## Week 4: Friday, Sep 18

# In-place Gaussian elimination

Last time, we described how the $L$ factor in Gaussian elimination is formed from the multipliers that appear during the algorithm. We can write the reduction to upper triangular form as

```
for j = 1:n-1
  % Subtract multiples lj of A(j,:) from rows that follow
  lj         = A(j+1:n,j)/A(j,j);
  A(j+1:n,:) = A(j+1:n,:)-lj*A(j,:);
end
```

Here `lj` represents the subdiagonal elements in the $j$th column of $L$. But notice that at step $j$, we no longer really need the storage for the subdiagonal elements of the transformed matrix $A$, as those subdiagonal elements are to be immediately eliminated. Therefore, we can use that storage for elements of $L$:

```
for j = 1:n-1
  % Compute a column of L, then update trailing submatrix
  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
```

At the end of this latter code, the subdiagonal elements of the rewritten $A$ correspond to entries of $L$, and the superdiagonal elements correspond to entries of $U$.

# Triangular solves

Once we have factored $A = LU$, we can solve linear systems $Ax = b$ by first solving $Ly = b$ (forward substitution), then $Ux = y$ (backward substitution). We would typically think of this as first computing $y_1, \ldots, y_n$ in order:

```
for i = 1:n
  y(i) = b(i)-L(i,1:i-1)*y(1:i-1);
end
```

and then computing $x_n, x_{n-1}, \ldots, x_1$ in reverse order:

```
for i = n:-1:1
  x(n) = ( y(i)-U(i,i+1:n)*x(i) )/U(n,n);
end
```

We can also use the same storage repeatedly, incrementally replacing $b$ with $y$ and then with $x$. If we use the packed storage for the $LU$ factors as well, we have

```
% Replace b with x solving Ax = b.
% Assume the matrix A stores L and U factors

for i = 1:n
  b(i) = b(i)-A(i,1:i-1)*b(1:i-1);
end
for i = n:-1:1
  b(i) = ( b(i)-A(i,i+1:n)*b(i+1:n) )/A(i,i);
end
```

This is a *row-oriented* version of the forward and backward substitution. We could also think of a *column-oriented* version. For example, rather than subtracting $L_{ij}b_j$ from $b_i$ as we process row $i$, instead do the subtraction immediately after processing row $j$:

```
% Replace b with x solving Ax = b.
% Assume the matrix A stores L and U factors

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

We might prefer the column-oriented version in languages like MATLAB or Fortran that use column-major storage formats, because it accesses the matrix elements in the order in which they are stored in memory, and stride 1 access is fast.

# 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.