MATH 221 Lecture Notes, Fall 2020

Advanced Matrix Computations

Professor: James Demmel
Fengzhe Shi
Ed. Grace Zdeblick

Contents

1 Lecture 1: Course Outline

1.1 Notation

Definition 1.1. $\|x\|_2 = \sqrt{\sum_i |x_i|^2}$ is 2-norm of x.
Definition 1.2. $\arg\min_x f(x)$ is the value of argument x that minimizes $f(x)$.
Definition 1.3. Asymptotic Notation:

- $f(n) = O(g(n))$ means that $|f(n)| \leq C|g(n)|$ for some constant $C > 0$ and n large enough.
- $f(n) = \Omega(g(n))$ means that $|f(n)| \geq C|g(n)|$ for some constant $C > 0$ and n large enough.
- $f(n) = \Theta(g(n))$ means that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

1.2 Introduction

To motivate the syllabus, we describe the "axes" of the design space of linear algebra algorithms:

1. The mathematical problems:
   - solve $Ax = b$;
   - least squares: $\arg\min_x \|Ax - b\|_2$;
   - eigenproblems: $Ax = \lambda x$;

   And many generalizations.
2. The structure of A: dense, symmetric, positive definite, sparse, structured[1].
3. The desired accuracy, a spectrum ranging over:
   - guaranteed correct: this is generally too expensive and won't be discussed
   - "guaranteed correct" except in "rare cases": e.g. iterative refinement for $Ax = b$ using Newton's method.
   - backward stable: exactly correct answer for a slightly wrong problem.
   - residual as small as desired
   - "probably ok" for randomized algorithms
   - error bounds using condition numbers: how sensitive is the problem to small changes?
4. As fast as possible on target architecture: your laptop (which may have a GPU), big parallel computer, cloud, cell-phone, ...

Each "problem" has a choice from each axis.

[1]E.g., Toeplitz: $A_{ij} = x_{i-j}$ (constant along diagonals).

1.3 The Mathematical Problems

1.3.1 Solve Ax=b

This is well-defined if A is square and invertible, but if it isn't (or A is close to a matrix that isn't), then least squares may be a better formulation.

1.3.2 Least Squares

- "What is the least I could change b to have a solution to $Ax = b$?"
- Overdetermined: $\arg\min_x \|Ax - b\|_2$ when A is $m \times n$ has full column rank; this mean $m \geq n$ and the solution is unique.
- Not full column rank (e.g. $m < n$): x not unique, so can pick x that minimizes $||x||_2$ too, to make it unique.
- Ridge regression for low rank matrices (Tikhonov regularization): $\arg\min_x \|Ax - b\|_2^2 + \lambda \|x\|_2^2$. This guarantees a unique solution when $\lambda > 0$.
- Constrained: $\arg\min_{x:\ \{Cx=d\}} \|Ax - b\|_2$, e.g. x may represent fractions of a population, so we require $\sum_i x_i = 1$ (additionally constraining $x_i \geq 0$ seems natural too, but this is a harder problem).
- Weighted: $\arg\min_x \left\|W^{-1}(Ax - b)\right\|_2$ where W, the weight matrix, has full column rank (also called Gauss-Markov linear model).
- Total least squares: $\arg\min_{x:\ (A+E)x=b+r} \|[E, r]\|_2$, useful when there is uncertainty in A as well as b[2].

1.3.3 Eigenproblems

Notation: If an $n \times n$ square matrix A has $Ax_i = \lambda_i x_i$ for $i = 1 : n$, write

$$X = [x_1, \cdots, x_n]$$

and

$$\Lambda = \text{diag}(\lambda_1, \cdots, \lambda_n);$$

then $AX = X\Lambda$. Assuming X is invertible, we write the eigendecomposition of A as

$$A = X\Lambda X^{-1}.$$

Recall that A may not have n independent eigenvectors, e.g. $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$. If we consider instead $A' = \begin{bmatrix} \epsilon & 1 \\ 0 & 0 \end{bmatrix}$ we see that the Jordan form is $\begin{bmatrix} 0 & 0 \\ 0 & \epsilon \end{bmatrix}$, which is discontinuous with the original problem. We will introduce something cheaper and more numerically stable later than Jordan form, called Schur form.

[2]where $[E, r]$ represents the concatenation of E and r into a larger matrix

1.3.3.1 SVD

$A = U\Sigma V^\top$ where A is $m \times n$, U is $m \times m$ and orthogonal[3], V is $n \times n$ and orthogonal, Σ is $m \times n$ diagonal, with the diagonal entries σ_i, called singular values, in sorted order. The columns of U and V are called left and right singular vectors respectively. Note that

$$\begin{aligned} AA^\top &= U\Sigma V^\top (U\Sigma V^\top)^\top \\ &= U\Sigma V^\top V \Sigma^\top U^\top \\ &= U(\Sigma\Sigma^\top)U^\top \\ &= \text{eigendecomposition of } AA^\top \end{aligned}$$

$$\begin{aligned} A^\top A &= V(\Sigma^\top \Sigma)V^\top \\ &= \text{eigendecomposition of } A^\top A \end{aligned}$$

SVD is the most "reliable" method for least squares, but also the most expensive.

1.3.3.2 Invariant Subspaces

If $x'(t) = Ax(t)$, $x(0)$ is given, and

$$Ax(0) = \lambda x(0),$$

then $x(t) = e^{\lambda t} x(0)$. So it is easy to tell if $x(t) \to 0$ as $t \to \infty$: depends on whether real$(\lambda) < 0$. And if $x(0) = \sum_i \beta_i x_i$ (a linear combination of eigenvectors), where $Ax_i = \lambda_i x_i$, and $x(t) = \sum_i \beta_i e^{\lambda_i t} x_i$, then

$$x'(t) = \sum_i \beta_i \lambda_i e^{\lambda_i t} x_i = Ax(t).$$

So whether $x(t) \to 0$ depends on whether real$(\lambda_i) < 0$ i.e. whether $x(0)$ is in the subspace spanned by eigenvectors x_i with $\text{Re}(\lambda_i) < 0$, called the "invariant subspace" spanned by those eigenvectors.

So we will want algorithms to compute invariant subspaces, which can be faster and more accurate than computing corresponding eigenvectors (which may not all exist).

1.3.3.3 Generalized Eigenproblems

Consider $Mx''(t) + Kx(t) = 0$. Here x could be positions of objects in a mechanical system, M could be a mass matrix, K a stiffness matrix. Or x could be currents in a circuit, M inductances, and K reciprocals of capacitances. We could again plug in $x(t) = e^{\lambda t} x(0)$ and get $\lambda^2 M x(0) + Kx(0) = 0$, which means $x(0)$ is a generalized eigenvector, and λ^2 a generalized eigenvalue of the pair of matrices (M, K). The usual definition of eigenvalue, being a root of $\det(K - \lambda' I)$, becomes being a root of $\det(K + \lambda' M) = 0$, where $\lambda' = \lambda^2$. All the ideas and algorithms above generalize to this case (Jordan form becomes Weierstrass form). Note that when M is singular, this is not the same as the standard eigenproblem for $M^{-1}K$. Singular M arise in "differential-algebraic systems", i.e. ODEs with linear constraints.

[3] $UU^\top = I$.

1.3.3.4 Nonlinear Eigenproblems

Consider

$$Mx''(t) + Dx'(t) + Kx(t) = 0.$$

Here D could be a damping matrix in a mechanical system, or resistances in a circuit. We could again plug in $x(t) = e^{\lambda t}x(0)$ and get

$$\lambda^2 Mx(0) + \lambda Dx(0) + Kx(0) = 0,$$

We will show how to reduce this one to a linear eigenproblem of twice the size.

1.3.3.5 Singular Eigenproblems

Consider the control system:

$$x'(t) = Ax(t) + Bu(t),$$

where A is $n \times n$ and B is $n \times m$ with $m < n$. Here $u(t)$ is a control input that you want to choose to guide $x(t)$. The question of what subspace $x(t)$ can lie in and be "controlled" by choosing $u(t)$ can be formulated as an eigenvalue problem for the pair of rectangular $nx(m+n)$ matrices $[B, A]$ and $[0, I]$. Again all the above ideas generalize (Jordan becomes Kronecker form).

1.3.4 Partial Solutions

Instead of a "complete" solution to an eigenvalue problem, that is computing all the eigenvalues (or singular values) and all the eigenvectors (or singular vectors), it is often only necessary to compute some of them, which may be much cheaper. This was illustrated above by invariant subspaces. Another example is a "low rank" approximation of a matrix, which could be just a few of the largest singular values, and their singular vectors.

1.3.5 Updating Solutions

Suppose we have solved $Ax = b$, a least squares problem, or an eigenproblem. Now we change A "slightly" and want to solve another problem, taking advantage of our previous work as much as possible. Here "slightly change" could mean changing a few entries, rows or columns, adding a few rows or columns, or even adding a low-rank matrix to A. The standard solution to SVD uses this.

1.3.6 Tensors

Instead of 2-dimensional arrays, i.e. matrices, data often comes in 3D or higher dimensional arrays, called tensors. Sometimes these can be "repacked" as matrices, and standard linear algorithms applied, but in many cases users prefer to retain the higher dimensional structure. There is a big literature on extending concepts and algorithms, from matrix multiplication to low-rank approximations, to tensors; these problems are sometimes much harder than for matrices (cf. Hillar and Lim (2013) "Most Tensor Problems ar NP-Hard").

1.4 The Structure of Matrices

To explore the structure of A, we tell a story about a typical office hours meeting:

A student (S) says: "I need to solve an $n \times n$ linear system $Ax = b$. What should I do?"

The Professor (P) replies: "The standard algorithm is *Gaussian Elimination* (GE), which costs $(2/3)n^3$ floating point operations (flops)."

S: "That's too expensive."

P: "Tell me more about your problem."

S: "Well, the matrix is real and symmetric, $A = A^\top$."

P: "Anything else?"

S: "Oh yes, it's positive definite, $x^\top Ax > 0$ for all nonzero vectors x."

P: "Great, you can use *Cholesky*, it costs only $(1/3)n^3$ flops, half as much."

The professor also begins to record their conversation in a "decision tree", where each node represents an algorithm, and edge represents a property of the matrix, with arrows pointing to nodes/algorithms depending on the answer. (This corresponds to Table 6.1 in the textbook)

S: "That's still too expensive."

P: "Tell me more about your matrix"

S: "It has a lot of zeros it, in fact all zeros once you're a distance $n^{2/3}$ from the diagonal."

P: "Great, you have a band matrix with bandwidth $\text{bw} = n^{2/3}$, so there is a version of *Cholesky* that only costs $O(\text{bw}^2 n) = O(n^{7/3})$ flops, much cheaper!"

S: "Still too expensive."

P: "So tell me more."

S: "I need to solve the problem over and over again, with the same A and different b, so should I just precompute A^{-1} once and multiply by it?"

P: "A^{-1} will be dense, so just multiplying by it costs $2n^2$ flops, but you can reuse the output of *Cholesky* (the L factor) to solve for each b in just $O(\text{bw} \cdot n) = O(n^{5/3})$".

S: "That's still too expensive."

P: "Tell me more."

S: "There are actually a lot more zero entries, just at most 7 nonzeros per row."

P: "Let's think about using an iterative method instead of a direct method, which just needs to multiply your matrix times a vector many times, updating an approximate answer until it is accurate enough."

S: "How many matrix-vectors multiplies will I need to do, to get a reasonably accurate answer?"

P: "Can you say anything about the range of eigenvalues, say $\kappa(A) = \lambda_{\max}/\lambda_{\min}$?"

S: "Yes, $\kappa(A)$ is about $n^{2/3}$ too."

P: "You could use the conjugate gradient method, which will need about $O(\sqrt{\kappa(A)})$ iterations, so $n^{1/3}$. With at most 7 nonzeros per row, matrix-vector multiplication costs at most $14n$ flops, so altogether $O(n^{1/3}n) = O(n^{4/3})$ flops. Happy yet?"

S: "No."

P: "Tell me more."

S: "I actually know the largest and smallest eigenvalues, does that help?"

P: "You know a lot about your matrix. What problem are you really trying to solve?"

S: "I have a cube of metal, I know the temperature everywhere on the surface, and I want to know the temperature everywhere inside."

P: "Oh, you're solving the 3D Poisson equation, why didn't you say so! Your best choice is either a direct method using an *Fast Fourier Transform* (FFT) costing $O(n \log n)$ flops, or an iterative method called multigrid, costing $O(n)$ flops. And $O(n)$ flops is $O(1)$ flops per component of the solution, you won't do better."

S: "And where can I download the software?" ...

This illustrates an important theme of this course, exploiting the mathematical structure of your problem to find the fastest solution. The Poisson equation is one of the best studied examples, but the number of interesting mathematical structures is bounded only by the imaginations of people working in math, science, engineering and other fields, who come up with problems to solve, so we will only explore some of the most widely used structures and algorithms in this course.

1.5 The Desired Accuracy

There is a range of choices, with a natural tradeoff with speed (more accuracy $\Rightarrow$ slower).

1.5.1 Guaranteed Accurate

For many problems, this would require a "proof" that a matrix is nonsingular, or exactly singular, requiring arbitrary precision arithmetic, so we won't consider this further. Feel free to use systems like *Mathematica* if this is what you need. We discuss some cheaper alternatives (with weaker "guarantees") below.

1.5.2 Backward Stable

This is the "gold standard" for most linear algebra problems, and means "get the exact answer for a slightly wrong problem," or more precisely:

If $\text{alg}(x)$ is our computed approximation of $f(x)$, then

$$\text{alg}(x) = f(x + \delta)$$

where δ is "small" compared to x. The definition of "small" will use matrix norms, but it should be proportional to the error in the underlying floating point arithmetic, e.g. 10^{-16} in double precision. In other words, if you only know your inputs to 16 digits (just rounding them to fit in the computer makes them this uncertain), then $\text{alg}(x)$ is "as good" as any other answer.

1.5.3 Residual as Small as Desired

For problems too large to use a backward stable algorithm, we can use an iterative algorithm that progressively makes a residual (e.g. $\|Ax - b\|$) smaller, until it is good enough for the user; we will see that the residual also estimates the size of δ, i.e. the backward error.

1.5.4 Probably OK

This refers to "randomized linear algebra" (RLA for short), where a large problem is cheaply replaced with a much smaller random approximation that we can then solve. Some of these approximations involve iterating, with a residual, so we can tell whether the answer is ok, and some do not, and only come with theorems like "the error is less than ε with probability $1 - \delta$ if the size of the random approximation is big enough, i.e. proportional to a quantity $f(\delta, \varepsilon)$ that gets larger as δ (the probability of failure) and ε (the error bound) get smaller."

$$f(\delta, \varepsilon) = \Omega\left(\frac{\log(1/\delta)}{\varepsilon^2}\right)$$

is a common result in this field, so choosing an error epsilon to be very small means these methods may not be competitive with existing methods. These algorithms are motivated by "big data", where problems are much larger than classical algorithms can handle fast enough, and approximate answers are good enough.

1.5.5 Alternatives for Guaranteed Accuracy

For users who would like more information about the reliability of their results, but cannot afford "guaranteed accuracy", here are two alternatives:

- Error bounds: Note that if a matrix is singular, or "close" to singular, a backward stable algorithm for $Ax = b$ may give a completely wrong answer (e.g. deciding A is (non)singular when the opposite is true). We will see that there is an error bound that is proportional to the "condition number"

  $$\kappa(A) = 1/(\text{distance from } A \text{ to the nearest singular matrix}),$$

  which can be estimated at reasonable additional cost, in particular when A is dense. There are analogous error bounds for most other linear algebra problems.

- "Guaranteed correct" except in "rare cases": combining error bounds with a few steps of Newton's method to improve the answer can often give small error bounds unless a matrix is very close to singular. The cost is again reasonable, for $Ax = b$ and least squares, for dense A. These techniques have recently become popular because of widespread deployment of low-precision accelerators for machine learning, which typically provide very fast 16-bit implementations of operations like matrix multiplication, which can be used as building blocks for other linear algebra operations.

There is one more kind of "accuracy" we will discuss briefly later, getting bit-wise identical results every time you run the program, which many users expect for debugging purposes. This can no longer be expected on many modern computers, for reasons we will discuss, along with some proposed solutions.

1.6 Implementations of Efficient Algorithms

The story illustrating "The Structure of Matrices" suggests that counting floating point operations is the right metric for choosing the fastest algorithm. In fact others may be much more important. Here are some examples.

1.6.1 Fewest Keystrokes

E.g. $A\backslash b$ to solve $Ax = b$. More generally, the metric is finding an existing reasonable implementation with as little human effort as possible. We will try to give pointers to the best available implementations, usually in libraries. There are lots of pointers on class webpage (e.g. *netlib*, *GAMS*). $A\backslash b$ invokes the *LAPACK* library, which is also used as the basis of the libraries used by most computer vendors, and has been developed in a collaboration by Berkeley and other universities over a period of years, with more work (and possible class projects) underway.

1.6.2 The Meaning of the Fewest Flops

What does fewest flops (floating point operations) really mean? How many operations does it take to multiply 2 $n \times n$ matrices?

- Classical: $2n^3$.
- Strassen (1969): $O(n^{\log_2 7}) \approxeq O(n^{2.81})$, which is sometimes practical, but only for large n, because of the constant factor hidden in the $O()$.
- Coppersmith/Winograd (1987): $O(n^{2.376})$, which is not practical so far, n needs to be enormous.
- Umans/Cohn: $O(n^{2.376})$, maybe $O(n^2)$? The search for a faster algorithm was reduced to a group theory problem (*FOCS2003*); Again not yet practical.
- Williams (2013): $O(n^{2.3728642})$.
- Le Gall (2014): $O(n^{2.3728639})$: World's record so far.
- Williams, et al. (2020): $O(n^{2.3728596})$
- Williams, et al. (2023): $O(n^{2.371552})$
- Demmel, Dumitriu, Holtz (2008): all the other standard linear algebra problems (solving $Ax = b$, eigenvalues, etc.) have algorithms with same complexity as matrix multiplication, $O(n^x)$ for some x, (and backward stable) - ideas behind some of these algorithms could be practical.

1.6.3 About Flops

Counting flops is not the only important metric in today's and the future world, for two reasons:

- Let's recall *Moore's Law*, which was a long-standing observation that the number of transistors on a chip kept doubling about every 2 years. This meant that until around 2004, computers kept doubling in speed periodically with no code changes. This has ended, for technological reasons, so instead the only way computers can run faster is by having multiple processors, so all code that needs to run faster (not just linear algebra!) has to change to run in parallel. Some of these parallel algorithms are mathematically the same as their sequential counterparts, and some are different; we will discuss some of these parallel algorithms, in particular those that are different.

- What is most expensive operation in a computer? Is it doing arithmetic? No: it is moving data, say between main memory (DRAM) and cache (smaller memory on the CPU, where arithmetic is done), or between parallel processors connected over a network. You can only do arithmetic on data stored in cache, not in DRAM, or on data stored on the same parallel processor, not different ones (draw pictures of basic architectures). It can cost 10x, 100x, 1000x or more to move a word than do an add/subtract/multiply, so our goal could be to minimize transferring data between different locations.

  Example 1.1

  Consider adding two $n \times n$ matrices $C = A + B$. The cost is n^2 reads of A (moving each A_{ij} from DRAM to cache, n^2 reads of B, n^2 additions, and n^2 writes of C (from cache back to DRAM). The reads and writes cost $O(100)$ times as much as the additions.

  Example 1.2

  Nvidia GPU (circa 2008) attached to a CPU: It cost 4 microseconds to call a subroutine in which time you could have done 1e7 flops since the GPU ran at 300 GFlops/s. Technology trends are making this worse: the speeds of arithmetic and getting data from DRAM are both still getting faster, but arithmetic is improving more quickly.

  Consequence: Two different algorithms for the same problem, even if they do the same number of arithmetic operations, may differ vastly in speed, because they do different numbers of data moves.

  Example 1.3

  The speed difference between matrix multipulication written in the naive way (3 nested loops) vs. optimized to minimize data moves is easily 100x, similarly for other operations.

  In recent years we and others have discovered new algorithms for most of the algorithms discussed in this class that provably minimize data movement, and can be much faster than the conventional algorithms. We are working on updating the standard libraries (called *LAPACK* and *ScaLAPACK*) used by essentially all companies (including *MATLAB*).

1.6.4 Other Metrics

So far we have been talking about minimizing the time to solve a problem. Is there any other metric besides time that matters? Yes: energy. It turns out that a major obstacle to *Moore's*

Law continuing as it had in the past is that it would take too much energy: the chips would get so hot they would melt, if we tried to build them the same way as before. And so whether you are concerned about the battery in your laptop dying, or the $1M per megawatt per year it costs to run your data center or supercomputer, or how long your drone can stay airborne, people are looking for ways to save energy. So which operations performed by a computer cost the most energy? Again, moving data can cost orders of magnitude more energy per operation than arithmetic, so the algorithms that minimize communication can also minimize energy.

2 Lecture 2: Floating Point Arithmetic and Error Analysis

Goals:

- Floating point arithmetic
- Roundoff error analysis for polynomial evaluation
- Beyond basic error analysis: exceptions, high/low/variable precision arithmetic, reproducibility, interval arithmetic, exploiting mathematical structure to get accuracy without high precision

Example 2.1 (Polynomial Evaluation, and Polynomial Zero Finding)

Review how bisection to find a root of $f(x) = 0$ works:

Algorithm 2.1.

```
1: start with an interval $[x_1, x_2]$ where $f$ changes sign: $f(x_1)f(x_2) < 0$
2: evaluate at midpoint: $f((x_1 + x_2)/2)$
3: keep bisecting subinterval where $f$ changes sign
```

You can get widely varying answers when using this algorithm to find the root of $(x-2)^{13}$ on different intervals containing 2. This is due to using Horner's rule to evaluate $(x-2)^{13}$, so we will try to bound the error in Horner's rule. To do so, we need to understand the basics of floating point arithmetic.

2.1 Floating Point

Long ago, computers did floating point in many different ways, making it hard to understand bugs and write portable code. Fortunately Prof. Kahan led an IEEE standards committee that convinced all the computer manufacturers to agree on one way to do it, called the IEEE 754 Floating Point Standard, for which he won the Turing Award. This was in 1985. The standard was updated in 2008, and again in 2019. We'll say more on the significant changes below.

Scientific Notation: $\pm \mathrm{d.d} \cdots \mathrm{d} \cdot \mathrm{radix}^{\mathrm{e}}$

Floating point usually uses radix $= 2$ (or 10 for financial applications), so you need to store the sign bit ($\pm$), exponent (e), and mantissa (d.ddd). Both $p = \#$digits in the mantissa and the exponent range are limited, to fit into 16, 32, 64 or 128 bits. Historically, only 32 and 64 bit precisions have been widely supported in hardware. But lately 16 bits have become popular, for machine learning, and companies like Google, Nvidia, Intel and others are also implementing a 16-bit format that differs from the IEEE Standard, called *bfloat16*, with even lower precision ($p = 8$ vs $p = 11$). How to use such low precision to reliably solve linear algebra (and other non-machine learning) problems is an area of current research. For simplicity, we will initially ignore the limit on exponent range, i.e. assume no overflow or underflow.

Normalization: We use 3.100e0 not 0.0031e3 - i.e. the leading digit is nonzero. Normalization gives uniqueness of representations, which is useful. And in binary, the leading digit must be 1, so it doesn't need to be stored, giving us a free bit of precision (called the "hidden bit").

Definition 2.1. $\text{rnd}(x)$ is the nearest floating point number to x. (Note: The default IEEE 754 rule for breaking ties is "nearest even", i.e. the number whose least significant digit is even (so zero in binary).)
Definition 2.2. Relative Representation Error (RRE): $\text{RRE}(x) = |x - \text{rnd}(x)|/|\text{rnd}(x)|$
Definition 2.3. Maximum Relative Representation Error: $\max_{x \neq 0} \text{RRE}(x)$, (also known as machine epsilon, macheps, ε). MaxRRE is the half distance from 1 to next larger number $1 + \text{radix}^{1-p}$, which equals to $.5 \cdot \text{radix}^{1-p}$ (2^{-p} in binary).

Roundoff error model, assuming no over/underflow:

$$\begin{aligned} \text{fl}(a \odot b) &= \text{rnd}(a \odot b) \\ &= \text{true result rounded to nearest even} \\ &= (a \odot b)(1 + \delta) \end{aligned}$$

where $|\delta| \leq \varepsilon$. The $\odot$ may be add, subtract, multiply or divide (or even $\sqrt{\ }$). We will use this throughout the course, it's all you need for most algorithms. It's also true for complex arithmetic (but using a bigger ε).

Existing IEEE formats:

- Single(S): 32 bits = 1 (for sign) + 8 (for exponent) + 23 (for mantissa). So there are $p = 24 = 1$ (hidden bit) + 23 bits to represent a number, and so $\varepsilon = 2^{-24} \sim 6\text{e}-8$. Also $-126 \leq \text{e} \leq 127$, so overflowthreshold(OV) $\sim 2^{128} \sim 1\text{e}38$, underflowthreshold(UN) $= 2^{-126} \sim 1\text{e}-38$
- Double(D): $64 = 1 + 11 + 52$ bits, so $p = 53$, $\varepsilon = 2^{-53} \sim 1\text{e}-16$, $-1022 \leq \text{e} \leq 1023$, OV $\sim 2^{1024} \sim 1\text{e}308$, and UN $= 2^{-1022} \sim 1\text{e}-308$
- Quad(Q): $128 = 1 + 15 + 112$ bits, $p = 113$, $\varepsilon = 2^{-113} \sim 1\text{e}-34$, $-16382 \leq \text{e} \leq 16383$, OV $\sim 2^{16384} \sim 1\text{e}4932$, and UN $= 2^{-16382} \sim 1\text{e}-4932$
- Half(H): $16 = 1 + 5 + 10$ bits, $p = 11$, $\varepsilon = 2^{-11} \sim 5\text{e}-4$, $-14 \leq \text{e} \leq 15$, OV $\sim 2^{15} \sim 1\text{e}4$, and UN $= 2^{-14} \sim 1\text{e}-4$

The new *bloat16* format has the following parameters:

$$16 = 1 + 8 + 7, \text{ so } p = 8, \ \varepsilon = 2^{-8} \sim 4\text{e}-3$$

The exponent e has the same range as IEEE single (by design: converting between *bfloat16* and *S* cannot overflow or underflow). Referring back to Lecture 1, where we referred to the approach of using a few steps of *Newton's method* to be "guaranteed correct except in rare cases," a common approach is to try to do most of the work in lower (and so faster) precision, and then do just a little work in higher (and so slower) precision, typically to compute accurate residuals (like $Ax - b$), during the *Newton* steps; the goal is to get the same accuracy as though the entire computation had been done in higher precision.

We briefly mention E(xtended), which was an 80-bit format on Intel x86 architectures, and was in the old IEEE standard from 1985, but is now deprecated. See also the IEEE 754 standard for details of decimal arithmetic (future C standards will include decimal types, as already in gcc).

That's enough information about floating point arithmetic to understand the plot of $(x-2)^{13}$, but more about floating point later.

2.1.1 Analyze Horner's Rule for evaluating $p(x)$

- Simplest expression: $p = \sum_{i=0}^{d} a_i x^i$
- Algorithm:

Algorithm 2.2.

```
1: $p = a_d$
2: for $i = d-1 : -1 : 0$ do
3:     $p_i = xp + a_i$
4: end for
```

Algorithm 2.3 (Label intermediate terms).

```
1: $p_d = a_d$
2: for $i = d-1 : -1 : 0$ do
3:     $p_i = xp_{i+1} + a_i$
4: end for
```

Algorithm 2.4 (Introduce roundoff).

```
1: $p_d = a_d$
2: for $i = d-1 : -1 : 0$ do
3:     $p_i = [xp_{i+1}(1+\delta_i) + a_i](1+\delta')$        ▷ $|\delta_i|, |\delta_i'| < \varepsilon$
4: end for
```

Thus,

$$\begin{aligned} p_0 &= \sum_{i=0}^{d-1} \left[(1+\delta_i') \prod_{j=0}^{i-1} (1+\delta_j)(1+\delta_j') \right] a_i x^i + \prod_{j=0}^{d-1} (1+\delta_j)(1+\delta_j') a_d x^d \\ &= \sum_{i=0}^{d-1} [\text{product of } 2i+1 \text{ terms like } 1+\delta]\, a_i x^i + [\text{product of } 2i \text{ terms like } 1+\delta]\, a_d x^d \\ &= \sum_{i=0}^{d} a_i' x^i \end{aligned}$$

where $a_i' = a_i \cdot$ terms like $1+\delta$

- In words: *Horner's Rule* is backward stable: you get the exact value of a polynomial at x but with slightly changed coefficients a_i' from input $p(x)$.

How to simplify to get error bound:

$$\begin{aligned} \prod_{i=1}^{n} (1+\delta_i) &\le \prod_{i=1}^{n} (1+\varepsilon) = (1+\varepsilon)^n \\ &= 1 + n\varepsilon + \underbrace{O(\varepsilon^2)}_{\text{usually be ignored}} \\ &\le 1 + \frac{n\varepsilon}{1-n\varepsilon} \text{ if } n\varepsilon < 1 \end{aligned}$$

Similarly,

$$\begin{aligned}\prod_{i=1}^{n}(1+\delta_i) &\geq (1-\varepsilon)^n \\ &= 1 - n\varepsilon + O(\varepsilon^2) \\ &\geq 1 - \frac{n\varepsilon}{1-n\varepsilon} \text{ if } n\varepsilon < 1\end{aligned}$$

Putting them together,

$$\left|\prod_{i=1}^{n}(1+\delta_i) - 1\right| \leq n\varepsilon$$

and thus,

$$|\text{computed } p_d - p(x)| \leq \sum_{i=0}^{d-1}(2i+1)\varepsilon\left|a_i x^i\right| + 2d\varepsilon\left|a_d x^d\right|$$

$$\begin{aligned}\text{relerr} &= \frac{|\text{computed } p_d - p(x)|}{|p(x)|} \\ &\leq \sum_{i=0}^{d}\frac{|a_i x^i|}{|p(x)|}2d\varepsilon \\ &= \text{ condition number } \cdot \text{ relative backward error}\end{aligned}$$

How many decimal digits can we trust?

$$\begin{aligned}\text{dd correct digits} &\iff \text{relative error } \leq 10^{-\text{dd}} \\ &\iff -\log_{10}(\text{relative error}) \geq \text{dd}\end{aligned}$$

How to modify *Horner's Rule* to compute (an absolute) error bound:

$$p = a_d,\ \text{ebnd} = |a_d|$$

Algorithm 2.5.

```
1: $p = a_d$, ebnd $= |a_d|$
2: for $i = d-1 : -1 : 1$ do
3:     $p = xp + a_i$
4:     ebnd $= |x| \cdot$ ebnd $+ |a_i|$
5: end for
6: ebnd $= 2d\varepsilon \cdot$ ebnd
```

MATLAB demo:

Picture 2.2 is a foreshadowing of what will happen in linear algebra: The vertical axis is the number of correct digits, both actual (the black dots) and lower-bounded using our error bound (the red

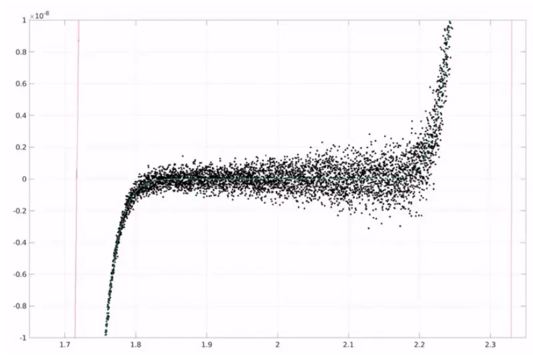


Figure 2.1: *MATLAB* demo

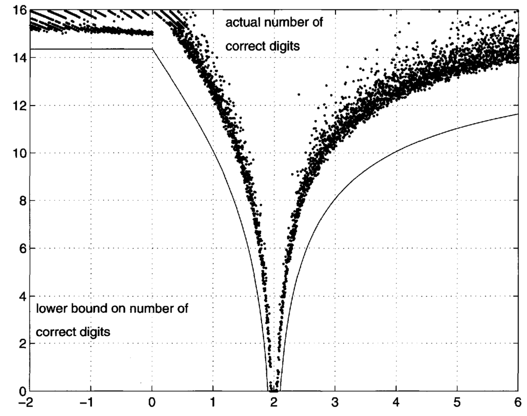


Figure 2.2: Plot of error bounds on the value of $y = (x-2)^{13}$ evaluated using *Horner's Rule.*

curve). The horizontal axis is the problem we are trying to solve, in this simple case the value of x at which we are evaluating a fixed polynomial $p(x)$.

The number of correct digits gets smaller and smaller, until no leading correct digits are computed, the closer the problem gets to the hardest possible problem, in this case the root $x = 2$ of the polynomial. This is the hardest problem because the only way to get a small relative error in the solution $p(2) = 0$, is to compute 0 exactly, i.e. no roundoff is permitted. And changing x very slightly makes the answer $p(x)$ change a lot, relatively speaking. In other words, the condition number, $\sum_i |a_i x^i|/|p(x)|$ in this simple case, approaches infinity as $p(x)$ approaches zero.

In linear algebra the horizontal axis still represents the problem being solved, but since the problem is typically defined by an $n \times n$ matrix, we need n^2 axes to define the problem. There are now many "hardest possible problems", e.g. singular matrices if the problem is matrix inversion. The singular matrices form a set of dimension $n^2 - 1$ in the set of all matrices, the surface defined by $\det(A) = 0$. And the closer the matrix is to this set, i.e. to being singular, the harder matrix inversion will be, in the sense that the error bound will get worse and worse the closer the matrix is to this set. Later we will show how to measure the distance from a matrix to the nearest singular matrix "exactly" (i.e. except for roundoff) using the SVD, and show that the condition number, and so the error bound, is inversely proportional to this distance.

Here is another way the above figure foreshadows linear algebra. Recall that we could interpret the computed value of the polynomial $p(x) = \sum_i a_i x^i$, with roundoff errors, as the exactly right value of a slightly wrong polynomial, that is

$$p_{\text{alg}(x)} = \sum_{i=0}^{d} [(1+e_i)a_i]x^i,$$

where $|e_i| \le 2d\varepsilon$. We called this property "backward stability", in contrast to "forward stability" which would means that the answer itself is close to correct. So the error bound bounds the difference between the exact solutions of two slightly different problems $p(x)$ and $p_{\text{alg}(x)}$.

For most linear algebra algorithms, we will also show they are backward stable. For example, if we want to solve $Ax = b$, we will instead get the exact solution of $(A+E)\hat{x} = b$, where the matrix E is "small" compared to A. Then we will get error bounds by essentially taking the first term of a Taylor expansion of $(A+E)^{-1}$:

$$\hat{x} - x = (A+E)^{-1}b - A^{-1}b$$

To do this, we will need to introduce some tools like matrix and vector norms (so we can quantify what "small" means) in the next lecture.

To extend the analysis of *Horner's Rule* to linear algebra algorithms, note the similarity between *Horner's Rule* and computing dot-products:

$$\begin{array}{lll} p = a_d & \text{for } i = d-1 : -1 : 0, & p = xp + a_i \\ s = 0 & \text{for } i = 1 : d, & s = x_i y_i + s \end{array}$$

Thus the error analysis of dot products, matrix multiplication, and other algorithms is very similar.

2.2 Details on Floating Point and Error Analysis

Next we briefly discuss some properties and uses of floating point that go beyond these most basic properties (even more details are at the end of these notes). Analyzing large, complicated codes by hand to understand all these issues is a challenge, so there is research in automating this analysis; there is a day-long tutorial on available tools that was held at *Supercomputing'19*.

2.2.1 Exception Handling

- Underflow: Tiny/Big = 0, (or "subnormal", special numbers at bottom of exponent range)
- Overflow and Divide-by-Zero: 1/0 = Inf = "infinity", represented as a special case in the usual format
- Natural computational rules: Big + Big = Inf, Big · Big = Inf, 3 − Inf = −Inf, etc.
- Invalid Operation: 0/0 = NaN = "Not a Number", also represented as special case: Inf − Inf = NaN, $\sqrt{-1}$ = NaN, 3 + NaN = NaN, etc. Flags are available to check if such exceptions occurred.

Impact on software:

- Reliability: Suppose we want to compute $s = \sqrt{\sum_{i=1}^{n} |x_i|^2}$: What could go wrong with:

Algorithm 2.6.

```
1: $s = 0$
2: for $i = i : n$ do
3:     $s = s + x_i^2$
4: end for
5: $s = \sqrt{s}$
```

Overflow or underflow could cause the wrong answer, even if the exact answer is within the threshold. To see how standard libraries deal with this, see *snrm2.f* in the *BLAS*. For a worst-case example, see Ariane 5 crash. We are currently investigating how to automatically guarantee that libraries like *LAPACK* cannot "misbehave" because of exceptions (go into infinite loops, give wrong answer without warning, etc.)

- Error analysis: it is possible to extend error analysis to take underflow into account by using the formula

$$\mathrm{fl}(a \odot b) = (1 + \delta)(a \odot b) + \eta$$

  where $|\delta| \le \varepsilon$ as before, and $|\eta|$ is bounded by a tiny number, depending on how underflow is handled.

- Speed: Run "reckless" algorithm that is fast but ignores possible exceptions. Check flags to see if exception occurred. In rare case of exception, rerun with slower, more reliable algorithms.

2.2.2 Higher Precision Arithmetic

Possible to simulate using either fixed or floating point, various packages available: *MPFR*, *ARPREC*, *XBLAS*. There are also special fast tricks just for high precision summation.

2.2.3 Lower Precision Arithmetic

Lower precision arithmetic: As mentioned before, many companies are building hardware accelerators for machine learning, which means they provide fast matrix multiplication. As mentioned in Lecture 1, and will be discussed later, it is possible to reformulate many dense linear algebra algorithms to use matrix multiplication as a building block, and so it is natural to want to use these accelerators for other linear algebra problems as well. As illustrated in our error analysis of *Horner's Rule*, many of our error bounds will be proportional to $d\varepsilon$, where d is the problem size (e.g. matrix dimension), and often $d^2\varepsilon$ or more. These bounds are only useful when $d\varepsilon$ (or $d^2\varepsilon$) are much smaller than 1. So when ε is about $4\mathrm{e}-3$ with *bfloat16*, this means d must be much smaller than $1/\varepsilon = 256$, or much smaller than $1/\sqrt{\varepsilon} = 16$, for these bounds to be useful. This is obviously very limiting, and raises several questions: Are our (worst case) error bounds too pessimistic? Can we do most of the work in low precision, and little in high precision, to get an accurate answer? There are some recent positive answers to both these questions.

2.2.4 Variable Precision

There have been various proposals over the years to support variable precision arithmetic, sometimes with variable word lengths, and sometimes encoded in a fixed length word. Variable word lengths make accessing arrays difficult. There has been recent interest in this area, with variable precision formats called *unums* (variable length) and posits (fixed length), proposed by John Gustavson. Posits allocate more bits for the mantissa when the exponent needs few bits (so the number is not far from 1 in magnitude) and fewer mantissa bits for long exponents. This so-called "tapered precision" complicates error analysis, and it is an open question of how error analyses or algorithms could change to benefit. See the class webpage for links to more details, including a youtube video of a debate between Gustavson and Kahan about the pros and cons of *unums*.

2.2.5 Reproducibility

Almost all users expect that running the same program more than once gives the bitwise identical answer; this is important for debugging, correctness, even legal reasons sometimes. But this can no longer be expected, even on your multicore laptop, because parallel computers (so nearly all now), may execute sums in different orders, and roundoff makes summation nonassociative:

$$\mathrm{fl}(\mathrm{fl}(1-1)+1\mathrm{e}-20)=1\mathrm{e}-20\neq 0=\mathrm{fl}(\mathrm{fl}(1+1\mathrm{e}-20)-1)$$

There is lots of work on coming up with algorithms that fix this, without slowing down too much, such as ReproBLAS. The 2019 version of the IEEE 754 standard also added a new "recommended instruction", to accelerate both the tricks for high precision arithmetic, and to make summation associative.

2.2.6 Guaranteed Error Bounds from Interval Arithmetic

Represent each floating point number by an interval $[x_{\mathrm{low}}, x_{\mathrm{high}}]$, and use special rounding modes in IEEE standard to make sure that lower and upper bounds are maintained throughout the computation:

$$[x_{\mathrm{low}}, x_{\mathrm{high}}]+[y_{\mathrm{low}}, y_{\mathrm{high}}]=[\mathrm{round\_down}(x_{\mathrm{low}}+y_{\mathrm{low}})+\mathrm{round\_up}(x_{\mathrm{high}}+y_{\mathrm{high}})]$$

Drawback: naively replacing all variables by intervals like this often makes interval widths grow so fast that they are useless. There have been many years of research on special algorithms that avoid this, especially for linear algebra.

2.2.7 Exploiting Structure to get High Accuracy

Some matrices have special mathematical structure that allows formulas to be used where roundoff is provably bounded so that you get high relative accuracy, i.e. most leading digits correct. For example, a Vandermonde matrix has entries $V_{ij}=x_i^j$, and arises naturally from polynomial interpolation problems. It turns out that this special structure permits many linear algebra problems, even eigenvalues, to be done accurately, no matter how hard ("ill-conditioned") the problem is using conventional algorithms. See "Accurate and efficient expression evaluation and linear algebra" for details.

3 Lecture 3: Norms, the SVD, and Condition Numbers

3.1 Accuracy

To summarize our approach to understanding accuracy in the face of roundoff (or other) error, our goal will be to prove that algorithms are "backward stable". For example if our algorithm for computing the scalar function $f(x)$:

$$\text{alg}(x) = f(x+\delta) \sim f(x) + f'(x)\delta$$

where δ is small compared to x. Then relative error can be approximated by:

$$\begin{aligned}\underbrace{\left|\frac{\text{alg}(x) - f(x)}{f(x)}\right|}_{\text{relative error in output}} &\lesssim \left|\frac{f'(x)\delta}{f(x)}\right| \\ &= \underbrace{\left|\frac{f'(x)x}{f(x)}\right|}_{\text{condition number}} \cdot \underbrace{\left|\frac{\delta}{x}\right|}_{\text{relative error in input}}\end{aligned}$$

Definition 3.1. The factor $\left|\frac{f'(x)x}{f(x)}\right|$ is called the condition number of the function f evaluated at x.

Same approach for solving $Ax = b$, but where we get $(A+\Delta)\hat{x} = b$ instead, where Δ is small compared to A.

To formalize the notion of "small", we need to understand vector and matrix norms. In both cases, we may say we want to compute $x = f(A)$, but get $\text{alg}(x) = \hat{x} = f(A+\Delta)$. So to bound the error, we write

$$\text{error} = \hat{x} - x = f(A+\Delta) - f(A).$$

Assuming Δ is small enough for *Taylor expansion*, we get

$$\text{error} \sim J_f(A)\Delta$$

where $J_f(A)$ is the Jacobian of f. If A and x were scalars, we could take absolute values and get an absolute error bound:

$$\|\text{error}\| \lessapprox \|J_f(A)\|\|\Delta\|$$

and proceed as above.

In the cases most relevant to linear algebra, A and x are not scalars but matrices and vectors (with obvious generalizations, depending on the problem). To generalize this error bound, we need to generalize absolute values, which leads us to norms.

3.2 Matrix and Vector Norms

Definition 3.2 (Inner Product)**.** Let $\mathcal{B}$ be a real linear space. $< \cdot, \cdot >: \mathcal{B} \times \mathcal{B} \to \mathbb{R}(\mathbb{C})$ is an inner product if all of the following apply:

1. $\langle x, y\rangle = \langle y, x\rangle$
2. $\langle x, y+z\rangle = \langle x, y\rangle + \langle x, z\rangle$
3. $\langle \alpha x, y\rangle = \alpha\langle x, y\rangle$ for any scalar α
4. $\langle x, x\rangle \geq 0$, and $\langle x, x\rangle = 0$ iff $x = 0$

Lemma 3.1

Cauchy-Schwartz inequality: $|\langle x, y\rangle| \leq \sqrt{\langle x, x\rangle\dot{}\langle y, y\rangle}$

Definition 3.3 (Norm)**.** Let $\mathcal{B}$ be a real (complex) linear space $\mathbb{R}^n$ (or $\mathbb{C}^n$). It is normed if there is a function $||\cdot|| :\ \mathcal{B} \to \mathbb{R}$ satisfying all of the following:

1. positive definiteness: $\|x\| \geq 0$ and $\|x\| = 0$ if and only if $x = 0$
2. homogeneity: $\|cx\| = |c|\,\|x\|$
3. the triangle inequality: $\|x+y\| \leq \|x\| + \|y\|$

Example 3.1

The most common norms:

- *p-norms*: $||x||_p = (\sum |x_i|^p)^{1/p}$ for $p \geq 1$
- *Euclidean norm*: 2-norm[a]
- *infinity-norm*: $||x||_\infty = \max_i |x_i|$
- *C-norm*: $||Cx||$ where C is any nonsingular matrix

Lemma 3.2

All norms are equivalent, i.e. given any $||\cdot||_a$ and $||\cdot||_b$, there are positive constants α and β such that

$$\alpha||\cdot||_a \leq ||\cdot||_b \leq \beta||\cdot||_a$$

Proof. Compactness. □

[a] If x is real, then $\|x\|_2^2 = \sum_{i=1}^n x_i^2 = x^\top x$

Definition 3.4 (Matrix Norm)**.**

1. $\|A\| \geq 0$ and $\|A\| = 0$ if and only if $A = 0$
2. $\|cA\| = |c|\,\|A\|$
3. $\|A + B\| \leq \|A\| + \|B\|$

Example 3.2

- Max norm: $\max_{ij} |A_{ij}|$
- Frobenius norm: $\|A\|_F = \left(\sum |a_{ij}|^2\right)^{1/2}$

Definition 3.5 (Operator Norm)**.** Given any vector norm $\|\cdot\|$, $\|A\| = \max_{x\neq 0} \frac{\|A_x\|}{\|x\|}$

Lemma 3.3

An operator norm is a matrix norm.

Lemma 3.4

If $\|A\|$ is an operator norm, then there exists x such that $\|x\| = 1$ and $\|Ax\| = \|A\|$.

Proof.

$$\begin{aligned}
\|A\| &= \max_{x\neq 0} \frac{\|Ax\|}{\|x\|} \\
&= \max_{x\neq 0} \|A(x/\|x\|)\| && \text{(by homogeneity)} \\
&= \max_{y:\|y\|=1} \|Ay\|
\end{aligned}$$

y attaining max exists since $\|Ay\|$ is a continuous function of y on compact (closed and bounded) set (unit ball). □

Lemma 3.5

For $x \in \mathbb{R}^n$:

$$\begin{aligned}
||x||_2 &\leq ||x||_1 \leq \sqrt{n}||x||_2, \\
||x||_\infty &\leq ||x||_2 \leq \sqrt{n}||x||_\infty, \\
||x||_\infty &\leq ||x||_1 \leq n||x||_\infty,
\end{aligned}$$

Definition 3.6. The norms are called mutually consistent if $||A \cdot B||_{m\times p} \leq ||A||_{m\times n} \cdot ||B||_{n\times p}$.

3.3 Orthogonal and Unitary Matrices

Notation: $Q^* = \overline{(Q^\top)}$, sometimes we write Q^H, H stands for *Hermitian.*
Definition 3.7 (Hermitian)**.** Q square, complex, $Q^* = Q$
Definition 3.8 (Orthogonal)**.** Q square, real, $Q^{-1} = Q^\top$
Definition 3.9 (Unitary)**.** Q square, complex, $Q^{-1} = Q^*$
Fact 3.1. Q orthogonal $\iff$ $Q^\top Q = I \iff$ all columns of Q are pairwise orthogonal and unit vectors. $QQ^\top = I$ implies same about rows.

Theorem 3.1 (Pythagorean Theorem)
$\|Qx\|_2 = \|x\|_2$

Proof.

$$\begin{aligned}\|Qx\|_2^2 &= (Qx)^\top Qx \\ &= x^\top Q^T Qx \\ &= x^\top x \\ &= \|x\|_2^2\end{aligned}$$

□

Fact 3.2. Q, Z orthogonal implies that QZ orthogonal.

Proof.

$$(QZ)^\top (QZ) = Z^\top Q^\top QZ = Z^\top Z = I$$

□

Fact 3.3. if Q is $m \times n$, $n < m$ and $Q^T Q = I_n$, then you can add $m - n$ columns to Q, to make it square and orthogonal, i.e. find an $m \times (m - n)$ matrix $\tilde{Q}$ such that $[Q, \tilde{Q}]$ is $m \times m$ and orthogonal.

