Conditioning and Numerical Stability

Eric Liu

Yelp
Applied Learning Group

Outline

Outline

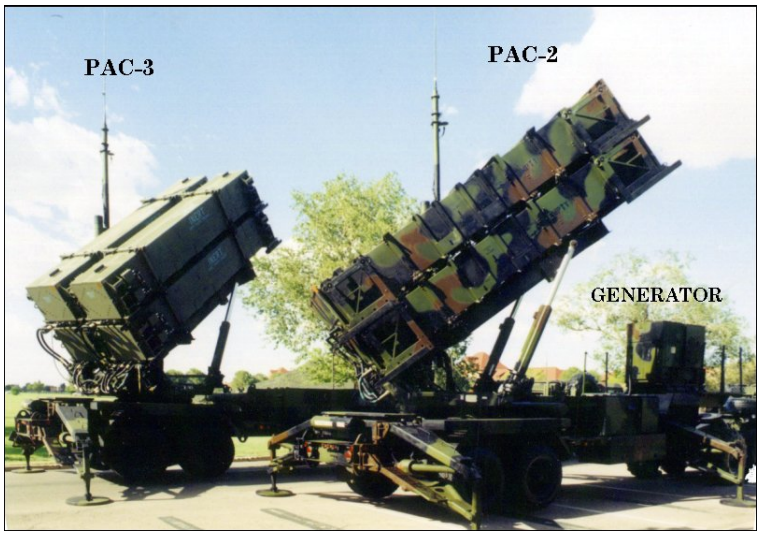
Three main sources of errors in (finite precision) numerical computations:

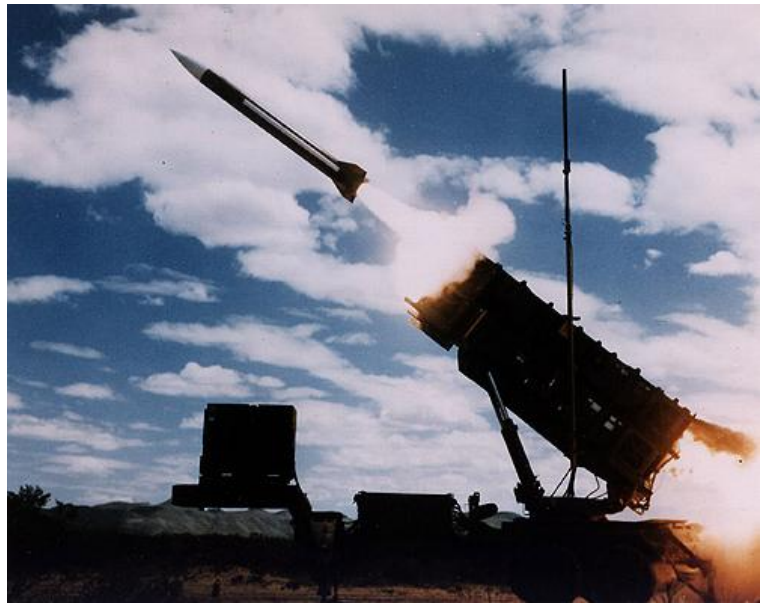
- rounding (today)
- data uncertainty (other people...)
- truncation (any time you work with non-polynomials, not today)

We'll look at how finite precision affects numerical algorithms, providing some mathematic basis and going through some common examples.

16 Feb 2011: A scud missile kills 28 US soldiers when the Patriot missile system fails to track/destroy it. (Desert Storm)

Patriot's internal clock measures time in $0.1s$ increments (as an integer offset). This is converted to floating-point (24-bit) and used in combination with the missile's velocity vector to compute the region of sky where the radar should search next.





Hours	Seconds	Calculated Time (s)	Error (s)	Error (m)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0025	55
20	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100	360000	359999.6667	.3433	687

After 20 hours of continuous operation, the error is so large that the Patriot is virtually useless. The system in Saudi Arabia was on for 100 hrs.

Conclusion: When we rely on floating point arithmetic, people die.
... Wait that doesn't sound right...
Conclusion v2.0: The effects of floating point arithemtic are complex and subtle. Assuming FP behaves like $\mathbb{R}$ is dangerous.

Linear Algebra Stuff

- A, A_{ij}: a matrix $\in \mathbb{C}^{n \times n}$, but we stick to $\mathbb{R}^{n \times n}$ for ease
- $\mathbf{x}, x_i, [a_1, a_2, \ldots, a_n]$: a vector $\in \mathbb{R}^n$
- a: a scalar $\in \mathbb{R}$
- $\|\cdot\|$: a norm
- A^T, A^{-1}, A^{-T}: matrix transpose, inverse, inverse transpose
  - An invertible matrix is called non-singular

Other Stuff

- Absolute Difference: $a - a_{reference}$
- Relative Difference: $\frac{a - a_{reference}}{a_{reference}}$
- $\tilde{x}$: the "numerically computed" value of x
- δx: an infinitesimal pertubation of x; e.g., $x + h, h \to 0$
- δf: an infinitesimal pertubation of $f(x), f(x + \delta x) - f(x)$

Vector Norms

For a vector space V, the mapping $\|\cdot\| : V \to \mathbb{R}$ is a norm on V if:

1. $\|a\mathbf{v}\| = a\mathbf{v}$ (scalability)
2. $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$ (triangle inequality)
3. $\|\mathbf{v}\| = 0 \implies \mathbf{v} = 0$

   We will use $\|\mathbf{v}\|_2 = \sqrt{\sum_n v_i^2}$ (2-norm or Euclidiean norm)

Induced Matrix Norms

$$\|A\| = \max\left(\frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} : x \neq 0\right)$$

- Find the vvector $\mathbf{x}$ for which $\|A\mathbf{x}\|$ is (relatively) largest; i.e., the direction that is amplified the most by A. $\|A\|$ is this value.
- For $\|\cdot\|_2$, $\|A\| = \sigma_1$, the largest singular value of A, aka sqrt of largest eigenvalue of AA^H.

Outline

Definition: Relative Condition Number

$$\kappa(x) = \sup_{\delta x} \left(\frac{\|\delta f\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right) = \frac{\|J\|}{\|f\| \, / \, \|x\|},$$

if J (Jacobian) exists.

In words...

- Sensitivity of the output, f, to pertubations of the data, x
- Measures ratio of relative changes in $f(x)$ to relative changes in x
- Large κ means (certain) changes in x can lead to much larger changes in f

Notes:

- Large κ is "ill-conditioned"; small κ is "well-conditioned"
- κ is dependent on norm-type; we will use $\|\cdot\|_2$.

Motivation: Using Pertubations

$\tilde{y} - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \frac{f''(x+\theta\Delta x)}{2}(\Delta x)^2 + \ldots$, so
$\frac{\tilde{y}-y}{y} = \frac{xf'(x)}{f(x)}\frac{\Delta x}{x}$, giving relative conditioning: $\kappa(x) = \frac{xf'(x)}{f(x)}$

Square Root: Well-conditioned

For $\sqrt{x}$, $\kappa = \frac{1}{2}$.
$\sqrt{x}$ is not sensitive to its inputs; the value of x never makes it harder or easier to compute.

Subtraction: (sometimes) Ill-conditioned

For $x_1 - x_2$, $\kappa = \frac{\sqrt{2}}{|x_1 - x_2|/\sqrt{x_1^2 + x_2^2}}$.
$\kappa \to \infty$ when $x_1 \approx x_2$, following intuition on cancellation error

$\kappa(A) \equiv \|A\| \left\|A^{-1}\right\|$. If $\|\cdot\|_2$, $\kappa(A) = \frac{\sigma_1}{\sigma_n}$. $\kappa(A) \geq 1$ by definition.

