Week 4: Wednesday, Sep 12

Basic LU factorization

Last time, we wrote Gaussian elimination in terms of a sequence *Gauss transformations* $M_j = I - \tau_j e_j^T$, where τ_j is the vector of multipliers that appear when eliminating in column j:

```
%
% Overwrites A with an upper triangular factor U, keeping track of
% multipliers in the matrix L.
%
function [L,A] = mylu(A)

  n = length(A);
  L = eye(n);
  for j=1:n-1

    % Form vector of multipliers
    L(j+1:n,j) = A(j+1:n,j)/A(j,j);

    % Apply Gauss transformation
    A(j+1:n,j) = 0;
    A(j+1:n,j+1:n) = A(j+1:n,j+1:n)-L(j+1:n,j)*A(j,j+1:n);

  end
```

Now, observe that at each step, the locations where we write the multipliers in L are exactly the same locations where we introduce zeros in A! Thus, we can re-use the storage space for A to store L (except for the diagonal ones, which are implicit) and U. Using this strategy, we have the following code:

```
%
% Overwrite A with L and U factors
%
function [A] = mylu(A)
  n = length(A);
  for j=1:n-1
```

```
    A(j+1:n,j) = A(j+1:n,j)/A(j,j);
    A(j+1:n,j+1:n) = A(j+1:n,j+1:n) - A(j+1:n,j)*A(j,j+1:n);
  end
```

If we wanted to extract the L and U factors explicitly, we could then do

```
  LU = mylu(A);
  L = eye(length(A)) + tril(A,-1);
  U = triu(A);
```

The bulk of the work at step j of the elimination algorithm is in the computation of a rank-one update to the trailing submatrix. How much work is there in total? In eliminating column j, we do $(n-j)^2$ multiplies and the same number of subtractions; so in all, the number of multiplies (and adds) is

$$\sum_{j=1}^{n-1} (n-j)^2 = \sum_{k=1}^{n-1} k^2 = \frac{1}{6}n^3 + O(n^2)$$

We also perform $O(n^2)$ divisions. Thus, Gaussian elimination, like matrix multiplication, is an $O(n^3)$ algorithm operating on $O(n^2)$ data.

Schur complements

The idea of expressing a step of Gaussian elimination as a low-rank submatrix update turns out to be sufficiently useful that we give it a name. At any given step of Gaussian elimination, the trailing submatrix is called a *Schur complement*. We investigate the structure of the Schur complements by looking at an LU factorization in block 2-by-2 form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{22}U_{22} + L_{21}U_{12} \end{bmatrix}.$$

We can compute L_{11} and U_{11} as LU factors of the leading sub-block A_{11}, and

$$\begin{aligned} U_{12} &= L_{11}^{-1} A_{12} \\ L_{21} &= A_{21} U_{11}^{-1}. \end{aligned}$$

What about L_{22} and U_{22}? We have

$$\begin{aligned} L_{22}U_{22} &= A_{22} - L_{21}U_{12} \\ &= A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12} \\ &= A_{22} - A_{21}A_{22}^{-1}A_{12}. \end{aligned}$$

This matrix $S = A_{22} - A_{21}A_{22}^{-1}A_{12}$ is the block analogue of the rank-1 update computed in the first step of the standard Gaussian elimination algorithm.

For our purposes, the idea of a Schur complement is important because it will allow us to write blocked variants of Gaussian elimination — an idea we will take up in more detail now.

Block Gaussian elimination

Just as we could rewrite matrix multiplication in block form, we can also rewrite Gaussian elimination in block form. For example, if we want

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

then we can write Gaussian elimination as:

1. Factor $A_{11} = L_{11}U_{11}$.
2. Compute $L_{21} = A_{21}U_{11}^{-1}$ and $U_{12} = L_{11}^{-1}A_{12}$.
3. Form the Schur complement $S = A_{22} - L_{21}U_{12}$ and factor $L_{22}U_{22} = S$.

This same idea works for more than a block 2-by-2 matrix. Suppose `idx` is a MATLAB vector that indicates the first index in each block of variables, so that block A_{IJ} is extracted as

```
I = idx(i):idx(i+1)-1;
J = idx(j):idx(j+1)-1;
A_IJ = A(idx(i):idx(i+1)-1, idx(j):idx(j+1)-1);
```

Then we can write the following code for block LU factorization:

```
M = length(idx)-1;          % Number of blocks
for j = 1:M
  J    = idx(j):idx(j+1)-1; % Indices for block J
  rest = idx(j+1):n;        % Indices after block J
  A(J,J) = lu(A(J,J));      % Factor submatrix A_JJ

  % Extract L and U (this could be implicit)
  L_JJ   = tril(A(J,J),-1) + eye(length(J));
```

```
    U_JJ   = triu(A(J,J));

    % Compute block column of L and row of U, Schur complement
    A(rest,J) = A(rest,J)/U_JJ;
    A(J,rest) = L_JJ\A(J,rest);
    A(rest,rest) = A(rest,rest)-A(rest,J)*A(J,rest);
  end
```

As with matrix multiply, thinking about Gaussian elimination in this blocky form lets us derive variants that have better cache efficiency. Notice that all the operations in this blocked code involve matrix-matrix multiplies and multiple back solves with the same matrix. These routines can be written in a cache-efficient way, since they do many floating point operations relative to the total amount of data involved.

