Applied Mathematics 205

Unit 0: Overview of Scientific Computing

Lecturer: Dr. David Knezevic

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!
- E.g. Navier–Stokes equations model fluid flow, but exact solutions only exist in a few simple cases
- Advances in algorithms and hardware over the past ~ 50 years have steadily increased prominence of scientific computing

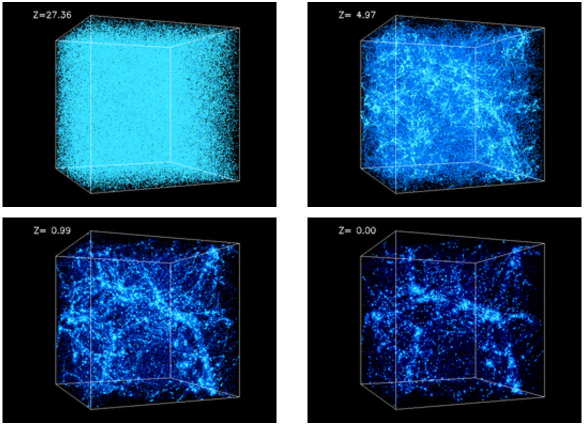
Scientific Computing

Computation is now very prominent in many different branches of science

For example...

Scientific Computing: Cosmology

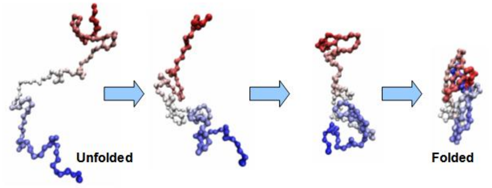
Cosmological simulations allow researchers to test theories of galaxy formation



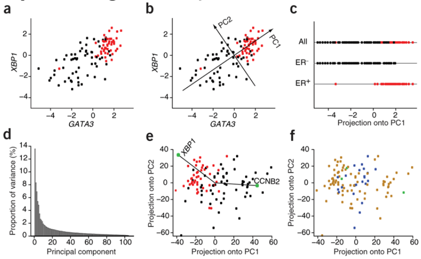
(cosmicweb.uchicago.edu)

Scientific Computing: Biology

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



Or statistical analysis of gene expression

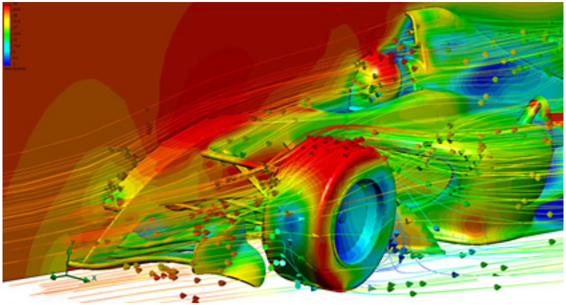


(Nature Biotechnology, 2008)

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

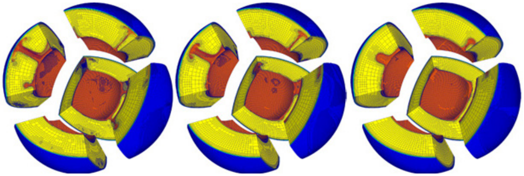


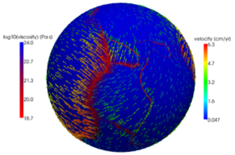
(www.mentor.com)

Scientific Computing: Geophysics

In geophysics we only have data on (or near) the earth's surface

Computational simulations allow us to test models of the interior





(www.tacc.utexas.edu)

What is Scientific Computing?

Scientific Computing (S.C.) is closely related to Numerical Analysis (N.A.)

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

N.A. is the study of these algorithms, S.C. emphasizes their application to practical problems

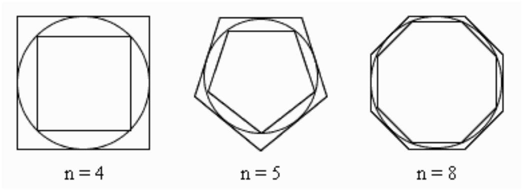
Continuous mathematics $\Longrightarrow$ algorithms involving real (or complex) numbers, as opposed to integers

N.A./S.C. is quite distinct from Computer Science, which usually focuses on discrete mathematics (graph theory, cryptography, ...)

