Section 1.7: Recall from last time the matrix

\[
L = \begin{bmatrix}
1 & 0 & 0 & \ldots & 0 \\
l_{21} & 1 & 0 & \ldots & 0 \\
l_{31} & l_{32} & 1 & \ldots & 0 \\
\vdots \\
l_{n1} & l_{n2} & l_{n3} & \ldots & 1 \\
\end{bmatrix}
\]

and let \( L_i \) be the lower triangular matrix with ones on the main diagonal, and the only other nonzero entries in the column \( i \), under the diagonal:

\[
L_i = \begin{bmatrix}
1 & 0 & \ldots & 0 & 0 & \ldots & 0 \\
0 & 1 & \ldots & 0 & 0 & \ldots & 0 \\
\vdots \\
0 & 0 & \ldots & 1 & 0 & \ldots & 0 \\
0 & 0 & \ldots & l_{(i+1)i} & 0 & \ldots & 0 \\
\vdots \\
0 & 0 & \ldots & l_{ni} & \ldots & 0 \\
\end{bmatrix}
\]

for \( 1 \leq i \leq (n-1) \).

If we denote by \( U_1 \) the matrix after the first iteration of the code has been completed (and zeros have been introduced in the first column, below the diagonal), \( U_i \) the matrix after the \( i \)th iteration has been completed (and now there are zeros under the diagonal in the 1st through \( i \)th columns), it is not too hard to see that

\[
A = L_1 U_1 = L_1 L_2 U_2 = \ldots = L_1 L_2 \ldots L_{n-1} U_{n-1}.
\]

We have shown the first equality last time, and the rest follow inductively.

What is less obvious, but not too hard to show, is that \( L_1 L_2 \ldots L_{n-1} = L \). You will have to do this in HW 2 for \( n = 4 \).

**Theorem 1.** If \( A \) is non-singular (that is, invertible) and if all of its principal submatrices are non-singular, one may factor \( A = LU \) with \( L \) a “unit” lower triangular matrix (that is, it has 1s on the diagonal) and \( U \) an upper triangular matrix. Moreover, this factorization is unique.

**Remark 1.** The condition that all of the principal submatrices of \( A \) are non-singular is equivalent to saying that there will be no error message when the code `ge_solve` is called with \( A \) as an input (none of the \( A(i, i) \) will be 0).
Remark 2. As part of HW 2, you will have to show this uniqueness; some results you will need are the fact that the inverse of a lower triangular matrix is lower triangular, similarly, the inverse of an upper triangular matrix is upper triangular (which you can get by transposition).

So now we are in a position to talk about the LU factorization for a matrix $A$; if the factorization exists and $A$ is non-singular, then the factorization is unique. Below is a simple tweak to `ge.solve`, which allows us to calculate $L$ and $U$:

```matlab
function [L, U] = lu_factor(A);

n = size(A,1); L = eye(n);
for i = 1:n
    if A(i,i) = 0, error('simple GE will not work on this matrix'), end
    for j = (i+1):n
        L(j,i) = A(j,i)/A(i,i);
        for k = (i+1):n
            A(j,k) = A(j,k) - L(j,i)*A(i,k);
        end
    end
    A(j,i) = 0;
end

U = A;
```

Flop (“floating point operation”) count. The third for loop performs 2 operations each time it is called, for a total $2(n-i)$ operations. To this we add another 1 inside the second for loop (the calculation of $L(j,i)$). The rest is an assignment that is not counted. We run the second for loop $(n-i)$ times, for a total of $(n-i)(2(n-i) + 1)$ operations, and we sum this over $i$ with $i$ from 1 through $n$:

$$
\sum_{i=1}^{n}(n-i)(2(n-i) + 1) = \sum_{i=1}^{n}(2(n-i)^2 + (n-i)) ,
$$

then we switch indexing to $j = n - i$ to obtain

$$
\sum_{j=0}^{n-1}2j^2 + j = 2 \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} = O\left(\frac{2}{3}n^3\right) + O(n^2) .
$$

The LDV factorization. If a matrix has a unique LU factorization, it has a unique LDV factorization, where $D$ is a diagonal matrix, $L$ is a unit lower triangular matrix, and $V$ is a unit upper triangular matrix. This can be seen immediately since in this case $DV = U$ and so $D$ is the matrix whose diagonal consists of $U(1,1)$ through $U(n,n)$, and $V = D^{-1}U$.

1.8: When $A(i,i)$ is 0 (or small). So far we have no at all addressed the obvious issue of what to do when one of the diagonal entries is 0 (or, given that we will work in floating point arithmetic, close to 0). This may and will happen.

The answer will be to employ one of the other elementary operations to help us out: swapping rows.

First a bit of notation. Following MATLAB, we will denote the following submatrices as follows
\( A(i : j, k : l) = \begin{bmatrix}
  a_{ik} & \cdots & a_{il} \\
  \vdots & \ddots & \vdots \\
  a_{jk} & \cdots & a_{jl}
\end{bmatrix}, \)

\( A(i : j,:) = A(i : j, 1 : n) = \begin{bmatrix}
  a_{i1} & \cdots & a_{in} \\
  \vdots & \ddots & \vdots \\
  a_{j1} & \cdots & a_{jn}
\end{bmatrix}, \)

\( A(i, j : k) = A(i, j : k) = \begin{bmatrix}
  a_{ij} & \cdots & a_{ik}
\end{bmatrix}. \)

To quickly outline the method, each time we iterate through the first for loop, we will search for the largest entry in \( A(i : n, i) \), swap the \( i \)th row with the row in which that largest entry is, and continue with the rest of the code, recording all the swaps. The result will be a three-pronged output, \( L, U, P \) such that \( LU = PA \), where \( P \) will be a permutation matrix. More on this next time.

**Motivation.** Why do we swap rows to make sure the largest entry is the one that becomes \( A(i, i) \)? The answer is that when we divide by small quantities (not necessarily non-zero), we incur significant errors, and these errors accumulate. As such, we will want to divide by the largest quantities possible, in order to minimize errors; and since we can keep track of row swaps and do almost no extra work, there is no downside to swapping rows to keep things as accurate as possible in a floating point computing environment. And so we shall, as will be explained next time.