Lemma 3.6

1. $||Ax|| \leq ||A|| ||x||$ for a vector norm and its operator norm
2. $||AB|| \leq ||A|| ||B||$ for any operator norm or for the Frobenius norm.
3. The max norm and Frobenius norm are not operator norms.
4. $||QAZ|| = ||A||$ if Q and Z are orthogonal or unitary for the Frobenius norm and for and for the operator norm induced by $|| \cdot ||_2$
5. Maximum absolute row sum: $\|A\|_\infty \equiv \max_{x \neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty} = \max_i \sum_j |a_{ij}|$
6. Maximum absolute col sum: $\|A\|_1 \equiv \max_{x \neq 0} \frac{\|Ax\|_1}{\|x\|_1} = \left\|A^T\right\|_\infty = \max_j \sum_i |a_{ij}|$

7. $\|A\|_2 \equiv \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sqrt{\lambda_{\max}\left(A^* A\right)}$

   Proof. With the fact that $A^\top A$ symmetric and real implies that there is eigendecomposition $A^\top A q_i = \lambda_i q_i$, where λ_i real, q_i all unit orthogonal vectors. $A^\top A = Q\Lambda Q^\top$:

$$\begin{aligned}
\|A\|_2 &= \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} \\
&= \max_{x\neq 0} \frac{\sqrt{(Ax)^\top (Ax)}}{\sqrt{x^\top x}} \\
&= \max_{x\neq 0} \sqrt{\frac{x^T A^T A x}{x^\top x}} \\
&= \sqrt{\max_{x\neq 0} \frac{x^\top Q\Lambda Q^\top x}{x^\top Q \underbrace{Q^\top x}_{y}}} \\
&= \sqrt{\max_{y\neq 0} \frac{y^\top \Lambda y}{y^\top y}} \\
&= \sqrt{\max_{y\neq 0} \frac{\sum \lambda_i y_i^2}{\sum y_i^2}} \\
&\leq \sqrt{\max_{y\neq 0} \frac{\lambda_{\max} \sum y_i^2}{\sum y_i^2}} \\
&= \sqrt{\lambda_{\max}}
\end{aligned}$$

   Additionally, we note that equality in the second-to-last line is attainable by $y = [0 \cdots 0\ 1\ 0 \cdots 0]^\top$, with the 1 located at the index corresponding to $\lambda_{\max}$ □

8. $\|A\|_2 = \left\|A^\top\right\|_2$
9. $\|A\|_2 = \max_i |\lambda_i(A)|$ if A is normal, i.e., $AA^* = A^*A$.
10. If A is n-by-n, then

$$\begin{aligned}
&n^{-1/2}\|A\|_2 \leq \|A\|_1 \leq n^{1/2}\|A\|_2 \\
&n^{-1/2}\|A\|_2 \leq \|A\|_\infty \leq n^{1/2}\|A\|_2 \\
&n^{-1}\|A\|_\infty \leq \|A\|_1 \leq n\|A\|_\infty \\
&n^{1/2}\|A\|_1 \leq \|A\|_F \leq n^{1/2}\|A\|_2
\end{aligned}$$

11. $\|Q\|_2 = 1$ if Q orthogonal

3.4 Singular Value Decomposition

The SVD is a Swiss Army Knife of numerical linear algebra. Given SVD, one can easily

- solve $Ax = b$;

- solve overdetermined or underdetermined least squares problems with rectangular A (whether A full rank or not);
- compute eigenvalues and eigenvectors of $AA^\top$, $A^\top A$;
- compute eigenvalues and eigenvectors of $A = A^\top$.

Furthermore, one can use the SVD to write down error bounds for all these problems. It is more expensive to compute than other algorithms specialized for these problems, so it may not be the algorithm of first resort.

Theorem 3.2

Suppose A is an $m \times m$ matrix. Then there an exists orthogonal matrix $U = [u_1, \cdots, u_m]$, a diagonal matrix $\Sigma = \operatorname{diag}(\sigma_1, \cdots, \sigma_m)$, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_m \geq 0$, and an orthogonal matrix $V = [v_1, \cdots, v_m]$ such that $A = U\Sigma V^\top$. The v_i are right singular vectors, the u_i are left singular vectors, and the σ_i are singular values.

More generally, A is an $m \times n$ matrix, where $m > n$, U is an $m \times m$ matrix and is orthogonal as before, V is a $n \times n$, orthogonal matrix, Σ is an $m \times n$ matrix with same diagonal. When $m > n$, sometimes write as following (thin SVD):

$$A = [u_1, \cdots, u_n] \operatorname{diag}(\sigma_1, \cdots, \sigma_n) V^\top$$

Geometric interpretation: A is a linear mapping from $\mathbb{R}^n \to \mathbb{R}^m$ with right orthogonal bases of $\mathbb{R}^n$ and $\mathbb{R}^m$, A is diagonal, $Av_i = \sigma_i u_i$

Proof. induction on n. 2 base cases:

$n = 1$: Let U have first column $= A/\|A\|_2$, other columns chosen in any way that makes U orthogonal, $\sigma_1 = \|A\|_2$, $v = 1$

$A = 0$: $U = I_m$, $\Sigma = 0$, $V = I_n$

Induction step (if $A \neq 0$)

$$\begin{aligned} \|A\|_2 &= \max_{x \neq 0} \|Ax\|_2 / \|x\|_2 \\ &= \max_{\|x\|_2 = 1} \|Ax\|_2 \end{aligned}$$

Let v_1 be x attaining max, $\sigma_1 = \|A\|_2 = \|Av_1\|_2$, $u_1 = Av_1/\|Av_1\|_2$. $V = \left[v_1, \hat{V}\right]$, $U = \left[u_1, \hat{U}\right]$, both square and orthogonal.

$$\begin{aligned}\hat{A} &= U^\top A V \\ &= \left[\begin{array}{c} u_1^\top \\ \hat{U}^\top \end{array}\right] A[v_1, \hat{V}] \\ &= \left[\begin{array}{cc} v_1^\top A v_1 & u_1^\top A \hat{V} \\ \hat{U}^\top A v_1 & \hat{U}^\top A \hat{V} \end{array}\right] \\ &= \left[\begin{array}{cc} \sigma_1 & A_{12} \\ A_{21} & A_{22} \end{array}\right]\end{aligned}$$

$A_{21} = 0$ by definition of $\hat{U}$, $A_{12} = 0$ by definition of $\sigma_1 = \|A\|_2$.

Use induction on $A_{22} = U_2 \Sigma_2 V_2^T$. To get A:

$$\begin{aligned}A &= U \hat{A} V^\top \\ &= U \left[\begin{array}{cc} \sigma_1 & A_{12} \\ A_{21} & A_{22} \end{array}\right] V^\top \\ &= U \left[\begin{array}{cc} \sigma_1 & 0 \\ 0 & U_2 \Sigma_2 V_2^\top \end{array}\right] V^\top \\ &= U \left[\begin{array}{cc} 1 & 0 \\ 0 & U_2 \end{array}\right] \left[\begin{array}{cc} \sigma_1 & 0 \\ 0 & \Sigma_2 \end{array}\right] \left[\begin{array}{cc} 1 & 0 \\ 0 & V_2^\top \end{array}\right] V^\top \\ &= \tilde{U} \Sigma \tilde{V}^\top\end{aligned}$$

□

3.5 Useful Properties of SVD

Fact 3.4. A is $m \times m$, nonsingular, we can solve $Ax = b$, given A's SVD, in $O(n^2)$ move operations.

Proof.

$$x = A^{-1} b = \left(U \Sigma v^T\right)^{-1} b = V \left(\Sigma^{-1} \left(U^\top b\right)\right)$$

□

But note: computing the SVD itself is expensive ($O(n^3)$). If all you want to do is solve $Ax = b$, Gaussian Elimination is cheaper. On the other hand, we will see that the SVD is more reliable when A is nearly singular, provides and error bound, and even lets us "solve" $Ax = b$ when A is exactly singular.

Fact 3.5. When $m > n$ and A is full rank, solve $\arg\min_x \|Ax - b\|_2$ use thin SVD $A = U \Sigma V^\top$, then

$$x = V \Sigma^{-1} U^\top b.$$

Proof. $A = \hat{U}\hat{\Sigma}V^\top$, $\hat{U} = [U, U']$, square and orthogonal $\hat{\Sigma} = \begin{bmatrix} \Sigma \\ 0 \end{bmatrix}$.

$$\begin{aligned}
\|Ax-b\|_2^2 &= \|\hat{U}\hat{\Sigma}V^\top x - b\|_2^2 \\
&= \|\hat{U}^\top(\hat{U}\hat{\Sigma}V^\top x - b)\|_2^2 && \text{(orthogonal matrix does not change the 2-norm)} \\
&= \left\|\hat{\Sigma}V^\top x - \hat{U}^\top b\right\|_2^2 \\
&= \left\|\begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^\top x - \begin{bmatrix} U^\top \\ U'^\top \end{bmatrix} b\right\|_2^2 \\
&= \left\|\begin{bmatrix} \Sigma V^\top x - U^\top b \\ -U'^T b \end{bmatrix}\right\|_2^2 \\
&= \left\|\Sigma V^\top x - U^\top b\right\|_2^2 + \|U'^\top b\|_2^2
\end{aligned}$$

Easy to minimize by choosing $x = V\Sigma^{-1}U^\top b$ to zero out the term depending on x. □

Definition 3.10 (Moore-Penrose pseudoinverse)**.** For $A = U\Sigma V^\top$ and is full rank,

$$A^+ = V\Sigma^{-1}U^\top.$$

This is the most natural extension of the definition of "inverse" to rectangular full-rank matrices. We can also use the SVD, and an appropriately defined Moore-Penrose pseudoinverse to solve the rank deficient least squares problems, or underdetermined problem ($m < n$), as we describe later.

Just to solve a least squares problem where you are not worried about rank deficiency, the QR decomposition is cheaper. On the other hand, we will see that the SVD is more reliable when A is nearly singular.

Fact 3.6. A symmetric with eigenvalues $\Lambda = \text{diag}(\lambda_1, \cdots, \lambda_n)$. Then SVD of A is

$$\begin{aligned}
A &= V\Lambda V^\top && \text{(eigenvalue decomposition)} \\
&= (VD)(D\Lambda) \cdot V^\top \\
&= U\Sigma V^\top
\end{aligned}$$

where $D = \text{diag}(\text{sign}(\lambda_i))$, and so $\Sigma = |\Lambda|$.

Fact 3.7. Using thin SVD,

$$A^\top A = (U\Sigma V^T)^\top (U\Sigma V^T) = V\Sigma^2 V^\top$$

which is the eigenvalue decomposition of $A^\top A$.

Fact 3.8. Using thin SVD,

$$AA^\top = (U\Sigma V^T)(U\Sigma V^T)^\top = U\Sigma^2 U^\top$$

which is the eigenvalue decomposition of $AA^\top$, but without writing down eigenvectors corresponding to the implicitly zero eigenvalues. It would be a perfectly adequate way to do if you only care about the space spanned by the nonzero eigenvectors.

Fact 3.9. Let $H = \begin{bmatrix} 0 & A^\top \\ A & 0 \end{bmatrix}$. Then H has eigenvalues $\pm\sigma_i$ and eigenvectors $\frac{1}{\sqrt{2}}\begin{bmatrix} v_i \\ \pm u_i \end{bmatrix}$

Proof. Plug in $A = U\Sigma V^\top$ into H. □

Fact 3.10. Suggests that algorithms for SVD and symmetric eigenproblem will be closely related.
Fact 3.11. $\|A\|_2 = \sigma_1$, $\|A^{-1}\|_2 = \frac{1}{\sigma_n}$.
Definition 3.11 (Condition number). $\kappa(A) = \frac{\sigma_1}{\sigma_n} = \|A\|_2\|A^{-1}\|_2$
Fact 3.12. Let S be unit sphere in $\mathbb{R}^n$. Then AS is an ellipsoid centered at O with principal axes in the directions u_i with length σ_i.

Proof. Suppose $s = [s_1, \ldots, s_n]$ where $\|s\|_2 = 1$, and write $As = U\Sigma V^\top s = U\Sigma\hat{s} = \sum_i u_i\sigma_i\hat{s}_i$. □

Example 3.3

$U_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $U_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sigma_1\hat{s}_1 \\ \sigma_2\hat{s}_2 \end{bmatrix}$,

$$\left(\frac{x}{\sigma_1}\right)^2 + \left(\frac{y}{\sigma_2}\right)^2 = \hat{s}_1^2 + \hat{s}_2^2 = 1$$

Fact 3.13. Suppose $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 = \sigma_{r+1} = \sigma_{r+2} = \cdots = \sigma_n$. Then A has rank r; the null space of A is $A = \operatorname{span}(v_{r+1}, \cdots, v_n)$; the range space of A is $A = \operatorname{span}(v_1, \cdots, v_r)$.
Fact 3.14. Matrix A_k of rank k closest to A in 2-norm is

$$A_k = \sum_{c=1}^{k} u_i\sigma_i v_i^\top = U\Sigma_k V^\top,$$

where

$$\Sigma_k = \operatorname{diag}(\sigma_1, \sigma_2, \cdots, \sigma_k, 0, \cdots, 0),$$

and the distance is

$$\|A_k - A\|_2 = \sigma_{k+1}.$$

In particular, the closest non-full rank matrix to A is at distance $\sigma_n = \sigma_{\min}$.

Proof. It's easy to see A_k has right rank. For the right distance to A: $\|A - A_k\|_2 = \left\|\sum_{i=k+1}^n u_i\sigma_i v_i^\top\right\|_2 = \sigma_{k+1}$. We need to show A_k is the closest among rank k matrices. Suppose B has rank k, so the null space of B has dimension $n-k$. The space spanned by the $\{v_1, \cdots, v_{k+1}\}$ has dimension $k+1$. Since the sum of the dimensions of spaces $(n-k)+(k+1) = n+1 > n$, these two spaces intersect in some unit vector h. Then

$$\begin{aligned}
\|A - B\|_2 &\geq \|(A-B)h\|_2 \\
&= \|Ah\|_2 && \text{(since } h \in \operatorname{null}(B)\text{)} \\
&= \left\|U\Sigma V^\top h\right\|_2 \\
&= \left\|\Sigma \underbrace{V^\top h}_{x}\right\|_2 && \text{(since } U \text{ is orthogonal)} \\
&= \left\|\Sigma\,[x_1, \cdots, x_{k+1}, 0, \cdots, 0]^\top\right\|_2 && \text{(since } h \in \operatorname{span}\{v_1, \cdots, v_{k+1}\}\text{)} \\
&\geq \sigma_{k+1} && \text{(since } x \text{ is a unit vector)}
\end{aligned}$$

□

3.6 Condition Number

Now we start using this material to analyze the condition number for matrix inversion and solving $Ax = b$: If A (and b) change a little bit, how much can A^{-1} (and $x = A^{-1}b$) change?

If $|x| < 1$, recall that

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots$$

Now generalize to matrices.

Lemma 3.7

If operator $\|X\| < 1$, then $I - X$ is nonsingular and

$$(I-X)^{-1} = \sum_{i=0}^{\infty} X^i, \ \left\|(I-X)^{-1}\right\| \leq \frac{1}{1-\|X\|}$$

Proof. Claim $I + X + X^2 + \cdots$ converges:

$$\left\|X^i\right\| \leq \|X\|^i \to 0$$

as $i \to \infty$. So by equivalence of norms there is some $C > 0$ such that

$$\max_{kj} |(X^i)_{kj}| \leq C\left\|X^i\right\| \leq C\|X\|^i$$

and so kj-th entry of $I + X + X^2 + \cdots$ is dominated by a convergent geometric series and so converges. Claim that

$$(I-X)\left(I + X + X^2 + \cdots + X^i\right) = I - X^{i+1} \to I$$

as $i \to \infty$, and so $\sum_{i=0}^{\infty} X^i = (I-X)^{-1}$.

$$\begin{aligned}
\left\|I + X + X^2 + \cdots\right\| &\leq \|I\| + \|X\| + \left\|X^2\right\| + \cdots && \text{by triangle inequality} \\
&\leq \|I\| + \|X\| + \|X\|^2 + \cdots && \text{by } \|AB\| \leq \|A\|\|B\| \\
&= 1 + \|X\| + \|X\|^2 + \cdots \\
&= \frac{1}{1-\|X\|}
\end{aligned}$$

□

Lemma 3.8

Suppose A invertible. Then $A - E$ invertible if

$$\|E\| < \frac{1}{\|A^{-1}\|},$$

in which case

$$(A-E)^{-1} = Z + Z(EZ) + Z(EZ)^2 + Z(EZ)^3 + \cdots$$

where $Z = A^{-1}$ and

$$\left\|(A-E)^{-1}\right\| \leq \frac{\|Z\|}{1 - \|E\|\,\|Z\|}$$

Proof.

$$(A-E)^{-1} = ((I - EZ)A) = Z(I - EZ)^{-1}$$

exists if $I - EZ$ invertible i.e. if $\|EZ\| < 1$, i.e. if $\|E\|\,\|Z\| < 1$ in which case

$$(A-E)^{-1} = Z(1 + EZ + (EZ)^2 + \cdots)$$

Then take norms. □

Finally, we can ask how much A^{-1} and $(A-E)^{-1}$ differ.

Lemma 3.9

Suppose A invertible. If $\|E\| < 1/\|Z\|$, then

$$\left\|(A-E)^{-1} - Z\right\| \leq \frac{\|Z\|^2\,\|E\|}{1 - \|E\|\,\|Z\|}.$$

Proof.

$$(A-E)^{-1} - Z = Z(EZ) + Z(EZ)^2 + \cdots = ZEZ(I + EZ + (EZ)^2 + \cdots)$$

and then take norms. So the relative change in A^{-1} is

$$\begin{aligned}\frac{\left\|(A-E)^{-1} - Z\right\|}{\|Z\|} &\leq \left[\|A\|\,\|Z\| \frac{1}{1 - \|E\| * \|Z\|}\right]\left[\frac{\|E\|}{\|A\|}\right] \\ &= [\|A\|\,\|Z\|]\left[\frac{\|E\|}{\|A\|}\right] + O(\|E\|^2) \\ &= \text{condition number} \cdot \text{relative change in } A\end{aligned}$$

□

Definition 3.12. Condition number $\kappa(A) = \|A\|\,\|Z\|$.
Fact 3.15. $\kappa(A) \geq 1$.

Proof.

$$1 = \|I\| = \|AZ\| \leq \|A\| \, \|Z\|$$

□

Theorem 3.3

$$\min\left\{\frac{\|E\|}{\|A\|} : \; A - E \text{ singular}\right\} = \text{relative\_distance}(A, \text{singluar matrices}) = \frac{1}{\kappa(A)}$$

Proof for 2-norm, using SVD,

$$\min\{\|E\| : \; A - E \text{ singular}\} = \sigma_{\min}(A),$$

so the relative distance for A to singularity is

$$\frac{\sigma_{\min}(A)}{\|A\|_2} = \frac{\sigma_{\min}(A)}{\sigma_{\max}(A)}.$$

And

$$\begin{aligned}\left\|A^{-1}\right\|_2 &= \left\|(U\Sigma V^\top)^{-1}\right\|_2 = \left\|V\Sigma^{-1}U^\top\right\|_2 \\ &= \left\|\Sigma^{-1}\right\|_2 = \max_i |\Sigma_{ii}^{-1}| = \frac{1}{\sigma_{\min}}\end{aligned}$$

□

We've looked at sensitivity of A^{-1}, now look at solving $Ax = b$. Consider $Ax = b$ vs $(A-E)x' = b+f$ where $x' = x + \delta x$. Subtract to get

$$\begin{aligned}A\delta x - Ex - E\delta x &= f \\ (A - E)\delta x &= f + Ex \\ \delta x &= (A - E)^{-1}(f + Ex)\end{aligned}$$

$$\begin{aligned}\|\delta x\| &= \left\|(A - E)^{-1}(f + Ex)\right\| \\ &\leq \left\|(A - E)^{-1}\right\| (\|f\| + \|E\| \, \|x\|) \\ &\leq \frac{\left\|A^{-1}\right\|}{1 - \|E\| \, \|A^{-1}\|} (\|f\| + \|E\| \, \|x\|)\end{aligned}$$

$$\begin{aligned}\frac{\|\delta x\|}{\|x\|} &\leq \frac{\left\|A^{-1}\right\| \|A\|}{1 - \|E\| \, \|A^{-1}\|} \left(\frac{\|f\|}{\|A\| \, \|x\|} + \frac{\|E\|}{\|A\|}\right) \\ &\leq \frac{\kappa(A)}{1 - \|E\| \, \|A^{-1}\|} \left(\frac{\|f\|}{\|b\|} + \frac{\|E\|}{\|A\|}\right)\end{aligned}$$

Our algorithms will attempt to guarantee that we get

$$(A - E)\hat{x} = b + f \qquad (\star)$$

when $\frac{\|f\|}{\|A\|\|x\|}$ and $\frac{\|E\|}{\|A\|}$ both equal to $O(\varepsilon)$. Recall that property $(\star)$ is called "backward stability".

Another practical approach: given x', how accurate a solution is it? Compute residual

$$r = Ax' - b = Ax' - Ax = A(x' - x) = A \cdot \text{error}.$$

So error $= A^{-1}r$ and $\|\text{error}\| \leq \left\|A^{-1}\right\| \|r\|$. $\|r\|$ also determines the backward error in A.

Theorem 3.4

The smallest E such that $(A + E)x' = b$ has

$$\|E\| = \frac{\|r\|}{\|x'\|}$$

Proof. $r = Ax' - b = -Ex'$, so

$$\|r\| \leq \|E\| \left\|x'\right\|$$

To attain lower bound on $\|E\|$, choose

$$E = -rx'^{\top} / \left\|x'\right\|^2$$

in 2 norm. In other words, if $Ax' - b$ is small, then the backwards error is small, which is probably the most you should ask for when the entries of A are uncertain. □

All our practical error bounds depend on $\left\|A^{-1}\right\|$. To actually compute A^{-1} costs several times as much as just solving $Ax = b$ ($2n^3$ versus $(2/3)n^3$), so we will use cheap estimates of $\left\|A^{-1}\right\|$ that avoid computing A^{-1} explicitly, and work with small probability of large error.

The idea is to use the definition

$$\begin{aligned} ||A^{-1}|| &= \max_{||x||=1} ||A^{-1}x|| \\ &= \max_{||x||\leq 1} ||A^{-1}x|| \end{aligned}$$

and do gradient ascent ("go uphill") on $||A^{-1}x||$ on the convex set $||x|| \leq 1$; one may also start with a random starting vector. For right choice of norm it is easy to figure out ascent direction, and each step requires solving $Ax = b$ for some b, which only costs $O(n^2)$, (assuming we have already done LU factorization). In practice it usually takes at most 5 steps or so to stop ascending so the total costs is $O(n^2)$.

In fact there is a theorem (Demmel, Diament, Malajovich, 2000) that says that estimating $\kappa(A)$ even roughly, but still with some guarantee (say "to within a factor of 10", or within any constant factor) is as about expensive as multiplying matrices, which in turn is about as expensive as doing Gaussian elimination in the first place. Since our goal is to spend just an additional $O(n^2)$ to estimate $\left\|A^{-1}\right\|$ given that we have already done *Gaussian Elimination* (LU factorization), this theorem implies that we need to settle for a small probability of getting a large error in our estimate of $\left\|A^{-1}\right\|$.

4 Lecture 4: Real Cost of an Algorithm and Matrix Multiplication

4.1 Cost Analysis

Goals: Understand the real cost (in time) of running an algorithm, so we can design algorithms that run as fast as possible.

Traditionally we just count the number of arithmetic operations, but in fact multiplication and addition are the cheapest operations an algorithm performs: it can be orders of magnitude more expensive to get that data from wherever it is stored in the computer (say main memory) and bring it to the part of the hardware that actually does the arithmetic.

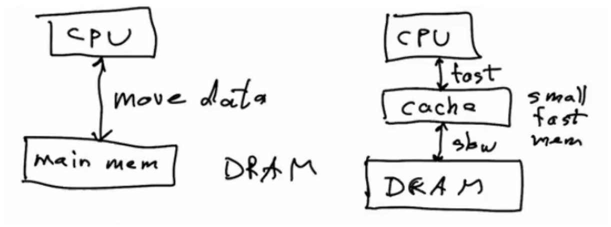


Figure 4.1: Simple models of sequential and parallel computers

For example, on a sequential computer, the main cost of a naive algorithm would be moving data between main memory (DRAM) and on-chip cache, and on a parallel computer, it is moving data[4] between processors connected over a network. We seek to minimize it if possible.

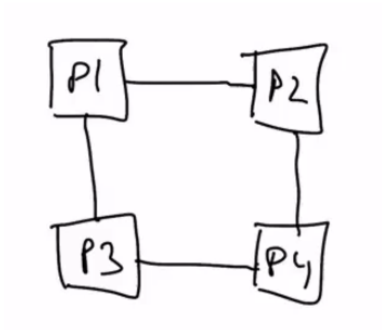


Figure 4.2: Moving data from p_i to p_j is slow

In the case of matrix multiplication, there is a theorem that gives a lower bound on the amount of communication required between DRAM and cache, assuming one can do the usual $O(n^3)$ operations in any order. And there is a well-known and widely-used algorithm that attains this lower bound. In 2011 we showed that this lower bound extends to any algorithm that "smells like" the 3-nested loops of conventional matrix multiplication, essentially covering all the usual linear algebra algorithms, for solving $Ax = b$, least squares, etc. ("Minimizing Communication in Numerical Linear Algebra") It turns out that the usual algorithms for these problem cannot always attain the lower bound, just by doing the same operations in a different order; instead one needs new algorithms. Some of these recently invented algorithms have given very large speedups ($O(\times)$) and are widely used.

[4]All forms of "moving data" are called "communication"

All these results (lower bounds and optimal algorithms) extend to other kinds of computer architectures, so with multiple layers of cache (so there is communication between every pair of of adjacent layers), and with parallel processors (so there is communication between different processors over a network). In this lecture we will talk about this lower bound and optimal algorithm for matrix multiplication, in the simplest case of having DRAM and cache. In later lectures, when we talk about more complicated linear algebra algorithms, we will just sketch how to redesign them to minimize communication.

There are many more algorithms, and computer architectures, to which these ideas could be applied, which are possible class projects. To get started, we want a simple mathematical model of what it costs to move data, so we can analyze algorithms easily and identify the one that is fastest. So we need to define two terms: bandwidth and latency.

First, to establish intuition, consider the time it takes to move cars on the freeway from Berkeley to Sacramento: Bandwidth measures how many cars/hour can get from Berkeley to Sacramento:

$$\#\text{cars/hour} = \text{density}(\#\text{cars/mile/lane}) \cdot \text{velocity}(\text{miles/hour}) \cdot \#\text{lanes}$$

Latency measures how long it takes for one car to get get from Berkeley to Sacramento:

$$\text{time} = \text{distance/velocity}$$

So the minimum time it takes n cars to go from Berkeley to Sacramento is when they all travel in a single "convoy", which is all as close together as possible given the density, and using all lanes:

$$\text{time} = \text{time for first car to arrive} + \text{time for remaining cars to arrive} = \text{latency} + n/\text{bandwidth}$$

The same idea (harder to explain the physics) applies to reading bits from main memory to cache: The data resides in main memory initially, but one can only perform arithmetic and other operations on it after moving it to the much smaller cache. The time to read or write w words of data from memory stored in contiguous locations (which we call a "message" instead of a convoy) is

$$\text{time} = \text{latency} + w/\text{bandwidth}$$

More generally, to read or write w words stored in m separate contiguous messages costs $m \cdot$ latency + w/bandwidth. Notation: We will write this cost as $m\alpha + w\beta$, and refer to it as the cost of "communication". We refer to w as *#words_moved* and m as *#messages*. We also let γ = time per flop, so if an algorithm does f flops, our estimate of the total running time will be:

$$\text{time} = m\alpha + w\beta + f\gamma$$

To reiterate our claim that communication is the most expensive operation:

1. $\gamma \ll \beta \ll \alpha$ on modern machines (factors of 10s or 100s for memory, even bigger for disk or sending messages between processors)
2. Gaps are increasing year over year, γ faster than β faster than α

And the same story holds for energy: the energy cost of moving data is much higher than doing arithmetic. So whether you are concerned about the battery in your cell phone dying, the O($1M) per megawatt per year it takes to run your data center, or how long your drone can stay airborne,

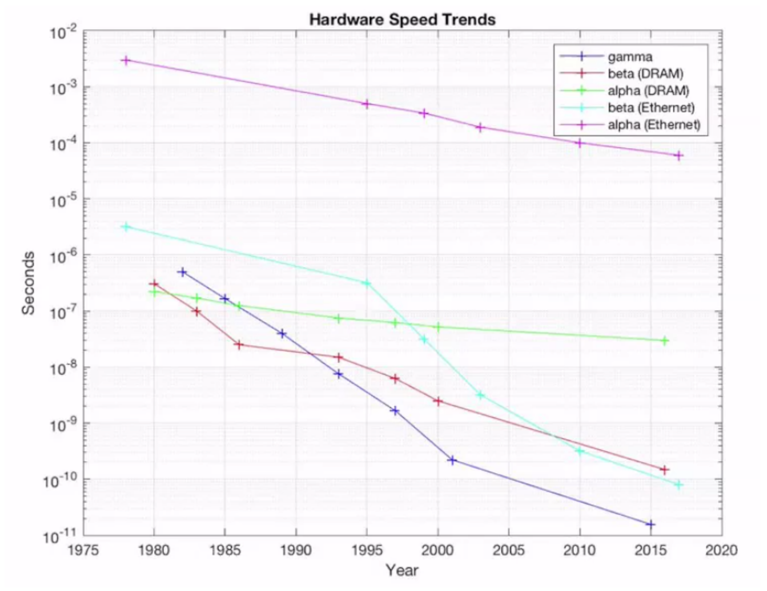


Figure 4.3: Hardware speed trends

you should minimize communication. So when we design or choose algorithms, we need to pay attention to their communication costs.

How do we tell if we have designed the algorithm well with respect to not communicating too much? If the $f\gamma$ term in the time dominates the communication cost $m\alpha + w\beta$:

$$f\gamma \geq m\alpha + w\beta$$

then the algorithm is at most 2 times slower than as though communication were free. When $f\gamma \gg m\alpha + w\beta$, the algorithm is running at near peak arithmetic speed, and can't go faster. When $f\gamma \ll m\alpha + w\beta$, communication costs dominate. Note that the Computational Intensity is $q = f/w$, "flops per word moved". This needs to be large to go fast, since $f\gamma > w\beta$ means $q = f/w > \beta/\gamma \gg 1$.

4.2 Some History

4.2.1 The Beginning and *EISPACK*

We will know we have done as well as possible if we can show that w and m are close to known lower bounds, that we will describe below. But first to describe how this trend has impacted algorithms over time, we give a little history. In the beginning, was the do-loop. This was enough for the first libraries (*EISPACK*, for eigenvalues of dense matrices, in the mid 1960s). People didn't worry about data motion, arithmetic was more expensive, they mostly just tried to get answer reliably, for $O(n^3)$ flops.

4.2.2 *BLAS-1* library (mid 1970s)

BLAS-1[5] was a standard library of 15 operations (mostly) on vectors, including:

[5]Basic Linear Algebra Subroutines

1. $y = \alpha x + y$, x and y vectors, α scalar ($AXPY$)

Algorithm 4.1 (Example: innermost loop in Gaussian Elimination).

```
1: for $k = i + 1 : n$ do
2:     $A_{jk} = A_{jk} - A_{ji} A_{ik}$
3: end for
```

2. Dot product
3. $\|x\|_2 = \sqrt{\sum_i x_i^2}$
4. Find largest entry in absolute value in a vector (for pivoting in Gaussian Elimination)

The motivations for the *BLAS-1* at the time were:

- easing programming
- readability, since these were commonly used functions
- robustness (e.g. avoid over/underflow in $\|x\|_2$)
- portable and efficient (if optimized on each architecture)

But there is no way to minimize communication in such simple operations, because they do only a few flops per memory reference, e.g. $2n$ flops on $2n$ data for a dot product, so with a computational intensity of $q = \frac{2n}{2n} = 1$. So if we implement matrix multiplication or Gaussian Elimination or anything else by loops calling $AXPY$, it will communicate a lot, the same as the number of flops.

4.2.3 *BLAS-2* library (mid 1980s)

This was a standard library of 25 operations (mostly) on matrix-vector pairs:

1. $y = \alpha y + \beta A x$ ($GEMV$[6]), with lots of variations for different matrix structures (symmetric, banded, triangular etc.) It allowed did obvious optimizations when α and/or $\beta = \pm 1, 0$
2. $A = A + \alpha x y^\top$ (GER[7]). It turns out the 2 innermost loops in GE can be written with GER (details later):

$$A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i)A(i, i+1:n)$$

3. Solve $Tx = b$ where T triangular (used by Gaussian Elimination to solve $Ax = b$) ($TRSV$). The motivation was similar to the *BLAS-1*, plus more opportunities to optimize on the vector computers of the day. But there is still not much reduction in communication, for example, *GEMV* reads $n^2 + 2n + 2$ words, writes n words, and does $2n^2 + 3n$ flops, for a computational intensity of about $q = 2$.

[6]General matrix vector multiply
[7]General rank one update

4.2.4 *BLAS-3* library (late 1980s)

This was a standard library of 9 operations on matrix-matrix pairs:

1. $C = \beta C + \alpha AB$ (*GEMM*[8])
2. $C = \beta C + \alpha AA^\top$ (*SYRK*[9])
3. Solve $TX = B$ where T triangular, X and B are rectangular matrices (*TRSM*)

Finally these have the potential for significant communication optimization, since for example the computational intensity q of *GEMM* applied to $n \times n$ matrices is:

$$q = \frac{f}{w} = \frac{2n^3}{3n^2\text{input} + n^2\text{output}} = \frac{n}{2}$$

But as we'll see, the straightforward 3-nested-loop version of *GEMM* is no better than *BLAS-2* or *BLAS-1*, so a different implementation is required, with very large speedups possible. In fact, there is an optimal way to implement *GEMM*, that provably does as little communication as possible (under certain assumptions). Note: *BLAS-k* does $O(n^k)$ operations, for $k = 1, 2, 3$, making the names easier to remember. These are supplied as part of the optimized math libraries on essentially all computers.

This led the community to seek algorithms for the rest of linear algebra (solving $Ax = b$, least squares, computing eigenvalues, SVD, etc.) that did much of their work by calling BLAS-3 routines, which would then run at high speed. This led to the *LAPACK* library (and its parallel version called *ScaLAPACK*) which is the basis of most dense linear algebra libraries provided by computer vendors, and used in many packages like *MATLAB*.

As stated earlier, we later discovered that the communication lower bounds attained by *GEMM* also apply to essentially the rest of linear algebra. Next we realized that the algorithms in *LAPACK* and *ScaLAPACK* usually did not attain these lower bounds, in fact the algorithms there could do asymptotically more communication than the bounds required. This in turn set off a search for new (or previously invented but ignored) algorithms that would attain these lower bounds. This has led to a number of new algorithms, some with large speedups ($O(10\times)$), and some still of just theoretical interest.

4.3 Analysis on Matrix multiplication

Goals: Prove communication lower bound for matrix multiplication

The following theorem for matrix multiplication was first proved in 1981 by Hong and Kung. The proof below is based on one by Irony, Tiskin and Toledo in 2004, which extends both to parallel algorithms, and to other 3-nested-loop-like algorithms.

[8]General matrix-matrix multiply
[9]Symmetric rank update

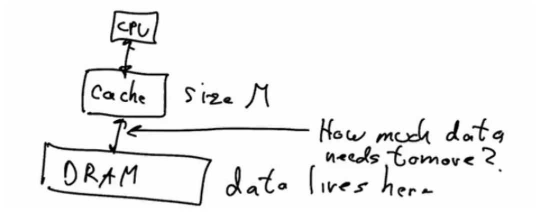


Figure 4.4: The goal

Theorem 4.1

Suppose one wants to multiply two nxn matrices $C = AB$ by doing the usual $2n^3$ multiplies and adds, on a computer with a main memory large enough to hold A, B and C, and a smaller cache of size M. Arithmetic can only be done on data stored in cache. Then a lower bound on the number of words W that need to move back and forth between main memory and cache to run the algorithm is

$$\Omega(\frac{n^3}{\sqrt{M}})$$

More generally, one does not need to multiply dense square matrices, the same proof works for rectangular and/or sparse matrices; the only change is that $\Omega(n^3/\sqrt{M})$ becomes $\Omega(\#\text{flops}/sqrtM)$.

Proof Sketch. Suppose we fill up the cache with M words of data, do as many flops as possible, store the results back in main memory, and repeat until we are done. Suppose we could upper bound by G the number of flops that are possible given M words of data. Then doing G flops would cost at least $2M$ words moved back and forth between main memory and cache. Since we have to do $2n^3$ flops altogether, we would need to repeat this at least $2n^3/G$ times, for a total cost of moving at least $(2n^3/G)2M$ words. So we need to find an upper bound G.

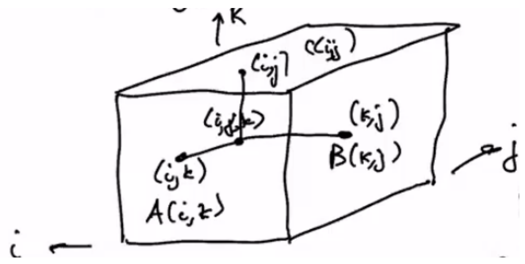


Figure 4.5: The geometry problem

We turn this into a geometry problem as follows. We represent each pair of flops (or inner loop iteration)

$$C(i, j) = C(i, j) + A(i, k)B(k, j)$$

as a lattice point (i, j, k) in 3D space, with $1 \leq i, j, k \leq n$, so an $n \times n \times n$ cube of points. The flops we can do with M words of data is represented by some subset V of this cube of lattice points. What data is required to execute the flops represented by (i, j, k)? It is $C(i, j)$, $A(i, k)$, and $B(k, j)$, which are represented by the 3 lattice points (i,j), (i,j) and (k,j) on 3 faces of the cube, i.e. the projections of (i,j,k) onto these three faces. We know we can have at most M words of data, i.e. M projected points. We now use a classical geometry theorem by Loomis and Whitney (1949), which says that if the set of lattice points is V whose cardinality $|V|$ we want to bound, and its 3 projections onto the faces of the cube are

- V_C (representing entries $C(i,j)$)
- V_A (representing entries $A(i,k)$)
- V_B (representing entries $B(k,j)$)

then $|V| \leq \sqrt{|V_A||V_B||V_C|}$. Since we can have at most M entries of A, B and C in cache (i.e. $|V_A| + |V_B| + |V_C| \leq M$), this means

$$|V| \leq \sqrt{M \cdot M \cdot M} = M^{3/2}$$

yielding our desired upper bound $G = M^{3/2}$. Finally, this yield our lower bound $W \geq (2 * n^3)/M^{3/2} 2M = \Omega(n^3/\sqrt{M})$ as claimed. We have not tried to be careful with the constant factors, but just tried to give the main idea. The best (nearly attainable) lower bound is actually $2n^3/\sqrt{M} - 2M$.

□

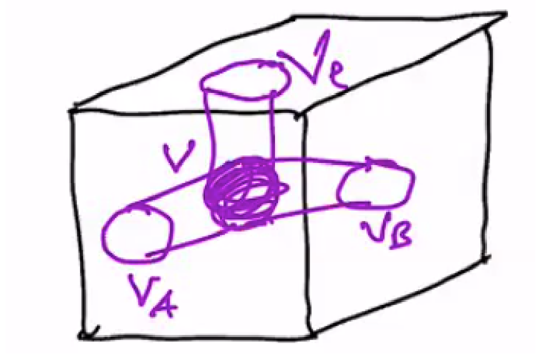


Figure 4.6: V_A, V_B and V_C

The proof also gives us a big hint as to how to design an optimal algorithm: What is the shape of the V that attains its upper bound $M^{3/2}$? Clearly V must be a cube of side length $\sqrt{M}$. This means that we will want to break A, B and C up into square submatrices of dimension at most $\sqrt{M/3}$, read 3 submatrices into cache (the factor of 3 means we can fit 3 such submatrices simultaneously), multiply them, and write the results back to main memory.

Algorithm 4.2.

```
1: Let $b$ be block size, small enough so $3b^2 \leq M$
2: Express $C$ as a block matrix where $C[i,j]$ is a $b \times b$ block, similarly for $A$ and $B$
3: for $i = 1 : n/b$ do
4:     for $j = 1 : n/b$ do
5:         read $C[i,j]$ into cache                    ▷ $b^2$ words moved
```

```
6:         for $k = 1 : n/b$ do
7:             read $A[i,k]$ and $B[k,j]$ into cache                ▷ $2b^2$ words moved
8:             $C[i,j] = C[i,j] + A[i,k]B[k,j]$   ▷ $b \times b$ matrix multiplication, so 3 more nested loops
9:         end for
10:        write $C[i,j]$ back to main memory                        ▷ $b^2$ words moved
11:    end for
12: end for
```

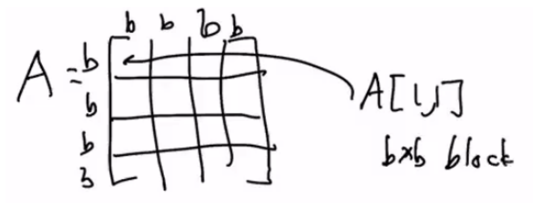


Figure 4.7: A into blocks

Counting the total number of words moved between cache and memory, we get

For reading $C[i,j]$	$(n/b)^2 b^2 = n^2$
For reading $A[i,k]$ and $B[k,j]$	$(n/b)^3 2b^2 = 2n^3/b$
For writing $C[i,j]$	$(n/b)^2 b^2 = n^2$

for a total of $2n^3/b + 2n^2$ words moved. We minimize this by making b as large as possible, i.e. $b = \sqrt{\frac{M}{3}}$, or the number of words moved $= O(n^3/\sqrt{M})$, which attains the lower bound (to within a constant factor).

What about "What about multiplying a $A \times B$ where B has just a few columns, or just one?" It is still 3 nested loops, but if B has fewer than b columns, then we can't break it into $b \times b$ blocks. But all the above lower bounds and optimal algorithms can be extended to the case of small loop bounds; the (attainable) lower bound becomes

$$\Omega(\max(\frac{\#\text{flops}}{\sqrt{M}}, \text{size(input)}, \text{size(output)}))$$

This approach to finding a communication lower bound and corresponding optimal algorithm has recently been extended to any algorithm that can be expressed as nested loops accessing arrays, with any number of loops, arrays, and subscripts, as long as the subscripts are "affine". The lower bound is always of the form:

$$\Omega(\frac{\#\text{loopiterations}}{M^e}),$$

where e is an exponent that depends on the problem ($e = 1/2$ for matrix multiplication). And the optimal block shapes can be general parallelograms. For linear algebra, we only need the special case described above. Here is another optimal matrix multiplication algorithm, that we will find simpler to generalize to other linear algebra problems, because it is "cache oblivious", i.e. it works for any cache size M, without needing to know M:

Algorithm 4.3 (Cache Oblivious).

```
1: Function $C = \text{RMM}(A, B)$[a b c]
2: Express $C$ as a block matrix where $C[i,j]$ is an $\frac{n}{2} \times \frac{n}{2}$ block
3: if $n = 1$ then
4:     $C = AB$
5: else
6:     write $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ where each $A_{ij}$ is $\frac{n}{2} \times \frac{n}{2}$
7:     write $B$ and $C$ similarly
8:     $C_{11} = \mathbf{RMM}(A_{11}, B_{11}) + \mathbf{RMM}(A_{12}, B_{21})$
9:     $C_{12} = \mathbf{RMM}(A_{11}, B_{12}) + \mathbf{RMM}(A_{12}, B_{22})$
10:    $C_{21} = \mathbf{RMM}(A_{21}, B_{11}) + \mathbf{RMM}(A_{22}, B_{21})$
11:    $C_{22} = \mathbf{RMM}(A_{21}, B_{12}) + \mathbf{RMM}(A_{22}, B_{22})$
12: end if
```

Cost: let $A(n)$ be the number of arithmetic operation on matrices of dimension n,

$$A(n) = 8A(\frac{n}{2}) + n^2$$

from the 8 recursive calls to **RMM** and the 4 additions of $\frac{n}{2} \times \frac{n}{2}$ matrices.

We claim that $A(n) = 2n^3 - n^2$, the same as the classical algorithm. Solve by changing variables from $n = 2^m$ to m

$$a(m) = 8a(m-1) + 2^{2m}$$

Divide by 8^m to get

$$\frac{a(m)}{8^m} = \frac{a(m-1)}{8^{m-1}} + 2^{-m}$$

Change variables again to $b(m) = a(m)/8^m$,

$$\begin{aligned} b(m) &= b(m-1) + 2^{-m} \\ &= \sum_{k=1}^{m} 2^{-k} + b(0) \end{aligned}$$

Let $W(n)$ be the number of words moved between slow memory and cache of size M,

$$\begin{aligned} W(n) &= 8w\left(\frac{n}{2}\right) + 12\left(\frac{n}{2}\right)^2 \\ &= 8w\left(\frac{n}{2}\right) + 3n^2 \end{aligned}$$

which looks bigger than $A(n)$. The base case is not at $W(1)$, rather

$$w(b) = 3b^2$$

[a]Recursive Matrix Multiplication
[b]Assume for simplicity that A and B are square of size n
[c]Assume for simplicity that n is a power of 2

if $3b^2 \leq M$, i.e. $b = \sqrt{\frac{M}{3}}$. Solve as before: change variables from $n = 2^m$ to m:

$$w(m) = 8w(m-1) + 3 \cdot 2^{2m}$$

Let $\bar{m} = \log_2 \sqrt{\frac{M}{3}}$. Divide by 8^m to get $v(m) = w(m)/8^m$:

$$v(m) = v(m-1) + 3 \cdot (1/2)^n$$
$$v(\bar{m}) = M/8^{\bar{m}} = M/(M/3)^{3/2} = 3^{3/2}/M^{1/2}$$

This is again a geometric sum but just down to $m = \bar{m}$, not 1:

$$\begin{aligned} v(m) &= \sum_{k=\bar{m}+1}^{m} 3(1/2)^k + v(\bar{m}) \\ &\leq 2 \cdot 3^{3/2}/\sqrt{M} \end{aligned}$$

So

$$W(n) = w(\log_2 n) = 8^{\log_2 n} v(\log_2 n) = 2 \cdot 3^{3/2} \frac{n^3}{\sqrt{M}}$$

as desired.

We mention briefly that the lower bound extends to the parallel case in a natural way: now the "cache" is the memory local to each processor, that it can access quickly, and "main memory" is the memory local to all the other processors, which is much slower to access. We assume each processor is assigned an equal fraction of the flops to do, $2\frac{n^3}{P}$, where P is the number of processors, and an equal fraction of the memory required to store all the data, so $M = 3\frac{n^2}{P}$. Then the same proof as above says that for each processor to perform $2\frac{n^3}{P}$ flops it needs to move

$$\Omega((n^3/P)/\sqrt{M}) = \Omega(n^2/\sqrt{P})$$

words into and out of its local memory. This lower bound is indeed attained by known algorithms, such as *SUMMA*[10].

4.4 The Strassen Algorithm

We can go asymptotically faster than n^3, not just to multiply matrices, but any linear algebra problem. There are many such algorithms, we discuss only the first one, which so far is the most practical one.

Matrix multiply is possible in $O(n^{\log_2 7}) \sim O(n^{2.81})$ operations. The algorithm is recursive, based on remarkable identities for multiplying 2×2 matrices.

Write $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ where each A_{ij} is $\frac{n}{2} \times \frac{n}{2}$, same for B and C.
$P_1 = (A_{12} - A_{22})(B_{21} + B_{22})$,
$P_2 = (A_{11} + A_{22})(B_{11} + B_{22})$,

[10] Scalable Universal Matrix Multiply Algorithm

$P_3 = (A_{11} - A_{21})(B_{11} + B_{12})$,
$P_4 = (A_{11} + A_{12})B_{22}$,
$P_5 = A_{11}(B_{12} - B_{22})$,
$P_6 = A_{22}(B_{21} - B_{11})$,
$P_7 = (A_{21} + A_{22})B_{11}$,
$C_11 = P_1 + P_2 - P_4 + P_6$,
$C_12 = P_4 + P_5$,
$C_21 = P_6 + P_7$,
$C_22 = P_2 - P_3 + P_5 - P_7$

Algorithm 4.4 (Strassen algorithm).

```
1: Function $C = \text{Strassen}(A, B)$[a][b][c]
2: Express $C$ as a block matrix where $C[i, j]$ is a $b \times b$ block
3: if $n = 1$ then
4:     $C = AB$
5: else
6:     $P_1 = \text{Strassen}(A_{11} - A_{22}, B_{21} + B_{22})$      ▷ And 6 more similar lines
7:     $C_{11} = P_1 + P_2 - P_4 + P_6$      ▷ And 3 more similar lines
8: end if
```

The arithmetic operations

$$A(n) = 7A\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$$

We solve as before: change variables from $n = 2^m$ to m, $A(n)$ to $a(m)$:

$$a(m) = 7a(m-1) + (9/2)2^{2m}$$

Divide by 7^m, change again to $b(m) = a(m)/7^m$,

$$b(m) = b(m-1) + (9/2) \cdot (4/7)^m$$

So

$$A(n) = a(\log_2 n) = b(\log_2 n)\, 7^{\log_2 n} = O(7^{\log_2 n}) = O(n^{\log_2 7})$$

About the number of words moved, again get a similar recurrence:

$$W(n) = 7W(n/2) + O(n^2),$$

and the base case $W(\bar{n}) = O(\bar{n}^2)$ when $\bar{n}^2 = O(M)$, i.e. the whole problem fits in cache. The solution is $O(n^x/M^{x/2-1})$ where $x = log_2 7$. Note that plugging in $x = 3$ gives the lower bound for standard matrix multiplication. Again, there is a theorem (with a very different proof), showing that this algorithm is optimal:

[a]Recursive Matrix Multiplication
[b]Assume for simplicity that A and B are square of size n
[c]Assume for simplicity that n is a power of 2

Theorem 4.2 (Demmel, Ballard, Holtz, Schwartz, 2010)
$\Omega(n^x/M^{x/2-1})$ is a lower bound on #words_moved for *Strassen* and algorithms "enough like" Strassen, where $x = \log_2 7$ for *Strassen*, and whatever it is for "Strassen-like" methods.

