Chapter 6

Linear Systems

The linear equation problem involves finding a vector $x \in \mathbb{R}^n$ so that $Ax = b$, where $b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$ are nonsingular. This problem is at the heart of many problems in scientific computation. We have already seen that the vector of coefficients that define a polynomial interpolant is a solution to a linear equation problem. In Chapter 3 the problem of spline interpolation led to a tridiagonal system. Most numerical techniques in optimization and differential equations involve repeated linear equation solving. Hence it is extremely important that we know how to solve this problem efficiently and that we fully understand what can be expected in terms of precision.

In this chapter the well-known process of Gaussian elimination is related to the factorization $A = LU$, where L is lower triangular and U is upper triangular. We arrive at the general algorithm in stages, discussing triangular, tridiagonal, and Hessenberg systems first. The need for pivoting is established, and this prompts a discussion of permutation matrices and how they can be manipulated in MATLAB. Finally, we explore the issue of linear system sensitivity and identify the important role that the condition number plays.

6.1 Triangular Problems

At one level, the goal of Gaussian elimination is to convert a given linear system into an equivalent, easy-to-solve triangular system. Triangular system solving is easy because the unknowns can be resolved without any further manipulation of the matrix of coefficients. Consider the following 3-by-3 lower triangular case:

$$\begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

The unknowns can be determined as follows:

$$\begin{aligned} x_1 &= b_1/\ell_{11} \\ x_2 &= (b_2 - \ell_{21}x_1)/\ell_{22} \\ x_3 &= (b_3 - \ell_{31}x_1 - \ell_{32}x_2)/\ell_{33} \end{aligned}$$

This is the 3-by-3 version of an algorithm known as *forward substitution*. Notice that the process requires $\det(A) = \ell_{11}\ell_{22}\ell_{33}$ to be nonzero.

6.1.1 Forward Substitution

Let's look at the general $Lx = b$ problem when L is lower triangular. To derive a specification for x_i, we merely rearrange the ith equation

$$\ell_{i1}x_1 + \cdots + \ell_{ii}x_i = b_i$$

to obtain

$$x_i = \left(b_i - \sum_{j=1}^{i-1} \ell_{ij}x_j \right) \Big/ \ell_{ii}.$$

If this is evaluated for $i = 1{:}n$, then a complete specification for x is obtained:

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

Note that the j-loop effectively subtracts the inner product

$$\sum_{j=1}^{i-1} \ell_{ij}x_j = L(i, 1{:}i-1) * x(1{:}i-1)$$

from b_i, so we can vectorize as follows:

```
x(1) = b(1)/L(1,1);
for i = 2:n
   x(i) = ( b(i) - L(i,1:i-1)*x(1:i-1) ) /L(i,i);
end
```

Since the computation of x_i involves about $2i$ flops, the entire process requires about

$$2\,(1 + 2 + \cdots + n) \approx n^2$$

flops. The forward substitution algorithm that we have derived is row oriented. At each stage an inner product must be computed that involves part of a row of L and the previously computed portion of x.

A column-oriented version that features the saxpy operation can also be obtained. Consider the $n = 3$ case once again. Once x_1 is resolved, it can be removed from equations 2 and 3, leaving us with a reduced, 2-by-2 lower triangular system. For example, to solve

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 5 & 0 \\ 7 & 9 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \\ 5 \end{bmatrix},$$

we find that $x_1 = 3$ and then deal with the 2-by-2 system

$$\begin{bmatrix} 5 & 0 \\ 9 & 8 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix} - 3 \begin{bmatrix} 1 \\ 7 \end{bmatrix} = \begin{bmatrix} -1 \\ -16 \end{bmatrix}.$$

