Notes for Math 221, Lecture 5, Sep 19, 2024

Last time we spoke about dense matrix multiplication, and how to make it go fast, by doing as little communication as possible, or by doing asymptotically fewer floating point operations too (eg Strassen's method). Our goals for the rest of dense linear algebra (solving Ax=b, least squares, etc) will include these. But we have more goals:

- Backward stability: get the right answer for a slightly different problem A + E where norm(E) = O(macheps) * norm(A)
- Exploit structure: go faster when matrix is symmetric, positive definite, or "sparse". Here sparse means that matrix depends on << n^2 parameters, either because most matrix entries are zero, or if it is dense but the entries depend on few parameters. For example, the Vandermonde matrix V(i,j) = x(i)^(j-1) is dense but depends on just n parameters x(1:n), and there is an O(n^2) algorithm to solve V*y=b (hint: it is just polynomial interpolation in disguise).

One of the linear algebra problems will be computing the SVD A = U*Sigma*V^T, which is one example of a matrix factorization of A into a product of simpler matrices. All the other algorithms we discuss will also explicitly or implicitly compute matrix factorizations:

- Gaussian elimination: A = P*L*U where P is a permutation, L lower triangular, U upper triangular (with variations if A is symmetric, or positive definite too)
- Least Squares: A = Q*R, Q orthogonal, R upper triangular
- Eigenvalue problem: A = Q*T*Q^H where Q unitary, T (nearly) triangular

And there are others, for example for computing modes of vibration (eigenvalues and eigenvectors) of a system with two matrices (mass and stiffness) or three (damping too), or deciding whether a linear control system with feedback can be made stable.

We begin with Gaussian elimination for a general (rectangular) matrix.
Def: Permutation matrix: identity matrix with permuted rows
Facts: let P, P1 etc be permutation matrices

- P*X = row permutation of X and X*P = column permutation of X
- P1*P2 also a permutation matrix
- inv(P) = P^T, i.e. P is orthogonal (enough to check that diag(P*P^T) = all ones)
- det(P) = +-1
- to store and multiply by P, just keep track of locations of ones (cheap)

Thm (LU decomposition): Given any m x n full rank matrix A, with m >= n, there exist

- m x m permutation matrix P
- m x n unit lower triangular matrix L (unit means L(i,i)=1)
- n x n nonsingular upper triangular matrix U

such that A = P*L*U
In the proof below, we will use induction, which is just Gaussian elimination.

Cor: If A n x n and nonsingular, there exist an n x n permutation P,
     unit lower triangular L, and nonsingular upper triangular U
     such that A = P*L*U

To solve A*x = b:
  (1) factor A = P*L*U (expensive part, cost = (2/3)n^3)
  (2) Solve P*L*U*x = b for L*U*x = P^T*b by permuting b
      ... cost = O(n)
  (3) Solve L*U*x = P^T*b for U*x = inv(L)*P^T*b using
      forward substitution with L ... cost = n^2 flops
  (4) Solve U*x = inv(L)*P^T*b for x = inv(U)*inv(L)*P^T*b = inv(A)*b
      using back substitution with U ... cost = n^2 flops
If we are given another b' vector, we can solve A*x'=b' in 2n^2 more
flops.

Note: We do not compute inv(A) and multiply x = inv(A)*b because it is
(1) 3x more expensive in dense case, can be much worse in sparse case
    (O(n^2) times!)
(2) not as numerically stable

Proof of Theorem:
  If A is full rank, the first column is nonzero, so there is a
  permutation P such that (P*A)(1,1) is nonzero. Write
   P*A = [A11 A12] = [1   0] * [A11 A12   ]
         [A21 A22]   [A21/A11 I]   [0 A22 -A21*A12/A11]
  where A11 is 1 x 1, A21 is (m-1) x 1, A12 is 1 x (n-1) and
  A22 is (m-1) x (n-1).
  Now A full (column) rank => P*A full rank => S = A22 - A21*A12/A11
  is full rank. (Otherwise, if some nonzero linear combination of
  columns of S were 0, say S*x=0, then a linear combination of columns
  of A would be zero, A*[-A12*x/A11 ; x] = 0, contradicting A being
  full column rank.) More simply in the square case:
     det(A) nonzero => 0 neq det(first factor) * det(second factor)
                             = 1 * A11 * det(S)   => det(S) nonzero)
  Notation: S called Schur complement
  Now apply induction: S = P'*L'*U', so
  P*A = [1   0] * [A11 A12   ]
        [A21/A11 I]   [0 P'*L'*U']
      = [1   0   ] * [A11 A12]
        [A21/A11 P'*L']   [0 U']
      = [1 0] * [1   0] * [A11 A12]
        [0 P']   [P'^T*(A21/A11) L']   [0 U']
      = P''   *   L   *   U
      = permutation * unit_lower_triangular * upper_triangular
 where U is nonsingular.
 So A = P^T*P'' * L * U = permutation * L * U   QED.

