Notes for Ma221 Lecture 1, Aug 29, 2024

Greetings!
   Class URL: people.eecs.berkeley.edu/~demmel/ma221_Fall24
   Prereqs: good linear algebra,
            programming experience in Matlab (other languages welcome,
               but homework assignments will be provided in Matlab)
            numerical sophistication at the level of Ma128a desirable
   Office Hours: see webpage
   First Homework on webpage - due Sep 9 (submit using bcourses)
   Grader - see webpage
   Grading: homework, final project, no exams
   Status of textbook in bookstore - see their webpage
   Fill out on-line survey - what we cover may depend on what you say
      you are interested in
   Lectures will be recorded, and posted on bcourses.
      We will try to post typed course notes before each lecture, and
      a pdf of the "virtual white board" after each on-line class
      meeting. One "lecture" covers one topic, which might take more
      or less than one 80-minute class meeting. You can find all the
      notes and virtual white boards from the last offering in
      Fall 23 at people.eecs.berkeley.edu/~demmel/ma221_Fall23
      In addition, a student generously typed up all the class notes
      from a recent offering in latex. These notes, which may turn
      into a new edition of the textbook, have not been completely
      proofread, so are only posted on bcourses. Suggested changes
      or corrections are welcome, in any of these materials!

A little notation, to simplify discussion:
   ||x||_2 = sqrt(sum_i |x_i|^2) is the 2-norm of vector x
   argmin_x f(x) = value of argument x that minimizes f(x)
   f(n) = O(g(n)) means that |f(n)| <= C*|g(n)| for some C>0 and
          n large enough
   f(n) = Omega(g(n)) means that |f(n)| >= C*|g(n)| for some C>0 and
          n large enough, or g(n) = O(f(n))
   f(n) = Theta(g(n)) means f(n) = O(g(n)) and f(n) = Omega(g(n))

To motivate the syllabus, we describe the "axes" of the design space
of linear algebra algorithms:
1) the mathematical problem you want to solve:
      solve A*x=b, least squares: argmin_x || A*x-b ||_2,
      eigenproblems: A*x = lambda*x, many generalizations ...
2) the structure of A:
      dense, symmetric, positive definite, sparse,
         structured, eg Toeplitz: A(i,j) = x_{i-j}
           (constant along diagonals)
3) the desired accuracy: spectrum ranging over
      guaranteed correct, "guaranteed correct" except in "rare cases",
      "backward stable", residual as small as desired, "probably ok";
      error bounds, ...
4) as fast as possible on target architecture
      your laptop (which may have a GPU), big parallel computer,
      cloud, cell-phone, ...

Depending on which "problem" = combination of 1), 2), 3) and 4) we
want to talk about, the answer might range from
   "type A\b" to "download standard software from this URL" to
   "project available to implement a proposed algorithm"
    to "open problem".
And if we tried to talk about every possible combination of
1), 2), 3) and 4), it would take rather more than 1 semester,
so we will choose a reasonably well-understood and widely-used
subset. But this could be tweaked depending on your survey responses.

Now we explore axis 1) a little more, the mathematical problem:
   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.

   Least squares:
     overdetermined: argmin_x || A*x-b ||_2 when A^{m x n} has
        full column rank; this mean m >= n and the solution is unique
     not full column rank (eg m < n): x not unique, so can pick x that
        minimizes || x ||_2 too, to make it unique.
     ridge regression: argmin_x || A*x-b ||_2^2 + lambda*|| x ||^2
        (also called Tikhonov regularization). This guarantees a
        unique solution when lambda > 0.
     constrained: argmin_{x: C*x=d} || A*x-b ||_2, eg x may represent
        fractions of a population, so we require sum_i x_i = 1
        (additionally constraining x_i >= 0 seems natural too, but
         this is a harder problem)
     weighted: argmin_x || W^{-1}*(A*x-b) ||_2 where W has full column
        rank (also called Gauss-Markov linear model)
     total least squares:
        argmin_{x such that (A+E)*x=b+r} || [E,r] ||_2
        useful when there is uncertainty in A as well as b