This result was generalized to even more "Strassen-like" methods in a PhD Thesis (Scott, 2015). Strassen is not (yet) much used in practice, but it can pay off for n not too large (a few hundred). The current world's record is by Le Gall (2014): $O(n^{2.3728639})$, but it is impractical: would need huge n to pay off.

Its error analysis slightly worse than usual algorithm: Usual matrix multiplication:

$$|\operatorname{fl}(AB) - (AB)| \leq n\varepsilon |A||B|$$

For *Strassen*:

$$\|\operatorname{fl}(AB) - (AB)\| \leq O(\varepsilon)|A||B|$$

which is good enough for lots of purposes. But when can *Strassen*'s error bound be much worse? (suppose one row of A, or one column of B, is very tiny). *Strassen* is also not allowed to be used in the "LINPACK Benchmark" which ranks machines by

$$\text{speed} = (2/3)n^3/\operatorname{time\_for\_GEPP}(n).$$

And if you use *Strassen* in *GEPP*, it does far smaller than $(2/3)n^3$ flops so the computed "speed" could exceed actually peak machine speed. As a result, vendors stopped optimizing *Strassen*, even though it's a good idea!

A similar Strassen-like trick can be used to make complex matrix multiplication a constant factor cheaper than you think. Normally, to multiply two complex numbers or matrices, you use the formula

$$(A + iB)(C + iD) = (AC - BD) + i(AD + BC)$$

costing 4 real multiplications and 2 real additions. Here is another way: $T1 = AC$,
$T2 = BD$,
$T3 = (A + B)(C + D)$.
Then it turns out that

$$(A + iB)(C + iD) = (T_1 - T_2) + i(T_3 - T_1 - T_2)$$

for a cost of 3 real multiplications and 5 real additions. But since matrix multiplication costs $O(n^3)$ or $O(n^x)$ for some $x > 2$, and addition costs $O(n^2)$, for large n the cost drops by a factor 3/4. The error analysis is very similar to the usual algorithm. Applying the above formula recursively in a different context yields an algorithm for multiplying two n-bit integers in $O(n^{\log_2 3}) = O(n^{1.59})$ bit operations, as opposed to the conventional $O(n^2)$.

5 Lecture 5: Gaussian Elimination

5.1 Gaussian Elimination

Exploit Structure:A symmetric, positive definite "sparse" depends on $\ll n^2$ parameters, so could have lots of zero entries, or be dense but depend on few parameters, e.g. Vandermonde.

Seek Matrix Factorizations: $A =$ product of simple matrices

- $A = U\Sigma V^\top =$ orthogonal $\cdot$ diagonal $\cdot$ orthogonal
- Gaussian Elimination: $A = PLU$
- For solving $Ax = b$, Least Squares: $A = QR$, Q orthogonal, R upper triangular
- eigenproblems: $A = QTQ^H$, Q unitary, T upper triangular

Definition 5.1 (permutation Matrix)**.** Identity matrix with permuted rows.
Fact 5.1. To store and multiply by P requires store locations of 1 (cheap)

1. P, P_1, P_2 permutation matrices. P has exactly one 1 in each row and each column.
2. $PX = X$ with permuted rows
3. $XP = X$ with permuted columns
4. $P_1P_2 =$ permutation
5. $P^{-1} = P^\top$ i.e. P orthogonal (check: $PP^\top$ has unit diagonal)
6. $\det(P) = \pm 1$

Theorem 5.1 (LU Decomposition)

Given any $m \times n$ full rank A, $m \geq n$, $\exists$ $m \times m$ permutation P, $m \times n$ unit lower triangular L, $n \times n$ nonsingular upper triangular U such that $A = PLU$.

Proof. If A is full rank, the first column is nonzero, so there is a permutation P such that $(PA)_{11}$ is nonzero. Write

$$\begin{aligned} PA &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ \frac{A_{21}}{A_{11}} & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} - \frac{A_{21}A_{12}}{A_{11}} \end{bmatrix} \end{aligned}$$

where A_{11} is 1×1, A_{21} is $(m-1) \times 1$, A_{12} is $1 \times (n-1)$ and A_{22} is $(m-1) \times (n-1)$. Now A full (column) rank $\implies PA$ full rank $\implies S = A_{22} - \frac{A_{21}A_{12}}{A_{11}}$ is full rank[a]. (Otherwise, if some nonzero linear combination of columns of S were 0, say $Sx = 0$, then a linear combination of columns of A would be zero, $A\,[-A_{12}x/A_{11}; x] = 0$, contradicting A being full column rank.

More simply in the square case:

$$\begin{aligned}0 = \det(A) &= \pm\det(PA) = \det(\text{first factor})\cdot\det(\text{second factor})\\ &= 1\cdot A_{11}\cdot\det(S)\\ &\implies \det(S)\neq 0\end{aligned}$$

Now apply induction: $S = P'L'U'$, so

$$\begin{aligned}PA &= \begin{bmatrix} 1 & 0 \\ \frac{A_{21}}{A_{11}} & I \end{bmatrix}\begin{bmatrix} A_{11} & A_{12} \\ 0 & P'L'U' \end{bmatrix}\\ &= \begin{bmatrix} 1 & 0 \\ \frac{A_{21}}{A_{11}} & P'L' \end{bmatrix}\begin{bmatrix} A_{11} & A_{12} \\ 0 & U' \end{bmatrix}\\ &= \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix}\begin{bmatrix} 1 & 0 \\ P'^{\top}\frac{A_{21}}{A_{11}} & L' \end{bmatrix}\begin{bmatrix} A_{11} & A_{12} \\ 0 & U' \end{bmatrix}\end{aligned}$$

$$A = P^{\top}\begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix}\begin{bmatrix} 1 & 0 \\ P'^{\top}\frac{A_{21}}{A_{11}} & L' \end{bmatrix}\begin{bmatrix} A_{11} & A_{12} \\ 0 & U' \end{bmatrix}$$

where the 3 matrices are permutation, unit lower triangular and upper triangular respectively.

□

Corollary 5.1

If A $n\times n$ and nonsingular, there exist an $n\times n$ permutation P, unit lower triangular L, and nonsingular upper triangular U such that $A = PLU$

To solve $Ax = b$:

1. factor: $A = PLU$ (expensive part, cost $= \frac{2}{3}n^3 + O(n^2)$)
2. Solve $PLUx = b$ for $LUx = P^{\top}b$ by permuting b, cost $= O(n)$
3. Solve $LUx = P^{\top}b$ for $Ux = L^{-1}P^{\top}b$ by forward substitution with L, cost $= n^2$
4. Solve $Ux = L^{-1}P^{\top}b$ for $x = U^{-1}L^{-1}P^{\top}b = A^{-1}b$ using back substitution with U, cost $= n^2$

If we are given another b', can solve $Ax' = b'$ in just $O(n^2)$ flops.
Note: We do not compute A^{-1} and multiply $x = A^{-1}b$ because

1. Three times more expensive in dense case (can be $O(n^2)$ more expensive in sparse case)
2. Not as numerically stable

How to pivot, i.e. choose A_{ii} to put on diagonal, goal being back stability:

$$PLU = A + E,\ \|E\| = O(\varepsilon)\|A\|$$

This not guaranteed by $A_{ii}\neq 0$.

[a] S is the Schur complement

5.2 Gaussian Elimination in Algorithm

Algorithm 5.1.

```
for $i = 1 : n$ do
    if $i = 1$ then
        Performs algebra shown above,
    else
        if $i > 1$ then
            Applies the same algorithm recursively to the Schur complement
        end if
    end if
    $L(i,i) = 1$
    $L(i+1:n, i) = A(i+1:n, i)/A(i,i)$                ▷ Ignore permutations for now
    $U(i, i:n) = A(i, i:n)$
    if $i = 1$ then
        $A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - L(i+1:n, i)U(i, i+1:n)$
    end if
end for
```

Add permutations after the first "for loop", add:

Algorithm 5.2.

```
if $A(i,i) = 0$ and $A(j,i) \neq 0$ then
    swap rows $i$ and $j$ of $L$ and $A$; record swap    ▷ How to choose $A(j,i)$ called "pivoting"
end if
```

Don't waste space:

- row i of U overwrites row i of A: omit $U(i, i:n) = A(i, i:n)$
- col i of L (below diagonal) overwrites same entries of A, which are zeroed out: change first line to $A(i+1:n, i) = A(i+1:n, i)/A(i,i)$
- only need to loop from $i = 1$ to $n-1$, and change last line from

$$A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - L(i+1:n, i)U(i, i+1:n)$$

  to

$$A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i)A(i, i+1:n)$$

Finally, we get

Algorithm 5.3.

```
for $i = 1 : n-1$ do
    if $A(i,i) = 0$ and $A(j,i) \neq 0$ then
        swap rows $i$ and $j$ of $L$ and $A$; record swap
        $A(i+1:n, i) = A(i+1:n, i)/A(i,i)$                ▷ call to BLAS1 routine scal
        $A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i)A(i, i+1:n)$  ▷ call to BLAS2 routine GER
```

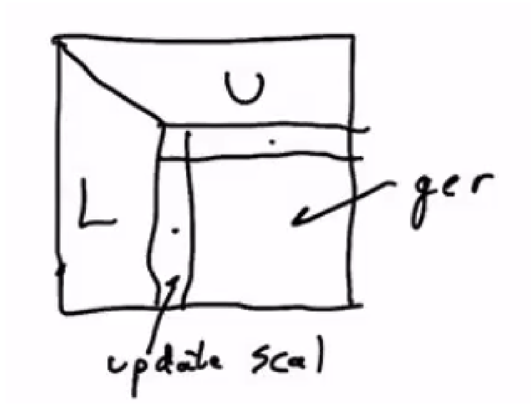


Figure 5.1: Intermediate step

```
6:     end if
7: end for
```

When we're done:

- U overwrites upper triangle and diagonal of A;
- L (below diagonal) overwrites A (below diagonal), the unit diagonal entries $L(i,i) = 1$ are not stored.

To see that this is the same Gaussian Elimination you learned long ago, start from

Algorithm 5.4.

```
1: for $i = 1 : n-1$ do                                  ▷ For each column $i$
2:     for $j = i+1 : n$ do    ▷ Add a multiple of row $i$ to row $j$ to zero out entry $(j,i)$ below diagonal
3:         $m = A(j,i)/A(i,i)$
4:         $A(j, i:n) = A(j, i:n) - m * A(i, i:n)$
5:     end for
6: end for
```

"Optimize" this by

1. Not bothering to compute the entries below the diagonal you know are zero: change last line to $A(j, i+1:n) = A(j, i+1:n) - m * A(i, i+1:n)$
2. Compute all the multipliers m first, store them in zeroed-out locations:

   Algorithm 5.5.

   ```
   1: for $i = 1 : n-1$ do
   2:     for $j = i+1 : n$ do
   3:         $A(j,i) = A(j,i)/A(i,i)$
   4:     end for
   5:     for $j = i+1 : n$ do
   ```

```
6:         $A(j,i+1:n) = A(j,i+1:n) - A(j,i)A(i,i+1:n)$
7:     end for
8: end for
```

3. Combine loops into single expressions to get same as before:

Algorithm 5.6.

```
1: for $i = 1 : n-1$ do
2:     $A(i+1:n,i) = A(i+1:n,i)/A(i,i)$
3:     $A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i)A(i,i+1:n)$
4: end for
```

The cost is $\sum_{i=1}^{n-1} 2(n-i)^2 = (2/3)n^3 + O(n^2)$ multiplies and adds.

We need to do pivoting carefully, i.e. choosing which nonzero to put on the diagonal, even if $A(i,i)$ is nonzero, to get a backward stable result, i.e. $PLU = A + E$ where E is small compared to A.

Example 5.1 (Single Precision)

If we run in single precision, so with 7 decimal digits, and take

$$A = \begin{bmatrix} 10^{-8} & 1 \\ 1 & 1 \end{bmatrix}, A^{-1} \approxeq \begin{bmatrix} -1 & 1 \\ 1 & -10^{-8} \end{bmatrix}$$

then $\kappa(A) \sim 2.6$, really small, so we expect a good answer. But

$$L = \begin{bmatrix} 1 & 0 \\ 10^8 & 1 \end{bmatrix}, U = \begin{bmatrix} 10^{-8} & 1 \\ 0 & \underbrace{\mathrm{fl}\left(1 - 10^{8.1}\right)}_{-10^8} \end{bmatrix}$$

thus

$$LU = \begin{bmatrix} 10^{-8} & 1 \\ 1 & 0 \end{bmatrix}$$

which is very different from A in the $(2,2)$ entry. In fact, we'd get the same (wrong) L and U if $A(2,2)$ were .5, -1, etc. because the operation $\mathrm{fl}(A(2,2) - 1\mathrm{e}8 * 1)$ "forgets" $A(2,2)$ if $A(2,2)$ is small enough, say $O(1)$. So going on to solve $Ax = b$ using L and U will give very wrong answers.

If we instead pivot (swap rows 1 and 2) so $A(1,1) = 1$, then we get full accuracy. The intuition is that we want to pick a large entry of A to be the "pivot" A_{11}, and repeat this at each step, for reasons we formalize below. To motivate the statement of the result, recall HW 1.10, where you showed $C = \mathrm{fl}(AB) = AB + E$ where

$$|E| \leq n\varepsilon|A||B|.$$

Since $A = PLU$ in exact arithmetic, the following theorem shoule be no surprise.

Theorem 5.2 (Backward error analysis of LU decomposition)

If P, L and are computed by the above algorithm, then

$$A - E = PLU$$

where $|E| \leq n\varepsilon P|L||U|$.

Proof. Recall that for simplicity we assume $P = I$. If we trace through the algorithm and ask how $U(i,j)$ is computed, we see that we start with $A(i,j)$ when $i \leq j$ and repeatedly subtract $L(i,k)U(k,j)$ for $k = 1, 2, \cdots, i-1$ until we get

$$U(i,j) = A(i,j) - \sum_{k=1}^{i-1} L(i,k)U(k,j)$$

which, not surprisingly, is what you get when you take the (i,j) entry of $A = LU$ and solve for $U(i,j)$. So $U(i,j)$ is essentially computed by a dot product of the (previously computed) i-th row L and j-th column of U, and using the same error analysis approach for dot products as in HW 1.10, we get the result.

Similarly, when $i > j$ we get $L(i,j)$ by starting with $A(i,j)$ and subtracting $L(i,k)U(k,j)$ for $k = 1, 2, \cdots, j-1$ dividing the resulting sum by $A(i,i)$ or

$$L(i,j) = (A(i,j) - \sum_{k=1}^{j-1} L(i,k)U(i,j))/A(i,i)$$

so again we have a dot product, followed by a division, and a similar approach works. □

Corollary 5.2

Solve $Ax = b$ by Gaussian elimination, following by forward and back substitution with L and U as described above. Then the computed results $\hat{x}$ satisfies

$$(A - F)\hat{x} = b$$

where $|F| \leq 3n\varepsilon P|L||U|$.

Proof. Here we assume $P = I$ for simplicity of notation (imagine running without pivoting on the matrix $P^\top A$). In HW 1.11, you showed that the computed solution $\hat{y}$ of $Ly = b$ satisfied

$$(L + \delta L)\hat{y} = b$$

with $|\delta L| \leq n\varepsilon|L|$, and that the computed solution $\hat{x}$ of $Ux = \hat{y}$ satisfied

$$(U + \delta v)\hat{x} = \hat{y}$$

with $|\delta U| \leq n\varepsilon|U|$. Thus

$$\begin{aligned} b &= (L+\delta L)\hat{y} = (L+\delta L)(U+\delta U)\hat{x} \\ &= (LU + (\delta L)U + L(\delta U) + (\delta L)(\delta U))\hat{x} \\ &= (A - E + (\delta L)U + L(\delta U) + (\delta L)(\delta U))\hat{x} \\ &= (A-F)\hat{x} \end{aligned}$$

where

$$\begin{aligned} |F| &\leq |E| + |(\delta L)U| + |L(\delta U)| + |(\delta L)(\delta U)| \\ &\leq |E| + |\delta L||U| + |L||\delta U| + |\delta L||\delta U| \\ &\leq \underbrace{n\varepsilon|L||U|}_{\text{by theorem}} + \underbrace{n\varepsilon|L||U|}_{\text{by bound on } |\delta L|} + \underbrace{n\varepsilon|L||U|}_{\text{by bound on } |\delta U|} + \underbrace{n^2\varepsilon^2|L||U|}_{\text{by both bounds}} \\ &= \left(3n\varepsilon + 0\left(\varepsilon^2\right)\right)|L||U| \end{aligned}$$

□

What the theorem and the corollary tell us is that we need

$$\|F\| \leq 3n\,\|\varepsilon|L||U|\| = O(\varepsilon\,\|A\|)$$

for the algorithm to be backward stable, i.e.

$$\||L||U|\| = O(\|A\|)$$

This in turn depends on pivoting carefully.

Now we discuss how to pivot so that $\||L||U|\| = O(\|A\|)$, or not much larger.
Definition 5.2. We call

$$g = \frac{\||L||U|\|}{\|A\|}$$

the pivot growth factor.[11]

The unstable 2×2 example above, where $L(2,1) = 1/1\text{e}-8 = 1\text{e}8$, suggests that we choose the pivot A_{11} to be as large as possible, so entries of L are as small as possible.

1. Simplest, and standard, approach, used in most libraries: "Partial pivoting" (also called *GEPP*). At each step, permute rows so A_{ii} is the largest (in absolute value) entry among $A(i:n,i)$. Then $L_{21} = A_{21}/A_{11}$ has all entries bounded by 1 in absolute value.

   - Bad news: in the worst case: even for $n = 24$ in singular precision, all wrong
   - Good news: hardly ever happens (only very small family of matrices where this occurs)

   Empirical observation, with some justification: $g < n^{2/3}$. If all entries of matrix were "random", this would be true; as you perform pivoting, they seem to get more random.

[11]This is defined somewhat differently in the literature, but is very similar.

2. Complete pivoting (*GECP*): permute rows and columns so that A_{11} largest entry in the whole matrix; again repeat at every step. Get $A = P_r LUP_c$ where P_r and P_c are both permutations.
    - Theoretical: $g < n^{\log n/4}$
    - Empirical: $g < n^{1/2}$
    - Long-standing Conjecture: $g < n$ (false, but nearly true)

    This is more expensive, hardly used, not in most libraries.
3. Tournament pivoting: something new, needed to minimize communication.
4. Threshold pivoting: this and similar schemes try to preserve sparsity while maintaining stability.

Altogether, our worst-case error bound for $Ax = b$ is

$$\begin{aligned}\frac{\|x-\hat{x}\|}{\|x\|} &\leq \kappa(A)\cdot \text{backward error}\\ &\leq \kappa(A)3n\varepsilon\cdot \text{pivot growth}\\ &\leq \kappa(A)3n\varepsilon g||A||\end{aligned}$$

where we can estimate $\kappa(A)$ with $O(n^2)$ flops after LU decomposition, and can bound pivot growth in $O(n^2)$ work too.

Theorem 5.3
With *GEPP*, $|L| \leq 1$ and

$$\max(|U(:,i)|) \leq 2^{n-1}\max(|A(:,i)|)$$

What if this error is too large for your application, or too slow? We can run iterative refinement, as known as *Newton's method*, using mixed precision, doing most of the work (the $O(n^3)$ part) in low (fast) precision, and a little more ($O(n^2)$) in high precision. In the following algorithm, low/high precision could mean single/double, half/single, bfloat16/single, double/quad, or other combinations:

Algorithm 5.7 (Do GEPP to solve $Ax = b$ in low precision).

```
1: $i = 1$
2: while not converged do
3:     $r = Ax_i - b$ [a]
4:     Solve $Ad = r$ [b]
5:     Update $x_{i+1} = x_i - d$ [c]
6: end while
```

[a]In high precision, but costs just $O(n^2)$; round final result r to low precision
[b]In low precision using existing LU factors, costs just $O(n^2)$
[c]In low precision, costs $O(n)$

We compute r in high precision, because otherwise the computed residual r may be mostly roundoff error, so the correction d is mostly noise, and there is no guarantee of progress (though some benefits have been proven, and so both versions are in *LAPACK*).

Testing "convergence" depends on one's goals. We mention two:

1. Getting a small backward error in high precision:

$$\|Ax_{\text{computed}} - b\| = O(\varepsilon_{\text{high}})(\|A\| \, \|x_{\text{computed}}\| + \|b\|)$$

   or a warning that the matrix is too ill-conditioned to converge. This is straightforward to implement, since we need to compute the residual anyway. This is motivated by the availability of hardware accelerators for machine learning (from Google, Nvidia, Intel, etc.) that can do 16-bit arithmetic much faster than higher precisions. Some of them also accumulate dot-products internally in 32-bit precision, giving us the high precision residual automatically. There is also recent work on alternatives to the simple *Newton's iteration* above, with better convergence behavior, see "A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic", or the recent Nvidia blog post.

2. Getting a small relative error in low precision:

$$\frac{\|x_{\text{computed}} - x_{\text{true}}\|}{\|x_{\text{true}}\|} = O(\varepsilon_{\text{low}})$$

   or a warning that the matrix is too ill-conditioned to do this. This is tricky, because we have to avoid getting fooled by a very ill-conditioned matrix that appears to "accidentally" converge. [12] Slides 33-35 show empirical results for millions of randomly generated test cases, using single/double, with condition numbers ranging from 1 to well beyond $1/\varepsilon_{\text{low}}$: The relative error of LU without refinement is indeed usually close to condition_number $* \, \varepsilon_{\text{low}}$, but LU with refinement is much better: the relative error is $O(\varepsilon_{\text{low}})$ as long as the condition number is less than about $1/\varepsilon_{\text{low}}$, which the algorithm reports to the user; when the condition number is larger, the relative error can rise as high as 1, and this lack of convergence is also reported to the user. This algorithm is available in *LAPACK* as *sgesvxx*, *dgesvxx*, etc.

5.3 Minimizing Communication

Now we return to minimizing communication. Historically, *GEPP* was rewritten to do most of its work by calling the matrix multiplication routine in the *BLAS-3*, which often led to high performance in libraries like *LAPACK* and *ScaLAPACK*. The idea is based on a similar induction proof as for classical LU decomposition, but instead of working on one column at a time, the algorithm works b columns at a time, where b is a block size analogous to one used in matrix multiplication. Ignoring pivoting for simplicity we write

$$\begin{aligned} A &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \\ &= \begin{bmatrix} L_{11}U_{11} & A_{12} \\ L_{21}U_{11} & A_{22} \end{bmatrix} \end{aligned}$$

[12] For details see `www.netlib.org/lapack/lawnspdf/lawn165.pdf`.

where A_{11} is b-by-b, A_{12} is b-by-$(n-b)$, A_{21} is $(n-b)$-by-b and A_{22} is $(n-b)$-by-$(n-b)$. And we have performed LU decomposition on the matrix

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11} \Rightarrow A = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & A_{22} \end{bmatrix}$$

where we have solved the triangular system of equations $A_{12} = L_{11}U_{12}$ for U_{12} by calling *BLAS-3* routine *TRSM*.

$$= \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & \underbrace{A_{22} - L_{21}U_{12}}_{\text{Schur complement } S} \end{bmatrix}$$

where we compute S by calling *BLAS-3* routine *GEMM*.

Since most of the work is done by the *BLAS-3*, it turns out that there are some combinations of dimension n and cache size M for which b cannot be chosen to attain the lower bound. Just as there was a recursive, cache oblivious version of matrix multiplication that minimized communication without explicitly depending on the cache size M, we can do the same for LU.

Here is a high level description of the algorithm:

- Do LU on left half of matrix
- Update right half (U at top, Schur complement at bottom)
- Do LU on Schur complement

Algorithm 5.8 (Recursive LU, Toledo, 1997).

```
1: Assume $A$ is $n \times m$ with $n \geq m$, $m$ a power of 2
2: if $m = 1$ then                                        ▷ One column
3:     Pivot so largest entry on diagonal, update rest of matrix
4:     $L = A/A_{11}$, $U = A_{11}$
5: else
6:     write $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $L_1 = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix}$, where $A_{11}$, $A_{12}$, $L_{11}$, $U_1$ and $U_2$ are $m/2$-by-$m/2$, $A_{21}$, $A_{22}$ and $L_{12}$ are $(n-m/2)$-by-$m/2$
7:     $[L_1, U_1] = \mathrm{RLU}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}\right)$        ▷ LU of left half of $A$
8:     Solve $A_{12} = L_{11}U_{12}$          ▷ Update $U$, in upper part of right half of $A$
9:     $A_{22} = A_{22} - L_{21}U_{12}$                 ▷ Update Schur complement
10:    $[L_2, U_2] = \mathrm{RLU}(A_{22})$                 ▷ LU on Schur complement
11:    $L = [L1, [0; L_2]^\top]$                ▷ return complete $n \times m$ $L$ factor
12:    $U = \begin{bmatrix} U_1 & U_{12} \\ 0 & U_2 \end{bmatrix}$               ▷ Return complete $m \times m$ $U$ factor
13: end if
```

Correctness follows by induction on m, as does showing when $m = n$:

$$A(n) = \#\text{arithmetic operations} = (2/3)n^3 + O(n^2),$$
$$W(n) = \#\text{words moved} = O(n^3/\sqrt{M})$$

As stated, this algorithm only minimizes the #words moved, not the #messages. To minimize #messages, we either

1. replace partial pivoting by tournament pivoting[13];
2. keep *GEPP*, but more complicated data structure[14].

Fact 5.2. If we do $L_{21}U_{12}$ by *Strassen*, and $L_{11}\backslash A_{12}$ by

1. inverting L_{11} by divide-and-conquer:
$$\begin{bmatrix} T_{11} & T_{12} \\ & T_{22} \end{bmatrix}^{-1} = \begin{bmatrix} T_{11}^{-1} & -T_{11}^{-1}T_{12}T_{22}^{-1} \\ 0 & T_{22}^{-1} \end{bmatrix}$$
2. multiplying $L_{11}^{-1}A_{12}$ etc. by *Strassen*

then the RLU algorithm costs $O(n^{\log_2 7})$ like *Strassen*, but can be slightly less stable than usual $O(n^3)$ version of *GEPP*.[15]

For the implementation of the algorithms, see the last page of the notes.

[13] "CALU: A communication optimal LU Factorization Algorithm"
[14] "Communication efficient Gaussian elimination with Partial Pivoting using a Shape Morphing Data Layout"
[15] See `arxiv.org/abs/math.NA/0612264`.

6 Lecture 6: Gaussian Elimination for Matrices With Special Structures

6.1 Gaussian Elimination for Special Structure

Goal: Save flops and memory

- Symmetric positive definite matrices (Cholesky) save half flops, space vs Gaussian elimination with partial pivoting
- Symmetric matrices save half flops, space vs Gaussian elimination with partial pivoting
- Band matrices cost goes from $O(n^3)$ to $O(\mathrm{bw}^2\, n)$, space from n^2 to $O(\mathrm{bw}\, n)$
- Sparse matrices cost, space can drop a lot, depends a lot on sparsity pattern complicated algorithms, many software libraries available
- "Structured" matrices are dense, but depend on $O(n)$ parameters, e.g.
  - Vandermonde $V_{ij} = x_i^{j-1}$, arising in polynomial interpolation
  - Toeplitz $T_{ij} = t_{i-j}$, so constant along diagonals, arising in time series

$$\begin{bmatrix} B_{11} & B_{12} & 0 & \cdots & \cdots & 0 \\ B_{21} & B_{22} & B_{23} & \ddots & \ddots & \vdots \\ 0 & B_{32} & B_{33} & B_{34} & \ddots & \vdots \\ \vdots & \ddots & B_{43} & B_{44} & B_{45} & 0 \\ \vdots & \ddots & \ddots & B_{54} & B_{55} & B_{56} \\ 0 & \cdots & \cdots & 0 & B_{65} & B_{66} \end{bmatrix}$$

Figure 6.1: A band matrix (tridiagonal) with bw = 1

6.2 Symmetric (Hermitian) Positive Definite

Definition 6.1. A is real and symmetric positive definite if and only if $A = A^\top$ and $x^\top Ax > 0$ for all $x \neq 0$.
A is complex and Hermitian positive definite if and only if $A = A^H$ and $x^H Ax > 0$ for all $x \neq 0$.

Lemma 6.1 (Just for real case)

1. X nonsingular implies that A is symmetric positive definite if and only if $X^\top AX$ is symmetric positive definite.

   Proof. A is symmetric positive definite and $x \neq 0$ implies that $Xx \neq 0$. Then $0 \neq$

$(Xx)^\top A\,(Xx)$, which means $X^\top AX$ is symmetric positive definite. □

2. A is symmetric positive definite and $H = A(j:k, j:k)$ implies that H is symmetric positive definite (H called a "principal submatrix").

   Proof. A is symmetric positive definite and $y \neq 0$ implies that

$$0 \neq x = \begin{bmatrix} 0 \\ y \\ 0 \end{bmatrix},$$

   where y in rows j to k. Then,

$$x^\top Ax = y^\top Hy \neq 0$$

□

3. A is symmetric positive definite if and only if $A = A^\top$ and all $\lambda_i > 0$.

   Proof. A is symmetric implies that $AQ = Q\Lambda$ where $Q^{-1} = Q^\top$ and $\Lambda = \operatorname{diag}(\lambda_1, \cdots, \lambda_n)$. $A = Q\Lambda Q^\top$ so A is symmetric positive definite if and only if Λ is symmetric positive definite. □

4. A is symmetric positive definite implies that $A_{ii} > 0$ and

$$\max_{i,j} |A_{ij}| = \max_i |A_{ii}|$$

   i.e. the largest entry is on diagonal.

   Proof. Let

$$e_i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix},$$

   then $e_i^\top Ae_i = A_{ii}$, so $A_{ii} > 0$. Suppose A_{ij}, $i \neq j$ were largest. Let x be a vector of zeros except for $x_i = 1$ and $x_j = -\operatorname{sign}(A_{ij})$. Then

$$x^\top Ax = A_{ii} + A_{jj} - 2|A_{ij}| \leq 0,$$

   a contradiction. □

5. (Basis of Cholesky) A is symmetric positive definite if and only if $A = LL^\top$ where L is a lower triangular matrix with positive diagonal entries.

Proof. Prove by induction, showing Schur complement is symmetric positive definite.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} \sqrt{A_{11}} & 0 \\ A_{21}/\sqrt{A_{11}} & I \end{bmatrix} \cdot \begin{bmatrix} \sqrt{A_{11}} & A_{12}/\sqrt{A_{11}} \\ 0 & S \end{bmatrix}$$

where A_{11} is a 1×1 matrix and

$$\begin{aligned} S &= A_{22} - A_{21}A_{12}/A_{11} \\ A &= \begin{bmatrix} \sqrt{A_{11}} & 0 \\ A_{21}/\sqrt{A_{11}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} \sqrt{A_{11}} & A_{21}/\sqrt{A_{11}} \\ 0 & I \end{bmatrix} \\ &= X \begin{bmatrix} 1 & 0 \\ 0 & S \end{bmatrix} X^\top \end{aligned}$$

So $\begin{bmatrix} 1 & 0 \\ 0 & S \end{bmatrix}$ is symmetric positive definite by part 1, which implies that S is symmetric positive definite by part 2.
By induction, $S = L_S L_S^\top$.

$$\begin{aligned} A &= \begin{bmatrix} \sqrt{A_{11}} & 0 \\ A_{21}/\sqrt{A_{11}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & L_S L_S^\top \end{bmatrix} \begin{bmatrix} \sqrt{A_{11}} & A_{21}/\sqrt{A_{11}} \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} \sqrt{A_{11}} & 0 \\ A_{21}/\sqrt{A_{11}} & L_S \end{bmatrix} \begin{bmatrix} \sqrt{A_{11}} & A_{21}/\sqrt{A_{11}} \\ 0 & L_S^\top \end{bmatrix} = LL^\top \end{aligned}$$

as desired. □

Definition 6.2. Suppose A is symmetric positive definite, then $A = LL^\top$ with L lower triangular with positive diagonal is called the Cholesky factorization.

Lemma 6.2

If A is symmetric positive definite and $A = LU$ with L unit lower triangular, let D be diagonal with $D_{ii} = \sqrt{U_{ii}} > 0$. Then $A = (LD)\left(D^{-1}U\right) = L_S L_S^\top$ is the Cholesky factorization.

This lemma suggests that any algorithm for LU can be modified to do Cholesky, and in fact use half the work and space since L and U are simply related.

Algorithm 6.1 (The simplest one).

```
1: for $j = 1 : n$ do
2:     $L_{jj} = \left(A_{jj} - \sum_{i=1}^{u-1} L_{ji}^2\right)^{1/2}$
3:     $L_{j+1:n,j} = \left(A_{j+1:n,j} - L_{j+1:n,j-1} \cdot L_{j,1:j-1}^\top\right) / L_{jj}$
4: end for
```

Error analysis: Same approach as to LU yields analogous bound:

$$(A+E)(x+\delta x)=b$$

where $|E| \leq 3n\varepsilon |L||L^\top|$. But since we do not pivot, we need to use a different approach to bound $|L||L^\top|$:

$$\begin{aligned}\left(|L||L^\top|\right)_{ij} &= \sum_k |L_{ik}|\,|L_{jk}| \\ &\leq \|L_{i,:}\|\,\|L_{j,:}\| \\ &= \sqrt{A_{ii}}\sqrt{A_{jj}} \\ &\leq \text{the max entry in } A\end{aligned}$$

6.3 Symmetric Indefinite

It is also possible to save half the space and flops by just preserving symmetry, without assuming positive definiteness. The traditional approach is called Bunch-Kaufman factorization:

$$A = PLDL^\top P^\top$$

where P is a permutation, L is unit lower triangular, and D has 1-by-1 and 2-by-2 diagonal blocks. More complicated pivoting (with 2x2 blocks) is needed to preserve symmetry: consider $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Instead of just searching a column for a pivot, once needs to search along a row too. It is more complicated to apply ideas from LU and Cholesky to this factorization (blocking, recursion, communication lower bounds): see *LAPACK* routine *ssytrf*.

A more numerically stable version is called rook-pivoting, which may search more than one row and column of A, but has better numerical stability guarantees, see *LAPACK* routine *ssytrf*.

And there is another approach (called Aasen factorization):

$$A = PLTL^\top P^\top$$

where now T is a symmetric tridiagonal matrix, and so costs just $O(n)$ to solve $Tx = b$. See "Implementing a Blocked Aasen's Algorithm with A Dynamic Scheduler on Multicore Architectures". Aasen is more amenable to optimizations to minimize communication.

6.4 Band Matrices

These are the simplest sparse matrices, and have solvers in *LAPACK*.

- Case without pivoting (e.g. Cholesky): $A = LU$, L, U bounded. $\text{ubw}(U) = \text{ubw}(A)$, $\text{lbw}(U) = \text{lbw}(A)$. The cost is $2n \cdot \text{ubw} \cdot \text{lbw} + n \cdot \text{lbw} = O(n)$ for narrow bandwidths.
- Case with pivoting: $A = PLU$, $\text{ubw}(U) = \text{ubw}(A) + \text{lbw}(A)$, " $\text{lbw}(L)$" $= \text{lbw}(A)$ (does not look banded, but uses same amount of storage)

Band matrices often arise from discretizing differential equations, or other physical simulations, where each unknown only depends on its neighbors, so A_{ij} is nonzero only when i and j are close, i.e. A_{ij} near diagonal.

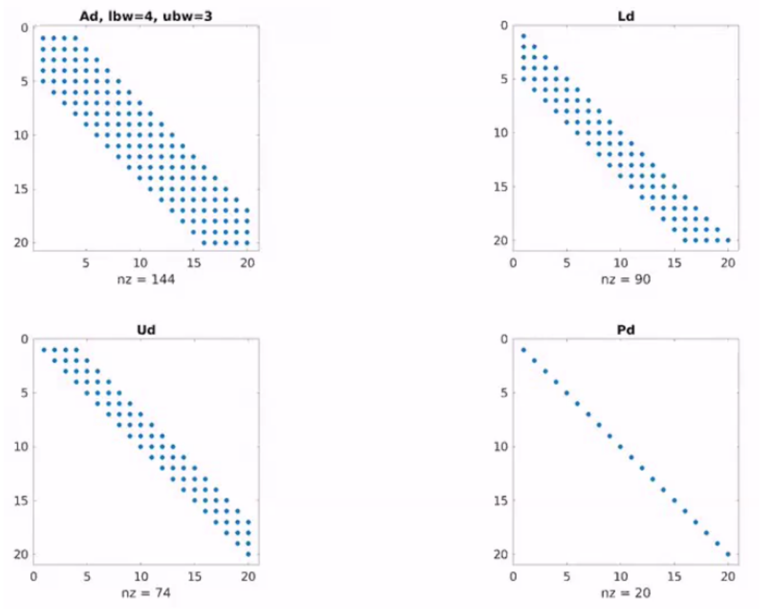


Figure 6.2: Case without pivoting

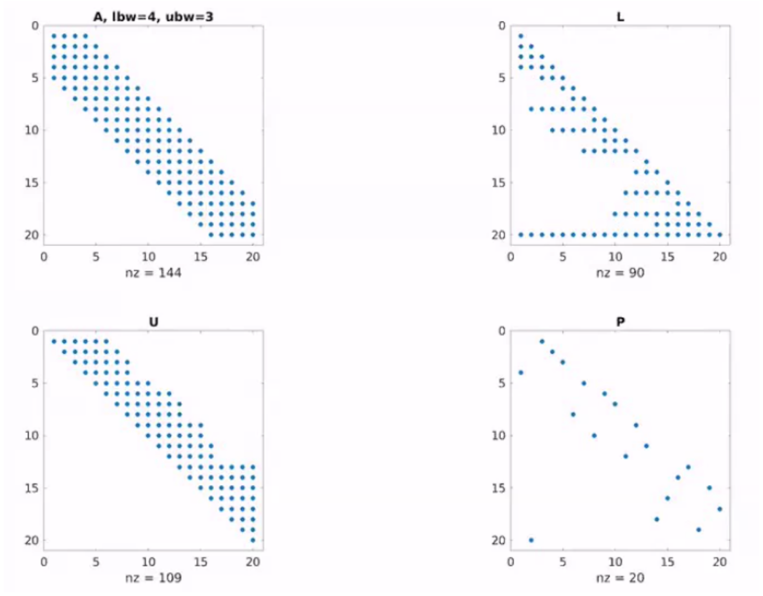


Figure 6.3: Case with pivoting

Theorem 6.1 (Gershgorin's Theorem)

All eigenvalues λ of A lie in n circles in the complex plane, where circle i has center A_{ii} and radius $\sum_{j=1,j\neq i}^{n} |A_{ji}|$.

Proof. If $Ax = \lambda x$, suppose $|x_i|$ is the largest entry in absolute value of x. Thus,

$$A_{ii} - \lambda = \sum_{j=1,j\neq i}^{n} \frac{A_{ji}x_j}{x_i}$$

and so

$$\begin{aligned} |A_{ii} - \lambda| &\leq \sum_{j=1,j\neq i}^{n} |A_{ji}||x_j/x_i| \\ &\leq \sum_{j=1,j\neq i}^{n} |A_{ji}| \end{aligned}$$

□

Example 6.1 (Sturm-Liouville problem)

$-y"(x)+q(x)y(x) = r(x)$ on $x \in [0,1]$, $y(0) = \alpha$, $y(1) = \beta$, $q(x) \geq \bar{q} > 0$, discretize at $x(i) = ih$, $h = \frac{1}{N+1}$. Unknowns $y(i) = y(x(i))$ for $i = 1, \cdots, N$. Approximate

$$y"(i) = \frac{(y(i+1) - 2y(i) + y(i-1))}{h^2}$$

So, letting $q(i) = q(x(i))$ and $r(i) = r(x(i))$, we get linear system

$$\frac{-(y(i+1) - 2y(i) + y(i-1))}{h^2} + g(i)y(i) = r(i)$$

or $Ay = v$ where

$$\begin{aligned} v &= r + \left[\frac{\alpha}{h^2}, 0, \ldots 0, \frac{\beta}{h^2}\right] \\ A &= \text{diag}(\frac{2}{h^2} + q(i)) + \text{diag}(\frac{-1}{h^2}, 1) + \text{diag}(\frac{-1}{h^2}, -1) \end{aligned}$$

We can prove A is symmetric positive definite, so we can use Cholesky, by using the theorem 6.1 (Gershgorin's Theorem) to $A = A^\top$, all eigenvalues in circles centered at $2/h^2 + q(i) \geq 2/h^2 + \bar{q}$, with radius $2/h^2$, so must be real and positive.

Example 6.2

Now consider Poisson's equation in 2 dimensions:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} + g(x,y)u(x,y) = r(x,y)$$

in a square $0 \le x, y \le 1$ with boundary conditions: $u(x,y)$ given on the boundary of square. Typical example: Given temperature on boundary of a square of uniform material, what is the temperature everywhere in the middle ($q = r = 0$)?
We discretize the square with an $n \times n$ mesh: $h = 1/(n+1)$, $x(i) = ih$, $y(i) = ih$, $u(i,j) = u(x(i), y(i))$, same for $4q(i,j)$ and $r(i,j)$. Again approximate

$$\frac{\partial^2 u(xy)}{\partial x^2} \cong [u(x-h,y) - 2u(x,y) + u(x+h,y)]/h^2$$
$$\frac{\partial^2 u(xy)}{\partial y^2} \cong [u(x,y-h) - 2u(x,y) + u(x,y+h)]/h^2$$

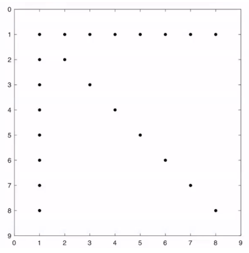
Add these, evaluate at $(x(i), y(i))$, so

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} \approx \left[u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) - 4u(i,j)\right]/h^2$$

Using "natural order" (rowwise or columnwise ordering of mesh points (i,j)), we get an spd band matrix of dimension n^2 and bandwidth n. A is banded but at with most 5 nonzeros per row, so should really think of it as sparse.

6.5 General Sparse Matrices

Here is a small *MATLAB* example to show important of ordering on "arrowhead matrix": Result is dense! So performing LU and storing L and U would cost as many flops and words as a dense matrix, $O(n^3)$ and $O(n^2)$.


(a) The original "arrowhead matrix"

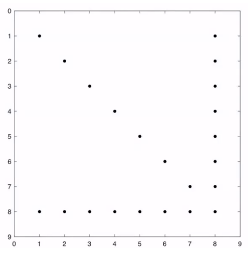

(b) The reversed "arrowhead matrix"

Figure 6.4: *MATLAB* results for "arrowhead matrix"

Now let's try LU on $PAP^\top$, where P is a permutation that reverses the order of rows and columns. Result is very sparse! And if you rewrite LU to take advantage of this, to neither store nor compute entries that are certain to be zero, the cost in flops and words would drop to $O(n)$, much smaller than before. *MATLAB* has many built-in sparse matrix operations (LU, Cholesky, etc), that take advantage of sparsity like this; do "help sparse" to learn more.

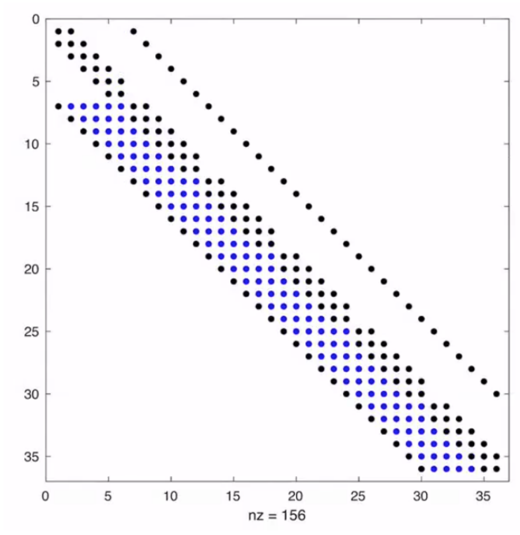


Figure 6.5: *MATLAB* result for 2D Poisson equation

Let's see what happens to the 2D Poisson equation:

So the band fills in entirely, costing

$$O(N\,\mathrm{bw}^2) = N\sqrt(N)^2 = N^2 = n^4$$

for an $n \times n$ mesh with $N = n^2$ unknowns. We'll see that we can do much better, $O(n^3)$ instead, for a clever choice of permutation P in $PAP^\top$.

Definition 6.3. A weighted, undirected graph G is a collection of 3 sets:

- V: Vertices (as known as nodes)
- E: Edges connected pairs of vertices (u, v) (note: (u, u) is allowed; undirected means (u, v) is the same as (v, u)
- W: Weight (number) associated with each edge

There is an obvious one-to-one relationship between a weighted undirected graph G and a symmetric matrix, and we will use the notations interchangeably:

- V: One row and column per vertex
- E: Locations of nonzeros (note: (u, u) on diagonal)
- W: Values of nonzeros

6.6 Challenges to Factoring Sparse Matrices

Store and operate on only nonzeros in an efficient way. The simplest imaginable data structure is Coordinate Format (COO): list of nonzero entries and their locations $(a(i, j), i, j)$.

Example 6.3

For matrix

$$A = \begin{bmatrix} 2 & 0 & 2 & 0 & 5 \\ 0 & 1 & 4 & 0 & 3 \\ 0 & 0 & 8 & 0 & 0 \end{bmatrix},$$

its COO $= ((2,1,1),(7,1,3),(5,1,5),(1,2,2),(4,2,3),(3,2,5),(8,3,3))$.

A better way is Compressed Sparse Row (CSR):

- val: The array of nonzero entries in each row, from row 1 to row n, from left to right
- col_index: Columns index of each nonzero (val(i) is in col(a) of sparse matrix)
- rowbegin: The pointer to start of each row: entries of row i lies in entries rowbegin(i) through rowbegin($i+1$) $-$ 1 of val() and col()

Example 6.4

With the same matrix from the above example, its CSR is

$$\begin{aligned} \text{val} &= (2,7,5,1,4,3,8) \\ \text{col\_index} &= (1,3,5,2,3,5,3) \\ \text{rowbegin} &= (1,4,7,8) \end{aligned}$$

We want to minimize time, memory, get the right answer (stability). Order effects fill-in, which effects time and memory (less is better!). So we want to pick order to minimize fill-in. Order effects backward stability for LU (and symmetric indefinite $LDL^\top$), but not Cholesky (recall that the stability proof for Cholesky did not depend on order). Cholesky is easiest case: can pick order just to minimize fill in.

What is the best order for Cholesky, i.e. choose permutation P, do Cholesky on P, do Cholesky on $PAP^\top$ to minimize flops.

Theorem 6.2

Picking the optimal order (out of all n! possibilities) is NP-hard, i.e. any algorithm will have cost exponential in n, overwhelming everything else.

Definition 6.4. A path in a graph from vertex v_1 to v_k is a set of edges $(v_1, v_2), (v_2, v_3), \cdots, (v_{k-1}, v_k)$ connected end-to-end.

Definition 6.5. The degree of a vertex in the graph is the number of edges for which it is an endpoint.

Definition 6.6. The distance between any two vertices is the length of the shortest path (fewest edges) connecting them.

So we need to use heuristics:

- Reverse Cuthill-McKee (RCM / Breadth-First Search):

  1. Pick any vertex, call it the root.
  2. Compute distance of all other vertices from root, and label them by distance, (so distance = 0 is the root, distance = 1 are the vertices with an edge to the root, etc), Cost = 0(#nonzeros in matrix), using Breadth-First-Search, cheap!
  3. Order vertices in reverse order by distance (farthest first)

  Fact 6.1. Vertices at distance k can only be directly connected to vertices at distance $k-1$ or $k+1$ (otherwise distance would be wrong!) Explains why RCM ordering tends to make matrix entries close to diagonal: matrix becomes block tridiagonal, with vertices at distance k making up diagonal block k. So it is like a band matrix, and limits fill-in to the band.

  Called *symrcm* in *MATLAB*.

- Minimum Degree (MD): At each step pick pivot to have minimum degree (note: as factorization proceeds, need to update matrix and its graph, so degree can change; use latest degree)
  Fact 6.2. If vertex corresponding to row/column i is used as a pivot, and has degree d, then we need to do d^2 multiplications and adds, and so can fill in at most d^2 entries (if they were originally zero, and ignoring symmetry).

- Nested Dissection: "Bisect" vertices of graph into 3 sets $V = V_1 \cup V_2 \cup V_s$ (s stands for "separator") such that:

  1. About the same number of vertices in V_1 and V_2
  2. V_s much smaller than V_1 and V_2
  3. There are no edges directly connecting vertices in V_1 and V_2

  Now reorder vertices with V_1 first, then V_2, then V_s:

$$A = \begin{bmatrix} A_{11} & 0 & A_{1s} \\ 0 & A_{21} & A_{23} \\ A_{13}^\top & A_{23}^\top & A_{33} \end{bmatrix},$$

  so after Cholesky,

$$L = \begin{bmatrix} L_1 & 0 & 0 \\ 0 & L_2 & 0 \\ L_{1s} & L_{2s} & L_s \end{bmatrix},$$

  in particular, $A_1 = L_1 L_1^\top$ and $A_2 = L_2 L_2^\top$ are independent Cholesky factorizations.

