Lecture `linalg3`: LU–factorization

September 30, 2022

Summary: Description of the basic algorithm without pivoting.

References: Taken in part from Sections 2.2 and 2.3, pages 84-89, of C. F. Van Loan (*Introduction to Scientific Computing, a Matrix–Vector Approach Using* MATLAB, second edition).

Gaussian elimination without pivoting

Basic idea

Let's start with a general n–by–n matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \cdots & a_{3n} \\ a_{41} & a_{42} & a_{43} & a_{44} & \cdots & a_{4n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{pmatrix}. \tag{1}$$

Our goal is to turn A into an upper triangular matrix U, working column–by–column (this is Gaussian elimination which we earlier discussed in the context of augmented matrices and row operations, see Lecture `linalg1`). So our first task is to set to zero all of the first–column entries below a_{11}. To achieve this task, we form the matrix

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -a_{21}/a_{11} & 1 & 0 & 0 & \cdots & 0 \\ -a_{31}/a_{11} & 0 & 1 & 0 & \cdots & 0 \\ -a_{41}/a_{11} & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ -a_{n1}/a_{11} & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -v_2 & 1 & 0 & 0 & \cdots & 0 \\ -v_3 & 0 & 1 & 0 & \cdots & 0 \\ -v_4 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ -v_n & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}. \tag{2}$$

Notice that we can't form M_1 if $a_{11} = 0$. If $a_{11} = 0$, we could get around this obstruction by first performing a row exchange on A. Indeed, if a_{k1} is the entry in A's first column with the largest magnitude $|a_{k1}|$, then we could exchange row k with row 1, and afterward form M_1. Such a row exchange is called a *pivot*; however, let's ignore this complication for the time being. Although we don't need it right now, for later purposes we note that

$$M_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ v_2 & 1 & 0 & 0 & \cdots & 0 \\ v_3 & 0 & 1 & 0 & \cdots & 0 \\ v_4 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ v_n & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}. \tag{3}$$

It's easy to check that this is the correct inverse of M_1. Make sure you do check! Now, M_1 has been tailored to zero out all first–column entries of A below a_{11}, that is to yield

$$M_1 A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & a'_{22} & a'_{23} & a'_{24} & \cdots & a'_{2n} \\ 0 & a'_{32} & a'_{33} & a'_{34} & \cdots & a'_{3n} \\ 0 & a'_{42} & a'_{43} & a'_{44} & \cdots & a'_{4n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & a'_{n2} & a'_{n3} & a'_{n4} & \cdots & a'_{nn} \end{pmatrix}, \tag{4}$$

where the primes indicate entries which have changed. That is, $a'_{ik} \neq a_{ik}$ in general. Next, in order to zero out the second–column entries of $M_1 A$ below a'_{22}, we construct

$$M_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & -a'_{32}/a'_{22} & 1 & 0 & \cdots & 0 \\ 0 & -a'_{42}/a'_{22} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & -a'_{n2}/a'_{22} & 0 & 0 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & -v'_3 & 1 & 0 & \cdots & 0 \\ 0 & -v'_4 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & -v'_n & 0 & 0 & \cdots & 1 \end{pmatrix}, \tag{5}$$

where again for simplicity we assume $a'_{22} \neq 0$ (or else another pivot is required). We won't need M_2^{-1} until later, but note quickly that it has the simple form

$$M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & v'_3 & 1 & 0 & \cdots & 0 \\ 0 & v'_4 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & v'_n & 0 & 0 & \cdots & 1 \end{pmatrix}. \tag{6}$$

By construction, we then have

$$M_2 M_1 A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & a'_{22} & a'_{23} & a'_{24} & \cdots & a'_{2n} \\ 0 & 0 & a''_{33} & a''_{34} & \cdots & a''_{3n} \\ 0 & 0 & a''_{43} & a''_{44} & \cdots & a''_{4n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a''_{n3} & a''_{n4} & \cdots & a''_{nn} \end{pmatrix}, \tag{7}$$

where $a''_{ik} \neq a'_{ik}$ in general. Maybe you see where this is going already, but let's go through one more step. We form

$$M_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & -a''_{43}/a''_{33} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & -a''_{n3}/a''_{33} & 0 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & -v''_4 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & -v''_n & 0 & \cdots & 1 \end{pmatrix}, \tag{8}$$

which has the simple inverse

$$M_3^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & v_4'' & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & v_n'' & 0 & \cdots & 1 \end{pmatrix}. \tag{9}$$