Motivating the Definition: $f(\mathbf{x}) = A\mathbf{x}$

$\kappa = \|A\| \frac{\|\mathbf{x}\|}{\|A\mathbf{x}\|}$, condition number of f with respect to pertubations in $\mathbf{x}$

- For non-singular A, $\kappa \leq \|A\| \left\|A^{-1}\right\|$.
- This pairing shows up in other fundamental linear algebra problems too: computing $\mathbf{x} = A^{-1}\mathbf{b}$ with respect to pertubations in $\mathbf{b}$ and A

What's wrong with the Normal Equations?

- Say I want to compute x that minimizes $\|\mathbf{b} - A\mathbf{x}\|$ for $A \in \mathbb{R}^{m \times n}, m \geq n$
- You might have been taught to solve: $A^T A\mathbf{x} = \mathbf{b}$. Is this wise?
- $\kappa(A^T A) = \left\|A^T A\right\| \left\|A^{-1}A^{-T}\right\| \leq \|A\|^2 \left\|A^{-1}\right\|^2 = \kappa(A)^2$. (Equality for certain classes of A and certain norms; constants are never large though.)
- So, terrible idea because it **squares the condition number.**

Singular vectors/values

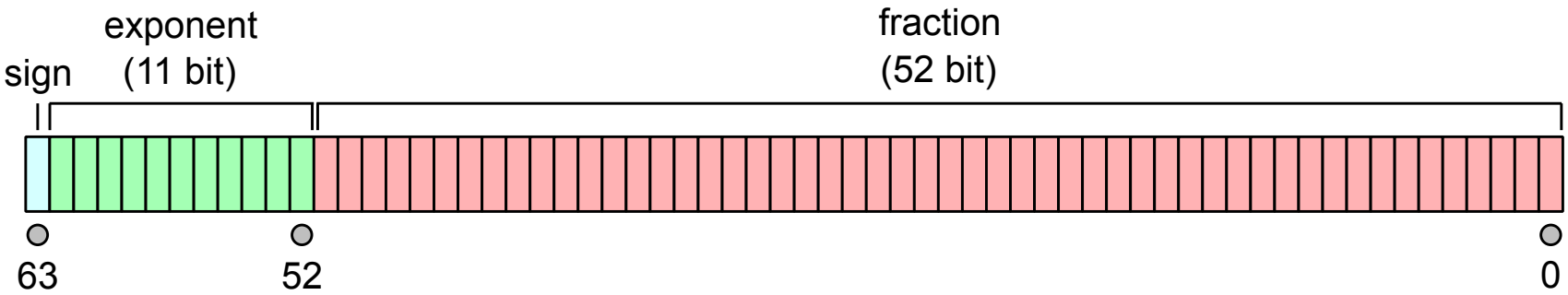
Consider computing matrix-vector product, $y = Ax$, $A \in \mathbb{R}^{n \times n}$:

- Let $x = s_1$. Suppose we can compute $\tilde{y} = A\tilde{x}$ exactly, with $\tilde{x} = s_1 + \epsilon s_n$
  - s_1 is the right singular vector corresponding to the smallest singular value, say $\sigma_1 = 1$.
- $A\tilde{x} = \sigma_1 s_1 + \sigma_n \epsilon s_n$
- If $\sigma_n \gg 1/\epsilon$, then $\|A\tilde{x} - Ax\|$ will be quite large.
- This is the same as having large $\kappa = \frac{\sigma_1}{\sigma_n}$.

Outline

64-bit (Double-Precision) IEEE754 Floating Point Numbers

1 bit **sign** (S); 11 bits **exponent** (E); 52 bits **mantissa** (M) aka significand
Stores a number: $(-1)^S \times 1.M \times 2^{E-1023}$



(a) 64-bit Double Storage Format

64-bit (Double-Precision) IEEE754 Floating Point Numbers

1 bit **sign** (S); 11 bits **exponent** (E); 52 bits **mantissa** (M) aka significand
Stores a number: $(-1)^S \times 1.M \times 2^{E-1023}$

Implications

- Non-representable numbers, finite-precision (later slide)
- FPs represent a WIDE RANGE of numbers with LIMITED PRECISION
  - Machine precision denoted by: $\epsilon_{machine}$
- Signed 0: usually obtained by $\frac{x}{\pm\infty}$ and exists so that $\frac{1}{\pm 0} = \pm\infty$
  - can be disabled by compiler (e.g., `--fno-signed-zeros`)
  - Language-dependent, but SHOULD follow $-0 == 0$
- Range (normalized): 2^{-1022} to $(2 - 2^{-52}) \times 2^{1023}$ or $\approx 2.2 \times 10^{-308}$ to $\approx 1.8 \times 10^{308}$
- Range (denormalized): 2^{-1074} to $(2 - 2^{-52}) \times 2^{1023}$ or $\approx 4.9 \times 10^{-324}$ to $\approx 1.8 \times 10^{308}$

Representable Numbers

For a fixed E, behaves like fixed-precision number. For doubles between 1 and 2, we can represent:

$$[1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, \ldots, 2]$$

and between 2 and 4:

$$[2, 2 + 2^{-51}, 2 + 2 \times 2^{-51}, \ldots, 4]$$

In general, each gap $[2^i, 2^{i+1}]$ contains the same number of numbers with gaps of size 2^{i-52}.

Denormalized Numbers

For a bitwise 0 exponent, the format changes to $0.M$

- Smallest: $2^{-52} \times 2^{-1023}$; Largest: $(1 - 2^{-52}) \times 2^{-1022}$
- Exist to reduce loss of precision when underflowing below the smallest normal number; without denormals, $2^{-1022}/2 = 0$, a total loss of precision

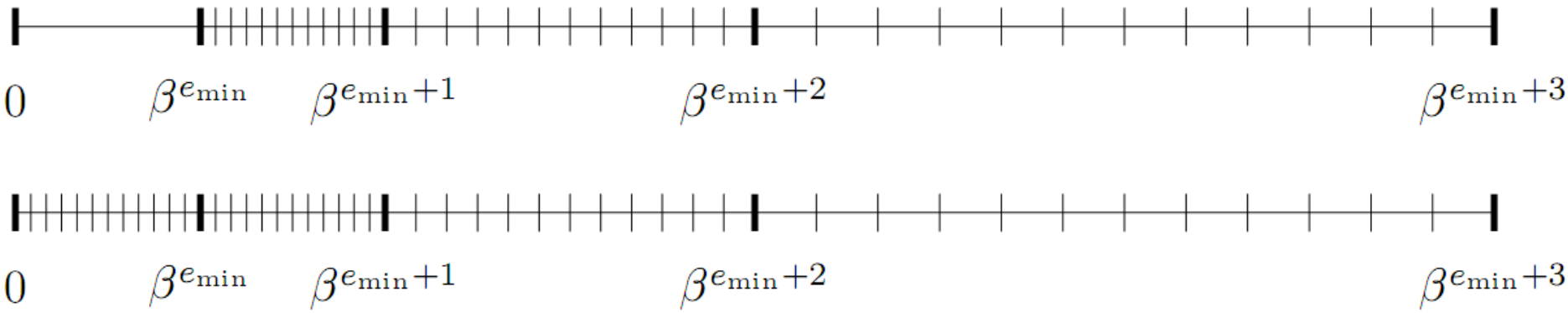


Figure : How normal (top) and denormal (bottom) numbers cover the real line. For `double`, $\beta = \frac{1}{2}$ and $\epsilon_{min} = -1022$, see previous slide.

Machine Precision: $\epsilon_{machine}$

Usually defined as the largest relative error due to round-off; for `double`:
$2^{-53} = 1.110223024625156 5e-16$

- RELATIVE! Be careful when operands substantially differ in magnitude
  - $a =$ random between [1,2]. $b = 10^8$.
  - How many digits of precision are in $(a - b) + b$?
  - What about $(b - a) + a)$?
- Some use the smallest relative gap between numbers; `double`:
  $2^{-52} = 2.220446049250313e-16$

Do not use FP loop counters: what happens here?

```
for(i=0.0; i<10^18; i+=1.0)
```

Additionally, compilers generally will not apply loop-optimizations to these, so it's almost always slower than having an unused integer counter.

$\pm$ Infinity

Exponent all 1s, Mantissa all 0s.