   Eigenproblems:
     Notation: If A*x_i = lambda_i * x_i for i=1:n, write
         X = [x_1,...,x_n] and Lambda = diag(lambda_1,...,lambda_n);
         then A*X = 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,
         eg A = [0, 1; 0, 0]. In earlier linear algebra courses, you
         probably studied something called the Jordan form, but we
         will introduce something cheaper and more numerically stable
         later, called Schur form.
     SVD: A = U*Sigma*V^T where
            A^{m x n},
            U^{m x m} and orthogonal (U*U^T = I),
            V^{n x n} and orthogonal,
            Sigma^{m x n} and diagonal, with diagonal entries
               sigma_i called singular values.
          The columns of U and V are called left and right singular
          vectors, resp. Note that
           A*A^T = U*Sigma*V^T*(U*Sigma*V^T)^T
                 = U*Sigma*V^T*V*Sigma^T*U^T
                 = U*(Sigma*Sigma^T)*U^T

```
            = eigendecomposition of A*A^T
      A^T*A = V*(Sigma^T*Sigma)*V^T
            = eigendecomposition of A^T*A
    SVD is the most "reliable" method for least squares, but also
    the most expensive.
Invariant subspaces: If x'(t) = A*x(t), x(0) given, and
    A*x(0) = lambda*x(0), then x(t) = exp(lambda*t)*x(0).
    So it is easy to tell if x(t) -> 0 as t -> infinity: depends
    on whether real(lambda) < 0. And if
        x(0) = sum_i beta_i * x_i, where A*x_i=lambda_i*x_i,
    and x(t) = sum_i beta_i * exp(lambda_i*t) * x_i, then
    x'(t) = sum_i beta_i * lambda_i * exp(lambda_i*t) * x_i
          = A*x(t),
    So whether x(t)->0 depends on whether real(lambda_i) < 0
    for all beta_i neq 0, i.e. whether x(0) is in the subspace
    spanned by eigenvectors x_i with 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).
Generalized eigenproblems: Consider M*x''(t) + K*x(t) = 0.
    For example, 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) = exp(lambda*t)*x(0) and get
    lambda^2*M*x(0) + K*x(0) = 0, which means x(0) is a
    generalized eigenvector, and lambda^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 inv(M)*K.
    Singular M arise in "differential-algebraic systems",
    i.e. ODEs with linear constraints.
Nonlinear eigenproblems: Consider
    M*x''(t) + D*x'(t) + K*x(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) = exp(lambda*t)*x(0) and get
    lambda^2*M*x(0) + lambda*D*x(0) + K*x(0) = 0,
    a nonlinear eigenproblem. We will show how to reduce this
    one to a linear eigenproblem of twice the size.
Singular eigenproblems: Consider the control system:
    x'(t) = A*x(t) + B*u(t), where A^{n x n}
    and B^{n x 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 n x (m+n) matrices [B,A] and [0,I]. Again all
    the above ideas generalize (Jordan becomes Kronecker form).
```

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.

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.

Tensors: Instead of 2-dimensional arrays, i.e. matrices, data often comes in 3-D 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. (We will not have time to consider tensors in this course.)

To explore axis 2), the structure of A, we tell a story about a typical office hours meeting: A student says: "I need to solve an n-by-n linear system Ax=b. What should I do?"

The Professor replies: "The standard algorithm is Gaussian Elimination (GE), which costs (2/3)*n^3 floating point operations (flops)."
Student: "That's too expensive."
Professor: "Tell me more about your problem."
S: "Well, the matrix is real and symmetric, A=A^T."
P: "Anything else?"
S: "Oh yes, it's positive definite, x^T*A*x > 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. (See Table 6.1 in the text, which we will talk about more later.)
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 bw=n^(2/3), so there is a version of Cholesky that only costs O(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 inv(A) once and multiply by it?"
P: "inv(A) will be dense, so just multiplying by it costs 2*n^2 flops, but you can reuse the output of Cholesky (the L factor) to solve for each b in just O(bw*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 the condition number = 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 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 FFT = Fast Fourier Transform 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.

Regarding axis 3), the desired accuracy, there is a range of choices, with a natural tradeoff with speed (more accuracy => slower).

"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.