This implies that $x_2 = -1/5$. The system is then reduced to

$$8x_3 = -16 - 9(-1/5),$$

from which we conclude that $x_3 = -71/40$. In general, at the jth step we solve for x_j and then remove it from equations $j + 1$ through n. At the start, $x_1 = b_1/\ell_{11}$ and equations 2 through n transform to

$$\begin{bmatrix} \ell_{22} & 0 & \cdots & 0 \\ \ell_{32} & \ell_{33} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{n2} & \ell_{n3} & \cdots & \ell_{nn} \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_2 - x_1\ell_{21} \\ b_3 - x_1\ell_{31} \\ \vdots \\ b_n - x_1\ell_{n1} \end{bmatrix} = b(2{:}n) - x_1 L(2{:}n, 1).$$

In general the j-th step computes $x_j = b_j/\ell_{jj}$ and then performs the saxpy update

$$b(j + 1{:}n) \leftarrow b(j + 1{:}n) - x_j L(j + 1{:}n, j).$$

Putting it all together, we obtain

```
  function x = LTriSol(L,b)
% x = LTriSol(L,b)
% Solves the nonsingular lower triangular system  Lx = b
% where L is n-by-n, b is n-by-1, and x is n-by-1.

n = length(b);
x = zeros(n,1);
for j=1:n-1
   x(j) = b(j)/L(j,j);
   b(j+1:n) = b(j+1:n) - L(j+1:n,j)*x(j);
end
x(n) = b(n)/L(n,n);
```

This version involves n^2 flops, just like the row-oriented, dot product version developed earlier.

6.1.2 Backward Substitution

The upper triangular case is analogous. The only difference is that the unknowns are resolved in reverse order. Thus to solve

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

we work from the bottom to the top:

$$\begin{aligned} x_3 &= b_3/u_{33} \\ x_2 &= (b_2 - u_{23}x_3)/u_{22} \\ x_1 &= (b_1 - u_{12}x_2 - u_{13}x_3)/u_{11} \end{aligned}$$

For general n, we obtain

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

As in the lower triangular case, the computations can be arranged so that a column-oriented, saxpy update procedure is obtained:

```
  function x = UTriSol(U,b)
% x = UTriSol(L,b)
% Solves the nonsingular upper triangular system  Ux = b.
% where U is n-by-n, b is n-by-1, and X is n-by-1.

n = length(b);
x = zeros(n,1);
for j=n:-1:2
   x(j) = b(j)/U(j,j);
   b(1:j-1) = b(1:j-1) - x(j)*U(1:j-1,j);
end
x(1) = b(1)/U(1,1);
```

This algorithm is called *backward substitution*.

6.1.3 Multiple Right-Hand Sides

In many applications we must solve a sequence of triangular linear systems where the matrix stays the same, but the right-hand sides vary. For example, if L is lower triangular and $B \in \mathbb{R}^{n\times r}$ is given, our task is to find $X \in \mathbb{R}^{n\times r}$ so that $LX = B$. Looking at the kth column of this matrix equation, we see that

$$LX(:,k) = B(:,k).$$

One way to solve for X is merely to apply the single right-hand side forward substitution algorithm r times:

```
X = zeros(n,r);
for k=1:r
   X(:,k) = LTriSol(L,B(:,k));
end
```

However, if we expand the call to `LTriSol`,

```
X = zeros(n,r);
for k=1:r
   for j=1:n-1
      X(j,k)     = B(j,k)/L(j,j);
      B(j+1:n,k) = B(j+1:n,k) - L(j+1:n,j)*X(j,k);
   end
   X(n,k) = B(n,k)/L(n,n);
end
```

and modify the order of computation, then we can vectorize "on `k`." To do this, note that in the preceding script we solve for $X(:,1)$, and then $X(:,2)$, and then $X(:,3)$, etc. Instead, we can solve for $X(1,:)$, and then $X(2,:)$, and then $X(3,:)$, etc. This amounts to reversing the order of the `k-` and `j-` loops:

```
X = zeros(n,r);
for j=1:n-1
  for k=1:r
      X(j,k)     = B(j,k)/L(j,j);
      B(j+1:n,k) = B(j+1:n,k) - L(j+1:n,j)*X(j,k);
   end
end
for k=1:r
   X(n,k) = B(n,k)/L(n,n);
end
```

Vectorizing the `k`-loops, we obtain

```
  function X = LTriSolM(L,B)
% X = LTriSolM(L,B)
% Solves the nonsingular lower triangular system  LX = B
% where L is n-by-n, B is n-by-r, and X is n-by-r.
[n,r] = size(B);
X = zeros(n,r);
for j=1:n-1
   X(j,1:r)     = B(j,1:r)/L(j,j);
   B(j+1:n,1:r) = B(j+1:n,1:r) - L(j+1:n,j)*X(j,1:r);
end
X(n,1:r) = B(n,1:r)/L(n,n);
```

In high-performance computing environments, maneuvers like this are often the key to efficient matrix computations.

