Welcome to Ma221!

Finish Lecture 1: 4 axes of NLA

1) math problem
2) structure
3) desired accuracy

tradeoff: more accurate ⇒ slower

1) "Guaranteed accuracy": For $Ax=b$, need proof that A is (non)singular
⇒ need arbitrary precision arithmetic
very expensive, try Mathematica

2) "Backward stability":
informally: "get exact answer for a slightly wrong problem

computed answer = $alg(x) = f(x+\delta)$
where δ "small" compared to x

Def of small requires matrix norms.
Size of δ should be proportional to error in arithmetic, eg 10^{-16} in double precision, then $alg(x)$ "as good" as true solution, because computing x changes x to $x+\delta$

3) Residual as small as desired:
for problems too big for a backward stable alg, iterates, progressively makes residual smaller
Ex: $\|Ax-b\|$ for solving $Ax=b$.
size of $\|Ax-b\|$ yields backward error
Lots of algorithms: Chaps 6+7

4) "Probably OK" → "randomized Linear Algebra": replace a large problem cheaply with a smaller one, by "random sampling", solve small approximation with a deterministic method. Some algs iterate, with residual, some don't:

Thm:
"the error is less than ε with probability $1-\delta$ if size of random approximation is big enough i.e. proportional to $f(\delta,\varepsilon)$ that gets larger as δ and ε get smaller
Often $f(\delta,\varepsilon) = \Omega\left(\log\left(\frac{1}{\delta}\right)\cdot\frac{1}{\varepsilon^2}\right)$"

$\Rightarrow$ if your ε too small, use deterministic alg.

Some users want more info about reliability

1) Error bounds: note that if A singular, or "close", a backward stable algorithm for $Ax=b$, may be completely wrong (eg deciding if A is (non)singular, when opposite true)

We will derive error bound that is proportional to

$$\text{"condition number"} = \frac{1}{\text{distance from } A \text{ to nearest singular matrix}}$$

efficient to compute

2) "Guaranteed correct" except in "rare cases": combine error bounds with iteration (few steps of Newton) to improve answer, until error bound small enough. Reasonable cost for $Ax=b$, LS

Popular again because of 16-bit

floating point accelerators for matrix multiply (Google's TPU)

3) One more kind of "accuracy" getting bit-wise identical results when running program again, for debugging. Not guaranteed on modern architectures: floating point sum may be done in different order, in parallel, floating point addition not associative. (new algorithms available)

Axis 4) How to implement an "efficient" algorithm

Example from axis 2 assumed counting flops most important

(1) Fewest keystrokes: eg "A\b" to solve $Ax=b$. Metric: minimizing human effort. We will give pointers to available software, lots on class webpage. (netlib, GAMS)

A\b depends on LAPACK local project, lots of class projects

(2) What does "fewest flops" mean?

How many flops to do $n \times n$ matmul

Classical: $2n^3$

Strassen (1969): $O(n^{\log_2 7}) = O(n^{2.81})$

arbitrarily faster for large n,

large hidden constant in $O(\ )$

Coppersmith/Winograd (1987): $O(n^{2.376})$

not practical, hidden constant huge

Umans/Cohen: $O(n^{2.376})$ maybe $O(n^{2+\epsilon})$

reduced search for algorithm to group theory

Williams (2013) $O(n^{2.3728642})$

Le Gall (2014) $O(n^{2.3728639})$

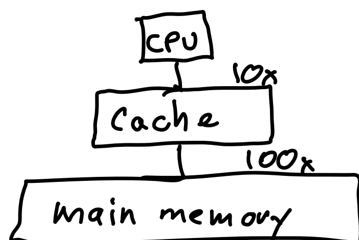
Alman-Williams (2020) $O(n^{2.3728596})$

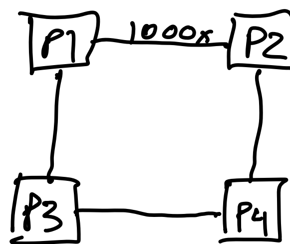
D., Dumitriu, Holtz (2008) all other standard problems ($Ax=b$, LS, eigen) can be solved in $O(n^x)$ given any matrix multiply running in $O(n^x)$ and backward stable; some practical

(3) But counting flops not only important metric in today's and future computers

(3.1) Recall Moore's Law: observation that # transistors on chip would double every 2 years. So until 2004, computers kept doubling in speed without code changes. This has ended, need to use parallel algorithms. Some parallel algs are mathematically same as sequential algs, some different. Will discuss some "different" ones (see CS 267 for more)

(3.2) What is most expensive operation a computer performs? Not arithmetic, moving data





Ex: "naive" matmul (3 nested loops)
can be 100x slower than optimized
matmul doing same arithmetic,
moving less data

Lots of recent algorithms that
minimize data movement (hit
lowerbounds), some do same arithmetic,
some different mathematically.

Lots of links on webpage, possible projects

4) So far we have minimized time.
Energy also important! Moore's Law
ended, because chips would have melted
if built same way. Reasons to care
about energy:

- battery in cell phone dying
- $1M / Megawatt/year to run data center
- how long your drone can fly

Turns out minimizing data movement also
minimizes energy

Summary of course syllabus

$Ax=b$, LS, eigenproblems, SVD, variations

Direct Methods

(aka matrix factorizations:
LU, Cholesky, QR, Schur form, etc)

LU: $A = P \cdot L \cdot U$
P permutation, L lower triangular
U upper triangular

Cholesky: $A = LL^T$, A sym pos def

QR: $A = Q \cdot R$ Q orthogonal
R upper triangular

Eigen Problems: not Jordan Form

Schur Form: $A = Q R Q^T$
Q orthogonal
R "upper triangular" with
eigenvalues on diagonal
(more complicated with complex
evals)

SVD $A = U \Sigma V^T$, U, V orthogonal
Σ diagonal

Iterative Methods: eg
Jacobi, Gauss-Seidel, Conjugate Gradients,
Multigrid, more
Common idea: do many matrix-vector
multiplies, find "best" linear
combination of resulting vectors
to approximate answer

Randomized Algs:
Approximate A by QA or
QAV^T where Q and V are "skinny"
sometimes orthogonal, sparse,
always random, solve smaller problem
with QA or QAV^T

+ structure
+ error bounds
+ algorithms for modern computers

Lecture 2:

Goals: Floating Point Arithmetic
Roundoff error analysis
Beyond basics
exceptions
high/low/variable precisions
reproducible
interval arithmetic

Ex: Polynomial evaluation and zero finding

Recall bisection:
want to solve $f(x)=0$
start with interval $[x_1, x_2]$ where $f(x_1)\cdot f(x_2)<0$
Bisection: evaluate $f\left(\frac{x_1+x_2}{2}\right)$
keep bisecting subinterval where f changes signs until interval small enough

Try: $(x-2)(x-3)(x-4)$
$= x^3 - 9x^2 + 26x - 24$

(matlab demo, code in typed notes)