Using M_3, we find

$$M_3M_2M_1A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & a_{22}' & a_{23}' & a_{24}' & \cdots & a_{2n}' \\ 0 & 0 & a_{33}'' & a_{34}'' & \cdots & a_{3n}'' \\ 0 & 0 & 0 & a_{44}''' & \cdots & a_{4n}''' \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & a_{n4}''' & \cdots & a_{nn}''' \end{pmatrix}. \tag{10}$$

In all, we repeat this procedure $n-1$ times, assuming along the way that all $a_{pp}^{(p-1)} \neq 0$, where (p) indicates p primes. This is just notation, for example $a_{55}^{(4)} = a_{55}''''$, and of course the primes are **not** derivatives. Primes just indicate that multiplication by M_k not only zeros out the kth–column entries below $a_{kk}^{(k-1)}$, but also scrambles up other entries in the matrix. As a result, we arrive at

$$M_{n-1}\cdots M_3M_2M_1A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & a_{22}' & a_{23}' & a_{24}' & \cdots & a_{2n}' \\ 0 & 0 & a_{33}'' & a_{34}'' & \cdots & a_{3n}'' \\ 0 & 0 & 0 & a_{44}''' & \cdots & a_{4n}''' \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{nn}^{(n-1)} \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & u_{24} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & u_{34} & \cdots & u_{3n} \\ 0 & 0 & 0 & u_{44} & \cdots & u_{4n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & 0 & \cdots & u_{nn} \end{pmatrix}. \tag{11}$$

Therefore, $M_{n-1}\cdots M_3M_2M_1A = U$, where U is upper triangular. But this means that

$$A = M_1^{-1}M_2^{-1}M_3^{-1}\cdots M_{n-1}^{-1}U, \tag{12}$$

and we will now show that

$$M_1^{-1}M_2^{-1}M_3^{-1}\cdots M_{n-1}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ v_2 & 1 & 0 & 0 & \cdots & 0 \\ v_3 & v_3' & 1 & 0 & \cdots & 0 \\ v_4 & v_4' & v_4'' & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ v_n & v_n' & v_n'' & v_n''' & \cdots & 1 \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ \ell_{21} & 1 & 0 & 0 & \cdots & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 & \cdots & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \ell_{n4} & \cdots & 1 \end{pmatrix}, \tag{13}$$

which is to say the product $M_1^{-1}M_2^{-1}M_3^{-1}\cdots M_{n-1}^{-1}$ is a *unit lower triangular matrix* (it's lower triangular with ones along the diagonal). To see why (13) holds, first consider

$$M_1^{-1}M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ v_2 & 1 & 0 & 0 & \cdots & 0 \\ v_3 & 0 & 1 & 0 & \cdots & 0 \\ v_4 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ v_n & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & v_3' & 1 & 0 & \cdots & 0 \\ 0 & v_4' & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & v_n' & 0 & 0 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ v_2 & 1 & 0 & 0 & \cdots & 0 \\ v_3 & v_3' & 1 & 0 & \cdots & 0 \\ v_4 & v_4' & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ v_n & v_n' & 0 & 0 & \cdots & 1 \end{pmatrix}. \tag{14}$$

But then

$$M_1^{-1}M_2^{-1}M_3^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ v_2 & 1 & 0 & 0 & \cdots & 0 \\ v_3 & v_3' & 1 & 0 & \cdots & 0 \\ v_4 & v_4' & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ v_n & v_n' & 0 & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & v_4'' & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & v_n'' & 0 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ v_2 & 1 & 0 & 0 & \cdots & 0 \\ v_3 & v_3' & 1 & 0 & \cdots & 0 \\ v_4 & v_4' & v_4'' & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ v_n & v_n' & v_n'' & 0 & \cdots & 1 \end{pmatrix}. \tag{15}$$

Such calculations can be continued, to reach (13). In (13) the second–to–last column [that's the $(n-1)$st column] of $M_1^{-1}M_2^{-1}M_3^{-1}\cdots M_{n-1}^{-1}$ is (in Matlab/standard hybrid notation)

$$\big(M_1^{-1}M_2^{-1}M_3^{-1}\cdots M_{n-1}^{-1}\big)(\underset{\uparrow\text{ all rows}}{:}, \overset{\downarrow (n-1)\text{st column}}{\mathtt{n-1}}) = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ v_n^{(n-2)} \end{pmatrix}, \tag{16}$$

with the superscript $(n-2)$ denoting that many primes in our early notation.