As an example, we use `LTriSolM` to compute the inverse of the n-by-n *Forsythe* matrix $F_n = (f_{ij})$ defined as follows:

$$f_{ij} = \begin{cases} 0 & \text{if } i < j \\ 1 & \text{if } i = j \\ -1 & \text{if } i > j \end{cases}.$$

This is accomplished by solving $F_n X = I_n$. For example,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} X = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Running the script file

```
% Script File: ShowTri
%
% Inverse of the 5-by-5 Forsythe Matrix.

   n = 5;
   L = eye(n,n) - tril(ones(n,n),-1)
   X = LTriSolM(L,eye(n,n))
```

we find

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 4 & 2 & 1 & 1 & 0 \\ 8 & 4 & 2 & 1 & 1 \end{bmatrix}.$$

Problems

P6.1.1 Modify `LTriSol` so that if $\ell_{11} = 0$, then it returns a vector x that satisfies $Lx = 0$ with $x_1 = -1$.

P6.1.2 Develop a vectorized method for solving the upper triangular multiple right-hand-side problem.

P6.1.3 Suppose $T \in \mathbb{R}^{n\times n}$ and $S \in \mathbb{R}^{m\times m}$ are given upper triangular matrices and that $B \in \mathbb{R}^{m\times n}$. Write a MATLAB function `X = Sylvester(S,T,B)` that solves the matrix equation $SX - XT = B$ for X. Note that if we compare kth columns in this equation, we obtain

$$SX(:,k) - \sum_{j=1}^{k} T(j,k)X(:,j) = B(:,k).$$

That is,

$$(S - T(k,k)I)X(:,k) = B(:,k) + \sum_{j=1}^{k-1} T(j,k)X(:,j).$$

By using this equation for $k = 1{:}n$, we can solve for $X(:,1), \ldots, X(:,n)$ in turn. Moreover, the matrix $S - T(k,k)I$ is upper triangular so that we can apply `UTriSol`. Assume that no diagonal entry of S is a diagonal entry of T.

P6.1.4 Repeat the previous problem, assuming that S and T are both lower triangular.

P6.1.5 Note that by solving the multiple right-hand-side problem $TX = B$ with $B = I$, then the solution is the inverse of T. Write a MATLAB function `X = UTriInv(U)` that computes the inverse of a nonsingular upper triangular matrix. Be sure to exploit any special patterns that arise because of B's special nature.

P6.1.6 As a function of n, i, and j, give an expression for the (i, j) entry of the inverse of the n-by-n Forsythe matrix.

P6.1.7 Complete the following function:

```
   function Z = PartInvU(A,p)
% Z = PartInvU(A,p)
% A is an n-by-n upper triangular nonsingular matrix and
% p is an integer that satisfies 1<=p<=n.
% Z   = X(1:p,1:p) where AX = I
```

Use `UTriSol(A,b)`.

6.2 Banded Problems

Before we embark on the development of Gaussian elimination for general linear systems, we take time out to look at the linear equation problem in two special cases where the matrix of coefficients already has a large number of zeros.

In the spline interpolation problem of §3.3, we have to solve a tridiagonal linear system whose matrix of coefficients looks like this:

$$A = \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 & 0 \\ 0 & \times & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix}.$$

(See page 125.) In such a system, each unknown x_i is coupled to at most two of its "neighbors." For example, the fourth equation relates x_4 to x_3 and x_5. This kind of local coupling among the unknowns occurs in a surprising number of applications. It is a happy circumstance because the matrix comes to us with zeros in many of the places that would ordinarily be zeroed by the elimination process.