Expressing this as an algorithm, we get the following (ignore
permutations first)
 for i = 1 to n
   ... i = 1 performs algebra shown above,
   ... i>1 applies the same algorithm recursively to the Schur
   ...   complement

```
   L(i,i) = 1, L(i+1:n,i) = A(i+1:n,i)/A(i,i)
   U(i,i:n) = A(i,i:n)
   if (i < n)  A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - L(i+1:n,i)*U(i,i+1:n)

Add permutations: after "for i=1 to n", add:
      if A(i,i) zero and A(j,i) nonzero, swap rows i and j of L and A;
          record swap
        ... choosing A(j,i) is called "pivoting", more on this below

Don't waste space:
   row i of U overwrites row of i 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
  for i=1 to n-1
    if A(i,i) zero and A(j,i) nonzero, swap rows i and j of 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 BLAS2 ger
(draw picture of intermediate step)

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
   for i=1 to n-1 ... for each column i
     for j=i+1 to n ... add a multiple of row i to row j to zero out
                    ... entry (j,i) below diagonal
        m = A(j,i)/A(i,i)
        A(j,i:n) = A(j,i:n) - m*A(i,i:n)
"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:
      for i=1 to n-1
        for j=i+1 to n
           A(j,i) = A(j,i)/A(i,i)
        for j=i+1 to n
           A(j,i+1:n) = A(j,i+1:n) - A(j,i)*A(i,i+1:n)
  (3) combine loops into single expressions to get same as before:
      for i=1 to n-1
        A(i+1:n,i) = A(i+1:n,i)/A(i,i)
        A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i)*A(i,i+1:n)
```

The cost is sum_{i=1 to 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. P*L*U = A + E where E is small compared to A.
For example, if we run in single precision, so with 7 decimal digits, and take

```
      A = [ 1e-8   1 ],  inv(A) ~ [ -1   1  ] ,
          [    1   1 ]            [  1 -1e-8]
```

then cond(A) ~ 2.6, really small, so we expect a good answer. But

```
      L = [    1   0 ]   U = [ 1e-8  1                 ]
          [  1e8   1 ]       [  0  fl(1-1e8*1) = -1e8  ]
     and so L*U = [  1e-8   1 ]
                  [   1     0 ]
```

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 fl(A(2,2)-1e8*1) "forgets" A(2,2) if A(2,2) is small enough, say O(1). So going on to solve A*x=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" A11, 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 = fl(A*B) = A*B + E where
|E| <= n*macheps*|A|*|B|.
Since A = P*L*U in exact arithmetic, it is perhaps no surprise that

Thm (backward error analysis of LU decomposition):
If P, L and are computed by the above algorithm, then
  A - E = P*L*U where |E| <= n*macheps*P*|L|*|U|.

Before proving this, we state and prove a corollary, bounding the backward error of solving A*x=b:

Corollary: Suppose we solve A*x=b by Gaussian elimination, following by forward and back substitution with L and U as described above. Then the computed results xhat satisfies
    (A-F)*xhat = b where |F| <= 3*n*macheps*P*|L|*|U|.

Proof: Here, and in the proof of the Thm, we assume P=I for simplicity of notation; just imagine running without pivoting on the matrix P^T*A. In HW 1.11, you showed that the computed solution yhat of L*y=b satisfied
  (L+dL)*yhat = b with |dL| <= n*macheps*|L|,
and that the computed solution xhat of U*x=yhat satisfied
  (U+dU)*xhat = yhat with |dU| <= n*macheps*|U|.
Thus

```
   b = (L+dL)*yhat = (L+dL)*(U+dU)*xhat
     = (L*U + dL*U + L*dU + dL*dU)*xhat
     = (A - E + dL*U + L*dU + dL*dU)*xhat    by the Thm
     = (A - F)*xhat
```

where F = E - dL*U - L*dU - dL*dU
so |F| <= |E| + |dL*U| + |L*dU| + |dL*dU|
       <= |E| + |dL|*|U| + |L|*|dU| + |dL|*|dU|
       <= n*macheps*|L|*|U| ... by the Thm
          + n*macheps*|L|*|U| ... from the bound on |dL|
          + n*macheps*|L|*|U| ... from the bound on |dU|
          + (n*macheps)^2*|L|*|U| ... from the bounds on |dL| and |dU|
       = (3*n*macheps + O(macheps^2))*|L|*|U|