Theorem 6.3 (George, Hoffman/Martin/Rose, Gilbert/Tarjan, 1970s & 1980s)
Any ordering for Cholesky on a 2D $n \times n$ mesh (so the matrix is $n^2 \times n^2$) has to do at least $\Omega(n^3)$ flops, and this is attained by nested dissection ordering. (If it were dense, it would cost $O(n^6)$, a lot more). This extends to "planar graphs", i.e. graph you can draw on paper without crossing any edges.

Theorem 6.4 (Ballard, Demmel, Holtz, Schwarz, 2009)
Lower bound on #words moved for sparse LU and Cholesky is $\Omega(\#\text{flops}/\sqrt{M})$, where #flops is number actually performed. So the lower bound on #words_moved for Cholesky on $n \times n$ mesh is $\Omega(n^3/\sqrt{M})$.

Theorem 6.5 (David, D., Grigori, Peyronnet 2010)
Possible to achieve lower bound for $n \times n$ of $\Omega(n^3/\sqrt{M})$ words moved by Cholesky with nested dissection ordering, if properly implemented.

Contrast: band solver would cost $O(dimension * \text{bw}^2) = n^{2+2} = n^4$ flops.

It's a good idea for 3D meshes too: $n \times n \times n$ mesh (so matrix is n^3 by n^3) costs n^6 to factor using Cholesky with nested dissection ordering (versus n^9 if it were dense, or $n^{3+2+2} = n^7$ if banded).

6.7 Summarize Sparse Cholesky

To summarize, the overall sparse Cholesky algorithm is as follows:

- Choose ordering (RCM, MD, ND)
- Build data structure to hold A and (to be computed) entries of L
- Perform factorization

Theorem 6.6
For any A there is a permutation P and diagonal matrices D_1 and D_2 such that $B = DD_1APD_2$ has $B_{ii} = 1$ and all other $|B_{ij}| \leq 1$.

What about nonsymmetric matrices, how do we pivot for sparsity and stability?

- Threshold pivoting: among pivot choices in column within a factor of 2 or 3 of the largest, pick the one with the least fill-in (so may need to change data structure on the fly).
- Static Pivoting (used in *SuperLU_DIST*):
  1. Permute and scale matrix to try to make diagonal as large as possible
  2. Reorder using similar ideas as for Cholesky: this leaves all the diagonal entries on the diagonal. Then build the data structures for L and U, before doing LU, rather than figuring them out on the fly, which would be slower.
  3. During factorization, if a prospective pivot is too small, just make it big (This happens rarely in practice). As a result we get the LU factorization of $A + a$ low rank change (the rank = the number of modified pivots), which we can fix with the Sherman-Morrison-Woodbury formula or an iterative method like GMRES.

This brief discussion suggests that there is a large body of software that has been developed for solving sparse $Ax = b$, and which one is faster may depend on your matrix (and computer). There is a survey of available sparse matrix software: "Updated Survey of sparse direct linear equation solvers".

Finally, we give a brief tour of structured matrices, which may be dense but depend only on $O(n)$ parameters. The number of structures that can arise, and be used to get faster algorithms, is large, e.g. depending on the underlying physics of the problem being modeled. We will do one set of common examples that share a structure:

- Vandermonde: $V_{ij} = x_i^{j-1}$ - arises in polynomial interpolation
- Cauchy: $C_{ij} = 1/(x_i + y_j)$ - arises in rational interpolation
- Toeplitz: $T_{ij} = x_{i-j}$, constant along diagonals, arises in convolution
- Hankel: $H_{ij} = x_{i+j}$, constant along "anti-diagonals", similar to Toeplitz

E.g.: Solving $Vz = b$ means

$$\sum_{j=1}^{n} x_i^{j-1} z_j = b_i$$

i.e. finding polynomial coefficients z_j so to match values b_i at points x_i, i.e. polynomial interpolation; we know how to do this in $O(n^2)$ time using, eg using "Newton interpolation." Another trick works for $V^\top z = b$.

E.g.: Multiplying by a Toeplitz matrix is same as convolution, which we know how to do fast by FFT.

E.g.: Solving with a Cauchy, also interpretable as interpolation.

Common structure of X: $AX + XB =$ low rank for some simple, nonsingular A and B.
Definition 6.7. The rank is called the "displacement rank".

Example 6.5

Let $D = \operatorname{diag}(x_i)$, then DV is "V shifted left" (except for the last column),

$$VP = V \begin{bmatrix} 0 & 0 & & 0 & 1 \\ 1 & 0 & & 0 & 0 \\ 0 & 1 & \ddots & \vdots & 0 \\ \vdots & 0 & & 0 & 0 \\ 0 & \vdots & & 1 & 0 \end{bmatrix}$$

is "V shifted left" (except for the last column), difference $DV - VP$ is zero except the last column, so the rank is 1.

Example 6.6

T Toeplitz implies that $PT - TP =$(T shifted down) $-$ (T shifted left), nonzero are only in first row, last col, so the rank is 2.

Example 6.7

C Cauchy implies that $\operatorname{diag}(x_i)C + C\operatorname{diag}(y_i)$ are all ones, then the rank is 1.

Theorem 6.7 (Kailath et al.)

There are $O(n^2)$ solvers if displacement rank low (stability can be a problem).

Theorem 6.8 (George & Liu, 1981)

L_{ij} is nonzero (assuming no exact cancellation) if there is a path from i to j in the graph G of A passing through vertices k satisfying $k < i$ and $k < j$.

Theorem 6.9 (Gilbert & Peierls)

Cost of determining structure of L no more than actually doing the factorization.

7 Lecture 7: Least Squares

7.1 Introduction to Least Squares

Standard notation:

$$\arg\min_x \|Ax-b\|_2,\ A\ m\times n,\ m>n$$

$m>n$ means overdetermined, don't expect $Ax=b$ exactly. Other variants of Least Squares::

- Constrained: $\arg\min_{x:\ Bx=y} \|Ax-b\|_2$ where #rows$(B)\le$ #cols$(A)\le$ #rows$(A)+$ #rows(B). So answer unique if A,B full rank
- Weighted Least Squares:: $\arg\min_x \|y\|_2$ such that $b=Ax+By$. It is weighted because if $B=I$, it would be usual LS. If B square, nonsingular, $\arg\min_{x:\ Bx=y} \|B^{-1}(Ax-b)\|_2$
- Underdetermined: #rows$(A)<$ #cols(A) or if A is not full column rank, then there is a whole space of solutions (add any z such that $Az=0$ to x). To make solution unique, $\arg\min_x \|x\|_2$ such that $Ax=b$
- Ridge Regression: $\arg\min_x \|Ax-b\|_2^2+\lambda\|x\|_2^2$, guarantees unique solution if $\lambda>0$
- Total Least Squares: $\arg\min_{x:(A+E)x=b+r} \|[E,r]\|_2$

7.2 Algorithm for Overdetermined Cases

- Solve Normal Equations (NE): $A^\top Ax=A^\top b$, $A^\top A$ s.p.d $\Rightarrow$ solve with Cholesky, the fastest in dense case, not stable if A ill-conditioned
- Use QR Decomposition: $A=QR$, Q is $m\times n$ and has orthonormal columns, R is upper triangular. $x=R^{-1}Q^\top b$.
  - Gram-Schmidt: Unstable if A ill-conditioned
  - Householder: Stable ($x=A\backslash b$ in *MATLAB*)
  - Blocked Householder: Reduce data movement
  - Cholesky QR: Getting QR via Normal Equations, fast but can be unstable.
- SVD: most "complete answer", gives condition number, error bounds, works in ill-conditioned, rank deficient cases too.
- Conversion to a linear system, where matrix contains A, $A^\top$, not $A^\top A$, allows us to exploit sparsity, iterative refinement (also possible to exploit sparsity in QR).

7.2.1 Normal Equations

Theorem 7.1 (Normal Equations)
If A full column rank, solution of $A^\top Ax = A^\top b$ minimizes $\|Ax-b\|_2$.

Proof. Assume x satisfies NE.

$$\begin{aligned}
\|A(x+e)-b\|_2^2 &= (A(x+e)-b)^\top(A(x+e)-b) \\
&= (Ax-b+Ae)^\top(Ax-b+Ae) \\
&= (Ax-b)^\top(Ax-b)+2e^\top A^\top(Ax-b)+(Ae)^\top(Ae) \\
&= \|Ax-b\|_2^2+0+\|Ae\|_2^2
\end{aligned}$$

which is minimized at $e=0$. □

7.2.2 QR Decomposition

$A=QR$, where A is $m\times n$, Q is $m\times n$ and has orthonormal columns, R is $n\times n$ and upper triangular. To minimize $\|Ax-b\|_2$, $x=R^{-1}Q^\top b$.

proof 1. $A=QR=\left[Q,\hat{Q}\right][R,0]^\top$, where $\left[Q,\hat{Q}\right]$ is square and orthogonal. Then

$$\begin{aligned}
\|Ax-b\|_2^2 &= \left\|\left[Q,\hat{Q}\right]^\top(Ax-b)\right\|_2^2 \\
&= \left\|\begin{bmatrix} Q^\top QRx-Q^\top b \\ \hat{Q}^\top QRx-\hat{Q}^\top b\end{bmatrix}\right\|_2^2 \\
&= \left\|\begin{bmatrix} Rx-Q^\top b \\ -\hat{Q}^\top b\end{bmatrix}\right\|_2^2 = \left\|Rx-Q^\top b\right\|_2^2+\left\|\hat{Q}^\top b\right\|_2^2
\end{aligned}$$

which is minimized when $x=R^{-1}Q^\top b$. □

proof 2. Plug into normal equations:

$$\begin{aligned}
x &= \left(A^\top A\right)^{-1}A^\top b \\
&= \left(R^\top Q^\top QR\right)^{-1}R^\top Q^\top b \\
&= \left(R^\top R\right)^{-1}R^\top Q^\top b \\
&= R^{-1}R^\top R^\top Q^\top b = R^{-1}Q^\top b
\end{aligned}$$

□

Algorithms for computing $A=QR$:

- Classical *Gram-Schmidt*: Much much less stable than the modified one.
- Modified *Gram-Schmidt*:

Algorithm 7.1 (Modified *Gram-Schmidt*).

```
1: Equate columns $i$ of $A$ and $QR$ to get $A(:,i) = \sum_{j=1}^{i} Q(:,j)R(j,i)$
2: $(Q(:,j))^\top A(:,i) = R(j,i)$
3: for $i = 1 : n$ do
4:     tmp $= A(:,i)$
5:     for $j = 1 : i-1$ do
6:         $R(j,i) = Q(:,j)^\top A(:,i)$            ▷ CGS[a], costs $2m$
7:         $R(j,i) = Q(:,j)^\top$tmp    ▷ MGS[b], costs $2m$, getting same result
8:         tmp = tmp $- R(j,i)Q(:,j)$               ▷ Costs $2m$
9:     end for
10:    $R(i,i) = \|\text{tmp}\|_2$                   ▷ Costs $2m$
11:    $Q(:,i) = \text{tmp}/R(i,i)$                  ▷ Costs $m$
12: end for
```

Two metrics of backward stability:

If backward stable should get accurate QR decomposition of slight perturbed input matrix $A + E = QR$ where

$$\frac{\|E\|}{\|A\|} = O(\varepsilon)$$

- This means $\|QR - A\| / \|A\|$ should be $O(\varepsilon)$.
- It also means $\left\|Q^\top Q - I\right\|$ should be $O(\varepsilon)$, an extra condition.

7.2.3 Singular Value Decomposition

Theorem 7.2

Suppose A is an $m \times m$ matrix. Then there exists orthogonal matrix $U = [u_1, \cdots, u_m]$, diagonal matrix $\Sigma = \operatorname{diag}(\sigma_1, \cdots, \sigma_m)$, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_m \geq 0$, and an orthogonal matrix $V = [v_1, \cdots, v_m]$ such that $A = U\Sigma V^\top$.

More generally, A is an $m \times n$ matrix, where $m > n$, U is an $m \times m$ matrix and is orthogonal as before, V is a $n \times n$, orthogonal matrix, Σ is an $m \times n$ matrix with same diagonal. When $m > n$, sometimes write as following (thin SVD):

$$A = [u_1, \cdots, u_n] \operatorname{diag}(\sigma_1, \cdots, \sigma_n) V^\top = \hat{U}\hat{\Sigma}V^\top$$

Definition 7.1. If $A = \hat{U}\hat{\Sigma}V^\top$, its Moore–Penrose pseudoinverse is

$$A^+ = V\hat{\Sigma}^{-1}\hat{U}^\top$$

Fact 7.1. $A^+ = \left(A^\top A\right)^{-1} A^\top$ because SVD and normal equations both solve least squares.

[a]Classical *Gram-Schmidt*
[b]Modified *Gram-Schmidt*

7.2.4 Perturbation Theory for Least Squares Problem

If $x = \arg\min_x \|Ax - b\|_2$, and we change A and b a little, how much can x change? Since square case is special case, expect $\kappa(A)$ to show up, but there is another source of sensitivity:

$$A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, b = \begin{bmatrix} 0 \\ b_2 \end{bmatrix} \Rightarrow x = [1, 0] \begin{bmatrix} 0 \\ b_2 \end{bmatrix} = 0$$
$$A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \Rightarrow x = [1, 0] \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = b_1$$

If b_2 very large, b_1 could be small compared to b_2 but still large, so a big change in x. More generally, if b is (nearly) orthogonal to span(A), then condition number very large.

If we have $x + e = \arg\min_x \|(A + \delta A)x - (b + \delta b)\|_2$, expanding gives:

$$\begin{aligned} e &= (x + e) - x \\ &= \Big((A + \delta A)^\top (A + \delta A)\Big)^{-1} (A + \delta A)^\top (b + \delta b) - (A^\top A)^{-1} A^\top b \end{aligned}$$

using approximation $(I - X)^{-1} = I + X + O(\|X\|^2)$ as before, only keeping linear terms (one factor of δA and/or δb), call

$$\varepsilon = \max\left(\frac{\|\delta A\|}{\|A\|}, \frac{\|\delta b\|}{\|b\|}\right)$$

to get

$$\frac{\|e\|}{\|x\|} \leq \varepsilon \left(2\kappa(A)\frac{1}{\cos\theta} + \tan\theta \kappa^2(A) + O(\varepsilon^2)\right)$$

where $\theta = \text{angle}(b, Ax)$, or

$$\sin\theta = \frac{\|Ax - b\|_2}{\|b\|_2}$$

So condition number can be large if

1. $\kappa(A)$ large
2. θ near $\pi/2$, i.e. $\frac{1}{\cos\theta} \sim \tan(\theta)$ large
3. error like $\kappa^2(A)$ when theta not near zero

Will turn out that using the right QR decomposition, or SVD, keeps $\varepsilon = O(\varepsilon)$, so backward stable. Normal equations are not backward stable; since we do Chol$(A^\top A)$ the error bound is always proportional to $\kappa^2(A)$, even when θ small.

7.3 Stable Algorithms for QR Decomposition

Recall that MGS and CGS produced Q that were far from orthogonal, and running them twice helped but sometime also failed to guarantee orthogonality. [16]

To guarantee stability of solving least squares problems (and later eigenproblems and computing the SVD) we need algorithms that guarantee that Q is (very nearly) orthogonal, i.e.

$$\left\| Q^\top Q - I \right\| = O(\varepsilon)$$

The basic idea is to express Q as product of simple orthogonal matrices $Q = Q_1 Q_2 \cdots Q_n$ where each Q_i accomplishes part of the task (multiplying it by A makes some entries zero, until A turns into R). Since a product of orthogonal matrices is orthogonal, there is no danger of losing orthogonality. There are two kinds of simple orthogonal matrices: *Householder transformations* and *Givens rotations.*

7.3.1 Householder Rotation

A *Householder transformation* (or reflection) is $H = I - 2uu^\top$, where u is a unit vector. We confirm orthogonality as follows:

$$HH^\top = (I - 2uu^\top)(I - 2uu^\top) = I - 4uu^\top + 4uu^\top uu^\top = I$$

It is called a reflection because Hx is a reflection of x in plane orthogonal to u.

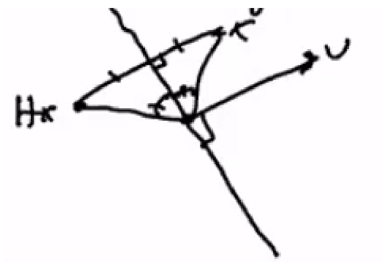


Figure 7.1: A reflection

Given x, we want to find u so that Hx has zeros in particular locations, in particular we want u so that

$$Hx = [c, 0, 0, \cdots, 0]^\top = c \cdot e_1$$

for some constant c. Since the 2-norm of both sides is $\|x\|_2$, then $|c| = \|x\|_2$. We solve for u as follows: Write

$$Hx = (I - 2uu^\top)x = x - 2u(u^\top x) = c \cdot e_1,$$

or

$$u = \frac{(x - c \cdot e_1)}{2u^\top x}.$$

[16] MGS still has its uses in some other algorithms, CGS does not.

The denominator is just a constant that we can choose so $\|u\|_2 = 1$, i.e.

$$y = x \pm \|x\|_2 e_1 \text{ and } u = \frac{y}{\|y\|_2}$$

We write this as $u = \text{House}(x)$. We choose the sign of $\pm$ to avoid cancellation (avoids some numerical problems)

$$y = [x_1 + \text{sign}(x_1)\|x\|_2, x_2, \cdots, x_n]^\top$$

How to do QR decomposition by forming Q as a product of *Householder transformations.*

Algorithm 7.2 (QR decomposition of $m \times n$ matrix A, $m \geq n$).

```
1: for $i = 1 : \min(m-1, n)$ do                    ▷ Only need to do last column if $m > n$
2:     $u(i) = \text{House}(A(i:m, i))$             ▷ Compute Householder vector
3:     $A(i:m, i:n) = (I - 2u(i)u(i)^\top)A(i:m, i:n) = A(i:m, i:n) - u(i)(2u(i)^\top A(i:m, i:n))$
4: end for
```

Note that We never actually form each Householder matrix

$$Q(i) = I - 2u(i)u(i)^\top,$$

or multiply them to get Q, we only multiply by them. We only need to store the $u(i)$, which are kept in A in place of the data they are used to zero out (same idea used to store L in Gaussian elimination).

The cost is

$$\sum_{i=1}^{n} 4(m-i+1)(n-i+1) = 2n^2m - \frac{2}{3}n^3$$

- $m > n$: dominated by $2n^2m$
- $m = n$: $\frac{4}{3}n^3$

So when we are done we get

$$Q_nQ_{n-1}\cdots Q_1A = R.$$

Then

$$\begin{aligned} A &= Q_1^\top Q_2^\top \cdots Q_n^\top R \\ &= Q_1Q_2\cdots Q_nR \\ &= QR \end{aligned}$$

and to solve the least squares problem $\underset{x}{\text{argmin}}||Ax - b||_2$ we do

$$x = R^{-1}(Q^\top b)$$

or

Algorithm 7.3 (QR decomposition of $m \times n$ matrix A, $m \geq n$).

```
1: for $i = 1 : n$ do
2:     $b = Q(i)b = (I - 2u(i)u(i)^\top)b = b + (-2u(i)^\top b)u(i)$
3:     $c = -2u(i)^\top b(i : m)$                ▷ Dot product
4: end for
5: $x = R^{-1}b$ by substitution
```

It costs $O(mn)$, much less than $A = QR$ (again analogous to GE)

In Matlab, $x = A\backslash b$ does *GEPP* if A is square, and solves the least squares problem using Householder QR when A has more rows than columns.

7.3.2 Optimizing QR

So far we have only described the *BLAS-2* version (analogous to simplest version of LU). We can and should use all the ideas that we used for matrix multiplication and LU to minimize how much data is moved by QR decomposition, which shares the same lower bound:

$$\#\text{words moved} = \Omega(\frac{\#\text{flops}}{\sqrt{M}}).$$

where M is the fast memory size.

The basic idea will be the same as for LU:

1. do QR on the left part of the matrix;
2. update the right part of the matrix using the factorization of the left part;
3. do QR on the right part.

As for LU, the left part can be a block of a fixed number of columns, or the whole left half (and then working recursively). Either way, we need to do step (2) above efficiently: apply the *Householder factorizations* from doing QR of the left part to the right part. In LU this was (mostly) updating the Schur complement using matrix multiplication, which is fast. So we need to figure out how to apply a product of *Householder transformations* $Q = Q_b Q_{b-1} \cdots Q_1$ with just a few matrix multiplications. Here is the result we need.

Theorem 7.3

If $Q_i = I - 2u_i u_i^\top$, then

$$Q = I - YTY^\top$$

where $Y = [u_1, u_2, \cdots, u_b]$ is $m \times b$, and T is $b \times b$ and upper triangular. So multiplying by Q reduces to 3 matrix multiplications.

There is one other new idea we need to deal with the common case when $m \gg n$, i.e. A is "tall and skinny", or the problem is "very overdetermined". When $n^2 \leq M$, the lower bound becomes

$$\begin{aligned}\#\text{words moved} &= \Omega(\frac{\#\text{flops}}{\sqrt{M}}) \\ &= \Omega(\frac{mn^2}{\sqrt{n^2}}) = \Omega(mn) \\ &= \Omega(\text{size of input})\end{aligned}$$

where mn is the size of the matrix A. In other words, the lower bound says we should just be able to access all the data once, which is an obvious lower bound for any algorithm. Using *Householder transformations* column by column will not work (unless the whole matrix fits in fast memory) since the basic QR algorithm scans over all the columns n times, not once. To access the data just once we instead do "Tall Skinny QR", or *TSQR*.

Sequential TSQR: To illustrate, Suppose we can only fit a little over 1/3 of the matrix in fast memory at one time. Then here is what we compute:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix}$$

just read A_1 into fast memory, do QR, yielding $A_1 = Q_1R_1$,

$$= \begin{bmatrix} Q_1R_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} Q_1 & & \\ & I & \\ & & I \end{bmatrix} \begin{bmatrix} R_1 \\ A_2 \\ A_3 \end{bmatrix} = \hat{Q}_1 \begin{bmatrix} R_1 \\ A_2 \\ A_3 \end{bmatrix}$$

where $\hat{Q}_1$ is square and orthogonal. Q_1 is still just represented as a collection of Householder vectors. Read A_2 into fast memory, pack R_1 on top of A_2 yielding $[R_1; A_2]$, do QR on $[R_1, A_2]^\top$, yielding $[R_1, A_2]^\top = Q_2R_2$

$$= \hat{Q}_1 \begin{bmatrix} Q_2R_2 \\ A_3 \end{bmatrix} = \hat{Q}_1 \begin{bmatrix} Q_2 & \\ & I \end{bmatrix} \begin{bmatrix} R_2 \\ A_3 \end{bmatrix} = \hat{Q}_1\hat{Q}_2 \begin{bmatrix} R_2 \\ A_3 \end{bmatrix}$$

Store $\hat{Q}_1$ and $\hat{Q}_2$ separately and read A_3 into fast memory, do $[R_2, A_3]^\top = Q_3R_3$

$$= \hat{Q}_1\hat{Q}_2[Q_3R_3] = \hat{Q}_1\hat{Q}_2\hat{Q}_3R_3 = QR$$

This is the QR decomposition, because $\hat{Q}_1\hat{Q}_2\hat{Q}_3$ is a product of orthogonal matrices, and so orthogonal, and R_3 is upper triangular. As described, we only need to read the matrix once (and write the Q factors once).

This may also be called a "streaming algorithm", because it can compute R as A "streams" into memory row-by-row, even if A is too large to store completely.

This algorithm is well suited for parallelism, because it doesn't matter in what order we take pairs of submatrices and do their QR decompositions. Here is an example of *parallel TSQR*, where each of 4 processors owns 1/4 of the rows of A:

$$\begin{aligned}
A &= \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} Q_1R_1 \\ Q_2R_2 \\ Q_3R_3 \\ Q_4R_4 \end{bmatrix} \\
&= \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} = \hat{Q}_1 \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \\
&= \hat{Q}_1 \begin{bmatrix} \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} \\ \begin{bmatrix} R_3 \\ R_4 \end{bmatrix} \end{bmatrix} = \hat{Q}_1 \begin{bmatrix} Q_{12}R_{12} \\ Q_{34}R_{34} \end{bmatrix} = \hat{Q}_1 \begin{bmatrix} Q_{12} \\ Q_{34} \end{bmatrix} \begin{bmatrix} R_{12} & \\ & R_{34} \end{bmatrix} \\
&= \hat{Q}_1\hat{Q}_2 \begin{bmatrix} R_{12} & \\ & R_{34} \end{bmatrix} = \hat{Q}_1\hat{Q}_2 \left[Q_{1234}R_{1234}\right] \\
&= \hat{Q}_1\hat{Q}_2\hat{Q}_3R_{1234}
\end{aligned}$$

Briefly, one could describe this as *MapReduce* where the reduction operation is QR.

Same idea to accelerate *GEPP* on tall-skinny matrix. Classical *GEPP*, like QR, scans over all columns time (finds max in columns, updates Schur complement, finds max in next column).

Goal: choose n pivots, touching data once.

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \to \begin{bmatrix} \beta_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \to \begin{bmatrix} R_{12} \\ R_{34} \end{bmatrix} \to R_{1234}$$

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_u \end{bmatrix} \to \begin{bmatrix} PA_1 \\ PA_2 \\ PA_3 \\ PA_4 \end{bmatrix}$$

where PA_1 are rows selected by *GEPP* on A_1, ditto for A_i. Each PA_i are "most linearly independent rows" of A_i.

$$\begin{bmatrix} PA_1 \\ PA_2 \\ PA_3 \\ PA_4 \end{bmatrix} \to \begin{bmatrix} PA_{12} \\ PA_{34} \end{bmatrix}$$

PA_{ij} are n most linearly independent rows of $\begin{bmatrix} PA_i \\ PA_j \end{bmatrix}$. Then we use the rows in PA_{1234} to do *Gaussian Elimination* on first n columns.

7.3.3 Givens Rotation

Besides Householder transformations, there is one other way to represent Q as a product of simple orthogonal transformations, called *Givens rotations*. They are useful in other cases than dense least squares problems (for which we use Householder) so we present them here.

A *Givens rotation* $R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ rotates x counterclockwise by θ.

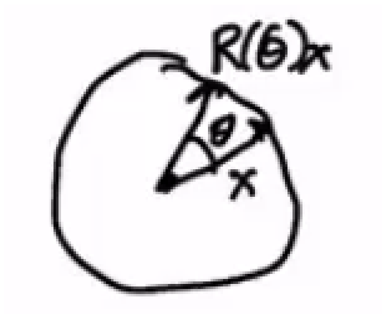


Figure 7.2: *Givens rotation*

More generally, we will take components i and j of a vector and rotate them:

$$R(i,j,\theta) = \text{ identity except for rows and columns } i \text{ and } j, \text{ containing entries of } R(\theta)$$

To do QR, we want to create zero entries. So given $x(i)$ and $x(j)$, we want θ so that $R(i,j,\theta)x$ is 0 in entry j:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x(i) \\ x(j) \end{bmatrix} = \begin{bmatrix} \sqrt{x(i)^2 + x(j)^2} \\ 0 \end{bmatrix}$$

or

$$\cos\theta = \frac{x(i)}{\sqrt{x(i)^2 + x(j)^2}},$$
$$\sin\theta = \frac{-x(j)}{\sqrt{x(i)^2 + x(j)^2}}$$

We do QR by repeatedly multiplying A by $R(i,j,\theta)$ matrices to introduce new zeros, until A becomes upper triangular. When A is sparse, this may cause less fill-in than *Householder transformations.*

7.3.4 Stability of Applying Orthogonal Matrices

Simple summary: Any algorithm that just works by multiplying an input matrix by orthogonal matrices is always backwards stable. This covers QR composition using *Householder* or *Givens rotations*, as well as later algorithms for the eigenproblem and SVD.

Here is why this is true: By using the basic rule $\mathrm{fl}(a \odot b) = (a \odot b)(1+\delta)$, $|\delta| \le \varepsilon$, one can show that for either multiplying by one *Householder* or *Givens transformation* Q one gets the following, where Q' is the floating point transformation actually stored in the machine, e.g.

$$Q' = I - 2u'u'^{\top}$$

where u' is the computed *Householder* vector:

$$\mathrm{fl}(Q'A) = Q'A + E$$

where $\|E\| = O(\varepsilon)\,\|A\|$ This follows from our earlier analysis of dot products, etc. As can be seen from the above formula for $u = \mathrm{House}(x)$, every component u'(i) is nearly equal to the exact value u(i), but roundoff keeps it from being perfect. Since Q' is nearly equal to the exact orthogonal Q, then $Q' = Q + F$ with $\|F\| = O(\varepsilon)$, so we can also write this as

$$\mathrm{fl}(Q'A) = (Q+F)A + E = QA + (FA+E) = QA + G$$

where $\|G\| \le \|F\|\,\|A\| + \|E\| = O(\varepsilon)\,\|A\|$. i.e. you get an exact orthogonal transformation QA plus a small error. The same result applies to *Givens rotations* and *block Householder transformations* (used to reduce communication). We can also write this as

$$\mathrm{fl}(Q'A) = QA + G = Q(A + Q^\top G) = Q(A+F),$$

where $\|F\| = \|G\| = \|f\| = O(\varepsilon)\,\|A\|$, i.e. multiplying by Q is backward stable: we get the exact orthogonal transformation of a slightly different matrix $A+F$. Now multiply by a sequence of orthogonal matrices, as in QR decomposition:

$$\begin{aligned}
\mathrm{fl}\left(Q_3'\left(Q_2'\left(Q_1'A\right)\right)\right) &= \mathrm{fl}\left(Q_3'\left(Q_2'\left(Q_1A + E_1\right)\right)\right) \\
&= \mathrm{fl}\left(Q_3'\left(Q_2\left(Q_1A+E_1\right)+E_2\right)\right) \\
&= Q_3\left(Q_2\left(Q_1A+E_1\right)+E_2\right)+E_3 \\
&= Q_3Q_2Q_1A + Q_3Q_2E_1 + Q_3E_2 + E_3 \\
&= (Q_3Q_2Q_1A) + E \\
&= \underbrace{Q_3Q_2Q_1}_{Q}\left(A + Q^\top E\right) \\
&= Q(A+F)
\end{aligned}$$

where

$$\begin{aligned}
\|F\| = \|E\| &\le \|Q_3Q_2E_1\| + \|Q_3E_2\| + \|E_3\| \\
&= \|E_1\| + \|E_2\| + \|E_3\| \\
&= O(\varepsilon)\,\|A\|
\end{aligned}$$

so multiplying by many orthogonal matrices is also backwards stable. This analysis explains not just why QR is stable, but also why computed Q nearly orthogonal.

There is one more very simple, and fast, QR decomposition algorithm, called *CholeskyQR*:

Algorithm 7.4 (CholeskyQR).

```
1: Factor $A^\top A = R^\top R$ using Cholesky
2: $Q = AR^{-1}$
```

It is clear that $QR = A$, and $Q^\top Q = R^{-\top}A^\top AR^{-1} = I$. But since we form $A^\top A$, this will clearly be unstable if A is ill-conditioned, and fails completely if Cholesky fails to complete, which can happen if we need to take the square root of a nonpositive number (which also means the trick of running it twice, as used with Gram-Schmidt, doesn't work either).

8 Lecture 8: Low Rank Matrices

Dealing with low rank (or nearly) matrices. Motivation: Real data is often low rank. So we want to

1. Take precautions to avoid inaccuracy in Least Squares.
2. Use it to compress data, go faster, both deterministically and using randomization.

Use LS as an application of compression, but many others.

8.1 Solving an LS Problem when Matrix Rank Deficient

Theorem 8.1

Let A be $m \times n,\ m \geq n,\ \operatorname{rank} A = r < n$.

$$A = U\Sigma V^\top = [U_1, U_2, U_3] \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \\ 0 & 0 \end{bmatrix} [V_1, V_2]$$

where U_1 is $m \times r$, U_2 is $m \times (n-r)$, U_3 is $m \times (m-n)$, Σ_1 is $r \times r$, Σ_2 is $(n-r) \times (n-r)$, V_1 is $n \times r$, and V_2 is $n \times (n-r)$. Here Σ_1 is full rank, $\Sigma_2 = 0$.

The set of vectors minimizing $\|Ax - b\|_2$ is $\left\{x = V_1\Sigma_1^{-1}U_1^\top b + V_2 y_2,\ \forall y_2 \in \mathbb{R}^{n-r}\right\}$. The unique x minimizing $\|Ax-b\|_2$ and $\|x\|_2$ is gotten by $y_2 = 0$:

$$x = V_1\Sigma_1^{-1}U_1^\top b$$

In practice, Σ_2 is often set to be all singular values less than a user-defined threshold.

Definition 8.1. $A^+ = V_1\Sigma_1^{-1}U_1^\top$ is Moore–Penrose pseudo inverse of A (include full rank case $n = r$)

So square or not, full rank or not, best solution $x = A^+ b$.

Proof.

$$\begin{aligned} \|Ax - b\|_2 &= \left\|U\Sigma V^\top x - b\right\|_2 \\ &= \left\|\Sigma V^\top x - U^\top b\right\|_2 \\ &= \left\|\Sigma y - U^\top b\right\|_2 \end{aligned}$$

$y = V^\top x$ and x have same norm. Since V orthogonal, so finding LS minimizing $\|y\|_2$ also minimizes the following:

$$\|Ax - b\|_2 = \left\| \begin{bmatrix} \Sigma_1 y_1 - U_1^T b \\ -U_2^T b \\ -U_3^T b \end{bmatrix} \right\|_2$$

where $y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ minimized by $y_1 = \Sigma^{-1} U_1^\top b$ and $\|y\|_2^2 = \|y_1\|_2^2 + \|y_2\|_2^2$ minimized by $y_2 = 0$.

$$\begin{aligned} x &= Vy \\ &= V_1 y_1 + V_2 y_2 \\ &= V_1 y_1 \\ &= V_1 \Sigma^{-1} U_1^\top b \end{aligned}$$

□

8.2 Solving LS when Matrix nearly Rank Deficient with Truncated SVD

We have defined the condition number of a matrix $\kappa(A)$ as $\frac{\sigma_{\max}}{\sigma_{\min}}$, where $\sigma_{\min} = 0$ if for a rank deficient matrix, so the condition number is formally infinite.

Example 8.1

Considering

$$\arg\min \| \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \|_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and

$$\arg\min \| \begin{bmatrix} 1 & 0 \\ 0 & e \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \|_2 = \begin{bmatrix} 1 \\ 1/e \end{bmatrix}$$

The solutions are arbitrarily different as $e \to 0$.

What does a solution mean if it can change discontinuously? Often A is not known exactly. Just up to some tolerance $\|A - A'\| \le \text{tol}$. If A nearly singular, what do you do?

Definition 8.2 (Truncated SVD)**.** If $A = U\Sigma V^\top$ is SVD of A, then,

$$A(\text{tol}) = U\Sigma(\text{tol})V^\top$$

where $\Sigma(\text{tol}) = \text{diag}\left(\sigma_1(\text{tol}), \sigma_2(\text{tol}), \cdots, \sigma_n(\text{tol})\right)$,

$$\sigma_i\left(\text{tol}\right) = \begin{cases} \sigma_i \text{ if } \sigma_i \ge \text{tol} \\ 0 \text{ if } \sigma_i < \text{tol} \end{cases}$$

i.e. we truncate small singular values to 0. Or, we replace A by lowest rank matrix $A(\text{tol})$ within distance tol. Using $A(\text{tol})$ for LS effectively reduces $k = \frac{\sigma_{\max}}{\sigma_{\min}}$ to $\frac{\sigma_{\max}}{\text{tol}}$. Tol is a knob that user can pick to trade off sensitivity and how well B can be approximated by $A(\text{tol})x$, because $\text{span}(A(\text{tol}))$ decreases as tol increases. Replacing A by an easier matrix also called regularization, several mechanisms.

Using the truncated SVD effectively replaces the usual condition number $\frac{\sigma_{\max(A)}}{\sigma_{\min(A)}}$ by $\frac{\sigma_{\max(A)}}{\text{tol}}$, which can be much smaller, depending on the choice of tol. In other words, tol is a "knob" one can turn to trade off conditioning with how closely Ax can approximate b (raising tol, and so lowering the rank of A, decreases the dimension of the space that can be approximated by Ax).

Replacing A by an "easier" matrix is also called "regularization" (using the truncated SVD is just one of several mechanisms). The following Lemma illustrates this in a special case, where just vector b changes. The general case depends on whether one picks tol in a "gap" between singular values, since otherwise a small perturbation could change the number of singular values is greater than tol.

Lemma 8.1

The difference between

$$x_1 = \arg\min_x \|A(\text{tol})x - b_1\|_2$$

and

$$x_2 = \arg\min_x \|A(\text{tol})x - b_2\|_2$$

where we choose the solution of smallest norm in both cases, is bounded by $\|b_1 - b_2\|_2 / \text{tol}$, i.e.

$$\|x_1 - x_2\|_2 \leq \frac{\|b_1 - b_2\|}{\text{tol}}$$

So choosing a larger tol limits the sensitivity of the solution.

Proof. Let $A = U\Sigma V^\top$ and $A(\text{tol}) = U\Sigma(\text{tol})V^\top$ as above. Then

$$\begin{aligned} \|x_1 - x_2\|_2 &= \left\|A(\text{tol})^+(b_1 - b_2)\right\|_2 \\ &= \left\|V(\Sigma(\text{tol}))^+U^\top(b_1 - b_2)\right\|_2 \\ &= \left\|\text{diag}\left(1/\sigma_1, 1/\sigma_2, \cdots, 1/\sigma_k, 0, \cdots\right)U^\top(b_1 - b_2)\right\| \end{aligned}$$

where $\sigma_k \geq \text{tol}$,

$$\begin{aligned} &\leq \frac{1}{\sigma_k}\left\|U^\top(b_1 - b_2)\right\|_2 \\ &= \frac{1}{\sigma_k}\|b_1 - b_2\|_2 \\ &\leq \frac{1}{\text{tol}}\|b_1 - b_2\|_2 \end{aligned}$$

□

So that tells us that increasing the tolerance lowers the sensitivity. We can also analyze the effect of changing A on $A(\text{tol})$ and see that it depends discontinuously on whether the rank of $A(\text{tol})$ changes.

Specifically, we can't have tol be too close to an existing singular value σ_i, otherwise changing the matrix slightly could make σ_i cross the threshold of tolerance and change the rank.

Setting small singular values to zero also compresses the matrix, since it costs just $(mk + k + kn)$ numbers to store, as opposed to mn. So tol is also a "knob" that trades off compression with approximation accuracy.

The SVD is the most precise and most accurate way to approximate a matrix by one of lower rank, costing $O(mn^2)$, with a big constant. Now we explore other cheaper ones, both deterministic and randomized.

8.3 Solving a Least Squares Problem when A is (nearly) Rank Deficient

8.3.1 With Ridge Regression

Another common way to deal with the drawback of the solution becoming unbounded as the smallest singular values of A approach 0, is regularization, or computing

$$\begin{aligned} x &= \arg\min_x \|Ax - b\|_2^2 + \lambda^2 \|x\|_2^2 \\ &= \arg\min_x \left\| \begin{bmatrix} A \\ \lambda I \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|_2^2 \end{aligned}$$

where λ is a "tuning parameter" to be chosen by the user. The larger λ is chosen, the more weight is placed on $\|x\|_2$, so the more important it is to minimize relative to $\|Ax - b\|_2$.

We can easily write down the normal equations:

$$x = \left(A^\top A + \lambda^2 I\right)^{-1} A^\top b$$

Adding λ^2 to the diagonal of $A^\top A$ before Cholesky just increases all the eigenvalues of $A^\top A$ by λ^2, pushing it farther away from being singular. If $A = U\Sigma V^\top$ is the thin SVD of A, then substituting this for A yields

$$\begin{aligned} x &= V(\Sigma(\Sigma^2 + \lambda^2 I)^{-1})U^\top b \\ &= V \operatorname{diag}\left(\frac{\sigma_i}{\sigma_i^2 + \lambda^2}\right) U^\top b \end{aligned}$$

which reduces to the usual solution when $\lambda = 0$. Note that when $\sigma_i \gg \lambda$, the above expression is close to $\frac{1}{\sigma_i}$ as expected, and when $\sigma_i < \lambda$, it is bounded by $\frac{\sigma_i}{\lambda^2}$, so it can't get larger than $\frac{1}{\lambda}$. Thus λ and tol in $A(\text{tol})$ play similar roles.

8.3.2 With QR Decomposition

Our next alternative to the truncated SVD is QR with column pivoting. Suppose we did $A = QR$ exactly, with $\text{rank}(A) = r < n$, what would R look like? If the leading r columns of A were full rank (true for "most" such A), then

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

with R_{11} is $r \times r$ and full rank, so R_{22} to zero. If A is nearly low-rank, we can hope that $\|R_{22}\| <$ tol, and set R_{22} to zero. Assuming that this works for a moment, write

$$A = QR = [Q, Q'] \begin{bmatrix} R_{11} \\ 0 \end{bmatrix},$$

with $[Q, Q'] = [Q_1, Q_2, Q']$ square and orthogonal as before, with Q_1 $m \times r$, Q_2 $m \times (n-r)$ and Q' $m \times (m-n)$. Thus

$$\begin{aligned} \arg\min_x \|Ax - b\|_2 &= \arg\min_x \left\| [Q_1, Q_2, Q'] \begin{bmatrix} R \\ 0 \end{bmatrix} x - b \right\|_2 \\ &= \arg\min_x \left\| \begin{bmatrix} R_{11}x_1 + R_{12}x_2 & -Q_1^T b \\ & -Q_2^\top b \\ & -Q'^\top b \end{bmatrix} \right\|_2 \end{aligned}$$

How do we pick x_2 to minimize $\|x\|_2$?

Example 8.2

$A = \begin{bmatrix} e & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$, $b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, $x = \begin{bmatrix} (b_1 - x_2)/e \\ x_2 \end{bmatrix}$. And so if e is tiny, we better pick x_2 carefully (close to b_1) to keep $\|x\|$ small. But if we permute the columns of A to AP and minimize $\|AP\hat{x} - b\|_2$ we get

$$AP = \begin{bmatrix} 1 & e \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}.$$

Thus

$$x = \begin{bmatrix} b_1 - ex_2 \\ x_2 \end{bmatrix}$$

So $\|x\|$ is much less sensitive to the choice of x_2.

What would a "perfect" R factor look like? We know the SVD gives us the best possible answer, so comparing $\begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$ to $\begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix}$ makes sense. These observations motivate the following (**INFORMAL**) definition.

Definition 8.3. A Rank Revealing QR Factorization ($RRQR$) of A is $AP = QR$ where P is a permutation, Q orthogonal, $R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$ with R_{11} $k \times k$ where

1. R_{22} is "small", ideally $\sigma_{\max}(R_{22}) = O(\sigma_{k+1}(A))$, i.e. R_{22} "contains" the $n-k$ smallest singular values of A

2. R_{11} is "large", ideally $\sigma_{\min}(R_{11})$ not much smaller than $\sigma_k(A)$

   If in additional to the above two, we have:

3. $\|R_{11}^{-1}R_{12}\|$ is "not too large", then we call $AP = QR$ a "strong rank revealing QR"

Theorem 8.2 (INFORMAL)

If the above 3 statements hold, then for $i = 1 : k$,

$$\sigma_i(A) \geq \sigma_i(R_{11}) \geq \frac{\sigma_i(A)}{\sqrt{1 + \left\| R_{11}^{-1} R_{12} \right\|_2^2}},$$

and for $i = k + 1 : n$,

$$\sigma_i(A) \leq \sigma_{\max}(R_{22}) \sqrt{1 + \left\| R_{11}^{-1} R_{12} \right\|_2^2}$$

In other words, the singular values of R_{11} are good approximations of the largest k singular values of A, and the smallest $n - k$ singular values of A are roughly bounded by $\|R_{22}\|$. Or, the leading k columns of AP contain most of the information of the range space of A:

$$\begin{aligned} AP = [A_1, A_2] = QR &= [Q_1, Q_2] \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \\ &\approxeq [Q_1, Q_2] \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \\ &= Q_1 R_{11} \left[I, R_{11}^{-1} R_{12} \right] \\ &= A_1 \left[I, R_{11}^{-1} R_{12} \right] \end{aligned}$$

8.4 Computing the Permutation

8.4.1 QR with Column Pivoting

QRCP is the oldest and simplest way, and often works, but like LU with partial pivoting, there are some rare matrices where it fails by a factor near 2^n. And it maximizes communication. At the first step, the algorithm chooses the column of A of largest norm; at each following step, the algorithm picks the column with the biggest component orthogonal to all the columns already chosen. Intuitively, this "greedy algorithm" picks the column with the biggest projection in a direction not spanned by previously chosen columns. The algorithm is simply

Algorithm 8.1 (QRCP).

```
for $i = 1 : \min(m-1, n)$ or or until the trailing submatrix ($R_2 2$) is small enough do
    Choose largest column in trailing matrix, i.e. $\arg\max_{j \geq i} \|A(i:m, j)\|$
    if $i \neq j$ then
        Swap columns $i$ and $j$
    end if
    Multiply $A(i:m, i:n)$ by a *Householder reflection* to zero out $A(i+1:m, i)$
end for
```

If the algorithm stops after k steps, because the trailing matrix $A(k+1 : m, k+1 : n)$ has no column larger than some tolerance, the cost is about $4mnk$, versus $O(mn^2)$ for the SVD, so much cheaper if $k \sim \text{rank}(A) \ll n$.

To understand why this works: the first multiplication by a Householder reflection decomposes trailing columns into a part parallel to first column (now just parallel to e_1) and an orthogonal part (in rows $2 : m$). Choosing the column of biggest norm in rows $2 : m$ chooses the column with the biggest component orthogonal to the first column.

At each step we similarly choose the trailing column with the biggest component orthogonal to the previously chosen columns. The arithmetic cost is low, since we can just update $\|A(i : m, j)\|$ from iteration to iteration for O(m*n) cost, rather than recomputing them for $O(mn^2)$ cost. But this simple algorithm "maximizes communication", since we read and write the entire trailing matrix $A(i : m, i : n)$ at each step. This is available in *LAPACK geqpf* and *MATLAB* `[Q,R,P]=qr(A)`.

Here is a typical example, which shows how well each $R(i, i)$ approximates σ_i:

As a (rare) example where *QRCP* fails to pivot well, consider $A = SCD$ where

i.e. C has ones on the diagonal and $-cs$ above the diagonal. sn and cs satisfy $sn^2 + cs^2 = 1$. D is not necessary in exact arithmetic, but avoids roundoff problems. A is upper triangular, and *QRCP* does not permute any columns, so $A = QR = IA$. Letting $k = n - 1$ yields $R_{22} = sn^{n-1}$ but

$$\begin{aligned}\sigma_n(A) &= \frac{1}{\|A^{-1}\|} \sim \frac{1}{\|C^{-1}S^{-1}\|} \le \frac{1}{\|C^{-1}(:, n)sn^{1-n}\|} \\ &< \frac{sn^{1-n}}{cs(1 + cs)^{n-2}}\end{aligned}$$

So $\sigma_n(A)$ can be smaller than R_{22} by an exponentially large factor $cs(1 + cs)^{n-2}$, which can grow as fast as 2^{n-2} when $cs \sim 1$. So *QRCP* has two weaknesses: (rare) failure to pivot well, and high communication cost. We address these in turn.

8.5 Fixing Pivoting for QR

8.5.1 Gu/Eisentat Strong RRQR Algorithm

This algorithm deals with the rare failure to pivot correctly. It uses a more complicated pivoting scheme, that depends on the norms of columns of R_{22}, rows of R_{11}^{-1}, and entries of $R_{11}^{-1}R_{12}$, and guarantees a strong *RRQR*. It does so by cheaply exchanging a column in R_{11} and another column not in R_{11} if that increases $\det(R_{11})$ by a sufficient factor. It still costs only $4mnk$ (plus lower order terms), about the same as *QRCP* when $m \gg n$.

8.5.2 Avoiding Communication in QR with Column Pivoting

Neither of the last two algorithms was designed with minimizing communication in mind, and so both access the entire matrix each time a column is chosen, and so the number of read/writes is also $O(mn^2)$, same as the number of flops, instead of the hoped for factor of $\sqrt{\text{fast\_memory\_size}}$ smaller.

The first attempt to fix this is in *LAPACK geqp3.f.* This uses matrix multiply to update the trailing submatrix, as do LU and plain QR, but only reduces the number of reads/writes by two times compared to the simple routine. Still, it is often faster.

But to approach the lower bound, we seem to need a different pivoting strategy, which can choose multiple pivot columns for each matrix access, not just 1. The approach is similar to the *TSLU* algorithm described earlier. We present the sequential version, which chooses b pivot columns with one pass through the matrix (the parallel version is analogous). b is a blocksize to be chosen for accuracy/performance.

