEE 754), adopted in 1985.

Development led by Prof. William Kahan (UC Berkeley)[6], who received the 1989 Turing Award for his work.

	total bits	p	L	U
IEEE single precision	32	23	-126	127
IEEE double precision	64	52	-1022	1023

Note that single precision has 8 exponent bits but only 254 different values of E, since some exponent bits are reserved to represent special numbers.

[6] It's interesting to search for `paranoia.c`.

Exceptional values

These exponents are reserved to indicate special behavior, including values such as Inf and NaN:

- Inf = "infinity", *e.g.* $1/0$ (also $-1/0 = -\text{Inf}$)
- NaN = "Not a Number", *e.g.* $0/0$, Inf/Inf

IEEE floating point arithmetic

Let $\mathbb{F}$ denote the floating point numbers. Then $\mathbb{F} \subset \mathbb{R}$ and $|\mathbb{F}| < \infty$.

Question: how should we represent a real number x, which is not in $\mathbb{F}$?

Answer: There are two cases to consider:

- Case 1: x is outside the range of $\mathbb{F}$ (too small or too large)
- Case 2: The mantissa of x requires more than p bits.

IEEE floating point arithmetic

Case 1: x is outside the range of $\mathbb{F}$ (too small or too large)

Too small:

- Smallest positive value that can be represented in double precision is $\approx 10^{-323}$.
- For a value smaller than this we get underflow, and the value typically set to 0.

Too large:

- Largest $x \in \mathbb{F}$ ($E = U$ and all mantissa bits are 1) is approximately $2^{1024} \approx 10^{308}$.
- For values larger than this we get overflow, and the value typically gets set to Inf.

IEEE floating point arithmetic

Case 2: The mantissa of x requires more than p bits

Need to round x to a nearby floating point number

Let $\texttt{round} : \mathbb{R} \to \mathbb{F}$ denote our rounding operator. There are several different options: round up, round down, round to nearest, *etc.*

This introduces a rounding error:

- absolute rounding error $x - \texttt{round}(x)$
- relative rounding error $(x - \texttt{round}(x))/x$

Machine precision

It is important to be able to quantify this rounding error—it's related to machine precision, often denoted as ϵ or ϵ_{mach}.

ϵ is the difference between 1 and the next floating point number after 1, *i.e.* $\epsilon = 2^{-p}$.

In IEEE double precision, $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$.

Rounding Error

Let $x = (1.d_1 d_2 \dots d_p d_{p+1})_2 \times 2^E \in \mathbb{R}_{>0}$.

Then $x \in [x_-, x_+]$ for $x_-, x_+ \in \mathbb{F}$, where
$x_- = (1.d_1 d_2 \dots d_p)_2 \times 2^E$ and $x_+ = x_- + \epsilon \times 2^E$.

$\texttt{round}(x) = x_-$ or x_+ depending on the rounding rule, and hence
$|\texttt{round}(x) - x| < \epsilon \times 2^E$ (why not "$\leq$")[7]

Also, $|x| \geq 2^E$.

[7] With "round to nearest" we have $|\texttt{round}(x) - x| \leq 0.5 \times \epsilon \times 2^E$, but here we prefer the above inequality because it is true for any rounding rule.

Rounding Error

Hence we have a relative error of less than ϵ, *i.e.*,

$$\left| \frac{\texttt{round}(x) - x}{x} \right| < \epsilon.$$

Another standard way to write this is

$$\texttt{round}(x) = x \left(1 + \frac{\texttt{round}(x) - x}{x} \right) = x(1 + \delta)$$

where $\delta = \frac{\texttt{round}(x) - x}{x}$ and $|\delta| < \epsilon$.

Hence rounding give the correct answer to within a factor of $1 + \delta$.

Floating Point Operations

An arithmetic operation on floating point numbers is called a "floating point operation": $\oplus$, $\ominus$, $\otimes$, $\oslash$ versus $+$, $-$, $\times$, $/$.

Computer performance is often measured in "flops": number of floating point operations per second.

Supercomputers are ranked based on number of flops achieved in the "linpack test," which solves dense linear algebra problems.

Currently, the fastest computers are in the 100 petaflop range:
1 petaflop $= 10^{15}$ floating point operations per second

Floating Point Operations

See http://www.top500.org for an up-to-date list of the fastest supercomputers.[8]

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
6	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
7	**Selene** - DGX A100 SuperPOD, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	272,800	27,580.0	34,568.6	1,344

[8] Rmax: flops from linpack test. Rpeak: theoretical maximum flops.

Floating Point Operations

Modern supercomputers are very large, link many processors together with fast interconnect to minimize communication time

Floating Point Operation Error

IEEE standard guarantees that for $x, y \in \mathbb{F}$, $x \circledast y = \texttt{round}(x * y)$ ($*$ and $\circledast$ represent one of the 4 arithmetic operations)

Hence from our discussion of rounding error it follows that for $x, y \in \mathbb{F}$, $x \circledast y = (x * y)(1 + \delta)$, for some $|\delta| < \epsilon$

Loss of Precision

Machine precision can be tested [*acc_test.py*, *acc_test.cc*]

Since ϵ is so small, we typically lose very little precision per operation

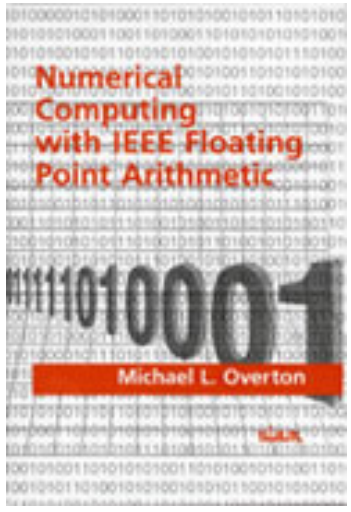
See Lecture: Example of benign loss of precision

But loss of precision is not always benign:

See Lecture: Significant loss of precision due to cancellation

IEEE Floating Point Arithmetic

For more detailed discussion of floating point arithmetic, see:



"Numerical Computing with IEEE Floating Point Arithmetic,"
Michael L. Overton, SIAM, 2001

Numerical Stability of an Algorithm

We have discussed rounding for a single operation, but in AM205 we will study numerical algorithms that require many operations

For an algorithm to be useful, it must be stable in the sense that rounding errors do not accumulate and result in "garbage" output

More precisely, numerical analysts aim to prove backward stability: The method gives the exact answer to a slightly perturbed problem

For example, a numerical method for solving $Ax = b$ should give the exact answer for $(A + \Delta A)x = (b + \Delta b)$ for small ΔA, Δb

Numerical Stability of an Algorithm

We note the importance of conditioning: Backward stability doesn't help us if the mathematical problem is ill-conditioned

For example, if A is ill-conditioned then a backward stable algorithm for solving $Ax = b$ can still give large error for x

Backward stability analysis is a deep subject which we do not cover in detail in AM205

We will, however, compare algorithms with different stability properties and observe the importance of stability in practice