- Generate by: $\infty \times \infty$, nonzero$/\infty$, $\infty + \infty$
- nonzero$/\pm\infty = \pm 0$

NaN (Not a Number)

Exponent all 1s, Mantissa nonzero. qNaN (quiet) when MSB=1, sNaN (signal) when MSB=0

- Arises when limits to 0 or ∞ are multi-valued.
- Gnenerate by: $0/0$, $\infty - \infty$, ∞/∞, $\infty \times 0$
- NaN $\neq$ NaN, NaN (any op) (anything) = NaN
  - So NaNs propagate through code quickly and can be a good debugging tool to check that certain memory is not accessed.

and denormalized numbers from earlier.

All of these special cases invoke non-standard FP circuitry on the CPU, making them slow (usually taking 100-200 cycles, the same cost as accessing main memory). AVOID THEM WHENEVER POSSIBLE![†]

† Almost. Sandy Bridge's SSE unit can deal with denormals (but not inf/nan) for some arithemtic ops ($+, -$) in hardware with no penalty.

Hadware Floating Point: **x87** vs **SSE2**

- **SSE2**: 64-bit data/arithmetic, vectorized in 128-bit registers (now AVX has 256 bits)
- **x87**: 64-bit data, 80-bit intermediate arithmetic. Not vectorized. Awkward register structure (non-named, in a stack)
  - x87 and SSE2 will get different results. But x87's higher intermediate precision DOES NOT make it more accurate!

Conclusion: Use SSE2. (`icc: -fpmodel fast=1` or `-vec`; `gcc: -mfpath=sse` or `-ftree-vectorize`)

FMA: Fused Multiply Add

Twice as fast, half as much round-off error:

- FMA: $fl(x \pm y * z) = (x \pm y * z)(1 + \delta)$
- Non-fused: $fl(x \pm y * z) = (x \pm yz(1 + \delta_1))(1 + \delta_2)$

Reasoning about floating point arithmetic in a hardware-independent way requires some abstraction.
Let $\mathbf{F} \in \mathbb{R}$ be the set of FP numbers; for some $|\epsilon| \leq \epsilon_{machine}$, we desire:

- $fl(x) = x(1+\epsilon) \in \mathbf{F}$ for $x \in \mathbb{R}$
- $x \odot y = fl(x \cdot y)$ for $x, y \in \mathbf{F}$

IEEE floating point satisfies this (in particular, current CPUs), yielding:

Standard Model aka Fundamental Axiom of FP Arithmetic

$$x \odot y = (x \cdot y)(1+\epsilon), x, y \in \mathbf{F},$$

where $\cdot = +, -, \times, \div$ and $\odot$ is the computed value.

So every op of FP math is exact up to a relative error $\leq \epsilon_{machine}$, or the computed value of $x \cdot y$ is as good as the rounded exact answer.
Does not require that $\epsilon = 0$ when $x \cdot y \in \mathbf{F}$, but this is definitely true on current hardware.

Outline

Accuracy

(Absolute/Relative) Error of an approximate quantity

- **Absolute Error**: $\|x - \tilde{x}\|$
- **Relative Error**: $\frac{\|x-\tilde{x}\|}{\|x\|}$
  Since we are working with *floating point* arithmetic, relative error is almost always the right way to meausre accuracy.

  Do not check floating point computations with absolute comparisons unless you know what you are doing.

Precision

Accuracy to which basic arithmetic ($+, -, \times, \div$) is peformed.

Backward Stability

$\tilde{f}(x) = f(\tilde{x})$ for $\tilde{x}$ with $\frac{|\tilde{x}-x|}{|x|} \leq O(\epsilon_{machine})$

- Akin to asking for what Δx can we compute $\tilde{y} = f(x + \Delta x)$ exactly?
- "...gives exactly the right answer to nearly the right question." (Trefethen) or "...exact solutoin of a pertubed problem." (Higham)

This notion is absolutely critical to numerical analysis. It's a fundamental property of the problem being solved and is effectively the source of our notions of conditioning.

(Mixed) Stability

$\frac{|\tilde{f}(x)-f(\tilde{x})|}{|f(\tilde{x})|} \leq O(\epsilon_{machine})$ for $\tilde{x}$ with $\frac{|\tilde{x}-x|}{|x|} \leq O(\epsilon_{machine})$

- Put another way, $\tilde{y} + \Delta y = f(x + \Delta x)$
- "...gives nearly the right answer to nearly the right question." (Trefethen)

Backward stability *implies* (mixed) stability but not vice versa, since backward stability asserts equality in the numerator of mixed stability.

Accuracy of a Backward Stable Method

Readily shown that a backward stable algorithm achieves the following error bound:

$$\frac{\left\|\tilde{f}(x) - f(x)\right\|}{\|f(x)\|} \leq O(\kappa * \epsilon_{machine})$$

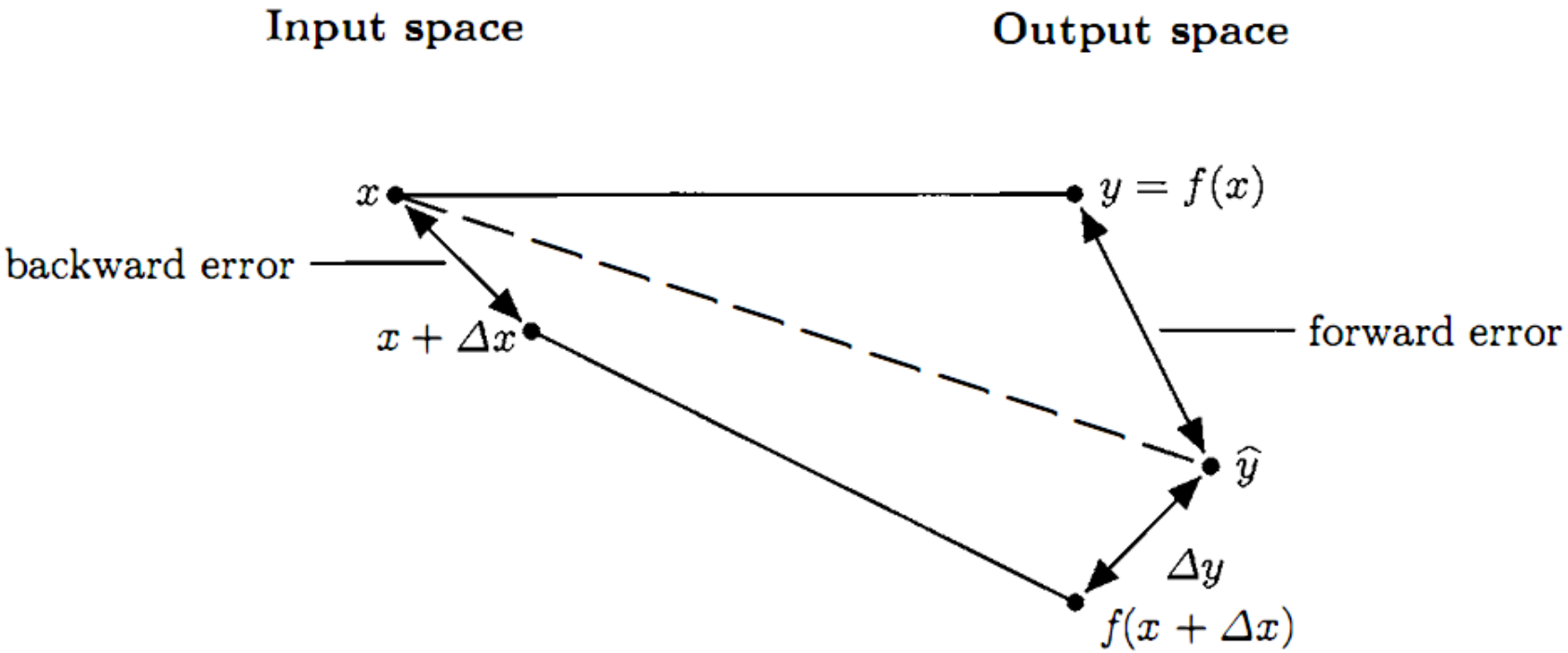
A method is **forward stable** if it produces results with errors that are similar in magnitude to results produced by a backward stable method.