What is Scientific Computing?

N.A./S.C. have been important subjects for centuries! (Though the names we use today are relatively recent...)

One of the earliest examples: Archimedes (287–212 BC) approximation of π using $n = 96$ polygon



Archimedes calculated that $3\frac{10}{71} < \pi < 3\frac{10}{70}$, an interval of 0.00201

What is Scientific Computing?

Key Numerical Analysis ideas captured by Archimedes:

- Approximate an infinite/continuous process (integration) by a finite/discrete process (polygon perimeter)
- Error estimate ($3\frac{10}{71} < \pi < 3\frac{10}{70}$) 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, mechanics, optics,...

And were very interested in accurate and efficient methods since hand calculations are so laborious

Scientific Computing vs. Numerical Analysis

S.C. and N.A. 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 $\implies$ 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 $f_{\mathrm{diff}}(x;h)$ in exact arithmetic

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

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

Hence we have the rounding error

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

Discretization Error vs. Rounding Error

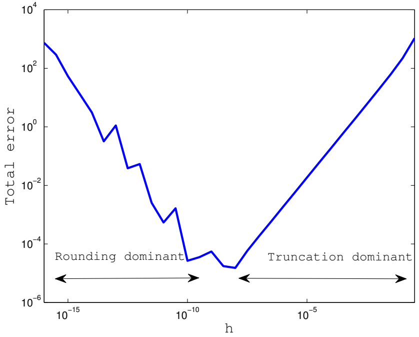
We can then use the triangle inequality ($|a+b| \leq |a| + |b|$) to bound the total error (discretization and rounding)

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

Since ϵ is so small, 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

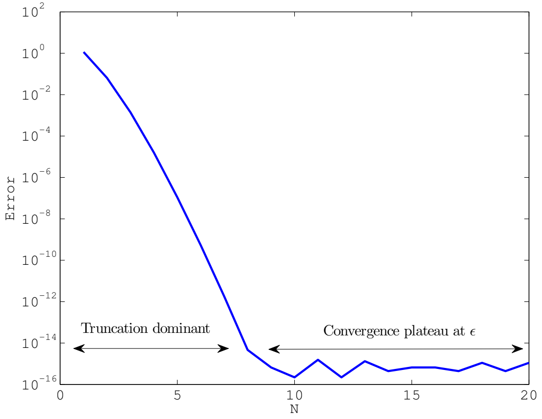
Note that this in this finite difference example, we observe error growth due to rounding as $h \to 0$

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

A more common situation (that we'll see in Unit I, 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 $|f'(x) - \tilde{f}_{\mathrm{diff}}(x;h)| \leq Mh/2 + \epsilon|f(x)|/h$

This is a bound on Absolute Error:

Absolute Error $\equiv$ true value - approximate value

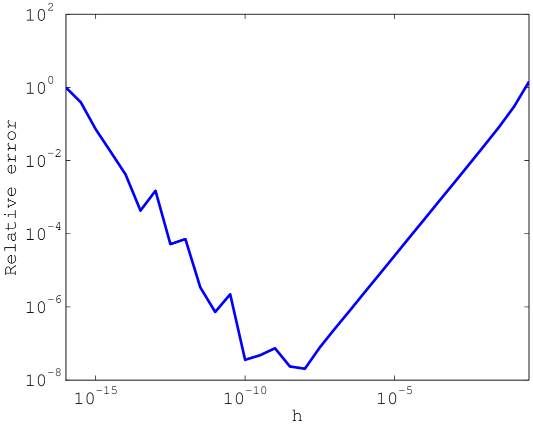
Generally more interesting to consider Relative Error:

Relative Error $\equiv$ Absolute Error / true value

Relative error takes the scaling of the problem into account

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

These types of plots are very important, since they allow us 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 $h \to 0$, 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 (`loglog` in Matlab) asymptotically yields a straight line with gradient β

Hence a good way to deduce algebraic convergence rate is by comparing error to αh^{β} on log-log axes (e.g. see Assignment 0)

Sidenote: Convergence plots

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

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

Hence for exponential convergence, better to use "semilog-y" axes (`semilogy` in Matlab), e.g. see 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 sensitivity is a property of f (i.e. not related to a numerical approximation of f)