Algorithm 8.2.

```
BestColumnsSoFar = $(1 : b)$
for $k = b + 1 : n - b + 1 : b$ do                    ▷ Assume $b|n$ for simplicity
    Form $m \times 2b$ matrix $A_{2b}$ from columns in BestColumnsSoFar and columns $k : k + b - 1$
    Choose the $b$ "best columns" among the $2b$ columns in $A_{2b}$[a]
end for
```

After each outer iteration, BestColumnsSoFar contains the indices of the b best columns found so far among columns 1 to $k + b - 1$. The parallel version takes pairs of $m \times b$ submatrices, chooses the best b columns from each set of $2b$, and proceeds to pair these up and choose the best (which is why we call it "tournament pivoting" by analogy to having a "tournament" where at each round we choose the best). However, the flop count roughly doubles compared to *QRCP*.

8.6 Low Rank Factorization without Orthogonality

So far we have only considered low rank factorizations where (at least) one factor is an orthogonal matrix, say Q in QR. Q can be thought of as linear combinations of columns of A, which approximate the column space of A. But not all data analysis questions can best be answered by such linear combinations. Suppose the rows represent individual people, and the columns represent some of their characteristics, like age, height and income. If one of the columns of Q happens to be

$$.2\text{age} - .3\text{height} + .1\text{income} + \cdots$$

then it can be hard to interpret what this means, as a "predictor" of the other columns/characteristics, like "been treated for disease X". And if there are thousands of columns, it is even harder. Instead, it would be good to be able to approximate the other columns by linear combinations of as few columns as possible, and analogously to approximate the other rows by a subset of the rows. This leads to the following decomposition.

Definition 8.4. A CUR decomposition of a matrix A is consists of the matrices:

- C is a subset of k columns of A
- R is a subset of k rows of A
- U is a $k \times k$ matrix

where $\|A - CUR\|$ is "small", i.e. close to the lower bound σ_{k+1}, which is attained by the SVD truncated to rank k.

A number of algorithms for computing a CUR decomposition have developed over time, see the class webpage for details. Here we highlight two, because they are easy to implement given the tools we have already presented:

[a] For example by: factor $A_{2b} = QR$, using *TSQR*. Choose best b columns of R (just $2b \times 2b$), using *RRQR* or *Strong RRQR*, update BestColumnsSoFar based on result.

1. Perform QR with some kind of column pivoting, to pick the k "most linearly independent" columns of A; let $J = [j_1, j_2, \cdots, j_k]$ be the indices of these columns, and let C consist of these columns of A.
2. Perform *GEPP*, or *TSLU*, on C, to pick the k most linearly independent rows of C; let $I = [i_1, i_2, \cdots, i_k]$ be the indices of these rows, and let R consist of these rows of A.

Having chosen C and R, we still need to choose U so that CUR approximates A. Here are two approaches:

1. The best possible U is given by the solution to HW 3.12: the U that minimizes the Frobenious norm of $A - CUR$ is $U = C^+AR^+$.
2. A cheaper approximation is just to choose U so that CUR equals A in columns J and rows I. Since the 3 k-by-k matrices $C(I, 1:k) = R(1:k, J) = A(I, J)$ are equal, we just let U be the inverse of this common matrix.

8.7 Randomized Linear Algebra

Now we consider randomized algorithms. Related reading is posted on the class webpage. The basic idea for many randomized algorithms is as follows: Let Q be an $m \times k$ random orthogonal matrix, with $k \ll n$. Then we approximate A by $Q\left(Q^\top A\right)$, the projection of A's columns onto the space spanned by Q. Since $Q\left(Q^\top A\right)$ has $k \ll n$ columns, solving the LS problem is much cheaper than with A. Multiplying $Q^\top A$ costs $2mnk$ flops if done straightforwardly, which is only about 2 times cheaper than *QRCP* above. So there has been significant work on finding structured or sparse Q to make computing $Q^\top A$ much cheaper. To date the best (and surprising result) says that you can solve a LS problem approximately, where A is sparse, with just $O(\text{nnz}(A))$ flops, where $\text{nnz}(A)$ is the number of nonzeros in A.

We give some motivation for why such a random projection should work by some low-dimensional examples, and then state the *Johnson-Lindenstrauss Lemma*, a main result in this area.

Example 8.3

Suppose x is a vector in $\mathbb{R}^2$, and q is a random unit vector in $\mathbb{R}^2$, i.e. $q = [\sin t, \cos t]^\top$ where t is uniformly distributed on $[0, 2\pi)$. What is the distribution of

$$|x^\top q|^2 = (\|x\|_2 \, |\cos(\angle(x,q))|)^2$$

It is easy to see that $\angle(x,q)$ is also uniformly distributed on $[0, 2\pi)$, so the expected value is

$$\mathbb{E}(|x^\top q|^2) = .5 \, \|x\|_2^2 \, ,$$

and more importantly, the probability that $|x^\top q|^2$ underestimates $\|x\|_2^2$ by a tiny factor $e \ll 1$, is $\mathbb{P}(|x^\top q|^2 < e\|x\|_2^2) = \mathbb{P}(|\cos(t))|^2 < e) \sim 2\sqrt{e}/\pi$, so tiny too.

Example 8.4

Now suppose x is a vector in $\mathbb{R}^3$, and Q represents a random plane, i.e. Q is a random 3×2 orthogonal matrix, and $x^\top Q$ is the projection of x onto the plane of Q. We again ask how well the size of the projection $\left\|x^\top Q\right\|_2^2$ approximates $\|x\|_2^2$.

Now the probability that $\left\|x^\top Q\right\|_2^2 < e\,\|x\|_2^2$ is the same as the probability that x is nearly parallel to the perpendicular to the plane of Q, which can be thought of as a random point on the unit sphere in $\mathbb{R}^3$. This probability is $O(e)$, much tinier, because x needs to be nearly orthogonal to both columns of Q.

Intuitively, as the number of columns of Q increases, the chance that $\left\|x^\top Q\right\|_2^2$ greatly underestimates $\|x\|_2^2$ decreases rapidly. The *Johnson-Lindenstrauss Lemma* captures this, not just for one vector x, but for many vectors.

Lemma 8.2 (Johnson-Lindenstrauss Lemma)

Let $0 < \varepsilon < 1$, and $x_1, \cdots, x_n$ be any n vectors in $\mathbb{R}^m$, and $k \geq 8\ln(n)/\varepsilon^2$. Let F be a random $k \times m$ orthogonal matrix multiplied by $\sqrt{m/k}$. Then with probability at least $1/n$, for all $1 \leq i$, $j \leq n$, $i \neq j$,

$$1 - \varepsilon \leq \|F(x_i - x_j)\|^2 \,/\, \|x_i - x_j\|^2 \leq 1 + \varepsilon$$

The point is that for Fx to be an ε approximation of x in norm, the number of rows of F grows "slowly", proportional to $\ln(n) = \ln(\#\text{vectors})$ and $1/\varepsilon^2$. The probability $1/n$ seems small, but being positive it means that F exists (the original goal of Johnson and Lindenstrauss). And it comes from showing that the probability of a large error for any one vector $x_i - x_j$ is tiny, just $2/n^2$. This justifies using a single random F in the algorithms below.

(The proof follows by observing that we can think of F as fixed and each vector $x = x_i - x_j$ as random, and simply take F as the first k rows of the mxm identity, and each entry of x an i.i.d. (independent, identically distributed) Gaussian $\mathcal{N}(0,1)$, i.e. with 0 mean and unit standard deviation, reducing the problem to reasoning about sums of squares of i.i.d. $\mathcal{N}(0,1)$ variables. See An Elementary Proof of a Theorem of Johnson and Lindenstrauss for details.

There is a range of different F matrices that can be used, which tradeoff cost and statistical guarantees, and are useful in different applications. [17]

One can construct a random orthogonal $m \times k$ matrix Q, with $m \geq k$, simply as follows: Let A be $m \times k$, with each entry i.i.d. $\mathcal{N}(0,1)$, and factor $A = QR$. Then $F = \sqrt{\frac{m}{k}}Q$ in the J-L Lemma. But this costs $O(mk^2)$ to form F, which is too expensive in general.

In some applications, it is enough to let each entry of F be i.i.d. $\mathcal{N}(0,1)$, without doing QR until later in the algorithm, we will see an example of this below. But multiplying Fx still costs $O(mk)$ flops, when x is dense.

The next alternative is the *subsampled randomized Fourier Transform* (*SRFFT*), for which Fx only costs $O(m\log(m))$ or even $O(m\log(k))$. In this case $F = R\mathrm{FFT}D$ where D is $m \times m$ diagonal with

[17] See Randomized Numerical Linear Algebra: Foundations and Algorithms and Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions for surveys.

entries uniformly distributed on the unit circle in the complex plane, FFT is the $m \times m$ *Fast Fourier Transform*, R is $k \times m$, a random subset of k rows of the $m \times m$ identity. There are also variations in the real case where the FFT is replaced by an even cheaper real (scaled) orthogonal matrix, the *Hadamard transform*, all of whose entries are ± 1, and D's diagonal is also ± 1; this is called *SRHT*. The only randomness is in D and R. The intuition for why this works is that multiplying by D and then the FFT "mixes" the entries of x sufficiently randomly that sampling k of them (multiplying by R) is good enough.

Finally, when x is sparse, we would ideally like an algorithm whose cost only grew proportionally to $\text{nnz}(x)$. In the real case, we can take $F = SD$ where D is $m \times m$ diagonal with entries randomly ± 1 (analogous to above), S is $k \times m$, where each column is a randomly chosen column of the $k \times k$ identity. Note that S and R above are similar, but not the same. Multiplying $y = SDx$ can be interpreted as initializing $y = 0$ and then adding or subtracting each entry of x to a randomly selected entry of y. If x is sparse, the cost is $\text{nnz}(x)$ additions or subtractions. This is called Randomized Sparse Embedding in the literature. This F is not as "statistically strong" as the F's described above, so we may need to choose a larger k to prove any approximation properties.

8.8 Apply to Least Squares

Consider the dense least squares problem $x_{\text{true}} = \arg\min_x \|Ax - b\|_2$ where A is $m \times n$ and $m \geq n$. We approximate this by

$$x_{\text{approx}} = \arg\min_x \|F(Ax - b)\|_2 = \arg\min_x \|(FA)x - Fb\|_2 .$$

Using results based on the J-L Lemma, we can take F to have $k = n\log(n)/\varepsilon^2$ rows in order to get

$$\|Ax_{\text{approx}} - b\|_2 \leq (1 + \varepsilon)\, \|Ax_{\text{true}} - b\|_2 ,$$

in other words the residual is nearly as small as the true residual with high probability. But there is no guarantee about how far apart x_{true} and x_{approx} are.

Now we consider the cost. Given a dense F, forming FA using dense matrix multiplication costs

$$O(kmn) = O(mn^2 \log(n)/\varepsilon^2),$$

so more than solving the original problem using QR, $O(mn^2)$. If we use *SRFFT* or *SRHT* and A is dense, forming FA costs just

$$O(mn\log(m)).$$

Next, FA has dimension $k \times n$ and so solving using QR costs $O(kn^2) = O(n^3 \log(n)/\varepsilon^2)$, for a total cost of

$$O(mn\log(m) + n^3 \log(n)/\varepsilon^2),$$

potentially much less than $O(mn^2)$ when $m \gg n$ and ε is not too small. In other words, if high accuracy is needed (ε is tiny), a randomized algorithm like this may not help.

Finally, we mention a least squares algorithm that costs just $O(\mathrm{nnz}(A))$, plus lower order terms. [18] This uses the F called a *Randomized Sparse Embedding* above, with

$$k = O((n/\varepsilon)^2 * log^6(n/\varepsilon))$$

Then forming FA and Fb costs $\mathrm{nnz}(A)$ and $\mathrm{nnz}(b)$ respectively, much less than $SRFFT$ when A is sparse. But note that k grows proportionally to n^2, much faster than with the $SRFFT$, where $k = O(n)$. If we solved

$$\arg\min_x \|(FA)x - Fb\|_2$$

using dense QR, that would cost

$$O(kn^2) = O(n^4 \log^6(n/\varepsilon)/\varepsilon^2),$$

which is much larger than with $SRFFT$. So instead we use a randomized algorithm again (say $SRFFT$) to solve the problem.

Theorem 8.3

With probability at least 2/3,

$$\|Ax_{\text{approx}} - b\|_2 \le (1+\varepsilon)\,\|Ax_{\text{true}} - b\|_2$$

How can we make the probability of getting a good answer much closer to 1? Just run the algorithm repeatedly: After s iterations, the probability that at least one iteration will have a small error rises to $1-(1/3)^s$, and of the s answers $x_1, \cdots, x_s$, the one that minimizes $\|Ax_i - b\|_2$ is obviously the best one.

8.9 Randomized Algorithms for Low Rank Factorizations

Next we consider the problem of using a randomized algorithm for computing a low rank factorization of the $m \times n$ matrix A, with $m \ge n$. We assume we know the target rank k, but since we often don't know k accurately in practice, we will often choose a few more columns $k+p$ to see if a little larger (or smaller) k is more accurate.

This algorithm is of most interest when $k \ll n$, i.e. the matrix is low rank. Note that in the following, the F matrices will be tall-and-skinny, and applied on the right (e.g. AF), so the transpose of the cases so far.

8.9.1 Randomized Low-Rank Factorization

1. Choose a random $n(k+p)$ matrix F.
2. Form $Y = AF$, which is $m(k+p)$, we expect Y to accurately span the column space of A.

[18]For details see Low Rank Approximation and Regression in Input Sparsity Time and Low-distortion Subspace Embeddings in Input-sparsity Time and Applications to Robust Linear Regression.

3. Factor $Y = QR$, so Q also accurately spans the column space of A.
4. Form $B = Q^\top A$, which is $(k+p)n$.

We now approximate A by $QB = QQ^\top A$, the projection of A onto the column space of Q. In fact by computing the small SVD $B = U\Sigma V^\top$, we can write

$$QB = (QU)\Sigma V^\top$$

as an approximate SVD of A.

The best possible approximation for any Q is when Q equals the first $k+p$ left singular vectors of $A = U_A \Sigma_A V_A^\top$, in which case

$$QQ^\top A = U_A(1:m, 1:k+p)\Sigma_A(1:k+p, 1:k+p)(V_A(1:n, 1:k+p))^\top,$$

and

$$\left\| A - QQ^\top A \right\|_2 = \sigma_{k+p+1}.$$

But our goal is to only get a good rank k approximation, so we are willing to settle for an error like $\sigma(k+1)$.

Theorem 8.4

If we choose each $F(i,j)$ to be i.i.d. $\mathcal{N}(0,1)$, then

$$\mathbb{E}\left(\left\| A - QQ^\top A \right\|_2\right) \le (1 + \frac{4\sqrt{k+p}}{p-1}\sqrt{\min(m,n)})\sigma_{k+1},$$

and

$$P\left(\left\| A - QQ^\top A \right\|_2\right) \le (1 + 11\sqrt{k+p}\sqrt{\min(m,n)})\sigma_{k+1} \ge 1 - \frac{6}{p^p}$$

So for example choosing just p=6 makes the probability about .9999.

When is a *Randomized Low Rank Factorization* cheaper than a deterministic algorithm like *QRCP*, which costs $O(mn(k+p))$? When A is sparse, with $\text{nnz}(A)$ nonzero entries, the last 3 steps of the algorithm cost:

2. $2(k+p)\,\text{nnz}(A)$
3. $2m(k+p)^2$
4. $2(k+p)\,\text{nnz}(A)$

each of which can be much smaller than $mn(k+p)$.

Whether the cost of (3) dominates (2) and (4) depends on the density of A: if A averages $(k+p)$ or more nonzeros per row, then (2) and (4) dominate (3).

When A is dense, we need another approach. As mentioned above, forming an explicit dense F, and multiplying it by a dense AF, will cost $2mn(k+p)$ flops, comparable to *QRCP*. If we use *SRFFT* or *SRHT* for F, the cost of forming $Y = AF$ drops to $O(mn\log(n))$, potentially much less than *QRCP*. Factoring $Y = QR$ is still $O(m(k+p)^2)$, also potentially much less than *QRCP* when $k+p \ll n$. But the cost of $B = Q^\top A$ is still $O(mn(k+p))$, like *QRCP*. So we need another idea.

8.9.2 Randomized Low-Rank Factorization via Row Extraction

1. Choose a random nx(k+p) matrix F.
2. Form $Y = AF$, which is $mx(k+p)$; we expect Y to accurately span the column space of A.
3. Factor $Y = QR$, so Q also accurately spans the columns space of A.
4. Find the "most linearly independent" $k+p$ rows of Q; write $PQ = [Q_1, Q_2]^\top$ where P is a permutation and Q_1 contains these $k+p$ rows. We can use *GEPP* on Q or *QRCP* on $Q^\top$ for this.
5. Let $X = PQQ_1^{-1} = [I; Q_2Q_1^{-1}]$. We expect $\|X\|$ to be $O(1)$ (e.g. if *QRCP* yields a strong rank revealing decomposition).
6. Let $PA = [A_1; A_2]$ where A_1 has $k+p$ rows; our low rank factorization of A is $A \sim P^\top X A_1$.
7.

The cost of the algorithm on a dense matrix is:

2. $O(mn\log(n))$ or $O(mn\log(k+p))$ if F is *SRFFT* or *SRHT*;
3. $2m(k+p)^2$;
4. $2m(k+p)^2$;
5. $O(m(k+p)^2)$

which is much better than the previous $O(mn(k+p))$ cost when the rank $k+p$ is low compared to n.

The next theorem tells us that if *QRCP* or GEPP works well in step 5, i.e. $\|X\| = O(1)$, then we can't weaken the approximation by a large factor.

Theorem 8.5

$$\left\|A - P^\top X A_1\right\|_2 \le (1 + \|X\|) \left\|A - QQ^\top A\right\|_2$$

Proof. Assume $P = I$ for simplicity. Then

$$\begin{aligned}
\|A - XA_1\| &= \left\|A - QQ^\top A + QQ^\top A - XA_1\right\| \\
&\leq \left\|A - QQ^\top A\right\|_2 + \left\|QQ^\top A - XA_1\right\|_2 \\
&= \left\|A - QQ^\top A\right\|_2 + \left\|XQQ^\top A - XA_1\right\|_2 \\
&= \left\|A - QQ^\top A\right\|_2 + \|X\|_2 \left\|QQ^\top A - A_1\right\|_2 \\
&\leq \left\|A - QQ^\top A\right\|_2 + \|X\|_2 \left\|QQ^\top A - A\right\|_2 \\
&\leq (1 + \|X\|) \left\|A - QQ^\top A\right\|_2
\end{aligned}$$

□

9 Lecture 9: Eigenproblems

9.1 Eigenvalue Problems

Goals:

- Canonical Forms (recall Jordan, why we want Schur instead)
- Variations on eigenproblems (not always just one matrix!)
- Perturbation Theory (can I trust the answer?)
- Algorithms (for a single nonsymmetric matrix)

Recall basic definitions for a square $n \times n$ matrix A:

Definition 9.1. $p(\lambda) = \det(A - \lambda I)$ is the characteristic polynomial, whose n roots are the eigenvalues of A.

Definition 9.2. If λ is an eigenvalue, a nonzero null vector x satisfying $(A - \lambda I)x = 0$ must exist, i.e. $Ax = \lambda x$, and is called a right eigenvector. Analogously a nonzero null vector y^H must exist such that $y^H A = \lambda y^H$, and is called a left eigenvector.

Definition 9.3. If S is nonsingular, and $B = SAS^{-1}$, then S is called a similarity transformation, and A and B are called similar matrices.

Lemma 9.1

If A and B are similar, they have the same eigenvalues, and the eigenvectors are related by multiplying by S.

Proof. $Ax = \lambda x$ if and only if $SAS^{-1}Sx = S\lambda x$ or $B(Sx) = \lambda(Sx)$, i.e. if and only if λ is also an eigenvalue of B and Sx is a right eigenvector of B.

Analogously, $y^H A = \lambda y^H$ if and only if $y^H S^{-1} SAS^{-1} = \lambda y^H S^{-1}$ or $(y^H S^{-1})B = \lambda(y^H S^{-1})$, i.e. if and only if $y^H S^{-1}$ is a left eigenvector of B. □

Our goal will be to take A and transform it to a simpler similar form B, from which its eigenvalues and eigenvectors are easy to extract. The simplest form, for which eigenvalues and eigenvectors are obvious, is a diagonal matrix D, since $De_i = D_{ii}e_i$, where e_i is the i-th column of I.

Lemma 9.2

Suppose $Ax = \lambda_i x_i$ for $i = 1$ to n, and that the matrix $S = [x_1, \cdots, x_n]$ is nonsingular, i.e. x_i are n linearly independent eigenvectors. Then $A = S \operatorname{diag}(\lambda_1, \cdots, \lambda_n)S^{-1}$. Conversely, if $A = S\Lambda S^{-1}$, where Λ is diagonal, then the columns of S are eigenvectors and the Λ_{ii} are eigenvalues.

Proof. $A = S\Lambda S^{-1}$ if and only if $AS = S\Lambda$, if and only if the i-th columns of both sides are the same, i.e. $AS(:, i) = S(:, i)\Lambda_{ii}$. □

But we can't always make $B = SAS^{-1}$ diagonal, for two reasons:

- It may be mathematically impossible (recall Jordan form, with multiple eigenvalues)
- It may be numerically unstable (even if the Jordan form is diagonal)

Recall Jordan Form: for any A there exists a similar matrix $J = SAS^{-1}$ such that $J = \operatorname{diag}(J_1, \cdots, J_k)$ where each J_i is a Jordan block:

$$J_i = \begin{bmatrix} \lambda_i & 1 & 0 & & \cdots & 0 \\ 0 & \lambda_i & 1 & 0 & \cdots & 0 \\ & & \cdots & \cdots & & \\ 0 & \cdots & & & \lambda_i & 1 \\ 0 & \cdots & & & 0 & \lambda_i \end{bmatrix}$$

Up to permuting the order of the J_i, the Jordan form is unique. Different J_i can have the same eigenvalue λ (e.g. $A = I$). There is only one (right) eigenvector per J_i (namely $[0, \cdots, 0, 1, 0, \cdots, 0]^\top$ with the 1 in the same row as the top row of J_i)). So a matrix may have n eigenvectors (if there are n 1×1 J_i's; the matrix is called diagonalizable in this case) or fewer (in which case it is called defective). The number of times one λ appears on the diagonal is called its multiplicity. But we will not compute the Jordan form, for numerical reasons (though algorithms do exist).

The best we can generally hope for, as in earlier chapters, is backwards stability: Getting exactly the right eigenvalues and similarity S for a slightly perturbed input matrix $A+E$, where $\|E\| = O(\varepsilon)\|A\|$.

9.2 Backward Stable Approach

In the last chapter we said that as long as you multiply a matrix by orthogonal matrices, it is backward stable, i.e.

$$\mathrm{fl}\left(Q_k\left(Q_{k-1}\left(\cdots\left(Q_1 A\right)\right)\cdots\right)\right) = Q(A+E)$$

where Q is exactly orthogonal, and $\|E\| = O(\varepsilon)\|A\|$.

If we apply this to computing an orthogonal similarity transformation, we get

$$\mathrm{fl}\left(Q_k\left(\cdots\left(Q_2\left(Q_1 A Q_1^\top\right)Q_2^\top\right)\cdots\right)Q_k^\top\right) = Q(A+E)Q^\top$$

i.e. the exact orthogonal similarity of the slightly perturbed input $A+E$. This means that if we can restrict the similarity transforms S we use in SAS^{-1} to be orthogonal, we get backwards stability. So the question is: if we restrict S to be orthogonal, how close to Jordan form can we get?

Theorem 9.1 (Schur Canonical Form)
Given any square A there is a unitary Q such that $Q^H A Q = T$ is upper triangular. The eigenvalues are the diagonals T_{ii}, which can be made to appear on the diagonal of T in any order.

Note that eigenvectors are easy to compute from the Schur form if you need them: $Tx = T_{ii}x$ turns into solving a triangular system:

$$\begin{bmatrix} T_{11} & T_{12} & T_{13} \\ & T_{ii} & T_{23} \\ & & T_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = T_{ii} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

If there is only one copy of eigenvalue T_{ii}, then we can get the only solution with respect to each x_i one by one starting from the last row. If T_{ii} is a multiple eigenvalue, then $T_{11} - T_{ii}I$ might be singular, so we might not be able to solve $(T_{11} - T_{ii}I)x_1 = -T_{12}$, as expected.

Proof. We use induction: Let x be a right eigenvector with $\|x\|_2 = 1$, and let $Q = [x, Q']$ be any unitary matrix with x as its first column. Then

$$\begin{aligned} Q^H AQ &= \begin{bmatrix} x^H \\ Q'^H \end{bmatrix} A \left[x, Q'\right] \\ &= \begin{bmatrix} x^H Ax & x^H AQ' \\ Q'^H Ax & Q'^H AQ' \end{bmatrix} \\ &= \begin{bmatrix} \lambda x^H x & x^H AQ' \\ \lambda Q'^H x & Q'^H AQ' \end{bmatrix} \\ &= \begin{bmatrix} \lambda & x^H AQ' \\ 0 & Q'^H AQ' \end{bmatrix} \end{aligned}$$

Now we apply induction to the smaller matrix $Q'^H AQ'$ to write it as $U^H TU$ where T is upper triangular and U is unitary, so

$$\begin{aligned} Q^H AQ &= \begin{bmatrix} \lambda & x^H AQ' \\ 0 & U^H TU \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & U^H \end{bmatrix} \begin{bmatrix} \lambda & x^H AQ'U^H \\ 0 & T \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & U \end{bmatrix} \end{aligned}$$

as desired. □

9.3 Schur Form for Real Matrices

Real matrices can have complex eigenvalues (unless, say, they are symmetric, the topic of Chap 5). So T may have to be complex even if A is real; we'd prefer to keep arithmetic real if possible, for various reasons (reduce #flops, less memory, make sure complex eigenvalues and eigenvectors come in conjugate pairs despite roundoff). So instead of a real triangular T, we will settle for a real block triangular T:

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1k} \\ & T_{22} & \cdots & T_{2k} \\ & & \ddots & \vdots \\ & & & T_{kk} \end{bmatrix}$$

The eigenvalues of T are just the union of the eigenvalues of all the T_{ii}. We will show that we can reduce any real A to such a block triangular T where each T_{ii} is either 1×1 (so a real eigenvalue) or 2×2 (with two complex conjugate eigenvalues).

Theorem 9.2 (Real Schur Canonical Form)

Given any real square A, there is a real orthogonal Q such that $QAQ^\top$ is block upper triangular with 1×1 and 2×2 blocks.

To prove this we need to generalize the notion of eigenvector to "invariant subspace":

Definition 9.4. Let $V = \operatorname{span}\{x_1, \cdots, x_m\} = \operatorname{span}(X)$ be a subspace of $\mathbb{R}^n$. It is called an invariant subspace if $AV = \operatorname{span}(AX)$ is a subset of V.

Example 9.1

$V = \operatorname{span}\{x\} = \{\alpha x\}$ for any scalar α where $Ax = \lambda x$, then for any α,

$$\begin{aligned} AV &= \{A(\alpha x),\ \forall \alpha\} \\ &= \{\alpha\lambda x,\ \forall\alpha\} \subseteq V \end{aligned}$$

Example 9.2

$V = \operatorname{span}\{x_1, \cdots, x_k\} = \left\{\sum_{i=1}^k \alpha_i x_i,\ \forall \alpha_i\right\}$ where $Ax_i = \lambda_i x_i$, then

$$\begin{aligned} AV &= \left\{A\sum_{i=1}^k \alpha_i x_i,\ \forall\alpha_i\right\} \\ &= \left\{\sum_{i=1}^k \alpha_i\lambda_i x_i,\ \forall\alpha_i\right\} \\ &\subseteq V \end{aligned}$$

Lemma 9.3

If $V = \operatorname{span}\{x_1, \cdots, x_m\} = \operatorname{span}(X)$ is an m-dimensional invariant subspace of A (i.e. $x_1, \cdots, x_m$ are independent). Then there is an $m \times m$ matrix B such that $AX = XB$. The eigenvalues of B are also eigenvalues of A.

Proof. The existence of B follows from the definition of invariant subspace: Ax_i in V means there are scalars $B_{1i}, \cdots, B_{mi}$ (the i-th column of B) such that $Ax_i = \sum_{j=1}^m x_i B_{ji}$, i.e. $AX = XB$. If $By = \lambda y$, then $AXy = XBy = Xy\lambda$, so Xy is an eigenvector of A with eigenvalue λ. □

Lemma 9.4

Let $V = \operatorname{span}(X)$ be an m-dimensional invariant subspace of A as above, with $AX = XB$. Let $X = QR$, and let $[Q, Q']$ be square and orthogonal. Then

$$[Q, Q']^\top A[Q, Q'] = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix}$$

is block upper triangular with $A_{11} = RBR^{-1}$ having the same eigenvalues as B.

Proof.

$$[Q, Q']^\top A[Q, Q'] = \begin{bmatrix} Q^\top AQ & Q^\top AQ' \\ Q'^\top AQ & Q^{\top\prime} AQ' \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where $AQ = AXR^{-1} = XBR^{-1} = QRBR^{-1}$, so $A_{11} = Q^\top AQ = Q^\top QRBR^{-1} = RBR^{-1}$, and $A_{21} = Q'^\top QRBR^{-1} = 0$. □

Proof in Schur Form. We use induction as before. If $Ax = \lambda x$ where λ and x are real, we reduce to a smaller problem using the last Lemma. If λ and x are complex, it is easy to confirm that the real and imaginary parts of $Ax = \lambda x$ are equivalent to the first and second columns of $AX = XB$, where

$$X = [\mathrm{re}(x), \mathrm{im}(x)] \text{ and } B = \begin{bmatrix} \mathrm{re}(\lambda) & \mathrm{im}(\lambda) \\ \mathrm{im}(\Delta) & \mathrm{re}(\lambda) \end{bmatrix}$$

For $AX = BX$, the first column is real and the second column is imaginary. X invariant subspace, eigenvalues of B are λ and $\bar{\lambda}$. So by the Lemma we can do an orthogonal similarity on A to get $\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix}$ where the eigenvalues of A_{11} are λ and $\bar{\lambda}$, completing the induction. □

9.4 More General Eigenvalue Problems

We briefly review other kinds of eigenvalues that can arise, beyond a single square matrix. In Lecture 1, we pointed out that ODEs can give rise to a range of eigenvalue problems:

1. In the simplest case, the ODE $x'(t) = Kx(t)$ leads to the eigenvalue problem for K: if $Kx(0) = \lambda x(0)$, then $x(t) = e^{\lambda t}x(0)$, and similarly if $x(0)$ is expressed as a linear combination of eigenvectors.

2. When $Mx^{"}(t) + Kx(t) = 0$ and $\lambda^2 Mx(0) + Kx(0) = 0$, then

   $$x(t) = e^{\lambda t}x(0)$$

   This is a "generalized eigenproblem" for the pair (M, K), with eigenvalue λ^2 and eigenvector $x(0)$. The usual definition of an eigenvalue becomes $\det(\lambda' M + K) = 0$, where $\lambda' = \lambda^2$.

3. When $Mx^{"}(t) + Dx'(t) + kx(t) = 0$, we get the "nonlinear eigenproblem":

   $$\lambda^2 Mx(0) + \lambda Dx(\sigma) + Kx(0) = 0$$

   which can be reduced to a linear generalized eigenproblem of twice the size.

4. When $x'(t) = A \times (t) + Bu(t)$, a linear control system, the question of how to choose $u(t)$ to control $x(t)$ turns into a "singular eigenproblem" for the pair of rectangular matrices $[B, A]$ and $[0, I]$.

More generally, all the ideas of this chapter (eigenvalues, eigenvectors, Jordan form, Schur form, algorithms) extend to these more general eigenvalue problems, see section 4.5 of the textbook for details. We will only discuss the eigenproblem for one square matrix in detail.

Recall that the best we can hope for is backward stability: right answer (eigenvalues) for a slightly wrong problem $A + E$, where $\|E\| = O(\varepsilon)\,\|A\|$. How much can this change the eigenvalues and vectors?

Last time: showed that if eigenvalues close together, eigenvectors can be very sensitive (or disappear, or be nonunique, as for I). How about the eigenvalues?

Definition 9.5. The epsilon pseudo-spectrum of A is the set of all eigenvalues of all matrices within distance ε of A:

$$\Lambda_\varepsilon(A) = \{\lambda : (A+E)x = \lambda x \text{ for some } x \neq 0 \text{ and } \|E\|_2 \leq \varepsilon\}$$

Ideal case: $\Lambda_\varepsilon(A)$ is the union of disks of radius ε centered at eigenvalues of A

Worst case:

Theorem 9.3 (Trefethen & Reichel)

Given any simply connected region R in the complex plane, and point x in R, and any $\varepsilon > 0$, there is an A with one eigenvalue at x such that $\Lambda_\varepsilon(A)$ is as close to filling out R as you like.

Proof. Use Riemann Mapping Theorem. □

Example 9.3

Perturb $n \times n$ Jordan block at 0 by changing $J_{n1} = \varepsilon$, get eigenvalues on circle of radius $\varepsilon^{1/n}$, which is much greater than ε. This example shows

1. eigenvalues are not necessarily differentiable functions of matrix entries (slope of $\varepsilon^{1/n}$ is infinite at $\varepsilon = 0$), although they are continuous (and differentiable when not multiple);
2. gives intuition that we should expect a sensitive eigenvalue when it is (close to) multiple, as was the case for eigenvectors.

Theorem 9.4

Let λ be a simple eigenvalue, with $Ax = \lambda x$ and $y^H A = \lambda y^H$, and $\|x\|_2 = \|y\|_2 = 1$. If we perturb A to $A + E$ the λ is perturbed to $\lambda + \delta\lambda$, and

$$\delta\lambda = \frac{y^H E x}{y^H \times} + O\left(\|E\|^2\right)$$

$$\delta\lambda \leq \frac{\|E\|^2}{y^H x} + O\left(\|E\|^2\right)$$

where θ is the acute angle between x and y. So $\sec(\theta)$ is the condition number of λ.

Proof. Subtract $Ax = \lambda$ x from $(A+E)(x+\delta x) = (\lambda+\delta\lambda)(x+\delta x)$ to get

$$Ax + A\delta x + Ex + E\delta x = \lambda x + \lambda\delta x + \delta\lambda x + \delta\lambda\delta x$$

Ignore second order terms $E\delta x$ and $\delta\lambda\delta x$, and multiply by y^H to get

$$y^H A\delta x + y^H Ex = \lambda y^H \delta x + \delta\lambda y^H x$$

Cancel $y^H A\delta x = \lambda y^H \delta x$ to get $y^H Ex = \delta\lambda y^H x$ as desired. Note that a Jordan block has $x = e(1)$ and $y = e(n)$, so $y^H x = 0$ as expected. □

An important special case are real symmetric matrices (or more generally normal matrices, where $A^H A = AA^H$), since these have orthogonal eigenvectors.

Corollary 9.1

If A is normal, perturbing A to $A+E$ means

$$|\delta\lambda| \leq \|E\| + O\left(\|E\|^2\right)$$

Proof. $A = Q\Lambda Q^H$ is the eigendecomposition, where Q is unitary, so $AQ = Q\Lambda$, and the right eigenvectors are the columns of Q, and since $Q^H A = \Lambda Q^H$, and the left eigenvectors are also the columns of Q, $x = y$. □

Later, in Chapter 5, for real symmetric matrices $A = A^\top$ (or more generally complex Hermitian matrices $A = A^H$), we will show that if E is also symmetric, then $|\delta\lambda| \leq \|E\|$ no matter how big $\|E\|$ is. The above theorem is true for small $\|E\|$. It is possible to change it slightly to work for any $\|E\|$.

Theorem 9.5 (Bauer-Fike)

Let A have all simple eigenvalues (i.e. be diagonalizable). Call them λ_i with right and left eigenvectors x_i and y_i, normalized so $\|x_i\|_2 = \|y_i\|_2 = 1$. Then for any E the eigenvalues of $A+E$ like in the union of disks D_i in the complex plane, where D_i has center λ_i and radius $n\,\|E\|_2\,/|y_i^H x_i|$.

Note that this is just n times larger than the last theorem. Also note that if two disks D_i and D_j overlap, all the theorem guarantees is that there are two eigenvalues of $A+E$ in the union $D_i \cup D_j$ (the same idea applies if more disks overlap).

9.5 Algorithms for the Nonsymmetric Eigenproblem

Our ultimate algorithm, the Hessenberg QR algorithm, takes a nonsymmetric A and computes the Schur form $A = QTQ^H$, in $O(n^3)$ flops. We will build up to it with simpler algorithms, that will also prove useful as building blocks for the algorithms for sparse matrices, where we only want to compute a few eigenvalues and vectors. The Hessenberg QR algorithm will also be used as a building block for large sparse matrices, because our algorithms for them will approximate them (in a certain sense) by much smaller dense matrices, to which we will apply Hessenberg QR.

The plan is as follows:

- Power Method: Just repeated multiplication of a vector by A; we'll show this makes the vector converge to the eigenvector for the eigenvalue of largest absolute value, which we also compute.
- Inverse Iteration: Apply power method to $B = (A - \sigma I)^{-1}$, which has the same eigenvectors as A, but now the largest eigenvalue in absolute value of B corresponds to the eigenvalue of A closest to σ (which is called the "shift"). By choosing σ appropriately, this lets us get any eigenvalue of A, not just the largest.
- Orthogonal Iteration: This extends the power method from one eigenvector to compute a whole invariant subspace.
- QR iteration: we combine Inverse Iteration and Orthogonal Iteration to get our ultimate algorithm.

There are a lot of other techniques needed to make QR iteration efficient (run in $O(n^3)$) and reliable, as well as to reduce data movement. We will only discuss some of these.

9.5.1 Power Method

Given $x(0)$ (potentially randomly),

Algorithm 9.1 (Power Method).

```
1: $i = 0$
2: while not converged do
3:     $y_{i+1} = Ax_i$
4:     $x_{i+1} = y_{i+1}/\|y_{i+1}\|_2$          ▷ Approximate eigenvector
5:     $\lambda'_{i+1} = x_{i+1}^\top A x_{i+1}$          ▷ Approximate eigenvalue
6:     $i = i + 1$
7: end while
```

We first analyze convergence when $A = \operatorname{diag}(\lambda_1, \cdots, \lambda_n)$ where $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots$ and then generalize:

$$\begin{aligned} x_i &= \frac{A^i x_0}{\|A^i x_0\|_2} \\ &= \frac{1}{\|A^i x_0\|_2} \left[\lambda_1^i x_0(1), \lambda_2^i x_0(2), \cdots, \lambda_n^i x_0(n)\right] \\ &= \frac{1}{\|A^i x_0\|_2} \lambda_1^i \left[x_0(1), \left(\frac{\lambda_2}{\lambda_1}\right)^i x_0(2), \left(\frac{\lambda_3}{\lambda_1}\right)^i x_0(3) \cdots\right] \end{aligned}$$

As i grows, each $(\lambda_j/\lambda_1)^i$ converges to 0, and x_i converges to $[1, 0, \cdots, 0]$ as desired, with error $O(|\frac{\lambda_2}{\lambda_1}|^i)$, assuming $x_0(1) \neq 0$.

More generally, suppose A is diagonalizable, with $A = S\Lambda S^{-1}$, so

$$A^i = S\Lambda^i S^{-1}$$

Let $z = S^{-1}x_0$, so

$$\begin{aligned} A^i x_0 &= S\left[\lambda_1^i z_1, \lambda_2^i z_2, \cdots\right] \\ &= \lambda_1^i S\left[z_1, \left(\frac{\lambda_2}{\lambda_1}\right)^2 z_2, \cdots\right] \end{aligned}$$

As i increases, the above equation converge,

$$\lambda_1^i S\left[z_1, 0, \cdots, 0\right] = z_1 \lambda_1^i S\left(:, 1\right)$$

i.e. since $AS = S\Lambda$, the eigenvector of A for λ_1, as desired.

For this to converge to the desired eigenvector at a reasonable rate, we need

1. $|\frac{\lambda_2}{\lambda_1}| < 1$, and the smaller the better. This is not necessarily true, e.g. for an orthogonal matrix $AA^\top = I$, all the eigenvalues have absolute value 1 (since $\|x\| = \|Ax\| = \|\lambda x\|$, so there is no convergence.
2. z_1 nonzero, and the larger the better. If we pick x_0 at random, it is very unlikely that z_1 will be very tiny, but there are no guarantees.

To deal with needing $|\lambda_1| \gg |\lambda_2|$ to get fast convergence, we use inverse iteration, i.e. the power method on $B = (A - \sigma I)^{-1}$, where σ is called the shift.

9.5.2 Inverse Iteration

Given $x(0)$,

Algorithm 9.2 (Inverse Iteration).

```
1: $i = 0$
2: while not converged do
3:     $y_{i+1} = (A - \sigma I)^{-1} x_i$
4:     $x_{i+1} = y_{i+1} / \|y_{i+1}\|_2$          ▷ Approximate eigenvector
5:     $\lambda'_{i+1} = x_{i+1}^\top A x_{i+1}$          ▷ Approximate eigenvalue
6:     $i = i + 1$
7: end while
```

The eigenvectors of B are the same as those of A, but its eigenvalues are $\frac{1}{\lambda_i - \sigma}$. Suppose σ is closer to λ_k than any other eigenvalue of A. Then the same kind of analysis as above shows that x_i is gotten by the taking the following vector divided by its norm:

$$\begin{bmatrix} \left[\left(\lambda_k - \sigma\right) / \left(\lambda_1 - \sigma\right)\right]^i \cdot \frac{z_1}{z_k} \\ \left[\left(\lambda_k - \sigma\right) / \left(\lambda_2 - \sigma\right)\right]^i \cdot \frac{z_2}{z_k} \\ \vdots \\ 1 \\ \vdots \\ \left[\left(\lambda_k - \sigma\right) / \left(\lambda_n - \sigma\right)\right]^i \cdot \frac{z_n}{z_k} \end{bmatrix}$$

where the k-th component is 1. So if we can choose σ much closer to λ_k than any other λ_j, we can make convergence as fast as we want. Where do we get σ? Once we start converging, the algorithm itself computes an improving estimate of λ_k at each iteration; we will see later that this makes convergence very fast, quadratic or even cubic in some cases.

The next step is to compute more than one vector at a time. We do this first for the analogue of the power method.

9.5.3 Orthogonal Iteration

Given Z_0, an $n \times p$ orthogonal matrix,

Algorithm 9.3 (Orthogonal Iteration).

```
1: $i = 0$
2: while not converged do
3:     $Y_{i+1} = AZ_i$
4:     factor $Y_{i+1} = Z_{i+1}R_{i+1}$    ▷ QR decomposition, $Z_{i+1}$ spans an approximate invariant subspace
5:     $i = i + 1$
6: end while
```

Note the similarity to the randomized algorithms discussed previously, where Z_0 was chosen randomly; these techniques can be combined, though we don't discuss any more details.

Here is an informal analysis, assuming $A = S\Lambda S^{-1}$ is diagonalizable and

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_p| > |\lambda_{p+1}| \geq \cdots$$

i.e. the first p eigenvalues are larger in absolute value than the others, which is needed for convergence. Note that

$$\begin{aligned} \text{span}(Z_{i+1}) &= \text{span}(Y_{i+1}) = \text{span}(AZ_i) \\ &= \text{span}(A^i Z_0) \text{ by induction} \\ &= \text{span}(S\Lambda^i S^{-1} Z_0) \end{aligned}$$

Now,

$$\begin{aligned} S\Lambda^i S^{-1} Z_0 &= S\lambda_p^i \, \text{diag}\left(\underbrace{\left(\frac{\lambda_1}{\lambda_p}\right)^i, \left(\frac{\lambda_2}{\lambda_p}\right)^i, \cdots,}_{\geq 1} 1, \underbrace{\left(\frac{\lambda_{p+1}}{\lambda_p}\right)^i}, , \cdots \right) S^{-1} Z_0 \\ &= S\lambda_p^i \begin{bmatrix} V_i \\ W_i \end{bmatrix} \end{aligned}$$

where $\left(\frac{\lambda_j}{\lambda_p}\right)^i$ goes to infinity if $j < p$ and goes to 0 if $j > p$. Thus W_i goes to zero, and V_i does not. If V_0 has full rank, so will V_i. Thus

$$A^i Z_0 = \lambda_p^i S \begin{bmatrix} V_i \\ W_i \end{bmatrix}$$

approaches $\lambda_p^i S \begin{bmatrix} V_i \\ 0 \end{bmatrix}$. So it is a linear combination of the first p columns of S, i.e. the first p eigenvectors, the desired invariant subspace. Note that the first $k < p$ columns of Z_i are the same as though we had run the algorithm starting with the first k columns of Z_0, because the k-th column of Q and R in the QR decomposition of $A = QR$ only depend on columns $1 : k$ of A. In other words, Orthogonal Iteration runs p different iterations simultaneously, with the first k columns of Z_i converging to the invariant subspace spanned by the first k eigenvectors of A. Thus we can let $p = n$ and $Z_0 = I$, and try to compute n invariant subspaces at the same time. This will give us Schur Form.

Theorem 9.6

Run Orthogonal iteration on A with $Z_0 = I$. If all the eigenvalues of A have different absolute values, and if the principal submatrices $S(1 : k, 1 : k)$ of the matrix of eigenvectors of A all have full rank, then $A_i = Z_i^H A Z_i$ converges to Schur form, i.e. upper triangular with eigenvalues on the diagonal.

Proof. By the previous analysis, for each k, the span of the first k columns of Z_i converge to the invariant subspace spanned by the first k eigenvectors of A. Write $Z_i = [Z_{i1}, Z_{i2}]$ where Z_{i1} has k columns so

$$\begin{aligned} Z_i^H A Z_i &= \begin{bmatrix} Z_{i1}^H \\ Z_{i2}^H \end{bmatrix} A \left[Z_{i1}^H, Z_{i2}^H\right] \\ &= \begin{bmatrix} Z_{i1}^H A Z_{i1} & Z_{i,}^H A Z_{i2} \\ Z_{i2}^H A Z_{i1} & Z_{i2}^H A Z_{i2} \end{bmatrix} \end{aligned}$$

Now AZ_{i1} converges to $Z_{i1}B_i$ since Z_{i1} is converging to an invariant subspace, so $Z_{i2}^H A Z_{i1}$ converges to $Z_{i2}^H Z_{i1} B_i = 0$. Since this is true for every k, $Z_i^H A Z_i$ converges to upper triangular form T_i. Since Z_i is unitary, this is the Schur form. □

9.5.4 QR Iteration

Given $A_0 = A$,

Algorithm 9.4 (QR Iteration).

```
1: $i = 0$
2: while not converged do
3:     factor $A_i = Q_i R_i$                    ▷ QR decomposition
4:     $A_{i+1} = R_i Q_i$
5:     $i = i + 1$
6: end while
```

Note that

$$A_{i+1} = R_i Q_i = Q_i^\top Q_i R_i Q_i = Q_i^\top A_i Q_i$$

so that all the A_i are orthogonally similar.

Theorem 9.7

A_i from QR iteration is identical to $Z_i^\top A Z_i$ from Orthogonal iteration, starting with $Z_0 = I$. Thus A_i converges to Schur Form if all the eigenvalues have different absolute values.

Proof. We use induction: assume $A_i = Z_i^\top A Z_i$. Then taking one step of Orthogonal iteration we write $AZ_i = Z_{i+1}R_{i+1}$, the QR decomposition. Then

$$\begin{aligned} A_i &= Z_i^\top A Z_i \\ &= Z_i^\top Z_{i+1} R_{i+1} \end{aligned}$$

which is an orthogonal matrix multiply an upper triangular, so this must also be the QR decomposition of A_i (by uniqueness). Then

$$\begin{aligned} Z_{i+1}^\top A Z_{i+1} &= Z_{i+1}^\top A (Z_i Z_i^\top) Z_{i+1} \\ &= (Z_{i+1}^\top A Z_i)(Z_i^\top Z_{i+1}) \\ &= R_{i+1}(Z_i^\top Z_{i+1}) \\ &= RQ \\ &= A_{i+1} \end{aligned}$$

where $QR = Z_i^\top A Z_i$, i.e. we have taken one step of QR iteration. □

Now we show how to incorporate inverse iteration: Given $A_0 = A$,

Algorithm 9.5 (QR Iteration with a Shift).

```
1: $i = 0$
2: while not converge do
3:     Choose a shift $\sigma_i$ near an eigenvalue of $A$
4:     factor $A_i - \sigma_i I = Q_i R_i$          ▷ QR decomposition
5:     $A_{i+1} = R_i Q_i + \sigma_i I$
6:     $i = i + 1$
7: end while
```

Lemma 9.5

A_i and A_{i+1} are orthogonally similar.