Note that these bounds are normed. Termwise they directly apply to the largest outputs, BUT the smallest ones may have large (unavoidable!) error.

"In short, it is an established fact that the best algorithms for most problems do no better, in general, than to compute exact solutions for slightly perturbed data." (Trefethen)

Forward/Backward Stability Pictorally

Backward stability is typically what all numerical algorithms aspire to be. Backward error analysis is also vastly simpler than forward analysis.



Backward Stable

- $\tilde{f}(x_1, x_2) = fl(x_1) \ominus fl(x_2)$: $\tilde{f}(x_1, x_2) = (x_1(1+\epsilon_1) - x_2(1+\epsilon_2))(1+\epsilon_3) = x_1(1+\epsilon_4) - x_2(1+\epsilon_5) = f(\tilde{x_1}, \tilde{x_2})$; (BACKWARD STABLE, $\implies$ STABLE).
  - This is backward stable, but we saw earlier that $\kappa = \infty$. So the forward error is *unbounded*!
- $f(\tilde{x}) = 0$ computed as $x \ominus x$: $\tilde{f}(x) = (x(1+\epsilon_1) - x(1-\epsilon_1))(1+\epsilon_2) = 0 = f(\tilde{x})$; (BACKWARD STABLE, $\implies$ STABLE).
- $x^T y$ with $x, y \in \mathbb{R}^n$: expand termwise with error representations and collect error terms; this is backward-stable with backward error $O(n\epsilon_{machine})$ (but n is fixed!)
- Most common linear algebra ops can be computed backward stably: $x^T y$, Ax, AB, QR, Eigen-decomposition, SVD, LU, Cholesky, and more. (Some of these become very hard if matrices are rank-deficient.)

Stable but not Backward Stable

- $f(\tilde{x}) = 1$ computed as $x \oslash x$: $\tilde{f}(x) = fl(x) \oslash fl(x) = \frac{x(1+\epsilon_1)}{x(1+\epsilon_1)}(1+\epsilon_2)$, so $\frac{|\tilde{f}(x)-f(\tilde{x})|}{|f(\tilde{x})|} = \epsilon_2$ (STABLE) but $\tilde{f}(x) = 1+\epsilon_2 \neq 1 = f(\tilde{x})$
- outer product, xy^T: only pairwise multiplication, clearly stable. It is NOT backward stable: it is incredibly unlikely that $rank(\tilde{A}) = 1$, so generally $(x+\delta x)(y+\delta y)^T$ will not exist
  - If the dimension of the output space $>$ dimension of input space, backward stabliity is unlikely
- $f(\tilde{x}) = x+1$ computed as $fl(x) \oplus 1$: $\tilde{f}(x) = x(1+\epsilon_1)+1(1+\epsilon_2)$. Stability is clear, but at $x \approx 0, f(\tilde{x}) = 1 \neq 1+\epsilon_2 = \tilde{f}(x)$.
  - Backward stability is not always possible. Even as inputs go to 0, round-off errors may not.
- Elementary functions (exp, log, cos, sin, etc) are stable, but not backward stable. e.g., for sin/cos, what happens when the slope is nearly 0?
  - Again this is a fundamental property of the functions; it cannot be somehow "fixed" by better library implementations

Unstable

Consider computing the eigenvalues of A by:

- Form the characteristic polynomial $\det(A - \lambda I) = 0$
- Find the roots of the characteristic polynomial

For example, $\lambda^2 - 2\lambda + 1 = 0$: ϵ pertubations in the coefficients can move the roots by $\sqrt{\epsilon}$. Try: $A = \begin{bmatrix} 1+\epsilon & 0 \\ 0 & 1 \end{bmatrix}$

Note: in fact, the problem here is root-finding. Even quadratic root-finding is ill-conditioned (think about cancellation in $b^2 - 4ac$).

For the problem of finding repeated roots, $\kappa = \infty$.

Outline

1 bit **sign** (S); 11 bits **exponent** (E); 52 bits **mantissa** (M) aka significand
Stores a number: $(-1)^S \times 1.M \times 2^{E-1023}$
$\epsilon_{machine} = 2^{-53} =$ 1.110223024625156e-16

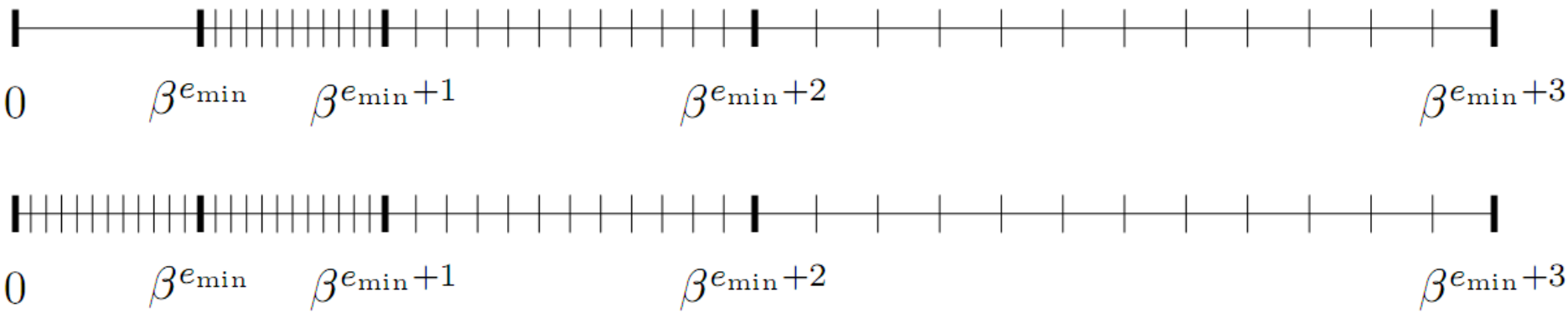


Figure : How normal (top) and denormal (bottom) numbers cover the real line. For `double`, $\beta = \frac{1}{2}$ and $\epsilon_{min} = -1022$, see previous slide.

Special numbers: denormalized, inf, NaN
FP represents a large range with *relatively limited* precision.

Definition: Relative Condition Number

$$\kappa(x) = \sup_{\delta x} \left(\frac{\|\delta f\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right) = \frac{\|J\|}{\|f\| \,/\, \|x\|},$$

if J (Jacobian) exists. (We'll use $\|\cdot\|_2$.)

In words...

- Sensitivity of the output, f, to pertubations of the data, x
- Measures ratio of relative changes in $f(x)$ to relative changes in x
- Large κ means (certain) changes in x can lead to much larger changes in f

Notes:

- Large κ is "ill-conditioned"; small κ is "well-conditioned"
- Matrix condition number: $\kappa(A) \equiv \|A\|_2 \left\|A^{-1}\right\|_2$

Backward Stability

$\tilde{f}(x) = f(\tilde{x})$ for $\tilde{x}$ with $\frac{|\tilde{x}-x|}{|x|} \leq O(\epsilon_{machine})$

- Akin to asking for what Δx can we compute $\tilde{y} = f(x + \Delta x)$ exactly?
- "...gives exactly the right answer to nearly the right question." (Trefethen) or "...exact solutoin of a pertubed problem." (Higham)

This notion is absolutely critical to numerical analysis. It's a fundamental property of the problem being solved and is effectively the source of our notions of conditioning.