What the Thm (proven below) and Corollary tell us is that we need
   norm(F) <= 3*norm(macheps*|L|*|U|) = O(macheps*norm(A))
for the algorithm to be backward stable, i.e.
   norm(|L|*|U|) = O(norm(A)).
This in turn depends on pivoting carefully.

Proof of Thm: 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 <= j and repeatedly subtract L(i,k)*U(k,j) for k=1,2,...,i-1 until we get
   U(i,j) = A(i,j) - sum_{k=1 to i-1} L(i,k)*U(k,j)
which, not surprisingly, is what you get when you take the (i,j) entry of A = L*U 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,...,j-1
  dividing the resulting sum by U(j,j)
or
  L(i,j) = (A(i,j) - sum_{k=1 to j-1} L(i,k)*U(i,j))/U(j,j)
so again we have a dot product, followed by a division, and a similar approach works.

Now we discuss how to pivot so that norm(|L|*|U|) = O(norm(A)), or not much larger.
Def: we call g = norm(|L|*|U|)/norm(A) the pivot growth factor.
(Note: this is defined somewhat differently in the literature, but is very similar).

The unstable 2x2 example above, where L(2,1) = 1/1e-8 = 1e8, suggests that we choose the pivot A11 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 A11 is the largest (in absolute value) entry in column
    Then L21 = A21/A11 has |L21| <= 1
    Theorem (easy): with GEPP, |L| <= 1 and
       max(|U(:,i)|) <= 2^(n-1)*max(|A(:,i)|)
    Bad news: worst case is terrible; 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: gPP < n^(2/3)
    If all entries of matrix were "random", this would be true; as you
      perform pivoting, they seem to get more random
(2) Complete pivoting (GECP): permute rows and columns so that A11
    largest entry in the whole matrix; again repeat at every step.
    Get A = P_r*L*U*P_c where P_r and P_c are both permutations.
    Theorem: gCP < n^(log n /4)
    Empirical: gCP < n^(1/2)
    Long-standing Conjecture: gCP < n (false, but nearly true)
    More expensive, hardly used, not in most libraries
(3) Tournament pivoting - something new, needed to minimize
    communication (#messages): will present it later.
(4) Threshold pivoting: this and similar schemes try to preserve
    sparsity while maintaining stability
Figures 2.1 and 2.2 in the book show empirical results that
demonstrate that these bounds for GEPP and GECP are "worst case", and
pessimistic in general.

Altogether, our worst-case error bound is
   norm(x-xhat)/norm(x) <= cond(A)*(backward error)
                        <= cond(A)*3*n*macheps*pivot_growth
where we can estimate cond(A) with O(n^2) flops after LU
decomposition, and can bound pivot growth in O(n^2) work too.

What if this error is too large for your application, or too slow?
We can run iterative refinement, aka 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:

```
  Do GEPP to solve Ax=b in low precision, call solution x(1)
     ... this is the O(n^3) part
  i = 1
  repeat
    r = A*x(i) - b     ... in high precision, but costs just O(n^2);
                       ... round final result r to low precision
    solve A*d = r      ... in low precision using existing LU
                           factors, costs just O(n^2)
    update x(i+1) = x(i) - d ... in low precision, costs O(n)
  until "convergence"
```

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:
     norm(A*x_computed - b) = O(macheps_high)*norm(A)*norm(x_computed)
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 the links to "A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic" on the class webpage, or the recent Nvidia blog post.
(The algorithm used, GMRES, will be discussed in Chap. 6.)

(2) Getting a small relative error in low precision:
      norm(x_computed - x_true)/norm(x_true) = O(macheps_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; for details see www.netlib.org/lapack/lawnspdf/lawn165.pdf. Slides 33-35 of www.cs.berkeley.edu/~demmel/Future_Sca-LAPACK_v7.ppt show empirical results for millions of randomly generated test cases, using single/double, with condition numbers ranging from 1 to well beyond 1/macheps_low: The relative error of LU without refinement is indeed usually close to condition_number * macheps_low, but LU with refinement is much better: the relative error is O(macheps_low) as long as the condition number is less than about 1/macheps_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.