Sensitivity and Conditioning

Hence for a sensitive problem: small input perturbation $\Longrightarrow$ large output perturbation

Can be made quantitive with concept of condition number[1]

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

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

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

Sensitivity and Conditioning

Condition number can be analyzed for different types of problem (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 Lecture: 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

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)

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

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"

This format assume that $d_0 = 1$ to save a mantissa bit, but sometimes $d_0 = 0$ is required[2]

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

[2]For example, to represent zero or for "subnormals" (see Assignment 1)

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 (Berkeley), received Turing Award in 1989 for this 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: some exponent values are "reserved"

Exceptional Values

These exponents are reserved to indicate special behavior, including "exceptional values": Inf, NaN

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

Matlab handles Inf and NaN in a natural way,
e.g. try "`Inf + 1`," "`Inf` $*$ `0`" or "`Inf / NaN`"

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[3] is $\approx 10^{-323}$
- For values smaller in magnitude than this we get "Underflow," and value typically gets set to 0

Too large:

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

[3] Smallest value is less than $2^{-1022} \approx 10^{-308}$ because IEEE allows "subnormal" numbers, see Assignment 1 for more details

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 `round` : $\mathbb{R} \to \mathbb{F}$ denote our "rounding operator" (several different options: chop, round up, round down, round to nearest)

This introduces a rounding error:

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

Machine precision

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

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

In IEEE Double Precision, $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$
("eps" in Matlab)

Rounding Error

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

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

$\text{round}(x) = x_-$ or x_+ (depending on rounding rule), hence
$|\text{round}(x) - x| < \epsilon \times 2^E$ (why not "$\leq$"?)[4]

Also, $|x| \geq 2^E$ (why?)

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

Rounding Error

Therefore we have a relative error of less than ϵ, i.e.:

$$\boxed{\left|\frac{\mathrm{round}(x)-x}{x}\right| < \epsilon}$$

Another standard way to write this is:

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

where $\delta \equiv \frac{\mathrm{round}(x)-x}{x}$, and $|\delta| < \epsilon$

Hence rounding gives 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": $\oplus$, $\ominus$, $\otimes$, $\oslash$ vs. $+, -, \times, /$

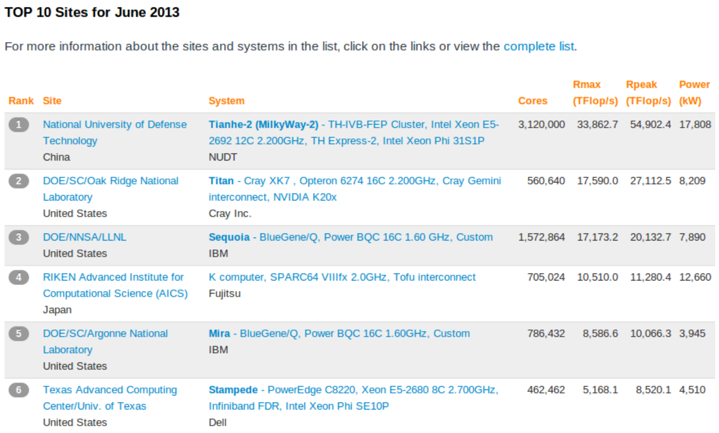
Computer performance is often measured in "flops": number of floating point operations per second

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

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

Floating Point Operations

See `www.top500.org` for up-to-date list of the fastest supercomputers[5]



TOP 10 Sites for June 2013

For more information about the sites and systems in the list, click on the links or view the complete list.

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National University of Defense Technology China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510

[5] Rmax: flops from linpack test, Rpeak: theoretical max 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 $x, y \in \mathbb{F}$, $x \circledast y = \mathrm{round}(x * y)$ (here $*$ and $\circledast$ represent any 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

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

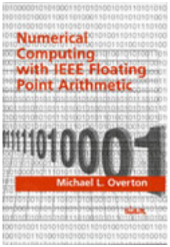
See Lecture: Example of benign loss of precision

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 which 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 $Ax = (b + \Delta b)$ for small Δb

Numerical Stability of an Algorithm

Hence the importance of conditioning is clear: 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 don't have time to cover in detail in AM205

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