Implementation

Before considering implementation of Gaussian elimination without pivoting, first notice that L and U can be stored "on top of" A. Indeed, consider the rectangular array (don't think of this as a matrix!)

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} & \cdots & u_{1n} \\ \ell_{21} & u_{22} & u_{23} & u_{24} & \cdots & u_{2n} \\ \ell_{31} & \ell_{32} & u_{33} & u_{34} & \cdots & u_{3n} \\ \ell_{41} & \ell_{42} & \ell_{43} & u_{44} & \cdots & u_{4n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \ell_{n4} & \cdots & u_{nn} \end{pmatrix} \tag{17}$$

which contains the information to trivially recover both L and U. This array is clearly the same size as the original matrix A. The algorithm we consider now does not form new matrices L and U, rather it manipulates A until it has the form given in the last equation. Once that form has been achieved, L and U can be (and are in Van Loan's implementation) easily extracted. It's possible to solve systems without extracting L and U, but just keeping the form above. Were we to work this way, we would never use any more memory than is necessary to store the original matrix A.

Here is Van Loan's implementation, which we'll comment on below

```
  function [L,U] = GE(A)
%           [L,U] = GE(A)
% The LU factorization without pivoting. If A is n-by-n and has an
% LU factorization, then L is unit lower triangular and U is upper
% triangular so A = LU.
[n,m] = size(A);
if n ~= m; error('Matrix A not square.'); end;
for k=1:n-1
   A(k+1:n,k) = A(k+1:n,k)/A(k,k);
   A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)*A(k,k+1:n);
end
L = eye(n,n) + tril(A,-1);
U = triu(A);
```

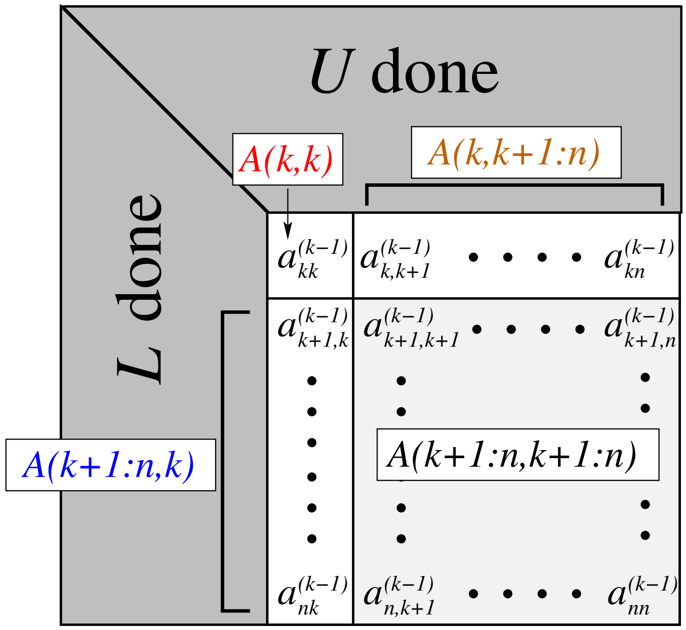

Figure 1: Schematic for LU–factorization. Here we show the matrix after $k-1$ completed steps. At this stage the lower $(n-k+1)$–by–$(n-k+1)$ matrix `A(k:n,k:n)` has been "scrambled" $k-1$ times, so the entries carry $k-1$ superscript primes, e.g. $a_{ij}^{(k-1)}$. These primes make no appearance in the actual algorithm.

The `for` loop in the algorithm transforms the matrix A into the form (17). Figure 1 gives a schematic for the algorithm, with relevant correspondences in matching colors. In the last two lines of `GE.m`, the matrices `L` and `U` are extracted. The `eye(n,n)` creates the n–by–n identity matrix, and `tril(A,-1)` takes the matrix `A` —at this point the form (17)— and returns `A` but with all entries on the diagonal and above set to zero. In other words, it pulls out the lower triangular (the `tril`) of `A` starting from the subdiagonal (the `-1`). So `eye(n,n) + tril(A,-1)` is indeed a unit lower triangular matrix. The `triu` similarly extracts the upper triangular part of `A`. We now explain why A enters the algorithm's `for` loop as (1) and exits the loop as (17). First, in the line `A(k+1:n,k) = A(k+1:n,k)/A(k,k);` the term $A(j,k)/A(k,k) = a_{jk}/a_{kk}$ for $j > k$ is just what we called v_k. Of course these a_{jk} are not the original entries of the matrix. They have been modified, and really are $a_{jk}^{(k-1)}/a_{kk}^{(k-1)} = v_k^{(k-1)}$ in our earlier notation with primes. The point is that they are being written over the zeroed out entries below the diagonal. This is the line responsible for filling in the ℓ_{jk} in (17). Perhaps the other line in the loop, `A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)*A(k,k+1:n);`, will make more sense if we express the operations equivalently as

```
for q = k+1:n
   v = A(q,k);  % A(q,k) was earlier over written with A(q,k)/A(k,k)
   for l = k+1:n
       A(q,l) = A(q,l) - v * A(k,l);
   end
end
```