Proof.

$$A_{i+1} = R_i Q_i + \sigma_i I = Q_i^\top Q_i R_i Q_i + \sigma_i I = Q_i^\top (A_i - \sigma_i I) Q_i + \sigma_i I = Q_i^\top A_i Q_i.$$

□

If R_i is nonsingular, we can also write

$$\begin{aligned} A_{i+1} &= R_i Q_i + \sigma_i I \\ &= R_i Q_i R_i R_i^{-1} + \sigma_i I \\ &= R_i (A_i - \sigma_i I) R_i^{-1} + \sigma_i I \\ &= R_i A_i R_i^{-1} \end{aligned}$$

If σ_i is an exact eigenvalue of A, we claim QR Iteration converges in one step: If $A_i - \sigma_i I$ is singular, then R_i is singular, so some diagonal entry of R_i must be zero. Suppose that the last diagonal entry $R_i(n,n) = 0$. Then the whole last row of R_i is zero, so the last row of $R_i Q_i$ is zero, so the last row of $A_{i+1} = R_i Q_i + \sigma_i I$ is zero except for $A_{i+1}(n,n) = \sigma_i$ as desired. This reduces the problem to one of dimension $n-1$, namely the first n-1 rows and columns of A_{i+1}. If σ_i is not an exact eigenvalue, we declare convergence when $A_{i+1}(n, 1:n-1)$ is small enough. From earlier analysis we expect this block to shrink in norm by a factor

$$|\lambda_k - \sigma_i| / \min_{j \neq k} |\lambda_j - \sigma_i|$$

where λ_k is the eigenvalue closest to σ_i.

Here is how to see that we are implicitly doing inverse iteration. For simplicity, we assume the eigenvalue is real. First, since $A_i - \sigma_i I = Q_i R_i$, we get

$$Q_i^\top (A_i - \sigma_i I) = R_i$$

So if σ_i is an exact eigenvalue, the last row of $Q_i^\top$ times $A_i - \sigma_i I$ is zero, and so the last column of Q_i is a left eigenvector of A_i for eigenvalue σ_i. Now suppose that σ_i is just close to an eigenvalue. Then $A_i - \sigma_i I = Q_i R_i$, so

$$\begin{aligned} (A_i - \sigma_i I)^{-1} &= R_i^{-1} Q_i^\top \\ (A_i - \sigma_i I)^{-\top} &= Q_i R_i^{-\top} \\ (A_i - \sigma_i I)^{-\top} R_i^\top &= Q_i \end{aligned}$$

and taking the last column of both sides we get that $(A_i - \sigma_i I)^{-\top} e_n$ and the last column of Q_i are parallel, i.e. the last column of Q_i is gotten by a step of inverse iteration, on $A_i^\top$, starting with e_n (the last column of I). Thus the last column of Q_i is closer to an eigenvector of $A_i^\top$.
$\Rightarrow$ the last column of $A_i^\top Q_i$ is closer to λ times the last column of Q_i;
$\Rightarrow$ the last column of $Q_i^\top A_i^\top Q_i$ is closer to λe_n;
$\Rightarrow$ the last row of $Q_i^\top A_i Q_i$ is closer to $\lambda e_n^\top$, i.e. tiny in the first $n-1$ entries, and close to λ on the diagonal.

So where do we get σ_i so that it is a good approximate eigenvalue? Since we expect $A_i(n,n)$ to converge to an eigenvalue, we pick $\sigma_i = A_i(n,n)$. We show that this in fact yields quadratic convergence, i.e. the error is squared at each step, so the number of correct digits doubles. To see why, suppose

$$\begin{aligned} \|A_i(n, 1:n-1)\| &= \varepsilon \ll 1 \\ |A_i(n,n) - \lambda_k| &= O(\varepsilon) \end{aligned}$$

for some eigenvalue λ_k, and that the other eigenvalues are much farther away than ε. By the above analysis, $\|A_i(n, 1 : n-1)\|$ will get multiplied by

$$|\lambda_k - \sigma_i| / \min_{j \neq k} |\lambda_k - \sigma_i| = O(\varepsilon),$$

and so on the next iteration $\|A_{i+1}(n, 1 : n-1)\| = O(\varepsilon^2)$.

9.6 Making QR Iteration Practical

With the following questions:

1. Each iteration has the cost of QR factorization plus matrix multipulication ($O(n^3)$). Even with just a constant number of iterations per eigenvalue, the cost is $O(n^4)$. We want a total cost of $O(n^3)$.
2. How do we pick a shift to converge to a complex eigenvalue, when the matrix is real, and we want to use real arithmetic? In other words, how do we compute the real Schur form?
3. How do we decide when we have converged?
4. How do we minimize data movement, the way we did for matrix multipulication, etc?

Here are the answers, briefly:

1. We preprocess the matrix by factoring it as $A = QHQ^\top$, where $A = QHQ^\top$ is orthogonal and H is upper Hessenberg, i.e. nonzero only on and above the first subdiagonal. It turns out that QR iteration on H leaves it upper Hessenberg, and lets us reduce the cost of one QR iteration from $O(n^3)$ to $O(n^2)$, and so the cost of n QR iterations to $O(n^3)$ as desired. When A is symmetric, so that H is upper Hessenberg and symmetric, it must be tridiagonal; this further reduces the cost of one QR iteration to $O(n)$, and of n iterations to $O(n^2)$.
2. Since complex eigenvalues of real matrices come in complex conjugate pairs, we can imagine taking one QR iteration with a shift sigma followed by one QR iteration with shift $\bar{\sigma}$. It turns out this bring us back to a real matrix, and by reorganizing the computation, merging the two QR iterations, we can avoid all complex arithmetic.
3. When any subdiagonal entry $H(i+1, i)$ is small enough,

   $$|H(i+1, i)| = O(\varepsilon) \|H\|,$$

   then we set it to zero, since this causes a change no larger than what roundoff does anyway. This splits the matrix into two parts, i.e. it is block upper Hessenberg, and we can deal with each diagonal block separately. If a block is 2×2 with complex eigenvalues, or 1×1, we are done.
4. No way is known to reduce the number of words moved to

   $$\Omega(\#\text{flops}/\sqrt{\text{memory\_size}}),$$

as we could for matrix multipulication, LU, and QR, using this algorithm, there is lots of recent research in trying to reduce memory traffic by trying to combine many QR iterations and interleave their operations in such a way as to get the same answer, but move less data (SIAM Linear Algebra Prize 2003, to Byers/Mathias/Braman). There are other algorithms that do move as little data as matrix multipulication, using randomization, but they do a lot more arithmetic (and may someday be of more practical importance, see "Minimizing Communication for Eigenproblems and the Singular Value Decomposition").

Here are some more details. *Hessenberg reduction* is analogous to QR decomposition: keep multiplying the matrix by Householder transformations P to create zeros. But now you have to multiply on the left and right: PAP (since $P = P^\top$) to maintain orthogonal similarity:

The code is analogous:

Algorithm 9.6.

```
for $i = 1 : n-2$ do                                        ▷ Zero out matrix entries $A(i+2:n, i)$
    $u = \text{House}(A(i+1:n, i))$
    $A(i+1:n, i:n) = A(i+1:n, i:n) - 2u(u^\top A(i+1:n, i:n))$      ▷ Multiply $A = PA$
    $A(1:n, i+1:n) = A(1:n, i+1:n) - 2(A(1:n, i+1:n)u)u^\top$      ▷ Multiply $A = AP$
end for
```

The cost is $\frac{10}{3}n^3 + O(n^2)$ just for A, or $\frac{14}{3}n^3 + O(n^2)$ if we multiply out the *Householder transformations* to get Q. This is a lot more than LU or QR, and is only the first, cheap phase of the algorithm.

When $A = A^\top$, then the resulting Hessenberg matrix $H = QAQ^\top$ is also symmetric, and so is in fact a tridiagonal T. This is called *tridiagonal reduction*, and is the starting point for solving the symmetric eigenvalue problems.

For the SVD, we do something similar, but with different orthogonal matrices on the left and right to make A bidiagonal:

$$QLAQR^\top = B,$$

i.e. nonzero only on the main diagonal of B and right above it. This will be the starting point for computing the SVD; once we have the SVD of $B = U\Sigma V^\top$ we get the SVD of A as

$$(QL^\top U)\Sigma(QR^\top V)^\top.$$

Lemma 9.6

Hessenberg form is maintained by QR iteration.

Proof. If A is upper Hessenberg, so is $A - \sigma I$, and if $A - \sigma I = QR$, it is easy to confirm that Q is also (column i of Q is just a linear combination of columns $1:i$ of $A - \sigma I$). Then it is also easy to see that RQ is also upper Hessenberg. □

Finally we explain how to do one step of QR iteration on an upper Hessenberg matrix $H = A - \sigma I$ in just $O(n^2)$ flops, not $O(n^3)$.

Definition 9.6. An upper Hessenberg matrix H is unreduced if all the subdiagonals $H(i+1,i)$ are nonzero. Note that if H has some $H(i+1,i)=0$, it is block triangular, and we can solve the eigenproblems for $H(1:i,1:i)$ and $H(i+1:n,i+1:n)$ independently.

Theorem 9.8 (Implicit Q Theorem)
Suppose that $Q^\top AQ$ is upper Hessenberg and unreduced. Then columns 2 through n of Q are determined uniquely (up to multiplying by ± 1) by column 1 of Q.

First we see how to use this to do one step of QR iteration in $O(n^2)$ flops, and then prove it. Recall that Q comes from doing $A-\sigma I = QR$, so the first column is simply proportional to the first column of $A-\sigma I$, namely

$$[A(1,1)-\sigma, A(2,1), 0, \cdots]^\top .$$

Let $Q_1^\top$ be a Givens rotation that acts on this column to zero out $A(2,1)$ and form $Q_1^\top AQ_1$; this fills in $A(3,1)$, a so-called "bulge", and makes the matrix no longer Hessenberg. Our algorithm will remove the bulge, multiplying by more Givens rotations Q_i, making

$$Q_n^\top Q_{n-1}^\top \cdots Q_1^\top * AQ_1 \cdots Q_{n-1}Q_n$$

upper Hessenberg again, at a cost of $O(n^2)$. Since Hessenberg form is uniquely determined by Q_1, this must be the answer. Since each Q_i moves the bulge down by one, until it "falls off", this algorithm is called "chasing the bulge".

Proof. Let q_i be column i of Q. Then $Q^\top AQ = H$ implies $AQ = QH$. Look at column 1 of both sides:

$$Aq_1 = H(1,1)q_1 + H(2,1)q_2.$$

This determines $H(1,1)$, $H(2,1)$ and q_2 uniquely, by doing QR on

$$[q_1, Aq_1] = [q_1, q_2]\begin{bmatrix} 1 & H(1,1) \\ 0 & H(2,1) \end{bmatrix}$$

More generally, suppose we have determined $q_2, \cdots, q_i$ and columns $1:i-1$ of H. We get the next column by equating the i-th columns of $AQ=QH$ to get

$$Aq_i = \sum_{j=1}^{i+1} q_j H(j,i).$$

So $q_j^\top Aq_i = H(j,i)$ for $j = 1:i$, and then

$$Aq_i - \sum_{j=1}^{i} q_j H(j,i) = q_{i+1}H(i+1,i)$$

gives us q_{i+1} and $H(i+1,i)$. This is the basis of the *LAPACK* code *xGEES* (for Schur form) or *xGEEV* (for eigenvectors), which is used by *eig()* in *MATLAB*. □

10 Lecture 10: Symmetric Eigenproblems and the SVD

10.1 Symmetric Eigenvalue and the SVD

Goals: Perturbation theory and algorithms for the symmetric eigenproblem and SVD

The theorems we present are useful not just for perturbation theory, but understanding why algorithms work. Everything from chapter 4 applies to symmetric matrices, but much more can be shown. We consider only real, symmetric matrix (the case of complex Hermitian is similar). Then we know that we can write $A = Q\Lambda Q^\top$ where Λ is the diagonal matrix with real eigenvalues, and $Q = [q_1, \cdots, q_n]$ consists of real orthonormal eigenvectors. We will assume $\lambda_1 \geq \cdots \geq \lambda_n$.

Most results hold for SVD of general matrix. Recall that the SVD of A is simply related to the eigendecomposition of the symmetric matrix:

$$B = \begin{bmatrix} 0 & A \\ A^\top & 0 \end{bmatrix} = B^\top$$

with the eigenvalues of B equal to the positive/negative singular values of A (plus some zeros if A is rectangular). So a small change from A to $A + E$ is also a small symmetric change to B. So if we show this doesn't change the eigenvalues of B very much (in fact by no more than norm(E)), then this means the singular values of A can change just as little.

Definition 10.1 (Rayleigh Quotient)**.**

$$\rho(v, A) = \frac{v^\top A v}{v^\top v} \qquad \forall v \neq 0$$

Properties: If $Av = \lambda v$, then $\rho(v, A) = \lambda$. If $v = \sum_{i=1}^n b_i q_i = Qb$, $b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$, then

$$\begin{aligned} \rho(v, A) &= \frac{(Qb)^\top A (Qb)}{(Qb)^\top (Qb)} \\ &= \frac{b^\top \Lambda b}{b^\top b} \\ &= \frac{\sum_{i=1}^n \lambda_i b_i^2}{\sum_{i=1}^n b_i^2} \\ &= \sum_{i=1}^n w_i \lambda_i \end{aligned}$$

where $\sum_{i=1}^n w_i = 1$, which is a convex combination of $\{\lambda_1, \cdots, \lambda_n\}$. Hence,

$$\begin{aligned} &\lambda_1 \geq \rho(v, A) \geq \lambda_n \\ &\lambda_1 \max_{v \neq 0} \rho(v, A) \\ &\lambda_n \min_{v \neq 0} \rho(v, A) \end{aligned}$$

In fact all the eigenvalues can be written using the Rayleigh quotient.

Theorem 10.1 (Courant-Fischer Minimax Theorem)

Let R^j denote a j-dimensional subspace of n-dimensional space, and S^{n-j+1} denote an $n-j+1$ dimensional subspace. Then,

$$\max_{R^j} \min_{0\neq r\in R^j} \rho(r,A) = \lambda_j = \min_{S^{n-j+1}} \max_{0\neq s\in S^{n-j+1}} \rho(s,A)$$

The max over R^j is attained when $R^j = \operatorname{span}(q_1,\cdots,q_j)$, and the min over r is attained for $r = q_j$. The min over S^{n-j+1} is attained for $S^{n-j+1} = \operatorname{span}(q_j, q_j+1,\cdots,q_n)$ and the max over s is attained for $s = q_j$ too.

Proof. For any pair R^j and S^{n-j+1}, since their dimensions add up to $n+1$, they must intersect in some nonzero vector x_{RS}. Thus,

$$\min_{0\neq r\in R^j} \rho(r,A) \leq \rho(x_{RS},A) \leq \max_{0\neq s\in S^{n-j+1}} \rho(s,A)$$

Let R' maximize $\min_{0\neq r\in R^j} \rho(r,A)$ and S' minimize $\max_{0\neq s\in S^{n-j+1}}$. Then,

$$\begin{aligned}\max_{R^j} \min_{0\neq r\in R^j} \rho(r,A) &= \min_{0\neq r\in R'} \rho(r,A)\\ &\leq \rho(X_{R'S'})\\ &\leq \max_{0\neq s\in S'} \rho(s,A)\\ &= \min_{S^{n-j+1}} \max_{0\neq s\in S^{n-j+1}} \rho(s,A)\end{aligned}$$

If we choose $R^j = \operatorname{span}(q_1,\cdots,q_j)$ and $r = q_j$,

$$\min_{0\neq r\in R^j} \rho(r,A) = \rho(q_j,A) = \lambda_j$$

If we choose $S^{n-j+1} = \operatorname{span}(q_j, q_j+1,\cdots,q_n)$ and $s = q_j$,

$$\max_{0\neq s\in S^{n-j+1}} \rho(s,A) = \rho(q_j,A) = \lambda_j$$

So the above inequalities are bounded below and above by λ_j, and so must all equal λ_j. □

Theorem 10.2 (Weyl's Theorem)

If A is symmetric, with eigenvalues $\lambda_1 \geq \cdots \geq \lambda_n$ and $(A+E) = (A+E)^\top$ with eigenvalues $\mu_1 \geq \cdots \geq \mu_n$, then for all i,

$$|\lambda_i - \mu_i| \leq \|E\|_2$$

Proof.

$$\begin{aligned}\mu_j &= \min_{S^{n-j+1}} \max_{0\neq s\in S^{n-j+1}} \rho(s, A+E) \\ &= \min_{S^{n-j+1}} \max_{0\neq s\in S^{n-j+1}} \frac{s^\top (A+E)s}{s^\top s} \\ &= \min_{S^{n-j+1}} \max_{0\neq s\in S^{n-j+1}} \frac{s^\top As}{s^\top s} + \underbrace{\frac{s^\top Es}{s^\top s}}_{\leq \|E\|_2} \\ &\leq \lambda_j + \|E\|_2\end{aligned}$$

Swapping roles of μ_j and λ_j, we get $\lambda_j \leq \mu_j + \|E\|_2$. □

Corollary 10.1

If A general, with singular values $\sigma_1 \geq \cdots \geq \sigma_n$ and $A+E$ has singular values $\tau_1 \geq \cdots \geq \tau_n$, then for all j,

$$|\sigma_i - \tau_i| \leq \|E\|_2$$

Definition 10.2.

$$\text{Inertia}(A) = (\#\text{negative eigenvalues}(A), \#\text{zero eigenvalues}(A), \#\text{positive eigenvalues}(A))$$

Theorem 10.3 (Sylvester's Theorem)

If A is symmetric and X nonsingular, then

$$\text{Inertia}(A) = \text{Inertia}(X^\top AX)$$

Proof. Suppose #eigenvalues$(A<0) = m$, and #eigenvalues$(X^\top AX < 0) = m'$, but $m' < m$. Prove by contradiction. Let N be the m-dimensional subspace of eigenvectors for the m negative eigenvalues of A, so x in N means $x^\top Ax < 0$. And let P be the $n-m'$ dimensional subspace of nonnegative eigenvalues of $X^\top AX$, so $x \in P$ means $x^\top X^\top AXx = (Xx)^\top A(Xx) \geq 0$, or $y \in XP$ means $y^\top Ay \geq 0$.

But dimension(XP) = dimension$(P) = n - m'$, and

$$\text{dimension}(N) + \text{dimension}(XP) = n - m' + m > n$$

so they intersect, i.e. there is some nonzero x in both spaces N and XP, i.e. $x^\top Ax < 0$ and $x^\top Ax \geq 0$, a contradiction. □

Fact. Suppose we do $A = LDL^\top$ (Gaussian elimination with symmetric or no pivoting). Then

$$\begin{aligned}\text{Inertia}(A) &= \text{Inertia}(D) \\ &= (\#D_{ii} < 0, \#D_{ii} = 0, \#D_{ii} > 0)\end{aligned}$$

Factorize $A - xI = L'D'L'^{\top}$,

$$\begin{aligned}\#D_{ii} < 0 &= \#(\text{eigenvalues}(A - xI) < 0) \\ &= \#(\text{eigenvalues}(A) < x)\end{aligned}$$

Factorize $A - yI$ for some $y > x$, compute $\#(\text{eigenvalues}(A) < y)$. Then,

$$\#(\text{eigenvalues}(A) \in [x, y)) = \#\text{eigenvalues}(A) < y - \#\text{eigenvalues}(A) < x$$

Thus we can count the number of eigenvalues in any interval $[x, y)$.

Now suppose we count $\#$eigenvalues $< (x + y)/2$, we can get the number number of eigenvalues in each half of the interval. By repeatedly bisecting the interval, we can can compute any subset of the eigenvalues as accurately as we like. We say more on how to do this cheaply later.

10.2 Perturbation Theory for Eigenvectors

Theorem 10.4

Let $A = Q\Lambda Q^{\top}$ be the eigendecomposition of A with $\Lambda = \text{diag}\,[\lambda_i]$ and $Q = [q_1, \cdots, q_n]$, and $A + E = Q'\Lambda Q'^{\top}$ with $\Lambda' = \text{diag}\,[\lambda_i']$ and $Q = [q_1', \cdots, q_n']$. Let θ_i be the angle between q_i and q_i'. Let

$$\text{gap}(i, A) = \min_{j \neq i} \|\lambda_j - \lambda_i\|$$

Then,

$$\|\frac{1}{2}\sin(2\theta_i)\| \leq \frac{\|E\|_2}{\text{gap}(i, A)}$$

When $\theta_i \ll 1$, $\frac{1}{2}\sin(2\theta_i) \approx \sin\theta_i \approx \theta_i$. Then, if $\|E\|_2 \ll \text{gap}(i, A)$, the change in direction of the eigenvector (i.e. θ_i) is small.

Proof of a weaker result: Let $q_i + d$ be an eigenvector of $A + E$ where $d^{\top}q_i = 0$, and $q_i' = (q_i + d) / \|q_i + d\|_2$. The goal is to bound $d = \tan\theta_i$.

$$(A + E)(q_i + d) = \lambda_i'(q_i + d)$$

Move q_i to one side and ignore Ed (since it is small) as

$$\left(A + E - \lambda_i' I\right) q_i = \left(\lambda_i' - A\right) d$$

Since $q_i^{\top}(A - \lambda_i I) = 0$,

$$\left(\lambda_i I + E - \lambda_i' I\right) q_i = \left(\lambda_i' - A\right) d$$

As $d = \sum_{j \neq i} d_j q_j$,

$$\left(\lambda_i I + E - \lambda_i' I\right) q_i = \sum_{j \neq i} \left(\lambda_i' - \lambda_j\right) d_j q_j$$

Then $\|(\lambda_i I + E - \lambda_i' I)\, q_i\| \leq 2\,\|E\|_2$ because $|\lambda_i - \lambda_i'| \leq \|E\|_2$ by theorem 10.2 (Weyl's Theorem). And $\left\|\sum_{j\neq i} (\lambda_i' - \lambda_j)\, d_j q_j\right\| \geq (\text{gap}(i, A) - \|E\|_2)\, \|d\|_2$. As

$$\lambda_i' - \lambda_j = \underbrace{\lambda_i' - \lambda_i}_{\leq \|E\|_2} + \underbrace{\lambda_i - \lambda_j}_{\geq \text{gap}}$$

we have the follows,

$$\frac{2\,\|E\|}{\text{gap}(i, A)} \sim \frac{2\,\|E\|_2}{\text{gap}(i, A) - \|E\|_2} \geq \|d\|_2 = |\tan\theta_i| \underbrace{\sim |\theta_i|}_{\text{if } |\theta_i| \ll 1}$$

if $\|E\|_2 \ll \text{gap}(i, A)$.

10.3 More Results on Rayleigh Quotients

The Rayleigh Quotient is a "best" approximation to an eigenvalue in a certain sense.

Theorem 10.5

$\|x\|_2 = 1$ and β given. Then A has an eigenvalue α

$$|\alpha - \beta| \leq \|Ax - \beta x\|_2$$

Given only x, the choice $\beta = \rho(x, A)$ minimizes $\|Ax - \beta x\|_2$. Given any unit vector x, there is always an eigenvalue λ such that $\|\lambda - \rho(x, A)\|_2 \leq \|Ax - \rho(x, A)x\|_2$ of $\rho(x, A)$, and $\rho(x, A)$ minimizes this bound. Now let λ_i be the eigenvalue of A closest to $\rho(x, A)$, and

$$\text{gap} = \min_{j\neq i} |\lambda_j - \rho(x, A)|$$

Then,

$$|\lambda_j - \rho(x, A)| \leq \frac{\|Ax - \rho(x, A)x\|_2^2}{\text{gap}}$$

i.e. error in $\rho(x, A)$ as an approximated eigenvalue is proportional to the square of the norm of the residual $Ax - \rho(x, A)x$.

Proof.

- Part 1: If x is a unit vector, assume $A = Q\Lambda Q^\top$,

$$\begin{aligned} 1 = \|x\|_2 &= \left\|(A - \beta I)^{-1}(A - \beta I)x\right\|_2 \\ &\leq \left\|(A - \beta I)^{-1}\right\|_2 \|Ax - \beta x\|_2 \\ &= \left\|(\Lambda - \beta I)^{-1}\right\|_2 \|Ax - \beta x\|_2 \\ &= \frac{1}{\min_i |\lambda_i - \beta|} \|Ax - \beta x\|_2 \end{aligned}$$

- Part 2: To show that $\rho(x,A)$ minimizes $\|Ax-\beta x\|_2$:

$$Ax-\beta x = \underbrace{Ax-\rho(x,A)x}_{y} + \underbrace{\rho(x,A)x-\beta x}_{z}$$

If $y^\top z = 0$, then (Pythagorean Theorem),

$$\|Ax-\beta x\|_2^2 = \|Ax-\rho(x,A)x\|_2^2 + \|\rho(x,A)x-\beta x\|_2^2$$

By theorem 3.1, for any β:

$$\|Ax-\rho(x,A)x\|_2 \leq \|Ax-\beta x\|_2$$

$$\begin{aligned} z^\top y &= (\rho(x,A)-\beta)\, x^\top (Ax-\rho(x,A)x) \\ &= (\rho(x,A)-\beta) \quad \underbrace{\left(x^\top Ax - \rho(x,A)x^\top x\right)}_{\text{equals 0 by the definition of } \rho(x,A)} \end{aligned}$$

- Special case of a 2×2 diagonal matrix A, which seems very special, but has all the ingredients of the general case. Assume $A = \text{diag}[\lambda_1,\lambda_2]$ and $x = [c,s]^\top$ where $c^2+s^2=1$. Hence,

$$\rho(x,A) = c^2\lambda_1 + s^2\lambda_2$$

Assume $c > s$, so $\rho(x,A)$ is closer to λ_1 than λ_2.

$$\begin{aligned} |\lambda_1-\rho(x,A)| &= \left|\lambda_1 - c^2\lambda_1 - s^2\lambda_2\right| \\ &= \left|s^2\lambda_1 - s^2\lambda_2\right| \\ &= s^2\,|\lambda_1-\lambda_2| \end{aligned}$$

$$\begin{aligned} \text{gap} &= |\lambda_2-\rho(x,A)| \\ &= \left|\lambda_2 - c^2\lambda_1 - s^2\lambda_2\right| \\ &= c^2\,|\lambda_1-\lambda_2| \end{aligned}$$

$$\begin{aligned} r &= Ax-\rho(x,A)x \\ &= \begin{bmatrix} \lambda_1 c \\ \lambda_2 s \end{bmatrix} - \left(c^2\lambda_1+s^2\lambda_2\right)\begin{bmatrix} c \\ s \end{bmatrix} \\ &= \begin{bmatrix} c^2(\lambda_1-\lambda_2) \\ s^2(\lambda_1-\lambda_2) \end{bmatrix} \end{aligned}$$

$$\|r\|_2^2 = s^2c^2(\lambda_1-\lambda_2)^2(s^2+c^2)$$

Thus,

$$\frac{\|r\|_2^2}{\text{gap}} = s^2 \left|\lambda_1 - \lambda_2\right| \qquad\qquad = \left|\lambda_1 - \lambda_2 \rho(x, A)\right|$$

i.e. the bound in the theorem is exact.

□

This result will be important later to help show that the QR algorithm from Chapter 4 converges cubically when applied to symmetric matrices (instead of "just" quadratically).

10.4 Overview of Algorithms

There are several approaches, depending on what one wants to compute:

1. Usual Accuracy: Backward stable in the sense that you get the exact eigenvalues and eigenvectors for $A + E$ where $\|E\| = O(\varepsilon)\|A\|$

   (a) Get all the eigenvalues (with or without the corresponding eigenvectors)

   (b) Just get all the eigenvalues in some interval $[x, y]$ (with or without the corresponding eigenvectors)

   (c) Just get λ_i through λ_j (with or without the corresponding vectors)

   (b) and (c) can be rather cheaper than (1.1) if only a few values wanted.

2. High Accuracy: Compute tiny eigenvalues and their eigenvectors more accurately than the "usual" accuracy guarantees.

   Example 10.1

   If A well-conditioned, so all singular values are large, then error bound $O(\varepsilon)\|A\|$ implies we can compute them with small relative errors. Now consider $B = DA$ where D is diagonal with some large and tiny entries. Then B will be ill-conditioned, and so have some tiny, some large singular values, and the usual error bound $O(\varepsilon)\|B\|$ implies the tiny singular values do not have many correct leading digits. But in this case, there is a "tighter" perturbation theory, and an algorithm that will still compute the tiny singular values of B with as accurately as large ones, as many correct leading digits as for A. For a survey of cases like this when high relative accuracy in tiny eigenvalues and singular values is possible. See the papers "Accurate and efficient expression evaluation and linear algebra", "New fast and accurate Jacobi SVD algorithm", "Computing the singular value decomposition with high relative accuracy" and "Jacobi's method is more accurate than QR".

3. Updating: Given the eigenvalues and eigenvectors of A, find them for A with a "small change", small in rank, e.g. $A \pm xx^\top$. It is more cheaply than starting from scratch.

All the above options also apply to computing the SVD, with the additional possibility of computing the left and/or the right singular vectors.

10.5 Algorithms and Their Costs

1. We begin by reducing A to $Q^\top AQ = T$ where Q is orthogonal and $T = T^\top$ is tridiagonal, costing $O(n^3)$ flops. This uses the same approach as in chapter 4, where $Q^\top AQ$ was upper Hessenberg, since a matrix that is both symmetric and upper Hessenberg must be tridiagonal. This is currently done by *LAPACK* routine *ssytrd*. All subsequent algorithms operate on T, which is much cheaper.

   There is an algorithm for tridiagonal reduction that does $O(n^3/\sqrt{\text{fast\_memory\_size}})$ moving words. See "Avoiding Communication in Successive Band Reduction".

   When A is banded, a different algorithm (in *ssbtrd*) takes advantage of the band structure to reduce to tridiagonal form in just $O(n^2\text{bandwidth})$ operations. This can also be done while doing much less communication (the lower bound itself is an open question).

   In the case of the SVD, we begin by reducing the general nonsymmetric (even rectangular) A to $U^\top AV = B$ where U and V are orthogonal and B is bidiagonal, i.e. nonzero on the main diagonal and right above it. All subsequent SVD algorithms operate on B. This is currently done by *LAPACK* routine *sgebrd*. All the ideas for minimizing communication mentioned above generalize to the SVD.

   (a) Given T we need to find all its eigenvalues (and possibly vectors) There are a variety of algorithms; all cost $O(n^2)$ just for eigenvalues, but anywhere from $O(n^2)$ to $O(n^3)$ for eigenvectors. Also some have better numerical stability properties than others.

      i. Oldest is QR iteration, as in chapter 4, but for tridiagonal T.

      Theorem 10.6 (Wilkinson Theorem)
      With the right shift, tridiagonal QR is globally convergent, and usually cubically convergent (the number of correct digits triples at each step!)

      It costs $O(n^2)$ to get all the eigenvalues, but costs $O(n^3)$ to get the vectors, unlike later routines.
      LAPACK routine *sgesvd* uses a variant of QR iteration for the SVD, which has the additional property of guaranteed high relative accuracy for all singular values, no matter how small, as long as the input is bidiagonal. See "Accurate singular values of bidiagonal matrices".

      ii. Another approach, which is much faster for computing eigenvectors ($O(n^2)$ instead of $O(n^3)$) but does not guarantee that they are orthogonal, works as follows:

      Algorithm 10.1.
      A. Compute the eigenvalues alone (*sstebz* in *LAPACK*)
      B. Compute the eigenvectors using inverse iteration (*sstein* in *LAPACK*)

$$x_{i+1} = (T - \lambda_i I)^{-1} x_i$$

      Since T is tridiagonal, one steps of inverse iteration costs $O(n)$, and since λ_j is an accurate eigenvalue, it should converge in very few steps. The trouble is that when

λ_j λ_{j+1} are very close, and so the eigenvectors are very sensitive, they may not be computed to be orthogonal, since there is nothing in the algorithm that explicitly enforces this (imagine what would happen if λ_j and λ_{j+1} were so close that they rounded to the same floating point number). Still, the possibility of having an algorithm that ran in $O(n^2)$ time but guaranteed orthogonality was a goal for many years, with the eventual algorithm discussed below (*MRRR*).

iii. Next is "divide-and-conquer", which is faster than QR, but not as fast as inverse iteration. It is used in LAPACK routine *ssyevd* (*sgesdd* for SVD). The speed is harder to analyze, but empirically it is like $O(n^g)$ for some $2 < g < 3$. The idea behind it is used for the updating problem, i.e. getting the eigenvalues and eigenvectors of $A + xx^\top$ given those of A.

iv. The most recent is Multiple Relatively Robust Representations, which can be thought as a version of inverse iteration that does guarantee orthogonal eigenvectors, and still cost just $O(n^2)$ ("Orthogonal Eigenvectors and Relative Gaps"). It is implemented in LAPACK routine *ssyevr*. The adaptation of this algorithm for the SVD appeared in the prize-winning thesis of Paul Willems, but some examples of matrices remain where it does not achieve the desired accuracy, so it is still an open problem to make this routine reliable enough for general use.

In theory, there is an even faster algorithm than $O(n^2)$, based on divide-and-conquer, by representing the eigenvector matrix Z of T implicitly rather than as n^2 explicit matrix entries, but since most users want explicit eigenvectors, and the cost of reduction to tridiagonal form T is already much larger, we do not usually use it:

Theorem 10.7

One can compute Z in $O(n \log^p n)$ time, provided we represent it implicitly (so that we can multiply any vector by it cheaply). Here p is a small integer.

Thus $A = QTQ^\top = QZ\Lambda Z^\top Q^\top = (QZ)\Lambda(QZ)^\top$, so we need to multiply QZ to get final eigenvectors (costs $O(n^3)$).

(b) Reduce A to $Q^\top AQ = T$ as above. Based on theorem 10.3 (Sylvester's Theorem), use bisection on T to get a few eigenvalues, and then inverse iteration to get their eigenvectors if desired. The cost is $O(n)$ per eigenvalue/vector, but does not guarantee orthogonality of eigenvectors of nearby eigenvalues *ssyevx* in *LAPACK*. MRRR (in *ssyevr*) could be used for this too, and guarantee orthogonality.

2. High Accuracy: Based on Jacobi's Method, the historically oldest algorithm. The modern version is by Drmac and Veselic, called *sgesvj* for the SVD.

3. Use same idea as divide-and-conquer: Assuming we have the eigenvalues and eigenvectors for $A = Q\Lambda Q^\top$, it is possible to compute the eigenvalues of $A \pm uu^\top$ in $O(n^2)$, much faster than starting from scratch.

10.6 QR Iteration

Cubic convergence follows from analysis of a simpler algorithm as follows.

Algorithm 10.2 (Rayleigh Quotient Iteration).

```
1: $i = 0$
2: choose unit vector $x_0$
3: while not converged do                          ▷ $s_i$ and/or $x_i$ stop changing
4:     $s_i = \rho(x_i, A) = x_i^\top A x_i$
5:     $y = (A - s_i I)^{-1} x_i$
6:     $x_{i+1} = y / \|y\|_2$
7:     $i = i + 1$
8: end while
```

This is what we called inverse iteration before, using the Rayleigh Quotient s_i as a shift, which we showed was the best possible eigenvalue approximation for the approximate eigenvector x_i. Suppose $Aq = \lambda q$, $\|q\|_2 = 1$, $\|x_i - q\| = e \ll 1$. We want to show that $\|x_{i+1} - q\|_2 = O(e^3)$. To bound $|s_i - \lambda|$:

$$\begin{aligned}
s_i &= x_i^\top A x_i \\
&= (x_i - q + q)^\top A\,(x_i - q + q) \\
&= (x_i - q)^\top A\,(x_i - q) + q^\top A(x_i - q) + (x_i - q)^\top Aq + q^\top Aq \\
&= (x_i - q)^\top A\,(x_i - q) + \lambda q^\top (x_i - q) + (x_i - q)^\top q\lambda + \lambda
\end{aligned}$$

So,

$$\begin{aligned}
s_i - \lambda &= (x_i - q)^\top A\,(x_i - q) + 2\lambda\,(x_i - q)^\top q \\
|s_i - \lambda| &\le O(\|x_i - q\|_2^2) + O(\|x_i - q\|) \\
&= O(\|x_i - q\|) \\
&= O(e) \\
|s_i - \lambda| &\le \|Ax_i - s_i x_i\|_2^2 \,/\, \text{gap}
\end{aligned}$$

where gap is the distance from s_i to next closest eigenvalue. We assume the gap is not too small. Then,

$$\begin{aligned}
|s_i - \lambda| &= \|A(x_i - q + q) - s_i(x_i - q + q)\|_2^2 \,/\, \text{gap} \\
&= \|(A - s_i I)(x_i - q) + (\lambda - s_i)q\|_2^2 \,/\, \text{gap} \\
&\le (\underbrace{\|(A - s_i I)(x_i - q)\|}_{o(e)} + \underbrace{\|(\lambda - s_i)q\|}_{o(e)})^2 / \,\text{gap} \\
&= O(e^2) / \,\text{gap}
\end{aligned}$$

Now we take one step of inverse iteration to get x_{i+1}; from earlier analysis we know

$$\begin{aligned}
\|x_{i+1} - q\| &\le \underbrace{\|x_i - q\|}_{e}\,\underbrace{\|s_i - \lambda\|}_{O(e^2)} \,/\, \text{gap} \\
&= O(e^3)
\end{aligned}$$

if gap is not too small.

To see that QR iteration is doing Rayleigh Quotient iteration in disguise, look at one step:

$$\begin{aligned} T - s_i I &= QR \\ (T - s_i I)^{-1} &= R^{-1}Q^{-1} \\ &= R^{-1}Q^\top \\ &= \left(R^{-1}Q^\top\right)^\top \\ &= QR^{-\top} \end{aligned}$$

So $(T - s_i I)^{-1}R^\top = Q$. Then the last column of Q is $(T - s_i I)^{-1}e_n R_{nn}$. Since $s_i = T_{nn} = e_n^\top T e_n = \rho(e_n, T)$, s_i is just the Rayleigh Quotient for e_n, and q_n is the result of one step of Rayleigh quotient iteration. Then the updated T is $RQ + s_i I = Q^\top T Q$, so the updated $T_{nn} = q_n^\top T q_n$ is the next Rayleigh quotient as desired.

In practice QR iteration is implemented analogously to Chap 4, by "bulge chasing", at a cost of $O(n)$ per iteration, or $O(n^2)$ to find all the eigenvalues. But multiplying together all the *Givens rotations* still costs $O(n^3)$, so something faster is still desirable.

10.7 Divide and Conquer for Tridiagonal eigenproblems

The important part is how to cheaply compute the eigendecomposition of a rank-one update to a symmetric matrix $A + \alpha uu^\top$, where we already have the eigendecomposition $A = Q\Lambda Q^\top$. Then

$$\begin{aligned} A + \alpha uu^\top &= Q\Lambda Q^\top + \alpha uu^\top \\ &= Q\left(\Lambda + \alpha(Q^\top u)(u^\top Q)\right)Q^\top \\ &= Q\left(\Lambda + \alpha vv^\top\right)Q^\top \end{aligned}$$

So we only need to think about computing the eigenvalues and eigenvectors of $\Lambda + \alpha vv^\top$. Let's compute its characteristic polynomial.

Lemma 10.1

$$\det\left(I + xy^\top\right) = 1 + y^\top x$$

Then the characteristic polynomial is,

$$\begin{aligned} \det\left(\Lambda + \alpha vv^\top - \lambda I\right) &= \det\left((\Lambda - \lambda I)(I + \alpha(\Lambda - \lambda I)^{-1}vv^\top)\right) \\ &= \det(\Lambda - \lambda I)\cdot\det(I + \alpha(\Lambda - \lambda I)^{-1}vv^\top) \\ &= \prod_i (\lambda_i - \lambda)\left(1 + \alpha\sum_j v_j^2/(\lambda_j - \lambda)\right) \\ &= \prod_i (\lambda_i - \lambda)\, f(\lambda) \end{aligned}$$

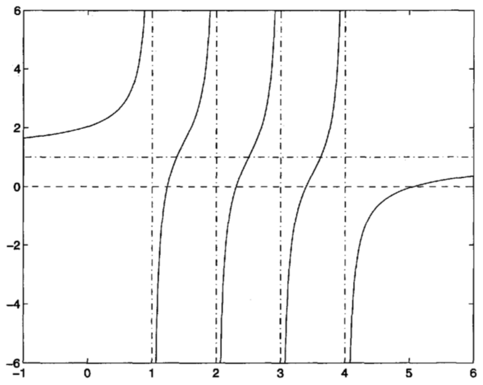


Figure 10.1: Graph of $f(\lambda) = 1 + \frac{.5}{1-\lambda} + \frac{.5}{2-\lambda} + \frac{.5}{3-\lambda} + \frac{.5}{4-\lambda}$

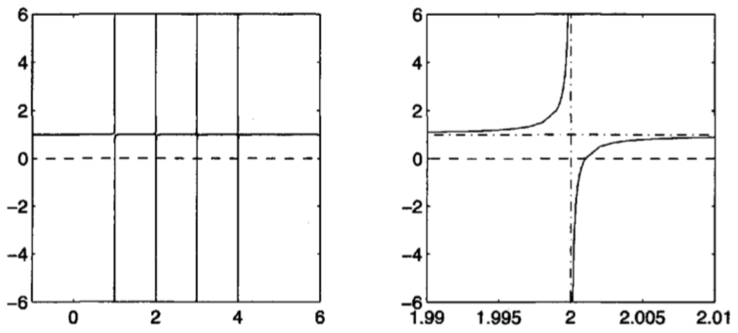


Figure 10.2: Graph of $f(\lambda) = 1 + \frac{10^{-3}}{1-\lambda} + \frac{10^{-3}}{2-\lambda} + \frac{10^{-3}}{3-\lambda} + \frac{10^{-3}}{4-\lambda}$

So our goal is to solve the so-called secular equation $f(\lambda) = 0$.

Consider figure 10.1, we see there is one root of $f(\lambda)$ between each pair $[\lambda_i, \lambda_{i+1}]$ of adjacent eigenvalues, and in each such interval $f(\lambda)$ is monotonic, so some *Newton*-like method should work well.

But now consider figure 10.2. Here $f(\lambda)$ no longer looks so benign, and simple *Newton* would not work, taking a big step out of the bounding interval. But we can still use *Newton*, but instead of approximating the $f(\lambda)$ by a straight line at each step, and getting the next guess by finding a zero of that straight line, we approximate $f(\lambda)$ by an simple but non-linear function that better matches the graph, with poles at λ_i and λ_{i+1}:

$$g(\lambda) = c_1 + \frac{c_2}{\lambda_i - \lambda} + \frac{c_3}{\lambda_{i+1} - \lambda}$$

where c_1, c_2 and c_3 are chosen as in *Newton* to match $g(x_i) = f(x_i)$ and $g'(x_i) = f'(x_i)$ at the last approximate 0. Solving $g(\lambda) = 0$ is then solving a quadratic for λ.

Lemma 10.2

If λ is an eigenvector of $\Lambda + \alpha vv^\top$, then $(\Lambda - \lambda I)^{-1}v$ is its eigenvector. Since Λ is diagonal, this costs $O(n)$ to compute.

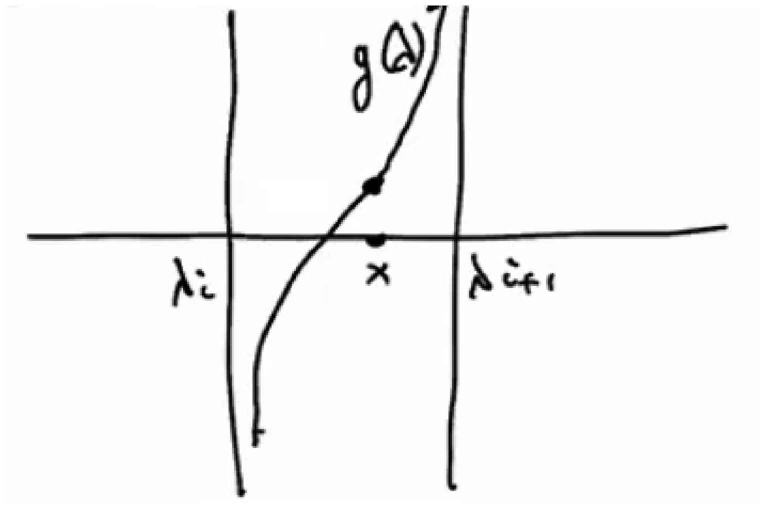


Figure 10.3: Graph of $g(\lambda)$

Proof.

$$\begin{aligned}(\Lambda+\alpha vv^\top)(\Lambda-\lambda I)^{-1}v &= (\Lambda-\lambda I+\lambda I+\alpha vv^\top)(\Lambda-\lambda I)^{-1}v\\ &= v+\lambda(\Lambda-\lambda I)^{-1}v+\underbrace{v\left(\alpha v^\top(\Lambda-\lambda I)^{-1}v\right)}_{-v \text{ as } f(\lambda)=0}\\ &= \lambda(\Lambda-\lambda I)^{-1}v\end{aligned}$$

□

Unfortunately this simple formula is not numerically stable; the text describes the clever fix (due to Ming Gu and Stan Eisenstat, for which Prof. Gu won the Householder Prize for best thesis in Linear Algebra in 1996).

When two eigenvalues λ_i and λ_{i+1} are (nearly) identical, then we know there is a root between them without doing any work. Similarly when v_i is (nearly) zero, we know that λ_i is (nearly) an eigenvalue without doing any work. In practice a surprising number of eigenvalues can be found quickly ("deflated") this way, which speeds up the algorithm.

Write $[Q',\Lambda'] = \text{Eig\_update}[Q,\Lambda,\alpha,u]$ as the function we just described that takes the eigendecomposition of $A = Q\Lambda Q^\top$ and updates it, returning the eigendecomposition of

$$A+\alpha uu^\top = Q'\Lambda'Q'^\top$$

We can also use *Eig_update* as a building block for an entire eigen solver, using Divide-and-Conquer. To do so we need to see how to divide a tridiagonal matrix into two smaller ones of half the size, just using a rank one change $\alpha vv^\top$. We just divide in the middle: If $T = \text{tridiag}\,(a_1,\cdots,a_n,b_1,\cdots,b_{n-1},)$ is a symmetric tridiagonal matrix with diagonals a_i and off-diagonals b_i, then we can write

$$T = \text{diag}(T_1,T_2)+b_i uu^\top$$

where T_1 and T_2 are two half-size tridiagonal matrices (if i is $n/2$) and u is a vector with $u_i = u_{i+1} = 1$ and the other $u_j = 0$. Using this notation, we can write our overall algorithm as follows:

Algorithm 10.3.

```
1: func[$Q, \Lambda$] = DC_eig($T$)
2: $n$ = dimension($T$)
3: if $n$ small enough then
4:     use QR iteration
5: else
6:     $i = \text{floor}(\frac{n}{2})$
7:     write $T = \text{diag}(T_1, T_2) + b_i uu^\top$ (just notation, no work)
8:     $[Q_1, \Lambda_1]$ = DC_eig($T_1$)
9:     $[Q_2, \Lambda_2]$ = DC_eig($T_2$) (where diag($Q_1, Q_2$) and diag($\Lambda_1, \Lambda_2$) are the eigendecomposition
   of diag($T_1, T_2$))
10:    $[Q, \Lambda]$ = Eig_update (diag($Q_1, Q_2$), diag($\Lambda_1, \Lambda_2$), $b_i$, $u$)
11: end if
12: return
```

10.8 Algorithms for Few Eigenvalues and Eigenvectors Desired

Now we turn to algorithms that are fastest when only a few eigenvalues and eigenvectors are desired. Afterward we return to briefly describe MRRR, the fastest algorithm when all eigenvalues and eigenvectors are desired.

Recall Sylvester's Theorem: Suppose A is symmetric and X nonsingular. Then A and $XAX^\top$ have the same Inertia:

$$(\#\text{evals} < 0,\ \#\text{evals} = 0,\ \#\text{evals} > 0).$$

So $A - \sigma I$ and $X(A - \sigma I)X^\top$ have the same Inertia, namely

$$(\#\text{evals of } A < \sigma,\ \#\text{evals of } A = \sigma,\ \#\text{evals of } A > \sigma)$$

So if $X(A-\sigma I)X^\top$ were diagonal, it would be easy to compute Inertia. By doing this for σ_1 and σ_2, we can count the number of eigenvalues in any interval $[\sigma_1, \sigma_2]$, and by repeatedly cutting intervals in half, we can compute intervals containing the eigenvalues that are as narrow as we like, or that only contain eigenvalues in regions of interest (e.g. the smallest).

But how do we cheaply choose X to make $X(A - \sigma I)X^\top$ diagonal? By starting with $A = T$ tridiagonal, and doing Gaussian elimination without pivoting:

$$T - \sigma I = LDL^\top,$$

so $L^{-1}(T - \sigma I)L^{-\top}$ and D have the same inertia.

However, LU without pivoting seems numerically dangerous. Fortunately, because of the tridiagonal structure, it is not, if done correctly.