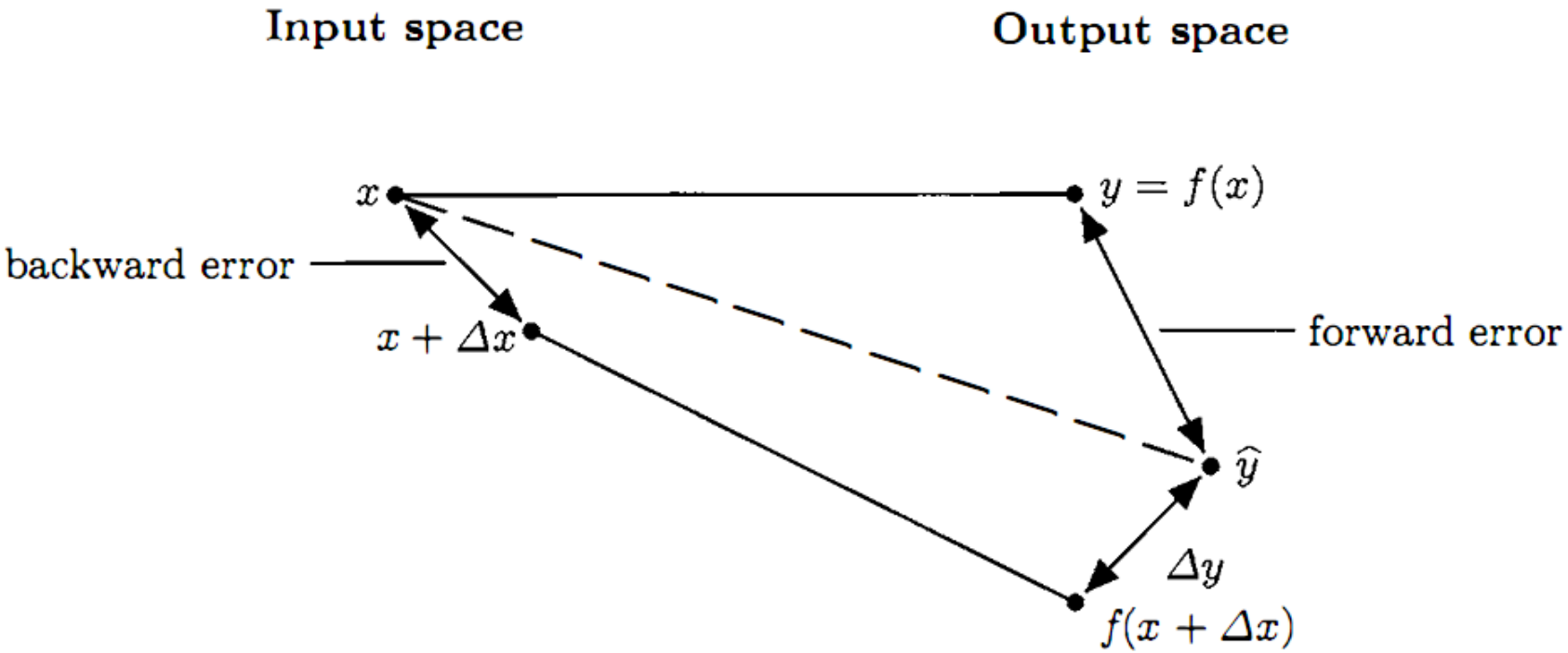
Accuracy of a Backward Stable Method

Readily shown that a backward stable algorithm achieves the following error bound:

$$\frac{\left\|\tilde{f}(x) - f(x)\right\|}{\|f(x)\|} \leq O(\kappa * \epsilon_{machine})$$

Forward/Backward Stability Pictorally

Backward stability is typically what all numerical algorithms aspire to be. Backward error analysis is also vastly simpler than forward analysis.



Outline

This is deceptively simple: even Wikipedia's discussion is very imprecise.
Most basic: `for i, s += x[i]`

- Result varies w/the order of summation (e.g., with $|x_1|, \ldots, |x_n|$ increasing/decreasing)
- a priori bound: $(n-1)\epsilon_{machine} \sum_{i=1}^{n} |x_i| + O(\epsilon_{machine}^2)$

Best order-independent solution: pairwise/recursive summation

- e.g., $s = ((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8))$
- (rough) a priori bound: $\log_2(n)\epsilon_{machine} \sum_{i=1}^{n} |x_i|$

Another interesting choice: insertion

- Sort x. Pop and sum x_1, x_2, then reinsert sum into sorted x
- a priori error bound: $(n-1)\epsilon_{machine} \sum_{i=1}^{n} |x_i| + O(\epsilon_{machine}^2)$

Ordering can be very tricky: to maximize accuracy, we want to minimize the absolute value of *all intermediate sums*. This is NP-hard.

Insertion is the optimal choice for summation when all entries are the same sign. (With the same sign, using increasing order is the best case for the recursive method.)
Summing by increasing size is extremely useful for evaluating infinite series (e.g., $\sum_{i=1}^{\infty} \frac{1}{i^2} = \pi^2/6$)

- Starting at $i = 1$ is only accurate to about 8 digits
- Going in reverse (where do you start?) yields about 15 digits

But it is not the best choice when signs differ. Consider $x = [1, M, 2M, -3M]$ with $fl(1 + M) = M$.

- increasing: $1 + M + 2M - 3M = 0$
- decreasing: $-3M + 2M + M + 1 = 1$

Hence decreasing is preferable when there is a lot of cancellation.

How would you solve a linear system of equations?

How would you solve $Ax = b$ for x?

- Cramer's Rule?
- $A^{-1} * b$?
- Guassian Elimination via Row Reduced Echelon Form?

Let's see if any of these are good ideas...
But first, our evaluation tool, the residual:

$$r(x) = \|b - Ax\|_2$$

This is not scale-invariant (scale b,A by α and r scales):

$$\rho(x) = \frac{\|b - Ax\|_2}{\|A\|_2 \|b\|_2}$$

It is not hard to show that $\rho = \min(\frac{\|\delta A\|_2}{\|A\|_2})$, where $(A + \Delta A)x = b$, so $\rho(x)$ *directly measures* backward error.

Let's try it out on the following ill-conditioned problem:

```
>> A = gallery('frank',14);
>> xreal=rand(14,1); b = A*xreal;
>> cond(A) % A is ill-conditioned
ans = 8.681498748868447e+11
```

method	$\rho(x) = \|b - Ax\| \,/\, \|A\| \,/\, \|b\|$	error ($\|x - x_{exact}\|_2$)
$A\backslash b$	1.652780611837637e-18	4.208888890411956e-05
SVD	3.585462648735363e-17	2.316190122807023e-04
$A^{-1}b$	9.263429874057655e-09	1.163491208778896e-04
Cramer	1.374663894689294e-08	5.540073944180914e-05

Residual, $\|b - Ax\|$, differences are huge!

Notes: **1)** $A\backslash b$ refers to: $PLU = A$, $x = U\backslash(L\backslash(Pb))$, with $\backslash$ overloaded as "backsolve" or "backward substitution"; **2)** SVD refers to $U\Sigma V^T = A$, $x = V(\Sigma^+(U^T b))$; **3)** These expressions just convey ideas, not efficient evaluation strategies.

But the errors aren't that different (`4.0e-5` to `2.0e-4`). Why?

Who cares? With $\kappa \approx 10^{11}$, we *cannot* expect anything more accurate than this.
BUT with PLU, Ax closely approximates b, which we know is the best we can hope for. There are lots of vectors y such that $\|x_{real} - y\| \approx 10^{-4}$, but for almost all of them, $Ay \neq b$.

More notes:

- SVD is the singular value decomposition: $U\Sigma V^H = A$, where U, V are unitary (left, right singular vectors) and Σ is diagonal.
- PLU is the row-pivoted LU decomposition: $LU = PA$. L is lower triangular, U is upper, and P is a permutation matrix (it just reorders rows of A or entries of b and is never formed explicitly).
- Cramer's Rule is $x_i = det(C_i)/det(A)$ where C_i is A with its i-th row replaced by b. This is unstable for $n > 2$. DO NOT USE CRAMER'S RULE.

Why did the residuals differ by so much?