This fragment clearly exhibits the action of M_k on the subblock $A(k+1:n, k+1:n)$ (no longer the subblock of the original matrix A, since it's entries will have been modified). Van Loan simply takes full advantage of Matlab syntax to compress this double `for` loop into a single line of code!

Here is an example of how to use `GE`.

```
>> A = [17 24  1  8 15;
        23  5  7 14 16;
         4  6 13 20 22;
        10 12 19 21  3;
        11 18 25  2  9];
[L, U] = GE(A)
L =
    1.0000         0         0         0         0
    1.3529    1.0000         0         0         0
    0.2353   -0.0128    1.0000         0         0
    0.5882    0.0771    1.4003    1.0000         0
    0.6471   -0.0899    1.9366    4.0578    1.0000
U =
   17.0000   24.0000    1.0000    8.0000   15.0000
         0  -27.4706    5.6471    3.1765   -4.2941
         0         0   12.8373   18.1585   18.4154
         0         0         0   -9.3786  -31.2802
         0         0         0         0   90.1734
>>
```

We check the quality of the factorization as follows.

```
>> A-L*U
ans =
   1.0e-14 *
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0    0.3553
         0         0         0         0         0
         0         0    0.3553         0         0
>>
```

So indeed this factorization faithfully reproduces A to nearly machine precision.

Gaussian elimination with pivots

Basic idea

The described LU–factorization can **fail**, even on a nonsingular matrix such as

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \neq \begin{pmatrix} 1 & 0 \\ \ell_{21} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{pmatrix}. \tag{18}$$

You can verify that no such factorization is possible by multiplying out the right-hand side, and then comparing terms. There is nothing wrong with this A, and it's easy to invert! Moreover, for some matrices like

$$A = \begin{pmatrix} \delta & 1 \\ 1 & 1 \end{pmatrix}, \tag{19}$$

where δ is very small, it turns out that —whereas the given `GE` algorithm does work and yields an LU factorization— in practice the algorithm's output is of poor quality. Again, the problem is not the matrix! Rather, the issue is with the algorithm `GE` itself. We will not fully explain why, but note that it has to do with the fact that, as described above, we need to do divisions by $a_{kk}^{(k-1)}$. If one of these terms is zero, then the algorithm will **fail**. This does not necessarily mean that the matrix is singular. Poor quality may result if one or more $a_{kk}^{(k-1)}$ is very small. We already mentioned an idea to get around this: when needed, perform row exchanges on the matrix to always ensure that $a_{kk}^{(k-1)}$ is the largest entry (in absolute value) at or below the diagonal in the kth column. For example, suppose after one step we reach

$$M_1 A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & \boxed{a'_{22}} & a'_{23} & a'_{24} & \cdots & a'_{2n} \\ 0 & a'_{32} & a'_{33} & a'_{34} & \cdots & a'_{3n} \\ 0 & \boxed{a'_{42}} & a'_{43} & a'_{44} & \cdots & a'_{4n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & a'_{n2} & a'_{n3} & a'_{n4} & \cdots & a'_{nn} \end{pmatrix}, \tag{20}$$

where $\boxed{a'_{22}}$ happens to be small or zero, while $\boxed{a'_{42}}$ happens to be the largest entry (in absolute value) in column 2 at or below the diagonal (in otherwords, we don't include a_{12} in the comparison). Then before forming the M_2 matrix, we would first exchange rows 2 and 4. To keep track of such row exchanges, we need another matrix, a permutation matrix P. A permutation matrix differs from the identity matrix only by row exchanges (see below for an example).