Algorithm 10.4 (Negcount: count the number c of eigenvalues of $T < s$).

```
▷ Only compute diagonal entries $d_i$ of $D$ in $T - sI = LDL^\top$
Assume diag($T$) = $(a_1, a_2, \cdots, a_n)$
Assume offdiag($T$) = $(b_1, b_2, \cdots, b_{n-1})$
$c = 0$
```

```
$b_0 = 0$
$d_0 = 1$
for $i = 1 : n$ do
    $d_i = (a_i - s) - b_{i-1}^2/d_{i-1}$                ▷ Obey parentheses!
    if $d_i < 0$ then
        $c = c + 1$
    end if
end for
```

We don't need l_i, the i-th off diagonal of L, because

$$(T - sI)_{i,i+1} = b_i = (LDL^\top)_{i,i+1} = d_i l_i$$

so we can replace the usual inner loop of $LDL^\top$ (analogous to LU)

$$d_i = a_i - s - l_{i-1}^2 d_{i-1}$$

by

$$d_i = a_i - s - \frac{b_{i-1}^2}{d_{i-1}}.$$

Theorem 10.8

Assuming we don't divide by zero, overflow or underflow, the computed c from Negcount(T, s) is exactly correct for a slightly perturbed $T + E$, where E is symmetric and tridiagonal, $E_{ii} = 0$ (the diagonal is not perturbed at all) and

$$|E_{i,i+1}| \leq 2.5\varepsilon |T_{i,i+1}|$$

Proof. As before, we replace each operation like $a + b$ in the algorithm with $(a + b)(1 + \delta)$ where $|\delta| < \varepsilon$. If we obey the parentheses in the algorithm, so $a_i - s$ is subtracted first, then we can divide out by all the $(1 + \delta)$ factors multiplying $a_i - s$ to get

$$d_i F_i = a_i - s - \frac{b_{i-1}^2 G_i}{d_{i-1}}$$

where F_i and G_i are the product of 2 or 3 (1+delta) factors. Now write this as

$$d_i F_i = a_i - s - \frac{b_{i-1}^2 G_i F_{i-1}}{d_{i-1} F_{i-1}}$$

or

$$d_i' = a_i - s - (b_{i-1}')^2 / d_{i-1}',$$

which is the exact recurrence for $T + E$. Since d_i and $d_i' = d_i F_i$ have the same sign, we get the same exact Negcount for either. □

In fact more is true: This works even if some pivot d_{i-1} is exactly zero, and we divide by zero, so the next d_i is infinite, because we end up dividing by d_i in the next step, and the "infinity" disappears. But it does mean that to correctly compute Negcount, we need to count -0 as negative, and +0 as positive (-0 is a feature of IEEE floating point arithmetic). Furthermore, the function $g(\sigma) = \#\text{evals} < \sigma$ is computed monotonically increasing, as long as the arithmetic is correctly rounded (otherwise the computed number of eigenvalues in a narrow enough interval might be negative!).

The cost of Negcount is clearly $O(n)$. To find one eigenvalue with Negcount via bisection, Negcount needs to be called at most 64 (in double) or 32 (in single) times, since each time the interval gets half as big, and essentially determines one more bit in the floating point format. So the cost is still $O(n)$. So to find k eigenvalues using bisection, the cost is $O(kn)$.

Given an accurate eigenvalue λ_j from bisection, we find its eigenvector by inverse iteration:

Algorithm 10.5.

```
Choose $x(0)$, $i = 0$
while $x_i$ not converged do
    $i = i + 1$
    Solve $(T - \lambda_j I)y = x(i-1)$ for $y$
    $x_i = y/||y||_2$
end while
```

which should converge in a few steps, and so at cost $O(n)$ per eigenvector. This seems like an optimal algorithm for all n eigenvalues and eigenvectors: at a cost of $O(n)$ per eigenpair, the total cost should be $O(n^2)$. But it has a problem: the computed eigenvectors, while they individually very nearly satisfy $Ax = \lambda_j x$ as desired, may not be orthogonal when λ_j is very close to another λ_{j+1}; there is nothing in the algorithm to enforce this, and when λ_j is close to λ_{j+1}, solving with $T - \lambda_j I$ is nearly the same as solving with $T - \lambda_{j+1} I$. Imagine the extreme case where λ_j and λ_{j+1} are so close that they round to the same floating point number! One could start with a random starting vector $x(0)$ and hope for the best but there is no guarantee of orthogonality.

At first people tried to guarantee orthogonality by taking all the computed eigenvectors belonging to a cluster of nearby eigenvalues and running QR decomposition on them, replacing them by the columns of Q. This guarantees orthogonality, but has two problems:

1. If the computed vectors are not linearly independent, the columns of Q may not satisfy $Aq = \lambda q$ very well.

2. If the size of the cluster of close eigenvalues is s, the cost of QR decomposition is $O(s^2 n)$, so if s is large, the cost could go back up to $O(n^3)$.

The MRRR (Multiple Relatively Robust Representations) algorithm was motivated by this problem: computing all the eigenvectors in $O(n^2)$ time such that are also guaranteed orthogonal. It is the fastest algorithm available (for large enough problems; it defaults to other algorithms for small problems), but is rather complicated to explain. Two lectures explaining it in detail are available from the 2004 Ma221 web page.

The final algorithm of importance is Jacobi's method, whose classical (and so slow) implementation is described in Lecture 5. Jacobi can be shown to be potentially the most accurate algorithm, with an error determined by a version of Weyl's Theorem for relative error, and so getting the tiny eigenvalues (and their eigenvectors) much more accurately. It was much slower than the other methods discussed so far, until work by Drmac and Veselic showed how it could be made much faster.

All the algorithms discussed so far extend to the SVD, typically by using the relationship between the SVD of A and the eigenvalues and eigenvectors of the symmetric matrix $\begin{bmatrix} 0 & A \\ A^\top & O \end{bmatrix}$. The one exception is MRRR, where some open problems remain, addressed in the 2010 PhD dissertation of Paul Willems, although there are still examples where Willems' code loses significant accuracy.

11 Lecture 11: Introduction to Iterative Methods

11.1 Introduction to Iterative Methods

Model Problem: Poisson's Equation
Goals: Contrast direct and iterative methods

- For $Ax = b$ or least squares: Use iterative methods when direct methods are too slow, or use too much memory (from fill-in during factorization) or you don't need as much accuracy.
- $Ax = \lambda x$ or SVD: Use iterative methods for the above reasons, or when only a few eigenvalues/vectors are desired (as well as above reasons).

Choosing best iterative method depends on mathematical structure of A. Summary of iterative methods as applied to Model problem, from simple (slow) to sophisticated (fast):

1. Simple ones apply to more problems than Model problem
2. Simple ones are building blocks for more sophisticated ones

Methods covered:

1. *"Splitting Methods"*: Split $A = M - K$ with M nonsingular. $Ax = b$ becomes $Mx = Kx + b$. Iterate: Solve $Mx_{i+1} = Kx_i + b$ for x_{i+1}. Convergence! Subtract $Mx = Kx + b$, let $e_i = x_i - x$, then:

$$\begin{aligned} Me_{i+1} &= Ke_i \\ e(i+1) &= M^{-1}Ke_i \\ &= \left(M^{-1}K\right)^i e_0 \end{aligned}$$

   Convergence depends on how fast $\left(M^{-1}K\right)^i \to 0$. Consider 3 splitting methods: Jacobi, Gauss-Seidel, and Successive Overrelaxation (SOR).

2. *Krylov Subspace Methods (KSMs)*: In the case that you have access to is a subroutine that computes Ax for any x, or A is so big that Ax is all you can afford to do. With KSMs, given x_0:

   (a) Compute

$$x_1 = Ax_0,\ x_2 = Ax_1,\ \cdots,\ x_k = Ax_{k-1}$$

   or a similar set of vectors that span the same subspace

   (b) Choose the linear combination $\sum_{i=0}^{k} a_i x_i$ of these vectors that "best" approximates the answer (for some definition of "best").

   (c) If the approximation is not good enough, increase k and repeat.

   Depending on:

- How the set of vectors is computed
- Properties of A (e.g. symmetry, positive definiteness, ...)
- The definition of "best approximation"

One gets a large number of different algorithms, all used in practice:

- Conjugate Gradients (CG)
- Generalized minimum residuals (GMRES)

and many others. Instead of asking for the "best approximation" to $Ax = b$, one can instead ask for the best approximation to $Ax = \lambda x$.

3. *Preconditioning*: The speed of the above methods for $Ax = b$ converge is complicated, but in general faster if $\kappa(A)$ smaller. Suppose we can find a "preconditioner", i.e. a matrix M:

   (a) Multiplying by M is cheap

   (b) MA is better conditioned than A

   Then one can apply the KSM to solve $MAx = Mb$. Finding good preconditioners can depend in complicated ways on A. For example, CG depends on A being s.p.d., whereas MA will rarely be symmetric, so need to reorganize CG to use Preconditioning.

4. *Multigrid*: This is one of the most effective preconditioners for problems with a suitable mathematical structure, e.g. Poisson's Equation. If A arises from solving a physical problem, then straight forward to find an approximation with, say, half as many mesh points in each direction, and use the solution of that smaller, and so easier, problem (e.g. 1/4 the dimension in 2D), as a preconditioner. But if the dimension n is very large, so is $n/4$. So we just apply the same idea recursively, using a problem of size $n/16$ to get a starting guess for the problem of size n/4, etc. When multigrid works, it can be optimal, solving an $n \times n$ linear system in $O(n)$ flops. But not every problem has the structure that permits this.

5. *Domain decomposition*: This can be thought of as a way to hybridize the above methods, using different methods in different "domains" or submatrices, where A may be easier to work with than as a whole.

Most methods assume the only thing you can afford to do is $y = Ax$. Computing Ax is generally the bottleneck, and usually because of communication: For example, if A is too large to fit in cache, it needs to be read out of main (slow) memory every iteration, with only one multiplication and addition per entry. So recent work has addressed communication-avoiding KSMs, where methods like CG are redesigned to take $s > 1$ steps for each access of A, instead of 1 step.

11.2 Model Problem: Poisson's Equation

1D with Dirichlet boundary conditions:

$$-\frac{\partial^2}{\partial x^2} v(x) = f(x)$$

on $x \in [0,1]$, $v(0) = v(1) = 0$. Discretizing with $h = \frac{1}{N+1}$ to get:

$$
\begin{aligned}
T_N \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix} &= T_N v \\
&= h^2 \begin{bmatrix} f_1 \\ \vdots \\ f_N \end{bmatrix} \\
&= h^2 f \\
T_N &= \begin{bmatrix} 2 & -1 & & \mathbf{O} \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ \mathbf{O} & & -1 & 2 \end{bmatrix}
\end{aligned}
$$

Lemma 11.1

$T_N z_j = \lambda_j z_j$ where $\|z_j\|_2 = 1$ and

$$
\begin{aligned}
\lambda_j &= 2\left(1 - \cos\frac{\pi j}{N+1}\right) \\
z_j(k) &= \sqrt{\frac{2}{N+1}} \sin\left(\frac{jk\pi}{N+1}\right)
\end{aligned}
$$

Corollary 11.1

The matrix Z where

$$
Z_{jk} = z_j(k) = \sqrt{\frac{2}{N+1}} \sin\left(\frac{jk\pi}{N+1}\right)
$$

is orthogonal (related to Fast Fourier Transform).

Eigenvalues range from the smallest one $\lambda_j \sim \left(\frac{\pi j}{N+1}\right)^2$ up to $\lambda_N \sim 4$, which tells the condition number of T_N:

$$
\begin{aligned}
\frac{\lambda_N}{\lambda_1} &\sim \left(\frac{2\,(N+1)^2}{\pi}\right)^2 \\
&= \left(\frac{2}{\pi}\right) h^{-2}
\end{aligned}
$$

2D with Dirichlet boundary conditions:

$$
\frac{-\partial^2}{\partial x^2} v(x,y) - \frac{\partial^2}{\partial y^2} v(x,y) = f(x,y)
$$

on square $[0,1]^2$ with $v(x,y)=0$ on boundary. Discretize as before, write $v_{i,j}=v(i\cdot h, j\cdot h)$, then

$$4v_{ij}-v_{i-1,j}-v_{i+1,j}-v_{i,j-1}-v_{i,j+1}=h^2 f_{ij}$$

Letting V be the $N\times N$ matrix of v_{ij}, we can also write

$$\begin{aligned} 2v_{ij}-v_{i-1,j}-v_{i+1,j} &= (T_N V)_{ij} \\ 2v_{ij}-v_{i,j-1}-v_{i,j+1} &= (V T_N)_{ij} \end{aligned}$$

Then the above equation becomes $(T_N V + V T_N)_{ij}$ or

$$T_N V + V T_N = h^2 F$$

a system of N^2 linear equations for the N^2 entries of V, though not in the usual "$Ax=b$" form. We can ask about the corresponding eigenvalues and eigenvectors in the analogous form:

$$T_N V + V T_N = \lambda V$$

Suppose $T_N z_i = \lambda_i z_i$ and $T_N z_j = \lambda_j z_j$ are any two eigen pairs of T_N. Letting $V = z_i z_j^\top$, we get

$$\begin{aligned} T_N V + V T_N &= (T_N z_i) z_j^\top + z_i (z_j^\top T_N) \\ &= (T_N z_i) z_j^\top + z_i (T_N^\top z_j)^\top \\ &= (T_N z_i) z_j^\top + z_i (T_N z_j)^\top \\ &= (\lambda_i z_i) z_j^\top + z_i (\lambda_j z_j^\top) \\ &= (\lambda_i + \lambda_j)\, z_i z_j^\top \\ &= (\lambda_i + \lambda_j)\, V \end{aligned}$$

Hence, $V = z_i z_j^\top$ is an eigenvector with eigenvalue $\lambda_i + \lambda_j$. Since there are N^2 pairs of (i,j), we are expecting N^2 pairs of eigenvalues and eigenvectors. We want to write V as a single vector in a way that easily extends to higher dimensional Poisson equations. In the 3×3 case, write V columnwise from left to right as:

$$v = [v_{11}; v_{21}; v_{31}; v_{12}; v_{22}; v_{32}; v_{13}; v_{23}; v_{33}]^\top$$

and $T_{N\times N}$ has the following patterns rowwise:

- 4 on the diagonal multiplies v_{ij}
- -1s next to the diagonal multiply $v_{i-1,j}$ and $v_{i+1,j}$
- -1s N away from the diagonal multiply $v_{i,j-1}$ and $v_{i,j+1}$

Note that some rows have fewer than $4-1s$, when these are on the boundary of the square. One may confirm that

$$T_{N\times N} V = h^2 f$$

Another way to write $T_{N\times N}$ is

$$\begin{bmatrix} T_N + 2I_N & -I_N & \\ -I_N & T_N + 2I_N & -I_N \\ & \ddots & \\ & -I_N & T_N + 2I_N \end{bmatrix}$$

a pattern we will generalize to arbitrary dimensions, using Kronecker products.

11.3 Poisson's Equation in d Dimensions

Definition 11.1. Let X be an $m \times n$ matrix. Then $\operatorname{vec}(X)$ is $mn \times 1$ gotten by stacking the columns of X on top of one another, from left to right.

Definition 11.2. Let A, B be $m \times n$, $p \times q$ matrices respectively. Then $A \otimes B$ is the $mp \times nq$ matrix

$$\begin{bmatrix} A_{11}B & \cdots & A_{1n}B \\ \vdots & \ddots & \vdots \\ A_{m1}B & \cdots & A_{mn}B \end{bmatrix}$$

is the Kronecker product of A and B.

Lemma 11.2

Let A, B, X be $m \times m$, $n \times n$, $m \times n$ matrices respectively. Then,

1. $\operatorname{vec}(AX) = (I_n \otimes A)\operatorname{vec}(X)$
2. $\operatorname{vec}(XB) = \left(B^\top \otimes I_m\right)\operatorname{vec}(X)$
3. The 2D Poisson's equation $T_N V + V T_N = h^2 F$ can be written as follows:

$$(I_N \otimes T_N + T_N \otimes I_N)\operatorname{vec}(V) = h^2 \operatorname{vec}(F)$$

Proof. 1. Since $I_n \otimes A = \operatorname{diag}\left(\underbrace{A, A, \cdots, A}_{n}\right)$,

$$\begin{aligned}(I_n \otimes A)\operatorname{vec}(X) &= \begin{bmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{bmatrix} \begin{bmatrix} X_{i,1} \\ X_{i,2} \\ \vdots \\ X_{i,n} \end{bmatrix} \\ &= \begin{bmatrix} AX_{i,1} \\ AX_{i,2} \\ \vdots \\ AX_{i,n} \end{bmatrix} \\ &= \operatorname{vec}(AX)\end{aligned}$$

2. Similar.
3. Apply part 1 to $T_N V$ and part 2 to $V T_N$, noting that T_N is symmetric. Then,

$$I_N \otimes T_N + T_N \otimes I_N = \begin{bmatrix} T_N & & & \\ & T_N & & \\ & & \ddots & \\ & & & T_N \end{bmatrix} + \begin{bmatrix} 2I_N & -I_N & \\ -I_N & 2I_N & -I_N \\ & \ddots & \\ & -I_N & 2I_N \end{bmatrix}$$

□

Lemma 11.3 1. Assume that AC and BD are well defined. Then,

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$$

2. If A and B are invertible, then,

$$(A \otimes B)^{-1} = (A^{-1}) \otimes (B^{-1})$$

3.

$$(A \otimes B)^\top = (A^\top) \otimes (B^\top)$$

Proposition 11.1
$T = Z\Lambda Z^\top$ be eigendecomposition of $N \times N$ symmetric T. Then the eigendecomposition of

$$I \otimes T + T \otimes I = (Z \otimes Z)(I_N \otimes \Lambda + \Lambda \otimes I_N)(Z^\top \otimes Z^\top)$$

where the first part on the right side of the equation is a diagonal matrix with entry

$$(I_N \otimes \Lambda + \Lambda \otimes I_N)_{(i-1)N+j,(i-1)N+j} = \lambda_i + \lambda_j$$

$Z \otimes Z$ is orthogonal with $((i-1)N + j)$th column

$$z_i \otimes z_j$$

Proof. Multiply out the factorization using the last lemma to get

$$\begin{aligned}(ZI_N \otimes Z\Lambda + Z\Lambda \otimes ZI_N)(Z^\top \otimes Z^\top) &= ZI_N Z^\top \otimes Z\Lambda Z^\top + Z\Lambda Z^\top \otimes ZI_N Z^\top \\ &= I_N \otimes T + T \otimes I_N\end{aligned}$$

□

Similarly, Poisson's equation in 3D leads to the matrix

$$T_{N\times N\times N} = (T_N \otimes I_N \otimes I_N) + (I_N \otimes T_N \otimes I_N) + (I_N \otimes I_N \otimes T_N)$$

with eigenvalue matrix

$$(\Lambda_N \otimes I_N \otimes I_N) + (I_N \otimes \Lambda_N \otimes I_N) + (I_N \otimes I_N \otimes \Lambda_N)$$

i.e. N^3 eigenvalues $\lambda_i + \lambda_j + \lambda_k$ for all triples (i, j, k), and corresponding eigenvector matrix $Z \otimes Z \otimes Z$. Poisson's equation equation in d dimensions is similarly represented.

11.4 Solving Poisson's Equation with Fast Fourier Transform

We now know enough to describe how to solve Poisson's equation with the FFT. This is a direct method, not iterative, but we want to compare it to the other methods. We describe it first for 2D Poisson, and then how to extend to higher dimensions. Start with 2D Poisson

$$T_N V + V T_N = F$$

which is a special case of the Sylvester equation described in Question 4.6: We substitute the eigendecomposition

$$T = Z \Lambda Z^\top$$

yielding

$$Z \Lambda Z^\top V + V Z \Lambda Z^\top = F$$

Pre-multiply by $Z^\top$ and post-multiply by Z, yielding

$$\Lambda Z^\top V Z + Z^\top V Z \Lambda = Z^\top F Z$$

Rename $V' = Z^\top V Z$ and $F' = Z^\top F Z$, yielding

$$\Lambda V' + V' \Lambda = F'$$

Since Λ is diagonal, this is a diagonal system of equations, with

$$\left(\Lambda V' + V' \Lambda\right)_{ij} = \lambda_i V'_{ij} + \lambda_j V'_{ij} = F'_{ij}$$

or

$$V'_{ij} = \frac{F'_{ij}}{\lambda_i + \lambda_j}$$

This yields the overall algorithm:

Algorithm 11.1.

1. Compute $F' = Z^\top F Z$
2. Compute $V'_{ij} = \frac{F'_{ij}}{\lambda_i + \lambda_j}$ for all i, j
3. Compute $V = Z V' Z^\top$

The main costs are the matrix multiplications by Z and $Z^\top$ in (1) and (3), which would cost $O(N^3)$ if done straightforwardly. But they can be done in $O(N^2 \log N)$ operations (recall N^2 is the number of unknowns) by noting that Z is closely related to the Fast Fourier Transform matrix:

$$\begin{aligned} \mathrm{FFT}_{ij} &= \exp\left(2\pi\sqrt{-1} ij/N\right) \\ &= \cos\frac{\pi ij}{N} + \sqrt{-1}\underbrace{\sin\frac{\pi ij}{N}}_{Z} \end{aligned}$$

for which there are $O(N \log N)$ methods for multiplication by a vector.
The same idea extends to high dimensions by using Kronecker product notation:

$$I_N \otimes T_N + T_N \otimes I_N = (Z \otimes Z)\,(I_N \otimes \Lambda + \Lambda \otimes I_N)\,(Z^\top \otimes Z^\top)$$

So,

$$\begin{aligned}(I_N \otimes T_N + T_N \otimes I_N)^{-1} &= (Z^\top \otimes Z^\top)^{-1}(I_N \otimes \Lambda + \Lambda \otimes I_N)^{-1}(Z \otimes Z)^{-1} \\ &= (Z \otimes Z)\,(D)^{-1}\,(Z^\top \otimes Z^\top)\end{aligned}$$

where D is a diagonal matrix with diagonal entries $\sum \lambda_i + \lambda_j + \lambda_k$.

11.5 Summary of performance of methods for Poisson's Equation

We will count #Flops, Memory needed, #Parallel Steps, and #Procs. "Parallel Steps" is the fewest possible, on a machine with as many processors as needed (given by #Procs), and so it refers only to the available parallelism, not how you would actually implement it on a real machine with a smaller number of processors.

The algorithms are sorted in two related orders: from slowest to fastest for the Poisson's Equation, and (roughly) from most general (fewest assumptions about the matrix) to least general.

Here are some explanations and abbreviations. Explicit inverse assumes we have pre-computed the exact inverse of the matrix A^{-1} (do not count cost of it) and only need to multiply by it.

- SOR: Successive overrelaxation
- SSOR/Chebyshev: Symmetric SOR with Chebyshev Acceleration
- FFT: Fast Fourier Transform
- BCR: Block Cyclic Reduction
- Lower Bound: The lower bound that assumes one flop per component of the output vector
- SpMV: Sparse-matrix times dense-vector multiplication, where the cost equals to the number of nonzeros.

For 2D Poisson on an $n \times n$ mesh,

$$N = n^2 = \#\text{unknowns}$$

For 3D Poisson on an $n \times n \times n$ mesh,

$$N = n^3 = \#\text{unknowns}$$

If table entry for 2D and 3D differ, 3D shown in parentheses. All the table entries are meant in a Big-Oh sense.

Method	Direct or Iterative	#Flops	Memory	#Parallel Steps	#Procs
Dense Cholesky[19]	D	N^3	N^2	N	N^2
Explicit Inverse	D	N^2	N^2	$\log N$	N^2
Band Cholesky[20]	D	N^2 $(N^{7/3})$	$N^{3/2}$ $(N^{5/3})$	N	N $(N^{4/3})$
Jacobi[21]	I	N^2 $(N^{5/3})$	N	N $(N^{2/3})$	N
Gauss-Seidel[22]	I	N^2 $(N^{5/3})$	N	N $(N^{2/3})$	N
Sparse Cholesky[23]	D	$N^{3/2}$ (N^2)	$N \log N$ $(N^{4/3})$	$N^{1/2}$ $(N^{2/3})$	N $(N^{4/3})$
Conjugate Gradients[24]	I	$N^{3/2}$ $(N^{4/3})$	N	$N^{1/2} \log N$ $(N^{1/3} \log N)$	N
SOR[25]	I	$N^{3/2}$ $(N^{1/3})$	N	$N^{1/2}$ $(N^{4/3})$	N
SSOR/Chebyshev[26]	I	$N^{5/4}$ $(N^{7/6})$	N	$N^{1/4}$ $(N^{1/6})$	N
FFT[27]	D	$N \log N$	N	$\log N$	N
BCR[28]	D	$N \log N$	N	?	?
Multigrid[29]	I	N	N	$\log^2 N$	N
Lower Bound		N	N	$\log N$	

[19]Works on any spd matrix

[20]Works on any banded spd matrix with cost $O(N\text{bw}^2)$. 2D: $\text{bw} = n = N^{1/2}$, 3D: $\text{bw} = n^2 = N^{2/3}$

[21]$O(\#\text{Flops(SpMV)} \cdot \#\text{iterations})$

[22]Cost analogous to Jacobi (constants differ). Works on any diagonally dominant or spd matrix

[23]Assumes we are using the best ordering known (nested dissection) Works on any spd matrix; complexity arises from cost of dense Cholesky on trailing $n \times n$ submatrix (in 2D) or $n^2 \times n^2$ submatrix (in 3D)

[24]Works on any spd matrix. $\#\text{Flops} = O(\#\text{Flops(SpMV)} \cdot \#\text{iterations})$, where $\#\text{iterations} = O(\sqrt{\text{cond}(A)})$

[25]Cost analysis analogous to Conjugate Gradients. Works on any spd matrix, convergence rate depends on cond

[26]Works on any spd matrix, but need to know range of eigenvalues. $\#\text{Flops} = O(\#\text{Flops(SpMV)} \cdot \#\text{iterations})$, where $\#\text{Flops(SpMV)} = O(\#\text{nonzeros in } A)$ and $\#\text{iterations} = \frac{1}{4}\,\text{cond}(A)$

[27]Works on Poisson

[28]Works on problems slightly more general than model problem

[29]Many variants, which work quite generally on problems arising from elliptic PDE and FEM models

12 Lecture 12: Splitting Methods

12.1 Splitting Methods

Goal: given an initial guess x_0 for a solution of $Ax = b$, cheaply compute a sequence x_i converging to $A^{-1}b$.

Definition 12.1. A splitting of A is a decomposition $A = M - K$ with M nonsingular.

This yields the following iterative method: Write $Ax = Mx - Kx = b$. so $Mx = Kx + b$. Then compute x_{i+1} from x_1 by solving

$$Mx_{i+1} = Kx_i + b$$

or

$$\begin{aligned} x_{i+1} &= M^{-1}Kx_i + M^{-1}b \\ &= Rx_i + c \quad (*) \end{aligned}$$

For this to work well, we need two conditions:

1. It should converge
2. Multiplying by R (i.e. solving $Mx_{i+1} = Kx_i + b$ for x_{i+1}) and computing $c = M^{-1}b$ should be much cheaper than solving with A itself

Lemma 12.1

Let $\|\cdot\|$ be any operator norm. Then if $\|R\| < 1$, $(*)$ converges to $A^{-1}b$ for any x_0.

Proof. Subtract $x = Rx + c$ from $x_{i+1} = Rx_i + c$ to get

$$x_{i+1} - x = R(x_i - x)$$

or

$$\|x_{i+1} - x\| \leq \|R\| \, \|x_{i+1} - x\| \leq \cdots \leq \|R\|^{i+1} \|x_0 - x\|$$

which converges to zero for any x_0 if $\|R\| < 1$. □

Definition 12.2. The spectral radius of R is $\rho(R) = \max |\lambda|$, the largest absolute value of any eigenvalue of R.

Lemma 12.2

For all operator norms, $\rho(R) \leq \|R\|$. For all R and all $\varepsilon > 0$, there exists an operator norm $\|\cdot\|^*$ such that

$$\|R\|^* \leq \rho(R) + \varepsilon$$

Proof. To show $\rho(R) \leq \|R\|$, note that $\|R\| = \max_{x \neq 0} \frac{\|Rx\|}{\|x\|}$, By picking x to be an eigenvector, we see that $\|R\| \geq |\lambda|$ for any eigenvalue λ of R. To construct $\|\cdot\|^*$, use the Jordan form of R:

$$S^{-1}RS = J$$

i.e. J is block diagonal with Jordan blocks. Let $D = \operatorname{diag}(1, \varepsilon, \varepsilon^2, \cdots, \varepsilon^{n-1})$. Then $J_\varepsilon = D^{-1}JD$ leaves the diagonal (eigenvalues) unchanged, and multiplies each superdiagonal entry of J by ε, so J has eigenvalues on the diagonal, and ε above the diagonal in any Jordan block. We now define a new vector norm by $\|x\|^* = \left\|(SD)^{-1}x\right\|_\infty$. Then,

$$\begin{aligned}
\|R\|^* &= \max_{x\neq 0} \frac{\|Rx\|^*}{\|x\|^*} \\
&= \max_{x\neq 0} \frac{\left\|(SD)^{-1}Rx\right\|_\infty}{\|\underbrace{(SD)^{-1}x}_{y}\|_\infty} \\
&= \max_{y\neq 0} \frac{\left\|(SD)^{-1}R(SD)y\right\|_\infty}{\|y\|_\infty} \\
&= \max_{y\neq 0} \frac{\|J_\varepsilon y\|_\infty}{\|y\|_\infty} \\
&= \|J_\varepsilon\|_\infty \\
&\leq \max|\lambda| + \varepsilon \\
&= \rho(R) + \varepsilon
\end{aligned}$$

□

Theorem 12.1

The iteration $x_{i+1} = Rx_i + c$ converges to $A^{-1}b$ for all starting vectors x_0 if and only if $\rho(R) < 1$.

Proof. If $\rho(R) \geq 1$, chose x_0 so that $x_0 - x$ is an eigenvector of R for its largest eigenvalue λ. Then $x_i - x = R^i(x_0 - x) = \lambda^i(x_0 - x)$ does not converge to zero since $|\lambda| \geq 1$. If $\rho(R) < 1$, use the last lemma to conclude that there is an operator norm $\|R\|^*$ and an ε such that

$$\|R\|^* \leq \rho(R) + \varepsilon < 1,$$

so that $x_i - x = R^i(x_0 - x)$ converges to zero for all x_0. □

Obviously, the smaller is $\rho(R)$, the faster is convergence. Our goals is to pick the splitting $A = M - K$ so that multiplying by $R = M^{-1}K$ is easy, and $\rho(R)$ is small. Choosing $M = I$ and $K = I - A$ makes multiplying by R easy, but will not generally make $rho(R)$ small. Choosing $K = 0$ and so $R = 0$ makes convergence immediate, but requires having the solution $c = A^{-1}b$.

12.2 Describe Jacobi & Gauss-Seidel & SOR

All share the notation $A = D - L' - U' = D(I - L - U)$, where D is the diagonal of A, L' is the negative of the strictly lower triangular part, and U' is the negative of the strictly upper triangular part of A.

12.2.1 Jacobi

- In words: For $j = 1$ to n, pick $x_{i+1}(j)$ to exactly satisfy equation j
- As a loop:

Algorithm 12.1.

```
1: for $j = 1 : n$ do
2:     $x_{i+1}(j) = \left(b_j - \sum_{k \neq j} A_{jk} x_i(k) / A_{jj}\right)$
3: end for
```

- As a splitting: $A = D - (L' + U') = M - K$, so

$$R_J = M^{-1}K = D^{-1}(L' + U') = L + U$$

- For the 2D Poisson equation $T_N V + V T_N = h^2 F$, to get from V_i to V_{i+1}:

Algorithm 12.2.

```
1: for $j = 1 : N$ do
2:     for $k = 1 : N$ do
3:         $V_{i+1}(j,k) = (V_i(j-1,k) + V_i(j+1,k) + V_i(j,k-1) + V_i(j,k+1) + h^2 F(j,k))/4$
           ▷ “Average” of 4 nearest neighbors and right-hand-side
4:     end for
5: end for
```

12.2.2 Gauss-Seidel

- In words: Improve on Jacobi by using most recently updated values of solution
- As a loop:

Algorithm 12.3.

```
1: for $j = 1 : n$ do
2:     $x_{i+1}(j) = \left(b_j - \sum_{k<j} A_{jk} x_{i+1}(k) - \sum_{k>j} A_{jk} x_i(k)\right) / A_{jj}$    ▷ Updated $x$ components minus older $x$ components
3: end for
```

- As a splitting: $A = (D - L') - U' = M - K$, so

$$\begin{aligned} R_{GS} &= (D - L')^{-1} U' \\ &= (D(I - D^{-1}L'))^{-1} U' \\ &= (D(I - L))^{-1} U' \\ &= (I - L)^{-1} D^{-1} U' \\ &= (I - L)^{-1} U \end{aligned}$$

Note that the order in which we update entries matters in GS (unlike Jacobi).

- For the 2D Poisson there are two orders to consider: natural order (rowwise or columnwise update of $V(i,j)$) Red-Black (RB) ordering: color the entries in the 2D mesh like a checkerboard, and number the Reds before all the Blacks. Note that each Red node only has Black neighbors, and vice-versa. Thus when updating Red nodes, we can update them in any order, since all the Black neighbors have old data. Then when we update the Black nodes, we can update them in any order, because all the Red neighbors have new data.

Algorithm 12.4.

```
1: for all $(j,k)$ nodes that are Red do                                  ▷ $j+k$ even
2:     $V_{i+1}(j,k) = (V_i(j-1,k) + V_i(j+1,k) + V_i(j,k-1) + V_i(j,k+1) + h^2F(j,k))/4$   ▷ Only use old data
3: end for
4: for all $(j,k)$ nodes that are Black do                                ▷ $j+k$ odd
5:     $V_{i+1}(j,k) = (V_{i+1}(j-1,k) + V_{i+1}(j+1,k) + V_{i+1}(j,k-1) + V_{i+1}(j,k+1) + h^2F(j,k))/4$   ▷ Only use new data
6: end for
```

12.2.3 Successive Overrelaxation (SOR)

- In words: This depends on a parameter w: We let the result of SOR be a weighted combination of the old (x_i) and new (x_{i+1}) solutions that GS would have computed:

$$x_{i+1}^{\text{SOR}(w)}(j) = (1-w)x_i(j) + wx_{i+1}^{\text{GS}}(j)$$

When $w = 1$, SOR(1) is just GS. When $w < 1$ we would call this underrelaxation, and when $w > 1$, we call it overrelaxation. We prefer to "overrelax" with the intuition that if moving in the direction from x_i to x_{i+1} was a good idea, moving even farther in the same direction is better. Later we will show how to pick w optimally for the model problem.

- As a loop:

Algorithm 12.5.

```
1: for $j = 1 : n$ do
2:     $x_{i+1}(j) = (1-w)x_i(j) + w\left(b_j - \sum_{k<j} A_{jk}x_{i+1}(k) - \sum_{k>j} A_{jk}x_i(k)\right)/A_{jj}$   ▷ Updated $x$ components minus older $x$ components
3: end for
```

- As a splitting: Multiply through by $D = A_{jj}$ to get

$$(D - wL')x_{i+1} = \left((1-w)D + wU'\right)x_i + wb$$

and then divide by w to get the splitting

$$\begin{aligned} A &= (D/w - L') - (D/w - D + U') \\ &= M - K \end{aligned}$$

or

$$\begin{aligned} R_{\mathrm{SOR}(w)} &= (D/w - L')^{-1}(D/w - D + U') \\ &= (I - wL)^{-1}\left((1-w)I + wU\right) \end{aligned}$$

- For the 2D Poisson with Red-Black Ordering:

Algorithm 12.6.

```
1: for all $(j,k)$ nodes that are Red do                                          ▷ $j+k$ even
2:     $V_{i+1}(j,k) = (1-w)V_i(j,k) + w(V_i(j-1,k) + V_i(j+1,k) + V_i(j,k-1) + V_i(j,k+1) + h^2F(j,k))/4$    ▷ Only use old data
3: end for
4: for all $(j,k)$ nodes that are Black do                                        ▷ $j+k$ odd
5:     $V_{i+1}(j,k) = (1-w)V_i(j,k) + w(V_{i+1}(j-1,k) + V_{i+1}(j+1,k) + V_{i+1}(j,k-1) + V_{i+1}(j,k+1) + h^2F(j,k))/4$    ▷ Only use new data
6: end for
```

12.3 Convergence of Splitting Methods

Now we analyze the convergence of these basic methods, in general and for the Model Problem (2D Poisson) in particular.

- Jacobi: For the Model problem, it is easy to analyze, since in the splitting

$$T_{n\times n} = M - K = 4I - (4I - T_{n\times n}),$$

so

$$R = M^{-1}K = I - T_{n\times n}/4,$$

so the eigenvalues of R are $1 - (\lambda_i + \lambda_j)/4$. For all pairs of eigenvalues,

$$\lambda_i = 2(1 - \cos(\frac{i\pi}{n+1})),$$

and so the spectral radius $\rho(R)$ of R is

$$\begin{aligned} 1 - \frac{\lambda_{\min}}{2} &= 1 - (1 - \cos\frac{\pi}{n+1}) \\ &= \cos\frac{\pi}{n+1} \\ &\approx 1 - \frac{\pi^2}{2(n+1)^2} \end{aligned}$$

which gets closer to 1 as n grows, i.e. Jacobi converges more slowly. Since the error after m steps is multiplied by $\rho(R)^m$, we estimate the speed of convergence by computing how many steps m are needed to reduce the error by a constant factor. Setting $\rho(R) = 1 - x$, $x \ll 1$,

$$\begin{aligned}\rho(R) &= (1-x)^m \\ &= (1-x)^{\frac{1}{x}mx} \\ &\approxeq (e^{-1})^{mx} \\ &= e^{-mx}\end{aligned}$$

$e^{-mx} \approx e^{-1}$ implies that $m = \frac{1}{x}$ steps are required to reduce error by a factor of e. For Jacobi,

$$\begin{aligned}\frac{1}{x} &= \frac{2(n+1)^2}{\pi^2} \\ &= O(n^2) \\ &= O(N)\end{aligned}$$

where $N = n^2$ is the number of unknowns. Note that cond$(T_{n\times n})$ $\frac{4N}{\pi^2}$ is also $O(N)$. So the number of iterations grows proportionally to the dimension N, and to the condition number. So to reduce the error by any constant factor costs $O(N)$ iterations times $O(N)$ flops_per_iteration, or $O(N^2)$ overall, explaining the entry in the table above. This is typical for many iterative methods: the number of iterations grows with the condition number (multigrid is an important exception!).

- Gauss Seidel: Assuming variables updated in right order for 2D Poisson, we get $\rho(R_{\text{GS}}) = \rho(R_J)^2$, i.e. one step of Gauss-Seidel reduces the error as much as two Jacobi steps; this is better but the overall complexity $O(N^2)$ is the same.
- SOR: Again with the right update order, and the right choice of overrelaxation parameter w, SOR(w) for 2D Poisson is much faster with

$$\rho(R_{\text{SOR}(w)}) \approxeq 1 - \frac{2\pi}{n}$$

for large n, which is close to 1 but much farther away than $\rho(R_J)$, and takes only $O(n) = O(\sqrt(N))$ steps to reduce the error by a constant factor, for an overall cost of $O(N^{\frac{3}{2}})$ as opposed to $O(N^2)$.

Now we present the general theory of these three methods applied to $Ax = b$.

Theorem 12.2

If A is strictly row diagonally dominant:

$$|A_{ii}| > \sum_{j\neq i} |A_{ij}|,$$

then both Jacobi and GS converge, and GS is at least as fast as Jacobi in the sense that

$$\|R_{\text{GS}}\|_\infty \leq \|R_J\|_\infty < 1$$

For Jacobi. Split $A = D - (D - A)$ so $R_J = D^{-1}(D - A) = I - D^{-1}A$. Then,

$$\sum_i |R_J(j,i)| = \left|1 - \frac{A_{jj}}{A_{jj}}\right| + \sum_{i\neq j} \frac{|A_{ji}|}{|A_{jj}|} = \sum_{i\neq j} \frac{|A_{ji}|}{|A_{jj}|} < 1$$

by strict row diagonal dominance. So $\|R_J\|_\infty < 1$ and Jacobi converges. □

Definition 12.3. If $|A_{jj}| \geq \sum_{i\neq j} |A_{ji}|$ in all rows, with strict inequality at least once, A is weakly row diagonally dominant.

This alone is not enough for convergence: consider $A = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ and $R = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ whose powers do not converge. So we need one more condition.

Definition 12.4. A matrix A is irreducible if there is no permutation P such that $PAP^\top$ is block triangular. Equivalently, the (directed) graph corresponding to entries of A is "strongly connected", i.e. there is a path from every vertex to every other vertex.

The model problem satisfies this definition, since a mesh is strongly connected. And for the vertices next to a boundary of the mesh,

$$|A_{ii}| > \sum_{j\neq i} |A_{ij}|$$

yielding.

Theorem 12.3

If A is weakly row diagonally dominant and irreducible (like the model problem) then

$$\rho(R_{\text{GS}}) < \rho(R_J) < 1,$$

so both Jacobi and GS converge, and GS is faster.

Theorem 12.4

IfA is symmetric positive definite, then SOR(w) converges if and only if $0 < w < 2$. In particular SOR(1) = GS converges.

12.4 Convergence of SOR for 2D Poisson

The previous results describe the most general situations in which we can prove convergence of splitting methods. To analyze the convergence of SOR(w) on the model problem we need to use another graph theoretic property of the matrix.

Definition 12.5. A matrix has Property A if there is a permutation P such that

$$PAP^\top = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

has the property that A_{11} and A_{22} are diagonal. In graph theoretic terms, the vertices V of the graph can be broken into two disjoint sets $V = V_1 \cup V_2$, where there are no edges between pairs of vertices in V_1, or between pairs of vertices in V_2. Such a graph is called bipartite.

Example 12.1

For a model problem on a 1D mesh, let the odd numbered nodes be V_1 and the even numbered be V_2; since there are only edges connecting even and odd nodes, the matrix has property A.

Example 12.2

For a model problem on a 2D mesh, think of the vertices forming a checkerboard so they are either Red or Black. Then the edges only connect Red to Black, so the matrix has Property A. Another way to say this is to put vertices v_{ij} with $i+j$ odd in V_1 and $i+j$ even in V_2. This works for the 3D model problem and in higher dimensions.

Theorem 12.5

Suppose matrix A has Property A, and we do $R_{\text{SOR}(w)}$ updating all the vertices in V_1 before all the vertices in V_2. Then the eigenvalues μ of R_J and λ of SOR(w) are related by

$$(\lambda + w - 1)^2 = \lambda w^2 \mu^2 \quad (*)$$

In particular, if $w = 1$, then $\lambda = \mu^2$, so

$$\rho(R_{\text{SOR}(1)}) = \rho(R_{\text{GS}}) = \rho(R_J)^2,$$

and GS converges twice as fast as Jacobi.

Proof. Assuming we number all vertices in V_1 before the vertices in V_2, matrix A will look like

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where A_{11} and A_{22} are diagonal, since property A tells us there are no edges connected vertices in V_1 to V_1 or V_2 to V_2. For Poisson this means

$$\begin{aligned} A &= 4I + \begin{bmatrix} 0 & A_{12} \\ A_{21} & 0 \end{bmatrix} \\ &= 4I + \begin{bmatrix} 0 & 0 \\ A_{21} & 0 \end{bmatrix} + \begin{bmatrix} 0 & A_{12} \\ 0 & 0 \end{bmatrix} \\ &= D - L' - U' \end{aligned}$$

Now since λ is an eigenvalue of SOR(w) we get

$$\begin{aligned}
0 &= \det(\lambda I - \mathrm{SOR}(w)) \\
&= \det(\lambda I - (I - wL)^{-1}((1-w)I + wU)) \\
&= \det((I - wL)(\lambda I - (I - wL)^{-1}((1-w)I + wU))) \\
&= \det(\lambda I - \lambda w L - (1-w)I - wU) \\
&= \det((\lambda - 1 + w)I - \lambda w L - wU) \\
&= \det(\sqrt{\lambda} w [\frac{\lambda - 1 + w}{\sqrt{\lambda} w} I - \sqrt{\lambda} L - \frac{1}{\sqrt{\lambda}} U]) \\
&= (\sqrt{\lambda} w)^n \det(\frac{\lambda - 1 + w}{\sqrt{\lambda} w} I - \sqrt{\lambda} L - \frac{1}{\sqrt{\lambda}} U)
\end{aligned}$$

Because of the sparsity of L and U, we can let $D = \begin{bmatrix} I & 0 \\ 0 & \frac{1}{\sqrt{2}} I \end{bmatrix}$, and note that $D\sqrt{\lambda} L D^{-1} = L$, and $D\sqrt{\lambda} U D^{-1} = U$, so we can premultiply the matrix inside det() by D and postmultiply by D^{-1}, without changing the determinant, to get

$$\begin{aligned}
0 &= \det(\frac{\lambda - 1 + w}{\sqrt{\lambda} w} I - L - U) \\
&= \det(\frac{\lambda - 1 + w}{\sqrt{\lambda} w} I - R_J)
\end{aligned}$$

i.e. $\frac{\lambda-1+w}{\sqrt{\lambda}w}$ is an eigenvalue μ of the Jacobi matrix R_J. More generally, since we know all eigenvalues μ of R_J for the model problem, we can use $(*)$ to figure out all eigenvalues λ of SOR(w), then then pick w_{opt} to minimize $\rho(R_{\mathrm{SOR}(w_{\text{opt}})})$. This yields. □

Theorem 12.6

Suppose A has property A, we do SOR(w) updating vertices in V_1 before V_2 as before, all eigenvalues of R_J are real, and $\mu = \rho(R_J) < 1$ (as in the model problem). Then

$$w_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \mu^2}}$$

and

$$\begin{aligned}
\rho(R_{\mathrm{SOR}(w_{\text{opt}})}) &= \frac{\cos^2\left(\frac{\pi}{n+1}\right)}{\left(1 + \sin\frac{\pi}{n+1}\right)^2} \\
&\approx 1 - \frac{2\pi}{n+1}
\end{aligned}$$

The proof follows by solving $(*)$ for λ (see Thm 6.7 in the text).

13 Lecture 13: Multigrid

14 Lecture 14: Introduction to Krylov Subspace Methods

Next we consider *Krylov Subspace Methods*, of which there are many, depending on the matrix structure. For symmetric positive definite matrices like our model problem, *Conjugate Gradients* (CG) is the most widely used, and the one we will discuss in most detail. We will also summarize the other methods with a "decision tree" that chooses the right algorithm depending on properties of your matrix.

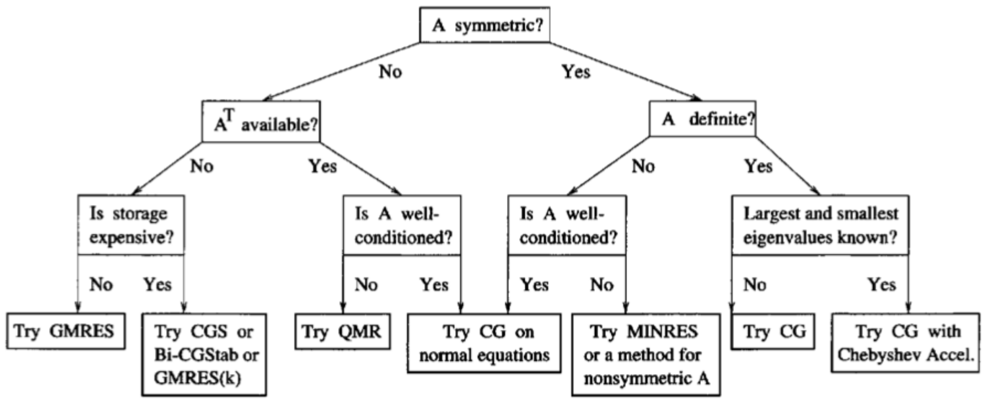

A symmetric?
No
Yes
A^T available?
A definite?
No
Yes
No
Yes
Is storage expensive?
Is A well-conditioned?
Is A well-conditioned?
Largest and smallest eigenvalues known?
No
Yes
No
Yes
Yes
No
No
Yes
Try GMRES
Try CGS or Bi-CGStab or GMRES(k)
Try QMR
Try CG on normal equations
Try MINRES or a method for nonsymmetric A
Try CG
Try CG with Chebyshev Accel.

Figure 14.1: Decision tree for choosing an iterative algorithm for $Ax = b$.

Unlike Jacobi, GS and SOR, a Krylov subspace method for $Ax = b$ or $Ax = \lambda x$ need only a "black-box" subroutine to multiply A times a vector by x (or $A^\top$ times x, for some methods). So you do not need to ask for the diagonal part, or subdiagonal part, etc. that you would need for methods like Jacobi.

This is particularly useful in two situations:

1. You can write algorithms that are very general, can access A only through subroutine for Ax.

2. You can solve problems where A may not actually be represented as a matrix with explicit entries, e.g. arises from doing a complicated simulation. For example, suppose you have a system, like a car engine or airplane wing, governed by a complicated set of PDEs. You'd like to optimize some set y of n outputs of the PDE as a function of some set x of n inputs, $y = f(x)$, say the pressure on the wing as a function of its shape. For some optimization methods, this means that you need to solve linear systems with the $n \times n$ coefficient matrix $A = \nabla f$. A is not written down, it is define implicitly as the derivative of the outputs of the simulation with respect to its inputs, and varies depending on the inputs. The easiest way to access A is to note that

$$Az = \nabla f z \sim \frac{(f(x+hz) - f(x))}{h}$$

when h is small enough, so multiplication by A requires running the simulation twice to get $f(x+hz)$ and $f(x)$ (care must be taken in choosing h small enough but not too small, because otherwise the result is dominated by error). There is no such simple way to get $A^\top z$, unfortunately. There is actually software that takes an arbitrary collection of C or

Fortran subroutines computing an arbitrary function y $= f(x)$, and produces new C or Fortran subroutines that compute $\nabla f(x)$, by differentiating each line of code automatically. A specialized and widely used example, for training neural nets, is TensorFlow.