- A^{-1} can *only* be computed with backward error $O(\kappa_A \epsilon_{machine})$
- Even if we could compute A^{-1} exactly, $A^{-1}x$ has $\|b - Ax\| \leq \kappa \|b\| \epsilon_{machine}$
  - b/c conditioning of Ax (matrix-mult) is κ_A!
- Whereas with PLU, $\|b - Ax\| \leq \|A\| \|x\| \epsilon_{machine}$
- This comparison is never unfavorable for PLU.

And for a single RHS, the PLU approach is *3 times* faster!
PLU is backward stable... sort of. (Cholesky is backward stable unconditionally.)

- We get $A + \Delta A = LU$ with $\frac{\|\Delta A\|}{\|A\|} \leq \rho \epsilon_{machine}$, where $\rho = O(2^n)$. Since n is fixed, ρ (growth factor) is constant and this is technically backward stable... but the bound is useless if ρ is near 2^n.
- However, the class of matrices with large ρ are pathologically rare. Some matrices have $\rho = 1$ (diagonally dominant, unitary, etc.)

Consider the basic LU (Gaussian Elimination) you learned in school:

$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$

Now pertub 0 by 10^{-20}. The perturbed matrix is still very well conditioned.

Factoring it, $L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{-20} \end{bmatrix}$.

With `double`, $1 - 10^{-20} = 1$, which means: $LU = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix}$,

a MASSIVE failure.

This is resolved by a process called **partial pivoting**.

Complete pivoting can solve the previous growth factor (ρ) issue, but it is very slow. And those bad cases are so rare that people only use complete pivoting when necessary.

The conditioning of polynomial root-finding is *atrocious*.

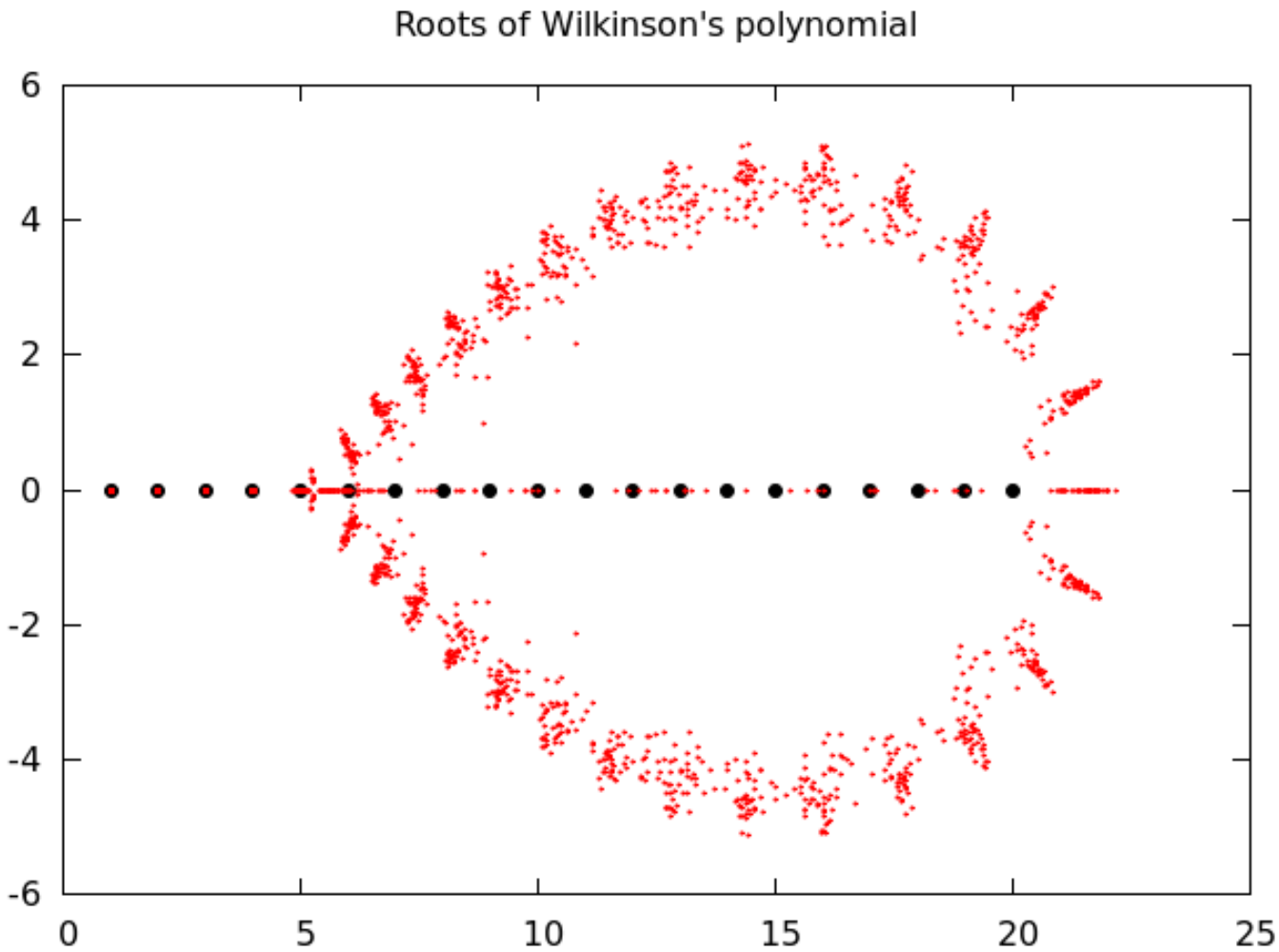
Wilkinson first studied this with his famous polynomial:

$$p(x) = \Pi_{i=1}^{20}(x - i) = a_0 + a_1 x + \ldots a_{19} x^{19} + a_{20} x^{20}$$

The most sensitive root is $x = 15$. $a_{15} \approx 1.67 \times 10^9$ and $\kappa \approx 5.1 \times 10^{13}$ for this root. The issue is the sensitivity of polynomial roots to pertubations in that poly's coefficients.

Recall from earlier that this gets vastly worse if roots are repeated, in which case $\kappa = \infty$.

Random pertubations of the coefficients of Wilkinson's polynomial in the 10th decimal lead to the following distribution of roots:



The simplest way to find eigenvectors/values is to perform **Power Iteration**:

- power (up): $x = Ax$
- normalize: $x = x/\|x\|$

x converges to s_1 (and $\left\|x^T Ax\right\|$ to λ_1) as long as $x^T s_1 \neq 0$ for the intial x (random). This is **Power Iteration**.

To compute an eigenvalue near μ (instead of the largest), we do **Inverse Iteration**:

- solve: $(A - \mu I)x = x$
- normalize: $x = x/\|x\|$

Which converges to the eigenvalue nearest μ (recall A's smallest eigevalues are A^{-1}'s largest.

And if we replace μ by the eigenvalue estimate at each step, $x^T Ax$, we get **Rayleigh Iteration**.

BUT what happens as $\mu \to \lambda$? We know that $(A - \lambda I)$ is singular! Is this a problem?

Actually, no. It can be shown that the error from solving this horribly-conditioned system lies in the direction of the desired eigenvector. **Rounding errors actually save us.**

The **QR Algorithm** (for eigenvalues, based on this) is one of the crowning achievements of numerical linear algebra and is only possible because of rounding.

Given a set of (linearly independent) vectors, produce an orthogonal basis.
Equivalently, factor $A = QR$ where Q is unitary and R is upper triangular.