"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 alg(x) is our computed approximation of f(x), then alg(x) = f(x + delta) where delta is "small" compared to x. The definition of "small" will use matrix norms, introduced later, but it should be proportional to the error in the underlying floating point arithmetic, eg 10^(-16) in double precision (again, more details later).
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 alg(x) is "as good" as any other answer.

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 (eg || A*x-b ||) smaller, until it is good enough for the user; we will see that the residual also estimates the size of delta, i.e. the backward error. There are many such algorithms, see Chaps 6 and 7 of the text.

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 epsilon with probability 1 - delta if the size of the random approximation is big enough, i.e. proportional to a quantity f(delta, epsilon) that gets larger as delta (the probability of failure) and epsilon (the error bound) get smaller." f(delta, epsilon) = Omega(log(1/delta)/epsilon^2) 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.

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 (eg deciding the matrix is (non)singular when the opposite is true). We will see that there is an error bound that is proportional to the "condition number" = 1/distance from A 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.

Finally we consider axis 4), how to implement an efficient algorithm. The story illustrating axis 2) 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) Fewest keystrokes: eg "A\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 (eg netlib, GAMS). A\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.

(2) What does fewest flops (floating point operations) really mean? How many operations does it take to multiply 2 nxn matrices?
Classical: 2*n^3
Strassen (1969): O(n^log2 7) ~ O(n^2.81)
This is sometimes practical, but only for large n, because of the constant factor hidden in the O().
Coppersmith/Winograd (1987): O(n^2.376)
This 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)
Alman & Williams (2020): O(n^2.3728596)
Williams, Xu, Xu,Zhou (2024): O(n^2.371552): World's record so far.
Demmel, Dumitriu, Holtz (2008): all the other standard linear algebra problems (solving Ax=b, eigenvalues, etc.) have algorithms with same complexity as matmul, O(n^x) for some x, (and backward stable) - ideas behind some of these algorithms could be practical.

(3) But counting flops is not the only important metric in today's and the future world, for two reasons:

(3.1) 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 ~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. (Parallel algorithms are discussed in more detail in CS194-15, taught this semester, and in CS267, taught in the spring semester.)

(3.2) 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.

Ex: Consider adding two nxn matrices C=A+B. The cost is n^2 reads of A (moving each A(i,j) 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.

Ex: 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.

Ex: The speed difference between matmul written in the naive way (3 nested loops) vs. optimized to minimize data moves is easily 40x, similarly for other operations (this is again a topic of CS194-15 and CS267)

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). So you (and many others) will be using these new algorithms in the next few years (whether you know it or not :)). This is ongoing work with open problems and possible class projects.

(4) 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 datacenter 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.

To summarize the syllabus of the course:

- Ax=b, least squares, eigenproblems, Singular Value Decomposition (SVD) and variations
  - Direct methods (aka matrix factorizations: LU, Cholesky, QR, Schur form etc.)
    - LU: A = P*L*U, P a permutation, L lower triangular, U upper triangular
    - Cholesky: A = L*L^T, when A is symmetric and positive definite
    - QR: A = Q*R, where Q is orthogonal (columns have unit norm and are orthogonal to one another), R upper triangular
    - Eigenproblems: Not Jordan form (A=V*Lambda*V^{-1}), but Schur form: A=Q*R*Q^T, where Q orthogonal, R upper triangular, eigenvalues on diagonal of R (a little more complicated when a real matrix has complex eigenvalues).
    - SVD: A = U*Sigma*V^T, where Sigma diagonal, U, V orthogonal
  - Iterative methods (eg Jacobi, Gauss-Seidel, Conjugate Gradients, Multigrid, etc.)
    - Common idea: do a bunch of matrix-vector multiplies, find "best" linear combination of resulting vectors that approximate the answer.
  - Randomized algorithms: Approximate A by Q*A or Q*A*V^T where Q and V are "skinny" matrices, often orthogonal

(These examples will motivate spending some time on the properties of orthogonal matrices.)

Some Shared themes:

- exploiting mathematical structure (eg symmetry, sparsity, ...)
- backward stability / condition numbers - how accurate is my answer?
- efficiency (minimizing flops and communication = data movement)
- finding good existing software