On the other hand, if we assume that the matrix A is available explicitly, then it is possible to reorganize many Krylov subspace methods to significantly reduce their communication costs, which are dominated by moving the data needed to represent the sparse matrix A. For example, when A is too large to fit in cache (a typical situation), each step of a conventional sequential method does a multiplication Ax, which requires reading the matrix A from slow memory to cache. Thus the communication cost is proportional to the number of steps, call it k. However the reorganized Krylov subspace methods can take k steps and only read A from slow memory once, a potential speedup of a factor of k. See "Communication lower bounds and optimal algorithms for numerical linear algebra" for more details. In practice k is limited by both the sparsity structure of A and issues of numerical stability.

14.1 Extracting information about A from a subroutine that does Az

We begin by discussing how to extract information about A from a subroutine that does Az. Given a starting vector y_1 (say $y_1 = b$ for solving $Ax = b$), we can compute $y_2 = Ay_1$, ..., $y_{i+1} = Ay_i$, $\dots y_n = Ay_{n-1} = A^{n-1}y_1$. Letting $K = [y_1, y_2, \cdots, y_n]$ we get

$$AK = [Ay_1, \cdots, Ay_n] = [y_2, y_3, \cdots, A^n y_1]$$

Assuming for a moment that K is nonsingular, write $c = -K^{-1}A^n y_1$ to get

$$AK = K\,[e_2, e_3, \cdots, e_n, -c] = KC$$

where

$$C = K^{-1}AK = \begin{bmatrix} 0 & 0 & & -c_1 \\ 1 & 0 & & -c_2 \\ 0 & 1 & & -c_3 \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & & -c_n \end{bmatrix}$$

is upper Hessenberg, and a companion matrix, i.e. its characteristic polynomial is simply

$$p(x) = x^n + \sum_{i=1}^{n} c_i x^{i-1}.$$

So just by matrix-vector multiplication we have reduced A to a simple form, and could imagine using C to solve $Ax = b$ or find eigenvalues. But this would be a bad idea for two reasons:

1. K is likely to be dense (if y_1 is dense), and solving with K is probably harder than solving with A.

2. K is likely to be very ill-conditioned, since it is basically running the power method, so the vectors y_i are converging to the eigenvector of the largest eigenvalue in absolute value.

Krylov subspace methods address these two drawbacks, implicitly or explicitly, as follows:

We will not compute K but rather an orthogonal Q such that the leading k columns of K and of Q span the same space, called a Krylov subspace:

$$\operatorname{span}\{y_1, y_2, \cdots, y_k\} = \operatorname{span}\left\{y_1, Ay_1, \cdots, A^{k-1}y_1\right\} = \mathcal{K}_k(A, y_1)$$

The relationship between K and Q is simply the QR decomposition: $K = QR$. Furthermore, we won't compute all n columns of K and Q, but just the first $k \ll n$ columns, and settle for the "best" approximate solution x of $Ax = b$ or $Ax = \lambda x$ that we can find in the Krylov subspace they span (the definition of "best" depends on the algorithm).

14.2 Reduction of A to Hessenberg form

To proceed, substitute $K = QR$ into $C = K^{-1}AK$ to get

$$C = R^{-1}Q^\top AQR$$

or

$$RCR^{-1} = Q^\top AQ = H \qquad (\star)$$

Since R and R^{-1} are triangular, and C is upper Hessenberg, then H is also upper Hessenberg. If A is symmetric, then so is H, and so H is tridiagonal.

To see how to compute the columns of Q and H one at a time, write $Q = [q_1, q_2, \cdots, q_n]$, rewrite $(\star)$ as

$$AQ = QH$$

and equate the j-th column on both sides to get

$$Aq_j = \sum_{i=1}^{j+1} q_i H(i, j)$$

Since the q_j are orthonormal, multiply both sides of the last equality by $q_m^\top$ to get

$$q_m^\top Aq_j = \sum_{i=1}^{j+1} H(i, j) q_m^\top q_i = H(m, j)$$

for $1 \le m \le j$. And so

$$H(j+1, j)q_{j+1} = Aq_j - \sum_{i=1}^{j} q_i H(i, j).$$

This yields the following algorithm:

Algorithm 14.1 (Arnoldi algorithm for (partial) reduction of A to Hessenberg form).

```
$q_1 = y_1/||y_1||_2$
for $j = 1 : k$ do
    $z = Aq_j$
    for $i = 1 : j$ do                      ▷ Run Modified Gram-Schmidt (MGS) on $z$
        $H(i,j) = q_i^\top z$
        $z = z - H(i,j)q_i$
    end for
    $H(j+1,j) = ||z||_2$
    if $H(j+1,j) = 0$ then
        quit
    end if
    $q_{j+1} = z/H(j+1,j)$
end for
```

The q_j vectors are called Arnoldi vectors, and the cost is k multiplications by A, plus $O(k^2 n)$ flops for MGS. If we stop the algorithm here, at $k < n$, what have we learned about A? Write $Q = [Q_k, Q_u]$ where $Q_k = [q_1, \cdots, q_k]$ is computed as well as the first column q_{k+1} of Q_u; the other columns of Q_u are unknown. Then

$$\begin{aligned} H = Q^\top AQ &= [Q_k, Q_u]^\top A[Q_k, Q_u] \\ &= \begin{bmatrix} Q_k^\top AQ_k & Q_k^\top AQ_u \\ Q_u^\top AQ_k & Q_u^\top AQ_u \end{bmatrix} \\ &= \begin{bmatrix} H_k & H_{uk} \\ H_{ku} & H_u \end{bmatrix} \end{aligned}$$

Since H is upper Hessenberg, so are H_k and H_u, and H_{ku} has a single nonzero entry in its upper right corner, namely $H(k+1,k)$. So H_{uk} and H_u are unknown, and H_k and H_{ku} are known.

14.3 Reduction of A to tridiagonal form when A is symmetric

When A is symmetric, so that H is symmetric and tridiagonal, we may write

$$H = T = \begin{bmatrix} \alpha_1 & \beta_1 & & O \\ \beta_1 & \alpha_2 & \beta_2 & \\ & \beta_2 & \alpha_3 & \ddots \\ & & \ddots & \ddots \\ O & & & \alpha_n \end{bmatrix}$$

Equating column j on both sides of $AQ = QT$ yields

$$Aq_j = \beta_{j-1}q_{j-1} + \alpha_j q_j + \beta_j q_{j-1}$$

Multiplying both sides by $q_j^\top$ yields

$$q_j^\top Aq_j = \alpha_j,$$

which leading to the following simpler version of the Arnoldi algorithm, called Lanczos:

Algorithm 14.2 (Lanczos algorithm for (partial) reduction of $A = A^\top$ to tridiagonal form).

```
$q_1 = y_1/||y_1||_2$
$\beta_0 = 0$
$q_0 = 0$
for $j = 1 : k$ do
    $z = Aq_j$
    $\alpha_j = q_j^\top z$
    $z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$
    $\beta_j = ||z||_2$
    if $\beta_j = 0$ then
        quit
    end if
    $q_{j+1} = z/\beta_j$
end for
```

14.4 Krylov subspace notation

Here is some notation to describe the Arnoldi and Lanczos algorithms.
Definition 14.1. The space spanned by $\{q_1, \cdots, q_k\}$ is called a Krylov subspace:

$$\mathcal{K}_k(A, y_1) = \operatorname{span}\{q_1, \cdots, q_k\}$$

or $\mathcal{K}_k$ for short.
Definition 14.2. We call H_k (T_k for Lanczos) the projection of A onto $\mathcal{K}_k$.

14.5 Solving eigenvalue problems using Krylov subspaces

Our goal is to solve $Ax = b$ or $Ax = \lambda x$ by somehow finding the "best" solution in the Krylov subspace. For the eigenvalue problem, the answer is simple: use the eigenvalues of H_k (or T_k) as approximate eigenvalues of A. To see what the error is, suppose $H_k y = \lambda y$ and write

$$H \begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} H_k & H_{uk} \\ H_{ku} & H_u \end{bmatrix} \begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} H_k y \\ H_{ku} y \end{bmatrix} = \begin{bmatrix} \lambda y \\ H_{k+1,k} y_k e_1 \end{bmatrix}$$

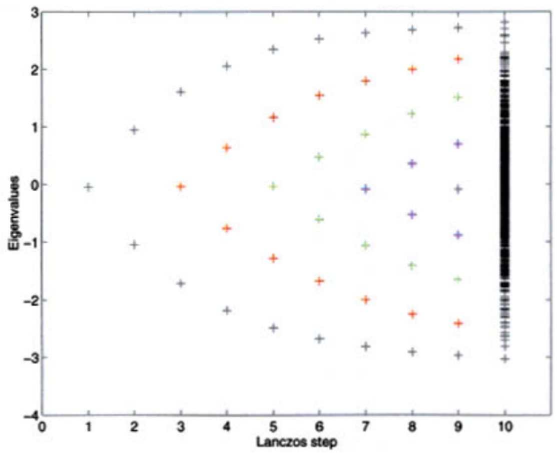
and since $AQ = QH$,

$$AQ \begin{bmatrix} y \\ 0 \end{bmatrix} = \lambda Q \begin{bmatrix} y \\ 0 \end{bmatrix} + q_{k+1} H_{k+1,k} y_k$$

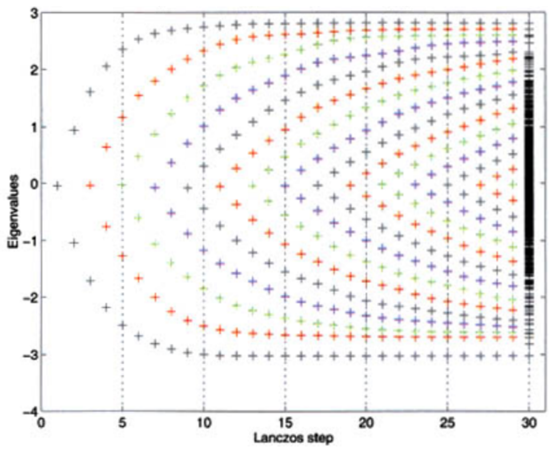
so if the product of $H_{k+1,k} y_k$ is small, the eigenvalue/vector pair $(\lambda, y') = (\lambda, Q[y, 0]^\top)$ has a small residual:

$$||Ay' - \lambda y'||_2 = |H_{k+1,k} y_k|.$$

When A is symmetric, we know that this is in fact a bound on the error in λ.



(a) 9 steps



(b) 29 steps

Figure 14.2: The Lanczos algorithm (full reorthogonalization) applied to A with different numbers of steps. Column k shows the eigenvalues of T_k, except that the rightmost columns show all the eigenvalues of A.

To see how the eigenvalues of T_k approximate the eigenvalues of A as k grows, we consider the symmetric case (Lanczos), and plot the eigenvalue of T_k versus k.

This lecture has ignored floating point errors so far. In practice, it is common for the vectors q_j to lose orthogonality as the number of iterations grows, leading to more complicated behavior, and interesting algorithmic changes.

15 Lecture 15: Krylov Subspace Methods–GMRES and CG

To solve $Ax = b$, given the orthogonal basis of the Krylov subspace $Q_k = [q_1, \cdots, q_k]$, our goal is to find the "best" approximate solution x_k in span(Q_k). To make progress, we need to define "best"; depending on what we choose, we end up with different algorithms:

1. Choose x_k to minimize $||x_k - x||_2$, where x is the true solution. Unfortunately, we don't have enough information in Q_k and
$$H_k = Q_k^\top A Q_k$$
(or T_k when A is symmetric) to compute this.

2. Choose x_k to minimize the norm $||r_k||_2$ of the residual $r_k = b - Ax_k$. We can do this, and the algorithm is called MINRES (for "minimum residual") when A is symmetric, and *GMRES* (for "generalized minimum residual") when it is not.

3. Choose x_k so that r_k is perpendicular to $\mathcal{K}_k$, i.e. $r_k^\top Q_k = 0$. This is called the orthogonal residual property, or a Galerkin condition, by analogy to finite elements. When A is symmetric, the algorithm is called *SYMMLQ*. When A is nonsymmetric, a variant of *GMRES* works.

4. When A is spd, it defines a norm
$$||r||_{A^{-1}} = (r^\top A^{-1} r)^{1/2}.$$
We say the best solution minimizes $||r_k||_{A^{-1}}$. Note that
$$\begin{aligned} ||r_k||^2_{A^{-1}} &= r_k^\top A^{-1} r_k \\ &= (b - Ax_k)^\top A^{-1}(b - Ax_k) \\ &= (Ax - Ax_k)^\top A^{-1}(Ax - Ax_k) \\ &= (x - x_k)^\top A(x - x_k) \\ &= ||x - x_k||^2_A \end{aligned}$$
This algorithm is called the *Conjugate Gradient Algorithm* (*CG*).

Theorem 15.1

When A is spd, definitions (3) and (4) of "best" are equivalent, and using the Conjugate Gradient algorithm, it is possible to compute x_k from previous iterates for the cost of one multiplication by A, and a small number of dot products and saxpys ($y = \alpha x + y$), keeping only 3 vectors in memory.

More generally, the choice of algorithm depends on the properties of A.

We now discuss *GMRES*, which uses the least structure of the matrix, and then *CG*, which uses the most.

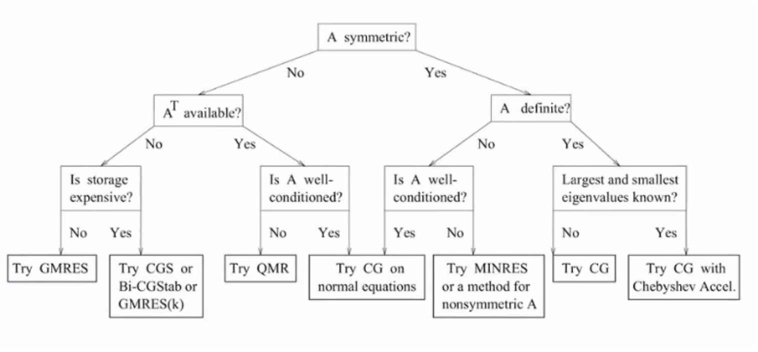

A symmetric?
No
Yes
A^T available?
No
Yes
A definite?
No
Yes
Is storage expensive?
No
Yes
Is A well-conditioned?
No
Yes
Is A well-conditioned?
Yes
No
Largest and smallest eigenvalues known?
No
Yes
Try GMRES
Try CGS or Bi-CGStab or GMRES(k)
Try QMR
Try CG on normal equations
Try MINRES or a method for nonsymmetric A
Try CG
Try CG with Chebyshev Accel.

Figure 15.1: Decision tree.

15.1 GMRES

In GMRES, at each step we choose x_k to minimize

$$||r_k||_2 = ||b - Ax_k||_2$$

where $x_k = Q_k y_k$, i.e. $x_k \in \mathcal{K}_k(A, b)$. Thus we choose y_k to minimize

$$\begin{aligned}
||r_k||_2 &= ||b - AQ_k y_k||_2 \\
&= \left\| b - A[Q_k, Q_u] \begin{bmatrix} y_k \\ 0 \end{bmatrix} \right\|_2 \\
&= \left\| Q^\top \left(b - A[Q_k, Q_u] \begin{bmatrix} y_k \\ 0 \end{bmatrix} \right) \right\|_2 \\
&= \left\| Q^\top b - H \begin{bmatrix} y_k \\ 0 \end{bmatrix} \right\|_2 \\
&= \left\| e_1 ||b||_2 - \begin{bmatrix} H_k & H_{uk} \\ H_{ku} & H_u \end{bmatrix} \begin{bmatrix} y_k \\ 0 \end{bmatrix} \right\|_2 \\
&= \left\| e_1 ||b||_2 - \begin{bmatrix} H_k \\ H_{ku} \end{bmatrix} \right\|_2
\end{aligned}$$

Since only the first row of H_k is nonzero, we just need to solve the $(k+1)$-by-k upper Hessenberg least squares problem

$$||r_k||_2 = \left\| e_1 ||b||_2 - \begin{bmatrix} H_k \\ 0 \cdots, 0, H_{k+1,k} \end{bmatrix} y_k \right\|_2$$

which can be done inexpensively with k Givens rotations, exploiting the upper Hessenberg structure, costing just $O(k^2)$ or $O(k)$ instead of $O(k^3)$.

15.2 Conjugate Gradient Algorithm (CG)

Now we return to CG, and begin by showing that when A is spd, it is "best" in two senses.

Lemma 15.1

When A is spd, (3) and (4) are equivalent: (3) choose x_k so that $r_k^\top Q_k = 0$. (4) choose x_k to minimize $||r_k||_{A^{-1}} = ||x_k - x||_{A^{-1}}$ which are solved by

$$x_k = Q_k T_k^{-1} Q_k^\top b = Q_k T_k^{-1} e_1 ||b||_2 \quad (\star),$$

where T_k is the tridiagonal matrix computed by Lanczos:

$$Q_k^\top A Q_k.$$

Also $r_k = \pm ||r_k|| q_{k+1}$.

Here is some intuition for $(\star)$:

- Multiplying $Q_k^\top b = e_1 ||b||_2$ projects b onto the Krylov subspace spanned by Q_k.
- T_k^{-1} is the inverse of the projection of A onto the Krylov subspace.
- Multiplying by Q_k maps back from the Krylov subspace to $\mathbb{R}^n$.

Proof. Drop the subscript k for simpler notation, so $Q = Q_k$, $T = T_k$, $x = QT^{-1}e_1||b||_2$, $r = b - Ax$ and $T = Q^\top AQ$ is spd (and so nonsingular), since A is. Then we need to confirm that

$$\begin{aligned} Q^\top r &= Q^\top (b - Ax) \\ &= Q^\top b - Q^\top Ax \\ &= e_1||b||_2 - Q^\top AQT^{-1}e_1||b||_2 \\ &= e_1||b||_2 - TT^{-1}e_1||b||_2 \\ &= 0 \end{aligned}$$

as desired.

Now we need to show that this choice of x minimizes $||r||^2_{A^{-1}}$, so consider a different $x' = x + Qz$ and $r' = b - Ax' = r - AQz$,

$$\begin{aligned} ||r'||^2_{A^{-1}} &= r'^\top A^{-1} r' \\ &= (r - AQz)^\top A^{-1}(r - AQz) \\ &= r^\top A^{-1} r - 2(AQz)^\top A^{-1} r + (AQz)^\top A^{-1}(AQz) \\ &= r^\top A^{-1} r - 2z^\top Q^\top AA^{-1} r + (AQz)^\top A^{-1}(AQz) \\ &= r^\top A^{-1} r - 2z^\top Q^\top r + (AQz)^\top A^{-1}(AQz) \\ &= r^\top A^{-1} r + (AQz)^\top A^{-1}(AQz) \\ &= ||r||^2_{A^{-1}} + ||AQz||^2_{A^{-1}} \end{aligned}$$

and this is minimized by choosing $z = 0$, so $x' = x$ as desired. Finally note that $r_k = b - Ax_k$ must be in $\mathcal{K}_{k+1}$, so that r_k is a linear combination of columns of $Q_{k+1} = [q_1, \cdots, q_{k+1}]$. But since r_k is perpendicular to the columns of $Q_k = [q_1, \cdots, q_k]$, r_k must be parallel to q_{k+1}, so $r_k = \pm||r_k||_2 q_{k+1}$. □

15.2.1 Deriving CG

There are several ways to derive CG. We will take the most "direct" way from the formula ($\star$) above, deriving recurrences for 3 sets of vectors, of which we only need to keep the most recent ones: x_k, r_k, and so-called conjugate gradient vectors p_k.

1. The p_k are called gradients because each step of CG can be thought of as moving x_{k-1} along the gradient direction p_k (i.e. $x_k = x_{k-1} + \nu p_k$) until it minimizes $||r_k||_{A^{-1}}$ over all choices of the scalar ν.
2. The p_k are called conjugate (or more precisely A-conjugate), because they are orthogonal in the A inner product: $p_k^\top A p_j = 0$ if $j \neq k$.

CG is sometimes derived by figuring out recurrences for x_k, r_k and p_k that satisfy properties (1) and (2), and then showing they satisfy the optimality properties in the Lemma. We will start with the formula for $x_k = Q_k T_k^{-1} e_1 ||b||_2$, from the lemma, and derive the recurrences from there.

Since $T_k = Q_k^\top A Q_k$ is spd, we can do Cholesky to get

$$T_k = L'_k {L'_k}^\top$$

where L'_k is lower bidiagonal, and

$$L'_k {L'_k}^\top = L_k D_k L_k^\top$$

where L_k has unit diagonal and D_k is diagonal, with

$$D_k(i,i) = \left(L'_k(i,i)\right)^2 .$$

Then from the Lemma,

$$\begin{aligned} x_k &= Q_k T_k^{-1} e_1 ||b||_2 \\ &= Q_k (L_k D_k L_k^\top)^{-1} e_1 ||b||_2 \\ &= Q_k L_k^{-\top} (D_k^{-1} L_k^{-1} e_1 ||b||_2) \\ &= P'_k y_k \end{aligned}$$

where we write $P'_k = [p'_1, \cdots, p'_k]$ and $y_k = D_k^{-1} L_k^{-1} e_1 ||b||_2$. The eventual conjugate gradients p_k will turn out to be scalar multiples of the p'_k. So we know enough to prove property (2) above.

Lemma 15.2

The p'_k are A-conjugate, i.e. ${P'_k}^\top A P'_k$ is diagonal.

Proof.

$$\begin{aligned} P_k'^{\top} A P_k' &= (Q_k L_k^{-\top})^{\top} A (Q_k L_k^{-\top}) \\ &= L_k^{-1} Q_k^{\top} A Q_k L_k^{-\top} \\ &= L_k^{-1} T_k L_k^{-\top} \\ &= L_k^{-1} (L_k D_k L_k^{\top}) L_k^{-\top} \\ &= D_k \end{aligned}$$

□

Now we derive simple recurrences for the column p_k' of P_k', and the components of y_k, which in turn give us a recurrence for x_k. We will show that P_{k-1}' is identical to the leading $k-1$ columns of P_k':

$$P_k' = [P_{k-1}', p_k'],$$

and similarly for $y_{k-1} = [s_1, \cdots, s_{k-1}]$ and $y_k = [s_1, \cdots, s_{k-1}, s_k]$. Assuming these are true for a moment, they will give us the recurrence for x_k, R_x:

$$\begin{aligned} x_k = P_k' y_k &= [P_{k-1}', p_k'][s_1; ...; s_k] \\ &= P_{k-1}'[s_1; ...; s_{k-1}] + p_k' s_k \\ &= x_{k-1} + p_k' s_k, \end{aligned}$$

assuming we can also get recurrences for p_k' and s_k.

Since Lanczos constructs T_k row by row, so T_{k-1} is the leading $k-1 \times k-1$ submatrix of T_k, and Cholesky also computes row by row, we get that L_{k-1} and D_{k-1} are the leading $k-1 \times k-1$ submatrices of L_k and D_k, resp:

$$\begin{aligned} T_k &= L_k D_k L_k^{\top} \\ &= \begin{bmatrix} l_1 & & & & 0 \\ 0 & \cdots & 0 & l_{k-1} & 1 \end{bmatrix} \begin{bmatrix} D_{k-1} & \\ & d_k \end{bmatrix} \begin{bmatrix} l_1 & & & & 0 \\ 0 & \cdots & 0 & l_{k-1} & 1 \end{bmatrix}^{\top}, \end{aligned}$$

so

$$L_k^{-1} = \begin{bmatrix} L_{k-1}^{-1} & 0 \\ \text{stuff} & 1 \end{bmatrix},$$

and

$$\begin{aligned} y_k &= D_k^{-1} L_k^{-1} e_1 \|b\|_2 \\ &= \begin{bmatrix} D_{k-1}^{-1} & 0 \\ 0 & d_k^{-1} \end{bmatrix} \begin{bmatrix} L_{k-1}^{-1} & 0 \\ \text{stuff} & 1 \end{bmatrix} e_1 \|b\|_2 \\ &= \begin{bmatrix} D_{k-1}^{-1} L_{k-1}^{-1} e_1 \|b\|_2 \\ s_k \end{bmatrix} = \begin{bmatrix} y_{k-1} \\ s_k \end{bmatrix}, \end{aligned}$$

which is a recurrence for y_k. Similarly,

$$\begin{aligned} P'_k &= Q_k L_k^{-\top} \\ &= [Q_{k-1}, q_k] \begin{bmatrix} L_{k-1}^{-1} & \text{stuff} \\ 0 & 1 \end{bmatrix} \\ &= \left[Q_{k-1} L_{k-1}^{-\top}, p'_k \right] \\ &= \left[P'_{k-1}, p'_k \right]. \end{aligned}$$

To get a formula for p'_k,

$$Q_k = P'_k L_k^{\top},$$

and equating the last columns we get

$$q_k = p'_k + p'_{k-1} L_k(k, k-1),$$

or

$$p'_k = q_k - l_{k-1} p'_{k-1}$$

as the desired recurrence for p'_k.

Finally we need a recurrence for r_k from R_x:

$$\begin{aligned} r_k &= b - Ax_k \\ &= b - A(x_{k-1} + p'_k s_k) \\ &= r_{k-1} - Ap'_k s_k. \end{aligned}$$

Putting these vector recurrences together we get

$$\begin{aligned} r_k &= r_{k-1} - Ap'_k s_k, \\ x_k &= x_{k-1} + p'_k s_k, \\ p'_k &= q_k - l_{k-1} p'_{k-1}. \end{aligned}$$

We then substitute

$$\begin{aligned} q_k &= r_{k-1} / ||r_{k-1}||_2 \\ p_k &= ||r_{k-1}||_2 p'_k, \end{aligned}$$

to get

$$\begin{aligned} r_k &= r_{k-1} - Ap_k (s_k / ||r_{k-1}||_2) = r_{k-1} - Ap_k \nu_k, \\ x_k &= x_{k-1} + p_k \nu_k, \\ p_k &= r_{k-1} - \frac{l_{k-1} ||r_{k-1}||_2}{|r_{k-2}||_2} p_{k-1} = r_{k-1} + \mu_k p_{k-1}. \end{aligned}$$

We still need formulas for the scalars μ_k and ν_k. There are several choices, some more numerically stable than others. See the text for the algebra, we just write here

$$\nu_k = \frac{r_{k-1}^\top r_{k-1}}{p_k^\top A p_k},$$
$$\mu_k = \frac{r_{k-1}^\top r_{k-1}}{r_{k-2}^\top r_{k-1}}.$$

Putting it all together, we get:

Algorithm 15.1 (Conjugate Gradient Method for solving $Ax = b$).

```
$k = 0$, $x_0 = 0$, $r_0 = b$, $p_1 = b$
while $||r_k||_2$ not small enough do
    $k = k + 1$
    $z = Ap_k$
    $\nu_k = \frac{r_{k-1}^\top r_{k-1}}{p_k^\top z}$
    $x_k = x_{k-1} + \nu_k p_k$
    $r_k = r_{k-1} - \nu_k z$
    $\mu_{k+1} = \frac{r_k^\top r_k}{r_{k-1}^\top r_{k-1}}$
    $p_{k+1} = r_k + \mu_{k+1} p_k$
end while
```

Note that minimization property (1) above follows from the fact that since x_k minimizes $||r_k||_A^{-1}$ over all possible x_k in Q_k, it must certainly also satisfy the lower dimensional minimization property (1).

15.2.2 Convergence of CG

As with most other iterative methods (besides multigrid), the convergence rate of CG depends on the condition number $\kappa(A)$.

Theorem 15.2

$$\frac{||r_k||_{A^{-1}}}{||r_0||_{A^{-1}}} \le \frac{2}{1 + \frac{2k}{\sqrt{\kappa(A)-1}}}$$

So when $\kappa(A)$ is large, one needs to take $O(\sqrt{\kappa(A)})$ steps to converge. For d-dimensional Poisson's equation on a grid with n mesh points on a side, $\kappa(A)$ is about n^2, so CG take $O(n)$ steps to converge, for any dimension d.

16 Lecture 16: Chebyshev Polynomials–Convergence Analysis of CG and Accelerating SOR

Recall that Krylov subspace methods (KSMs) seek the best x approximating $Ax = b$ in a k-dimensional Krylov subspace:

$$\mathcal{K}_k(A, B) = \text{span}\left\{b,\ Ab,\ A^2b, \cdots, A^{k-1}b\right\} = \text{span}\left\{p_{k-1}(A)b\right\},$$

where p is a polynomial of degree less or equal to $k-1$.

In the case of CG, recall that the best x_k is defined to minimize:

$$\begin{aligned}||Ax_k - b||_{A^{-1}}^2 &= (Ax_k - b)^\top A(Ax_k - b) \\ &= (Ap_{k-1}(A)b - b)^\top A^{-1}(Ap_{k-1}(A)b - b) \\ &= (q_k(A)b)^\top A^{-1}(q_k(A)b),\end{aligned}$$

where $q_k(A) = I - Ap_{k-1}(A)$ is a polynomial of degree k with

$$q_k(0) = 1 = b^\top q_k(A)^2 A^{-1} b$$

Since A is spd, we can write its eigendecomposition $A = Q\Lambda Q^\top$ and get

$$\begin{aligned}y &= Q^\top b \\ &= y^\top q_k(\Lambda)^2 \Lambda^{-1} y \\ &= \sum_i q_k(\lambda_i)^2 \lambda_i^{-1} y_i^2.\end{aligned}$$

So the best x_k corresponds to the best polynomial q_k, i.e. that minimizes this weighted sum of its squared values $q_k(\lambda_i)^2$ at the eigenvalues of A. To simplify, we upper bound this by

$$\begin{aligned}q_k(\lambda_i)^2 &\le \max_i |q_k(\lambda_i)|^2 y^\top \Lambda^{-1} y \\ &= \max_i |q_k(\lambda_i)|^2 ||b||_{A^{-1}}^2\end{aligned}$$

or

$$\frac{||Ax_k - b||_{A^{-1}}}{||b||_{A^{-1}}} \le \max_i |q_k(\lambda_i)|.$$

So seek q_k where $q_k(0) = 1$ and $|q_k(\lambda_i)|$ small for all eigenvalues of A. Note that 0 can not be an eigenvalue of A, since A is spd. Then we can get a useful bound on the convergence rate. Chebyshev polynomials, suitably scaled depending on A, will serve this purpose, and give us a bound on how fast CG converges.

Another example where polynomials can help is accelerating a splitting method, like SOR. Recall that a splitting method produces a sequence of approximate solutions

$$x_k = Rx_{k-1} + c,$$

where the exact solution of $Ax = b$ satisfies $x = Rx + c$, and so

$$x_k - x = R(x_{k-1} - x) = R^k(x_0 - x).$$

We are seeking a linear combination

$$y_k = \sum_{i=0}^{k} c_{ki} x_i$$

of these approximations that converged faster. Note that if $x_0 = x$, then all subsequent $x_i = x$, so we need to have $x = \sum_{i=0}^{k} c_{ki} x$ or $1 = \sum_{i=0}^{k} c_{ki}$ for all k. Then

$$\begin{aligned} y_k - x &= \sum_{i=0}^{k} c_{ki} x_i - x \\ &= \sum_{i=0}^{k} c_{ki}(x_i - x) \\ &= \sum_{i=0}^{k} c_{ki} R^i (x_0 - x) \\ &= p_k(R)(x_0 - x), \end{aligned}$$

which we denote by (**).

So now our goal is again to find a degree-k polynomial $p_k(R)$ that is small at the eigenvalues of R, but with the slightly different constraint that the sum of its coefficients is 1, i.e. $p_k(1) = 1$. Again, suitably scaled Chebyshev polynomials will serve this purpose. Furthermore, Chebyshev polynomials have one more important property that will make computing y_k cheap: we will only need to keep y_{k-1} and y_{k-2} in memory and operate on them to get y_k, not x_0 through x_k.

16.1 Chebyshev Polynomials

Now we define Chebyshev polynomials, and just the few of their many properties that we will need for our purposes.

Definition 16.1. The m-th Chebyshev polynomial is defined by the 3-term recurrence $T_0(z) = 1$, $T_1(z) = z$, and $T_m(z) = 2zT_{m-1}(z) - T_{m-2}(z)$.

Lemma 16.1

Chebyshev polynomials have the following properties:

(a) $T_m(1) = 1$

(b) $T_m(z) = 2^m z^m + O(z^{m-1})$

(c) $T_m(\cos y) = \cos(my)$, which applies to $T_m(z)$ when $|z| \le 1$, hence $|T_m(z)| \le 1$if $|z| \le 1$.

(d) $T_m(\cosh y) = \cosh(my)$; this applies to $T_m(z)$ when $z \ge 1$, hence $T_m(z) \ge 1$ if $z \ge 1$.

(e) If m is even (odd), then $T_m(z)$ is an even (odd) polynomial.

(f) If $|z| \geq 1$ then $T_m(z) = \frac{1}{2}[(z + \sqrt{z^2-1}))^m + (z + \sqrt{z^2-1}))^{-m}]$

(g) $T_m(1+\varepsilon) \geq \frac{1}{2}\left(1 + m\sqrt{2\varepsilon}\right)$ if $\varepsilon > 0$

16.2 Convergence Analysis of CG

We apply this to understanding the convergence of CG using (*) as follows. We need to pick a polynomial $q_k(z)$ with the properties that $q_k(0) = 1$, and $|q_k(\lambda_i)|$ is small for eigenvalues λ_i of A. Let $0 < \lambda_{\min} \leq \lambda_{\max}$ be the eigenvalues of the spd matrix A, and note that if $\lambda_{\min} \leq z \leq \lambda_{\max}$, then

$$-1 \leq \frac{\lambda_{\max} + \lambda_{\min} - 2z}{\lambda_{\max} - \lambda_{\min}} \leq 1,$$

and so

$$q_k(z) = \frac{T_k\left(\frac{\lambda_{\max}+\lambda_{\min}-2z}{\lambda_{\max}-\lambda_{\min}}\right)}{T_k\left(\frac{\lambda_{\max}+\lambda_{\min}}{\lambda_{\max}-\lambda_{\min}}\right)},$$

satisfies $q_k(0) = 1$ as required, and for $\lambda_{\min} \leq z \leq \lambda_{\max}$, by property (c) of the lemma above, it satisfies

$$\begin{aligned}|q_k(z)| &\leq \frac{1}{T_k\left(\frac{\lambda_{\max}+\lambda_{\min}}{\lambda_{\max}-\lambda_{\min}}\right)}\\ &= \frac{1}{T_k\left(\frac{\kappa+1}{\kappa-1}\right)}\\ &= \frac{1}{T_k\left(1+\frac{2}{\kappa-1}\right)}\\ &\leq \frac{2}{1+\frac{2k}{\sqrt{\kappa-1}}},\end{aligned}$$

where $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ is the condition number of A. Thus we can conclude that $k = O(\sqrt{\kappa})$ steps of CG are enough to decrease the error by a constant factor ¡ 1, and so that $O(\sqrt{\kappa})$ steps of CG are needed to converge to any fixed residual norm $||r||_{A^{-1}}/||b||_{A^{-1}}$. This proves the convergence result for CG claimed earlier.

16.3 Accelerating SOR

Now we consider how to accelerate the convergence of splitting methods like SOR. Now we need to pick a polynomial $p_k(z)$ with the properties that $p_k(z)$ is small for z an eigenvalue of R, and $p_k(1) = 1$. Since we are assuming the splitting method converges, this mean the spectral radius $\rho(R) < 1$, i.e all the eigenvalues of R are less than 1 in absolute value. Since the useful properties of Chebyshev polynomials proven above only apply to real arguments, we need to assume R only has real eigenvalues. Assuming this is true, let ρ satisfy

$$-1 < -\rho \leq \lambda_{\min} \leq \lambda_{\max} \leq \rho < 1$$

where $\lambda_{\min}$ and $\lambda_{\max}$ are the smallest and largest eigenvalues of R. In contrast to the previous analysis of CG, where these eigenvalues were only used for analysis, we actually need their values (or a bound ρ) in order to implement an accelerator. This limits the usability of this approach, but in some cases (eg Poisson equation and variations), these are known (along with faster algorithms, like multigrid). Given ρ, we let

$$p_k(z) = \frac{T_k(z/\rho)}{T_k(1/\rho)}.$$

Clearly $p_k(1) = 1$ as desired, and if $|z| \le \rho$ then $|p_k(z)| \le \frac{1}{T_k(1/\rho)}$.

Assuming we pick ρ as small as possible, so $\rho = \rho(R) = \max_i |\lambda_i(R)|$, and $\rho = 1 - \varepsilon$, then again using property (g) of the Lemma we get

$$\begin{aligned} \frac{1}{T_k(1/\rho)} &= \frac{1}{T_k(1/(1-\varepsilon))} \\ &= \frac{1}{T_k(1+\varepsilon/(1-\varepsilon))} \\ &\le \frac{2}{1+k\sqrt{\frac{2\varepsilon}{1-\varepsilon}}} \\ &\sim 2(1-k\sqrt{2\varepsilon}). \end{aligned}$$

This is to be contrasted with the original splitting method, which after k steps decreases the error by

$$\rho^k = (1-\varepsilon)^k \sim 1 - k\varepsilon.$$

In other words, when ε is small, Chebyshev acceleration can reduce the number of iterations by a square root, an asymptotic improvement. To see how to do this cheaply, we use the 3-term recurrence

$$T_{k+1}(z) = 2zT_k(z) - T_{k-1}(z)$$

defining Chebyshev polynomials, and scale it to get a 3-term recurrence for $p_{k+1}(z)$. Letting $\mu_k = \frac{1}{T_k(1/\rho)}$ for brevity, we get

$$\begin{aligned} p_{k+1}(z) &= mu_{k+1} T_{k+1}(z/\rho) \\ &= \mu_{k+1}(2z/\rho T_k(z/\rho) - T_{k-1}(z/\rho)) \\ &= \mu_{k+1}(2z/\rho p_k(z)/\mu_k - p_{k-1}(z)/\mu_{k-1})) \\ &= \frac{2\mu_{k+1}}{\rho\mu_k} z p_k(z) - \frac{\mu_{k+1}}{\mu_{k-1}} p_{k-1}(z) \\ &= \alpha_k z p_k(z) + \beta_k p_{k-1}(z). \end{aligned}$$

Note that $1/mu_k = T_k(1/\rho)$ also can be computed using a 3-term recurrence, so allowing us to compute α_k and β_k cheaply too. Applying this 3 term recurrence for $p_{k+1}(z)$ to (**) yields

$$\begin{aligned} y_{k+1} - x &= p_{k+1}(R)(x_0 - x) \\ &= \alpha_k R p_k(R)(x_0 - x) + \beta_k p_{k-1}(R)(x_0 - x) \\ &= \alpha_k R(y_k - x) + \beta_k(y_{k-1} - x) \\ &= \alpha_k R y_k + \beta_k y_{k-1} - \alpha_k R x - \beta_k x, \end{aligned}$$

or

$$y_{k+1} = \alpha_k R y_k + \beta_k y_{k-1} - \alpha_k R x - \beta_k x + x.$$

To simplify the last three terms which depend on x, use $x = Rx + c$ to get

$$\begin{aligned} -\alpha_k R x - \beta_k x + x &= -alpha_k(x - c) - \beta_k x + x \\ &= (-\alpha_k - \beta_k + 1)x + \alpha_k c \\ &= \alpha_k c, \end{aligned}$$

which follows from the definitions of α_k and β_k. This yields the final algorithm:

$$y_{k+1} = \alpha_k R y_k + \beta_k y_{k-1} + \alpha_k c$$

whose cost is rough the same as the original splitting method.

Recall that this depended on R having all real eigenvalues, as well as knowing a reasonably tight bound on $\rho(R)$. Let us first try applying this to Jacobian's method for our model problem, Poisson equation on an $N \times N$ mesh. Recall that R_J was symmetric, so all real eigenvalues, and that $\rho(R_J)$ was

$$\cos(\pi/(N+1)) \sim 1 - \frac{\pi^2}{2(N+1)^2} = 1 - O(1/N^2),$$

where N is the number of mesh points in each dimension. This means Jacobian would take $O(N^2)$ iterations to converge versus $O(N)$ for the Chebyshev accelerated version, a big improvement. But we already know that we can use SOR(w_{opt}) to converge in $O(N)$ Iterations, so it is natural to try to applying Chebyshev acceleration to SOR(w_{opt}) to converge in just $O(\sqrt{N})$ iterations. Unfortunately $R_{\text{SOR}}(w)$ has complex eigenvalues, so does not satisfy our assumption. Fortunately there is a clever fix: we apply $R_{\text{SOR}}(w)$ once, updating the entries of x in the usual order, and then apply it again, updating them in the reverse order. This effectively SOR, yielding SSOR(w), and it can be shown that SSOR(w) has all real eigenvalues. The choice of optimal w is no longer the same for SOR(w).

But a good choice is known with

$$\rho(R_{\text{SSOR}}(w)) = 1 - O(1/N),$$

allowing us to use Chebyshev acceleration to converge in $O(\sqrt{N})$ iterations instead of $O(N)$ as desired. For the $N \times N$ Poisson equation, this means $O(N^2\sqrt{N}) = O(n^{5/4})$ flops are needed to converge, where $n = N^2$ is the size of the matrix, versus $O(n^{3/2})$ for SOR(w_{opt}).

17 Lecture 17: Preconditioning to Accelerate Iterative Methods

The last topic in the course is preconditioning, i.e. changing A in a cheap way to make it better conditioned, to accelerate convergence. The simplest idea is to solve $M^{-1}Ax = M^{-1}b$, where M^{-1} is cheap to multiply by, and $M^{-1}A$ is better conditioned than A. Given M, this is straightforward for algorithms like GMRES, but not CG, since $M^{-1}A$ will not generally be spd. But if M is spd, with eigendecomposition $M = Q\Lambda Q^\top$, then we could define

$$M^{1/2} = Q\Lambda^{1/2}Q^\top,$$

and imagine applying CG to the equivalent spd system:

$$M^{-1/2}AM^{-1/2}M^{1/2}x = M^{-1/2}b.$$

Since $M^{-1}A$ and $M^{-1/2}AM^{-1/2}$ are similar, if one is well-conditioned, both are. It turns out that we can apply CG implicitly to this system without needing $M^{1/2}$ or $M^{-1/2}$:

Algorithm 17.1 (Preconditioned CG).

```
$k = 0$, $x_0 = 0$, $r_0 = b$, $p_1 = M^{-1}b$, $y(0) = M^{-1}r(0)$
while $||r_k||_2$ not small enough do
    $k = k + 1$
    $z = Ap_k$
    $\nu_k = \frac{y_{k-1}^\top r_{k-1}}{p_k^\top z}$
    $x_k = x_{k-1} + \nu_k p_k$
    $r_k = r_{k-1} - \nu_k z$
    $y_k = M^{-1}r(k)$
    $\mu_{k+1} = \frac{y_k^\top r_k}{y_{k-1}^\top r_{k-1}}$
    $p_{k+1} = y_k + \mu_{k+1}p_k$
end while
```

Recall that our goal is to choose M so that (1) it is cheap to multiply a vector by M^{-1} and (2) $M^{-1}A$ is (much) better conditioned than A. Obviously choosing $M = I$ satisfies goal (1) but not goal (2), and $M = A$ is the opposite. So there is a wide array of preconditioners M that have been proposed, the best depending very strongly the structure of A. We give some examples below, from simple to more complicated.

17.1 Jacobi preconditioning

If A has widely varying diagonal entries,

$$M = \mathrm{diag}(a_{11}, a_{22}, \cdots, a_{nn})$$

works. This is also called Jacobi preconditioning. Note that $M^{-1}A = I - R_J$, so if the spectral radius $\rho(R_J)$ is small, i.e. Jacobi converges quickly, then $M^{-1}A$ is well-conditioned.

17.2 Block Jacobi preconditioning

More generally, one can take

$$M = \mathrm{diag}(A_{11},\, A_{22},\, \cdots ,\, A_{kk}),$$

where each A_{ii} is a diagonal block of A of some size. Choosing larger blocks makes multiplying by M^{-1} more expensive, but $M^{-1}A$ better conditioned and so convergence is faster. This is called block Jacobi preconditioning. For example, the A_{ii} might be chosen corresponding to different physical regions of a simulation in which different methods may be used to solve each A_{ii} (a similar idea is used in domain decomposition below). Note that by replacing A by $PAP^\top$ where P is a permutation, the blocks A_{ii} can correspond to any subset of rows and columns.

17.3 Incomplete Cholesky and Incomplete LU (ILU)

"Incomplete Cholesky" means computing an approximate factorization of a spd

$$A \sim LL^\top = M,$$

where for example, one might limit L to a particular sparsity pattern to keep it cheap, such as A's original nonzeros, and then multiplying by

$$M^{-1} = (LL^\top)^{-1} = L^{-\top}L^{-1},$$

by doing two (cheap) triangular solves. The same idea for general A is called "incomplete LU", or ILU.

17.4 Multigrid

One or more multigrid V cycles can be used as a preconditioner, given a suitable A (or A_{ii} in (2)).

17.5 Domain Decomposition

Finally, we mention domain decomposition. To motivate, imagine we have a sparse matrix that, like 2D Poisson, has a sparsity structure whose graph is a 2D $n \times n$ mesh. However, suppose the matrix entries corresponding to left half of the mesh are quite different from the right half, perhaps because they correspond to different parts of a physical domain being modeled (eg finite element models of steel and concrete). And the mesh points corresponding to the boundary between these two domains, representing the interactions between steel and concrete, are different again. Suppose we number the mesh points (matrix rows and columns) in the following order: left half (steel), right half (concrete), and interface, yielding the matrix

$$A = \begin{bmatrix} A_{ss} & 0 & A_{si} \\ 0 & A_{cc} & A_{ci} \\ A_{is} & A_{ic} & A_{ii} \end{bmatrix}$$

The subscripts refers to mesh points (rows and columns) in the steel, c in the concrete, and i in the interface. Thus A_{ss} and A_{cc} are square of dimension $n(n-1)/2$ (assuming n is odd) and A_{ii} is square of dimension n, so much smaller. The zero blocks arise because there is no direct connection between steel and concrete, only via the interface. Thus one may factor

$$A = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{is}A_{ss}^{-1} & A_{ii}A_{cc}^{-1} & I \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S \end{bmatrix} \begin{bmatrix} A_{ss} & 0 & A_{si} \\ 0 & A_{cc} & A_{ci} \\ 0 & 0 & I \end{bmatrix}$$

where

$$S = A_{ii} - A_{is}A_{ss}^{-1}A_{si} - A_{ic}A_{cc}^{-1}A_{ci}$$

is the Schur complement. Thus

$$A^{-1} = \begin{bmatrix} A_{Ss}^{-1} & & -A_{ss}^{-1}A_{si} \\ & A_{cc}^{-1} & -A_{cc}^{-1}A_{ci} \\ & & I \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ -A_{is}A_{ss}^{-1} & -A_{ic}A_{cc}^{-1} & I \end{bmatrix}$$

Given this factorization, it is natural to use any available preconditioners for A_{ss} and A_{cc} to approximate multiplication by the submatrices like

$$(-A_{is}A_{ss}^{-1})x = -A_{is}(A_{ss}^{-1}x).$$

S would be expensive to compute, and factorize, even though it is much smaller than A_{ss} or A_{cc}. On the other hand, multiplying (approximately) by S can also take advantage of the ability to multiply (approximately) byA_{ss}^{-1} and A_{cc}^{-1} using their preconditioners. And if we can multiply (approximately) by S, we can multiply by S^{-1} by using one of the other iterative methods already discussed, such as CG. This often works well because S can be much better conditioned than A (condition number growing like $O(N)$ instead of $O(N^2)$ for Poisson).

The above idea generalizes naturally when there are more than 2 domains (like s and c above), and it is called "nonoverlapping" domain decomposition, because the s, c and i domains are disjoint. It is also possible to have "overlapping" domain decomposition, which can lead to faster convergence in some cases. For example, consider the 2D Poisson equation on a domain which is not a rectangle, but two partially overlapping rectangles, say forming an L-shaped domain. Given the availability of fast solvers for rectangular domains, how can we best combine them to solve the problem on overlapping domains? We could obviously use nonoverlapping domain decomposition, treating the L-shaped domains as two rectangles sharing an interface, but it turns out one can converge faster by making one rectangle larger, so it overlaps the other. Intuitively, this increases how fast information can move from one domain to the other, analogous to the motivation for multigrid, addressing the problem of data moving to just one neighboring mesh point per iteration illustrated in Fig 6.9 in the text. To express this mathematically, suppose we number the mesh points just inside one of the rectangles first, then the mesh point inside both, and then the mesh points just inside the second rectangle, yielding the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & 0 \\ A_{21} & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{bmatrix}.$$

In other words,

$$R_1 = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

is the entire first rectangle, and

$$R_2 = \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}$$

is the entire second rectangle. This suggests using the following preconditioner, sometimes also called "additive Schwartz" or "overlapping block Jacobi":

$$M^{-1} = \begin{bmatrix} R_1^{-1} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & R_2^{-1} \end{bmatrix}$$

In words, this means solving (or approximately solving) the two rectangles separately. Just as standard Jacobi can be improved by using the most recent updates (Gauss-Seidel), the same idea can be used here, first updating rectangle 1, and then rectangle 2 (also called "multiplicative Schwartz"). And combining this technique with a multigrid-like idea of using a coarser grid approximation is also possible.