Though some of you might make use of cache blocking ideas in your own work, most of you will never try to write a cache-efficient Gaussian elimination routine of your own. The routines in LAPACK and MATLAB(really the same routines) are plenty efficient, so you would most likely turn to them. Still, it is worth knowing how to think about block Gaussian elimination, because sometimes the ideas can be specialized to build fast solvers for linear systems when there are fast solvers for sub-matrices

For example, consider the *bordered* matrix

$$A = \begin{bmatrix} B & W \\ V^T & C \end{bmatrix},$$

where B is an n-by-n matrix for which we have a fast solver and C is a p-by-p matrix, $p \ll n$. We can factor A into a product of *block* lower and upper triangular factors with a simple form:

$$\begin{bmatrix} B & W \\ V^T & C \end{bmatrix} = \begin{bmatrix} B & 0 \\ V^T & L_{22} \end{bmatrix} \begin{bmatrix} I & B^{-1}W \\ 0 & U_{22} \end{bmatrix}$$

where $L_{22}U_{22} = C - V^T B^{-1} W$ is an ordinary (small) factorization of the trailing Schur complement. To solve the linear system

$$\begin{bmatrix} B & W \\ V^T & C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix},$$

we would then run block forward and backward substitution:

$$\begin{aligned} y_1 &= B^{-1} b_1 \\ y_2 &= L_{22}^{-1} \left(b_2 - V^T y_1 \right) \end{aligned}$$

$$\begin{aligned} x_2 &= U_{22}^{-1} y_2 \\ x_1 &= y_1 - B^{-1}(W x_2) \end{aligned}$$

Perturbation theory

Recall the general strategy for error analysis that we described earlier: we want to derive forward error bounds by combining a sensitivity estimate (in terms of a *condition number*) with a *backward* error analysis that explains the computed result as the exact answer to a slightly erroneous problem. We will follow that strategy here, so it will be useful to work through the sensitivity analysis of solving linear systems.

Suppose that $Ax = b$ and that $\hat{A}\hat{x} = \hat{b}$, where $\hat{A} = A + \delta A$, $\hat{b} = b + \delta b$, and $\hat{x} = x + \delta x$. Then

$$\delta A \, x + A \, \delta x + \delta A \, \delta x = \delta b.$$

Assuming the delta terms are small, we have the linear approximation

$$\delta A \, x + A \, \delta x \approx \delta b.$$

We can use this to get δx alone:

$$\delta x \approx A^{-1}(\delta b - \delta A \, x);$$

and taking norms gives us

$$\|\delta x\| \lesssim \|A^{-1}\|(\|\delta b\| + \|\delta A\|\|x\|).$$

Now, divide through by $\|x\|$ to get the relative error in x:

$$\frac{\|\delta x\|}{\|x\|} \lesssim \|A\|\|A^{-1}\| \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\|\|x\|} \right).$$

Recall that $\|b\| \leq \|A\|\|x\|$ to arrive at

$$\frac{\|\delta x\|}{\|x\|} \lesssim \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right),$$

where $\kappa(A) = \|A\|\|A^{-1}\|$. That is, the relative error in x is (to first order) bounded by the condition number times the relative errors in A and b.

Residual good!

The analysis in the previous section is pessimistic in that it gives us the worst-case error in δx for *any* δA and δb. But what if we are given data that behaves better than the worst case?

If we know A and b, a reasonable way to evaluate an approximate solution $\hat{x}$ is through the residual $r = b - A\hat{x}$. The approximate solution satisfies

$$A\hat{x} = b + r,$$

so if we subtract of $Ax = b$, we have

$$\hat{x} - x = A^{-1}r.$$

We can use this to get the error estimate

$$\|\hat{x} - x\| = \|A^{-1}\|\|r\|;$$

but for a given $\hat{x}$, we also actually have a prayer of *evaluating* $\delta x = A^{-1}r$ with at least some accuracy. It's worth pausing to think how novel this situation is. Generally, we can only *bound* error terms. If I tell you "my answer is off by just about 2.5," you'll look at me much more sceptically than if I tell you "my answer is off by no more than 2.5," and reasonably so. After all, if I knew that my answer was off by nearly 2.5, why wouldn't I add 2.5 to my original answer in order to get something closer to truth? This is exactly the idea behind *iterative refinement*:

1. Get an approximate solution $A\hat{x}_1 \approx b$.
2. Compute the residual $r = b - A\hat{x}_1$ (to good accuracy).
3. Approximately solve $A\,\delta x_1 \approx r$.
4. Get a new approximate solution $\hat{x}_2 = \hat{x}_1 + \delta x_1$; repeat as needed.