Classical Gram-Schmidt

for $j = 1$ **to** n

$v_j = a_j$

for $i = 1$ **to** $j - 1$

$r_{ij} = q_i^* a_j$

$v_j = v_j - r_{ij} q_i$

$r_{jj} = \|v_j\|_2$

$q_j = v_j / r_{jj}$

Modified Gram-Schmidt

for $i = 1$ **to** n

$v_i = a_i$

for $i = 1$ **to** n

$r_{ii} = \|v_i\|$

$q_i = v_i / r_{ii}$

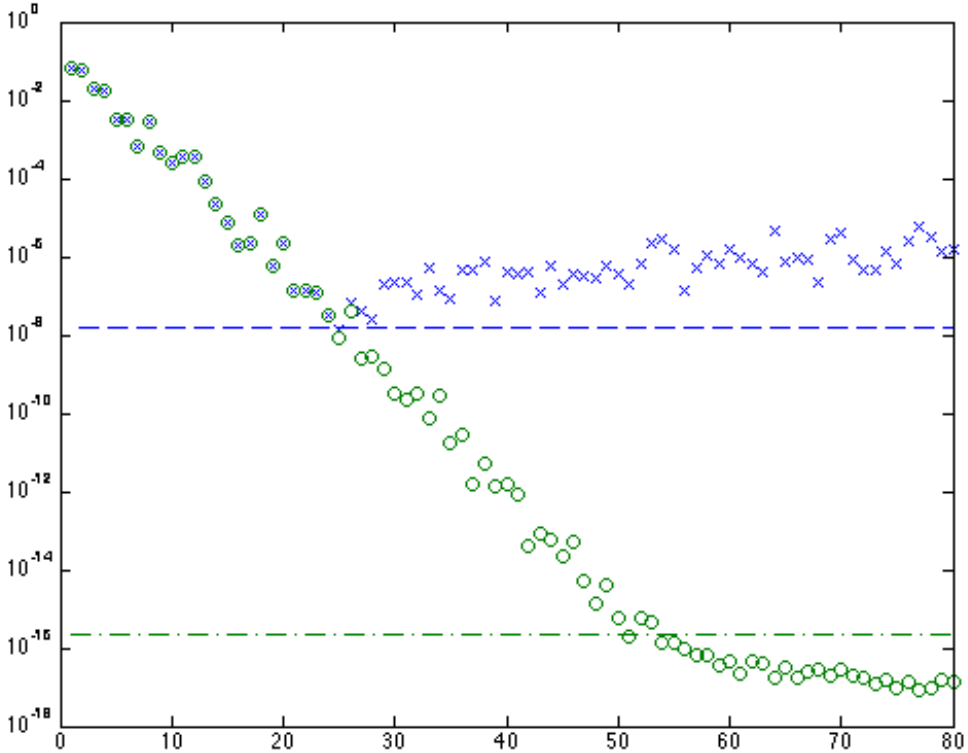
for $j = i + 1$ **to** n

$r_{ij} = q_i^* v_j$

$v_j = v_j - r_{ij} q_i$

CGS vs MGS for a matrix with exponentially graded singular values. r_{jj} is plotted vs j; the values should roughly fall on the line 2^{-j}.

- CGS fails at roughly $\sqrt{\epsilon_m}$
- MGS fails at roughly ϵ_m



- Blue = CGS, Green = MGS
- In general, MGS is still not the most accurate choice: that honor falls on **Householder Reflections**.

So what happened? CGS and MGS are *mathematically* equivalent.

- CGS suffers severely from cancellation errors. At each step, we subtract vectors from other vectors. If these were such that $x_1^T x_2 \approx 1$, CGS is mostly acting on noise.
- As a result, CGS can lose orthogonality: the final vectors produced will not be orthogonal to the initial vectors. CGS makes no effort to guarantee this, instead relying on a mathematical property which need not hold in finite precision.
- MGS updates each vector *once per step* instead of having all of its updates done on a single step (a la CGS). (So after step i, MGS finishes w/i-th input and CLGS finishes w/i-th output.)
- In CGS, the updates are $r_{ij} = q_i^H * a_j$, which involves one of the original vectors in A; MGS replaces a_j with the partially orthogonalized vector v_j

But CGS still gets some love in real life because it is trivially parallelizable (whereas MGS is quite hard to parallelize).

Now let us look at a common use of the QR factorization: Least Squares problems:

$$\arg\min_x \|b - Ax\|_2$$

We'll use the (ill-conditioned) Vandermonde matrix as our example:

$A = \begin{bmatrix} 1 & \alpha_1 & \cdots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \cdots & \alpha_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \cdots & \alpha_m^{n-1} \end{bmatrix}$, which is a function of $[\alpha_1, \ldots, \alpha_n]$ and $A \in \mathbb{R}^{m \times n}$.

Multiplying against a coefficient vector $v \in \mathbb{R}^n$ is the same as evaluating the polynomial $1 + v_1\alpha + \ldots + v_n\alpha^{n-1}$ at m points.

- Hence the Vandermonde matrix can be used for computing best-fit polynomials.
- This matrix is also very ill-conditioned due to monomial basis being terrible.

```
>> m=50; n=15;
>> p = linspace(0,1,m);
>> A = vander(p);
>> A = fliplr(A);
>> A = A(:,1:n);
>> xexact=linspace(1,n,n)'; % to get residual 0
>> b=A*xexact;
>> cond(A)
ans = 2.327103439491880e+10
```

A is full-rank but ill-conditioned

Full Rank Least Squares Problems

method	$\rho(x)$	error, $\|x - x_{exact}\|_2$
$A\backslash b$	4.408088754699393e-14	3.035230539102043e-06
SVD	8.041172999670976e-14	1.694428274018899e-05
A^+	2.906305885792272e-05	2.414822053248719e-05
Normal Eqn	7.431101364089831e-07	2.427883946195195e+02
NE with inv	2.350450537581168e+02	2.277376161039734e+02
CLGS	3.991378407562542e-02	1.103716447425526e+05
MGS	3.069366439247523e-05	7.286631128416664e+04
MGS Aug	4.306406135702305e-14	5.962763963100584e-06

Notes: **1)** $A\backslash b$ here refers to (Householder) QR factorization, $QR = A$ where Q is unitary and R is upper triangular; **2)** SVD has the same meaning as it did with linear systems: $V(\Sigma^+(U^T b))$; **3)** A^+, the pseudoinverse (next section) is formed explicitly using the SVD; SVD and A^+ are analagous to PLU and A^{-1} here; **4)** NE with inv means $(A^T A)^{-1}$ was formed explicitly; **5)** MGS Augmented refers to `Q,R]=mgs([A,b])` with `Qb=R(:,n+1)`, `R=R(1:n,1:n)`, and $x = R\backslash Qb$

In general, for **full-rank** least-squares problems: (in order of computational expense)

- Classical GS: unstable
- Modified GS: backward stable, as long as Q is not formed explicitly (can still compute Qx for any vector(s) x though by appending)
- Householder: backward stable even if Q is formed explicitly (Householder works with all orthogonal transformations)
- Normal Equations: unstable; never, ever use this
- SVD: backward stable (in fact SVD pretty much never fails)

SVD is the only totally safe option for rank-deficient problems.
Column-pivoted Householder works most of the time but is not guaranteed.

Note: MGS does not produce the Q matrix in a backward stable fashion. The orthogonality of Q is not guaranteed explicitly; instead both get it implicitly (and unreliably). 5) on prev page, Augmented MGS, fixes this.

aka pseudoinverse, denoted A^+
(reverse) motivation: $0^{-1} = \infty$ is not useful; so let us define $x^+ = x^{-1}$ if $x \neq 0$ and $= 0$ if $x = 0$.