MATLAB's `lu` function

Van Loan describes the algorithm with pivots (meaning with the extra matrix P). Here we just describe MATLAB's built in LU–factorization and how to use it. The idea is to give up on $A = LU$, and to instead look for the factorization $PA = LU$. Sometimes the convention is $A = PLU$, but we will stick with the first one, since that's what MATLAB uses. A permutation matrix P has inverse P^T (the transpose of P) which is also a permutation matrix. So another way to express the factorization is $A = P^T LU$. In MATLAB the factorization is performed as follows.

```
[L, U, P] = lu(A);
```

Let's return the example we considered above. In the command window we enter the following.

```
>> A = [17 24  1  8 15;
        23  5  7 14 16;
         4  6 13 20 22;
        10 12 19 21  3;
        11 18 25  2  9];
>> [L,U,P] = lu(A)
L =
    1.0000         0         0         0         0
    0.7391    1.0000         0         0         0
    0.4783    0.7687    1.0000         0         0
    0.1739    0.2527    0.5164    1.0000         0
    0.4348    0.4839    0.7231    0.9231    1.0000
U =
   23.0000    5.0000    7.0000   14.0000   16.0000
         0   20.3043   -4.1739   -2.3478    3.1739
         0         0   24.8608   -2.8908   -1.0921
         0         0         0   19.6512   18.9793
         0         0         0         0  -22.2222
P =
     0     1     0     0     0
```

```
     1     0     0     0     0
     0     0     0     0     1
     0     0     1     0     0
     0     0     0     1     0
```

Note the permutation matrix P, and that all entries of L are less than 1 in absolute value (a feature of using pivots). We check the quality of the factorization as follows

```
>> A-P'*L*U
ans =
   1.0e-14 *
         0         0         0         0         0
         0         0         0         0         0
         0         0         0    0.3553         0
         0         0         0   -0.3553    0.3553
         0         0    0.3553         0         0
>>
```

Numerically, the entries of $A - P^T LU$ are of order 10^{-14}, about the same as we found using `GE`.

Let's look at another example, to show why pivots are needed practically. Say we want to numerically solve the simple system

$$\begin{pmatrix} \alpha & 1 & 1 \\ 1 & -1 & 1 \\ 0.5 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 2.5 \end{pmatrix}, \tag{21}$$

where α is a parameter. The *exact solution* is

$$x_1 = 1 + \frac{4\alpha}{2-4\alpha}, \qquad x_2 = 1 + \frac{\alpha}{2-4\alpha}, \qquad x_3 = 1 - \frac{3\alpha}{2-4\alpha}. \tag{22}$$

When $\alpha = 0$, notice that $(x_1, x_2, x_3) = (1, 1, 1)$, and that the matrix above is *not* singular. Indeed

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & -1 & 1 \\ 0.5 & 1 & 1 \end{pmatrix} \implies A^{-1} = \begin{pmatrix} -2 & 0 & 2 \\ -0.5 & -0.5 & 1 \\ 1.5 & 0.5 & -1 \end{pmatrix} \tag{23}$$

Now it's clear that the `GE` algorithm will **fail** if $\alpha = 0$. Let's investigate its performance when $\alpha \neq 0$, but is very small nevertheless. The following script compares solution via Gaussian elimination with and without pivoting (and via MATLAB's slash, which does Gaussian elimination with pivoting).

```
%  Fix the linear system.
alpha = 1e-12;
A = [alpha  1 1;
         1 -1 1;
       0.5  1 1];
b = [2; 1; 2.5];

%  Exact solution.
x_exact = [1 + 4*alpha/(2-4*alpha);
           1 + alpha/(2-4*alpha);
           1 - 3*alpha/(2-4*alpha)];

%  Solution from GE with no pivoting.
[L U] = GE(A);
y = LTriSol(L,b);
```

```
x = UTriSol(U,y)
error = norm(x - x_exact)

%  Solution from GE with pivoting.
[L U P] = lu(A);
Pb = P*b;
y = LTriSol(L,Pb);
x = UTriSol(U,y)
error = norm(x - x_exact)

%  Solution from Matlab slash (uses pivoting).
x = A\b
error = norm(x - x_exact)
```

Here is the output from the script.

```
>> TestLU
x =
     9.999778782798785e-01
     9.999999999990000e-01
     1.000000000000000e+00
error =
     2.212172212145990e-05
x =
     1.000000000002000e+00
     1.000000000000500e+00
     9.999999999984999e-01
error =
     2.482534153247273e-16
x =
     1.000000000002000e+00
     1.000000000000500e+00
     9.999999999984999e-01
error =
     2.482534153247273e-16
```

We see that the solutions obtained with pivots are considerably more accurate.