Now we return to minimizing communication. Historically, GEPP was rewritten to do most of its work by calling the matrix multiplication routine in the BLAS3, 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

```
  A = [ A11, A12 ]
      [ A21, A22 ]
... where A11 is b-by-b, A21 (n-b)-by-b, A12 b-by-(n-b) and
...       A22 (n-b)-by-(n-b)
    = [ L11*U11,  A12 ]
      [ L21*U11,  A22 ]
... where we have performed LU decomposition on the n-by-b matrix
...     [ A11 ] = [ L11 ] * U11
...     [ A21 ]   [ L12 ]
...     by using the previous "unblocked" algorithm
    = [ L11*U11,  L11*U12 ]
      [ L21*U11,  A22     ]
... where we have solved the triangular system of equations
... A12 = L11*U12 for U12 by calling BLAS3 routine TRSM
    = [ L11,  0 ] * [ U11, U12          ]
      [ L21,  I ]   [ 0  , A22 - L21*U12]
... where we compute A22hat = A22 - L21*U12 by calling BLAS3 GEMM
```

The inductive step then applies the same procedure to A22hat.
While this often works well in practice, since most of the work is
done by the BLAS3, 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 matmul that
minimized communication without explicitly depending on the
cache size M, we can do the same for LU (due to Toledo, 1997):

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"

```
function [L,U] = RLU(A) ... RLU = Recursive LU
 ... assume A is n by m with n >= m, m a power of 2
 if m=1 ... one column
   pivot so largest entry on diagonal, update rest of matrix
   L = A/A11, U = A11
 else
   ... write A = [ A11 A12 ], L1 = [ L11 ]
   ...           [ A21 A22 ]       [ L12 ]
   ... where A11, A12, L11, U1 and U2 are m/2 by m/2
   ...       A21, A22 and L12 are n-m/2 by m/2
   [L1,U1] = RLU([ A11 ]) ... LU of left half of A
                 [ A21 ]
   Solve A12=L11*U12    ... update U, in upper part of right half of A
   A22 = A22 - L21*U12  ... update Schur complement
   [L2,U2] = RLU( A22 } ... LU on Schur complement
   L = [ L1, [0;L2] ]   ... return complete nxm L factor
   U = [ U1  U12 ]      ... return complete mxm U factor
       [  0  U2  ]
```

Correctness follows by induction on m, as does showing when m=n:
  A(n) = #arithmetic operations = (2/3)n^3 + O(n^2), and
  W(n) = #words moved = O(n^3/sqrt(M))
As stated, this algorithm only minimizes the #words moved, not the
#messages. To minimize #messages, we either need to use a different
pivoting scheme (tournament pivoting, to be discussed later, see
"CALU: A communication optimal LU Factorization Algorithm")
or a more complicated data structure, (see
"Communication efficient Gaussian elimination with Partial Pivoting
using a Shape Morphing Data Layout");
both papers are at bebop.cs.berkeley.edu.

Fact: if we do L21*U12 by Strassen, and L11\A12 by
     (1) inverting L11 by divide-and-conquer:

$$\begin{bmatrix} T11 & T12 \\ 0 & T22 \end{bmatrix}^{-1} = \begin{bmatrix} inv(T11) & -inv(T11)*T12*inv(T22) \\ 0 & inv(T22) \end{bmatrix}$$

     (2) multiplying inv(L11)*A12 etc by Strassen
then the RLU algorithm costs O(n^(log2 7)) like Strassen, but can be
slightly less stable than usual O(n^3) version of GEPP

(see arxiv.org/abs/math.NA/0612264).

Where to find all this implemented (all "blocked" algorithms, unless marked as "recursive"):

- Matlab: A\b or [P,L,U]=lu(A), rcond or condest
- LAPACK:
  - xGETRF just for GEPP where x = S/D/C/Z
  - xGETRF2 recursive
  - xGESV to solve A*x=b
  - xGESVX for condition estimation, iterative refinement with no extra precision
  - xGESVXX for condition estimation, iterative refinement with extra precision
  - xGECON for condition estimation alone
- ScaLAPACK: PxGETRF etc