Pseudoinverse is defined with the following properties:

- $AA^+A = A$
- $A^+AA^+ = A^+$
- $(AA^+)^H = AA^+$
- $(A^+A)^H = A^+A$

Full-rank special cases:

- A has full column rank: $A^+ = (A^HA)^{-1}A^H$
- A has full row rank: $A^+ = A^H(AA^H)^{-1}$

Only backward-stable method for computing this is via the SVD ($U\Sigma V^H$, pseudoinvert Σ using the scalar definition above).
Can form using QR factorization, but this has all the problems of the normal equations.
As with A^{-1}, DO NOT FORM EXPLICITLY whenver possible

Here A is 50×30 with only 15 linearly independent columns. Numerically, $\kappa_A = 9.880135812164258e+32$

method	$\rho(x)$	solution norm, $\|x\|_2$
$A\backslash b$	4.724975630252374e-13	1.722180019700298e+02
SVD	1.491970558662751e-12	4.086118112800787e+02
A^+	9.057444862168540e-05	9.486410693511561e+01
Normal Eqn	1.049806584713839e-05	3.218559175502463e+03
NE with inv	3.793775325623811e+04	6.744588645721843e+03
CLGS	8.301462000278773e-01	1.773019946649426e+06
MGS	3.536670106234968e+19	8.937641920562169e+34
MGS Aug	4.239895946596119e+03	1.026766798876118e+19
House QR no piv	1.616290736371978e+03	4.253520765033544e+18

Notes: **1)** the last column says $\|x\|_2$ and not $\|x - x_{exact}\|_2$ on purpose. For rank deficient problems, if $y \in null(A)$, then $\|Ay\| = 0$ hence $\|A(x+y)\| = \|Ax\|$; **2)** $A\backslash b$ refers to (Householder) QR with column-pivoting, this method yields x with $rank(A)$ zeros; **3)** SVD is as before, it yields solution such that $\|x\|_2$ is minimal b/c this soln is perpindicular to $null(A)$.

Expanding on the rank-deficient notes from the previous page... now we have $B \in \mathbb{R}^{50\times 30}$ where the second 15 columns are linearly dependent on the first 15.

Caveats:

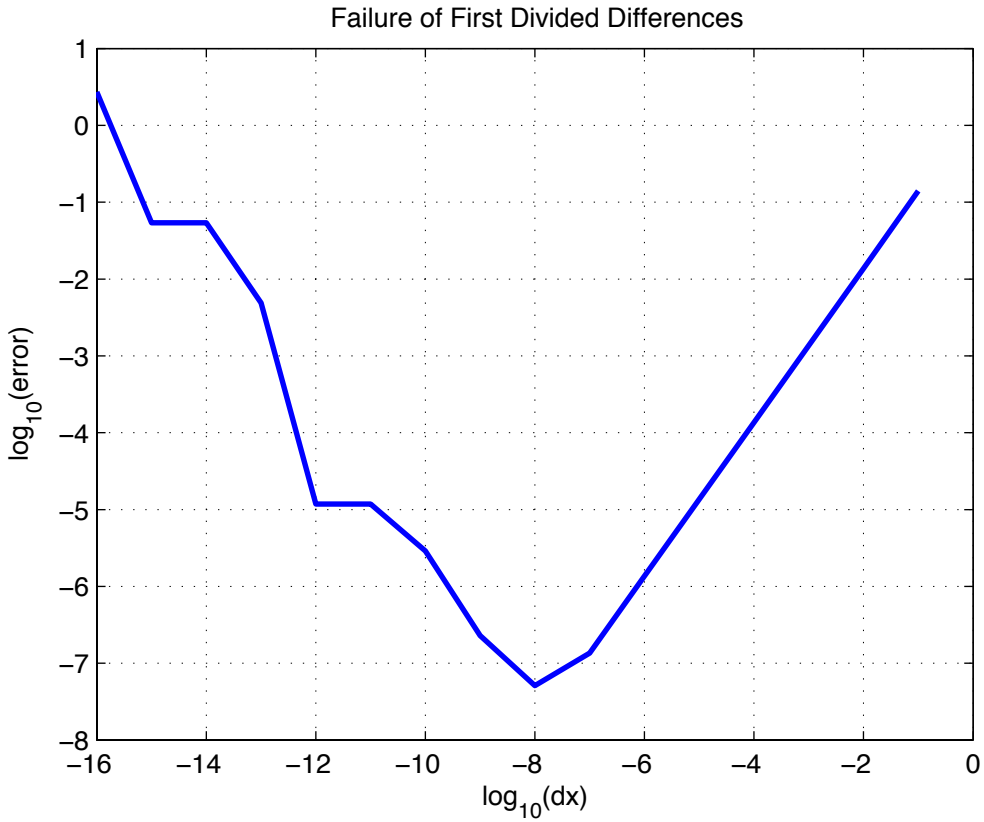
- Need to know what "0" means for any method to work. This is not obvious since it is unclear what ϵ is from round-off and what is from real data.
  - LAPACK (and thus MATLAB, num/sciPy) uses $\epsilon_{machine} \max(m,n) \max_i(\sigma_i)$ by default
- RRQR (Rank-Revealing QR, i.e., column pivoting) can fail, but it is rare.
  - I have no personal experience with this, but LAPACK (and again things based on it) have this as a common (or default) choice
- SVD is the safest route, but it is also (by far) the most expensive. Costs are similar if $m \gg n$

Derivatives with Finite Differences

Intro calculus taught us to compute $\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$ (first order finite differences).

On a computer, what value of h should be used for maximum accuracy to compute $\frac{\partial e^x}{\partial x}\Big|_{x=1} = e$?

On a computer, what value of h should be used for maximum accuracy to compute $\frac{\partial e^x}{\partial x}\Big|_{x=1} = e$?



(a) Error computing $\frac{\partial e^x}{\partial x}\Big|_{x=1}$ with first order finite differences

$h = \sqrt{\epsilon_{machine}}$ is about the limit

You might have thought to use $h = \epsilon_{machine}$ since we cannot pick a smaller value (or $x + h = x$), and you might think smaller $\implies$ more accuracy.

- Finite difference computations are notoriously hard to use.
- Fighting 2 opposed forces: truncation error (smaller h better) and cancellation from subtraction (larger h better)
- Solutions:
  - Use higher order FD formulas; e.g., $\frac{f(x+h)-f(x-h)}{2h}$ is second order and tolerates $h \approx 10^{-10}$. Even more accurate formulas exist at the cost of more function evaluations. (Extrapolation fits under this category.)
  - Find transformations that lead to better numerical results (e.g., substitute $y = e^x$)
  - Use complex-variables: $f'(x) \approx f(x + ih)/h$, where $i = \sqrt{-1}$. Here, $h = \epsilon_{machine}$ is safe.

Outline

Tips for Algorithm Design (Higham)

- Avoid subtracting quantities contaminated by error. (results in dangerous cancellation; see finite differences ex)
- Minimize the size of intermediate quantities. (Huge intermediates swamp your data with noise; see $(a - b) + b$ ex)
- Find other approaches that are mathematically but not numerically equivalent. (e.g., CGS vs MGS vs Householder vs SVD for $\min \|b - Ax\|_2$)
- Use well-conditioned transformations. (e.g., work with orthogonal matrices, see Householder or SVD)
- Avoid under/overflow

And one from me: Don't re-implement naive versions of standard methods (e.g., LU w/o pivoting) unless you know what you're doing.

Myths (Higham)

- Cancellation is always bad
- Rounding errors only matter when you're doing a lot of computations (e.g., computing $e = \lim_{n\to\infty}(1 + 1/n)^n$)
- Short computations, free from cancellation and under/overflow are acccurate (LU factorization w/o pivoting, adding numbers)
  - corollary: well-conditioned matrix $\implies$ anything works
- Increasing precision $\implies$ improved accuracy (need to understand where your sources of error are)
- The final result is only as accurate as any intermediate quantity (correlated errors can *cancel*)
- Rounding errors are never helpful to a computation (inverse iteration for eigenvectors)

References

My favorite books on this topic:

- L. N. Trefethen and D. Bau. Numerical Linear Algebra. SIAM. 1997. Philadelphia, PA.
  - Extremely well-written and easy to read (for a math book). Focuses on concepts/flavor, skimming over ultra-fine details.
- N. J. Higham. Accuracy and Stability of Numerical Algorithms. 2nd Ed. SIAM. 2002.
  - Very dense and extremely detailed analysis of how round-off affects... things. Look here when Trefethen doesn't scratch your itch for stability/conditioning.
- J. W. Demmel. Applied Numerical Linear Algebra. SIAM. 1997.
  - Very dense on all the nooks and crannies of numerical linear algebra, from theory to high performance implementations.
- G. H. Golub and C. F. Van Loan. Matrix Computations. 3rd Ed. JHU Press. 1996.
  - THE definitive reference on numerical linear algebra.

Questions?