Upper Hessenberg systems provide a second family of specialized problems that are useful to consider. An *upper Hessenberg* matrix has lower bandwidth 1. For example,

$$A = \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix}.$$

Upper Hessenberg linear systems arise in many applications where the eigenvalues and eigenvectors of a matrix are required.

For these two specially structured linear equation problems, we set out to show how the matrix of coefficients can be factored into a product $A = LU$, where L is lower triangular and U is upper triangular. If such a factorization is available, then the solution to $Ax = b$ follows from a pair of triangular system solves:

$$\left.\begin{array}{r} Ly = b \\ \\ Ux = y \end{array}\right\} \Rightarrow\ Ax = (LU)x = L(Ux) = Ly = b.$$

In terms of the functions `LTriSol` and `UTriSol`,

```
y = LTriSol(L,b);
x = UTriSol(U,y);
```

For tridiagonal and Hessenberg systems, the computation of L and U is easier to explain than for general matrices.

6.2.1 Tridiagonal Systems

Consider the following 4-by-4 tridiagonal linear system:

$$\begin{bmatrix} d_1 & f_1 & 0 & 0 \\ e_2 & d_2 & f_2 & 0 \\ 0 & e_3 & d_3 & f_3 \\ 0 & 0 & e_4 & d_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}.$$

One way to derive the LU factorization for a tridiagonal A is to equate entries in the following equation:

$$\begin{bmatrix} d_1 & f_1 & 0 & 0 \\ e_2 & d_2 & f_2 & 0 \\ 0 & e_3 & d_3 & f_3 \\ 0 & 0 & e_4 & d_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_2 & 1 & 0 & 0 \\ 0 & \ell_3 & 1 & 0 \\ 0 & 0 & \ell_4 & 1 \end{bmatrix} \begin{bmatrix} u_1 & f_1 & 0 & 0 \\ 0 & u_2 & f_2 & 0 \\ 0 & 0 & u_3 & f_3 \\ 0 & 0 & 0 & u_4 \end{bmatrix}.$$

Doing this, we find

$$\begin{array}{lllll} (1,1): & d_1 = u_1 & \Rightarrow & u_1 = d_1 \\ (2,1): & e_2 = \ell_2 u_1 & \Rightarrow & \ell_2 = e_2/u_1 \\ (2,2): & d_2 = \ell_2 f_1 + u_2 & \Rightarrow & u_2 = d_2 - \ell_2 f_1 \\ (3,2): & e_3 = \ell_3 u_2 & \Rightarrow & \ell_3 = e_3/u_2 \\ (3,3): & d_3 = \ell_3 f_2 + u_3 & \Rightarrow & u_3 = d_3 - \ell_3 f_2 \\ (4,3): & e_4 = \ell_4 u_3 & \Rightarrow & \ell_4 = e_4/u_3 \\ (4,4): & d_4 = \ell_4 f_3 + u_4 & \Rightarrow & u_4 = d_4 - \ell_4 f_3 \end{array}$$

In general, for $i = 2{:}n$ we have

$$\begin{array}{lllll} (i,i-1): & e_i = \ell_i u_{i-1} & \Rightarrow & \ell_i = e_i/u_{i-1} \\ (i,i): & d_i = \ell_i f_{i-1} + u_i & \Rightarrow & u_i = d_i - \ell_i f_{i-1} \end{array}$$

which leads to the following procedure:

```
  function [l,u] = TriDiLU(d,e,f)
% [l,u] = TriDiLU(d,e,f)
% Tridiagonal LU without pivoting. d,e,f are  n-vectors and assume
% A = diag(e(2:n),-1) + diag(d) + diag(f(1:n-1),1) has an LU factorization.
% l and u are n-vectors with the property that if L = eye + diag(l(2:n),-1)
% and U = diag(u) + diag(f(1:n-1),1), then A = LU.
n = length(d); l = zeros(n,1); u = zeros(n,1);
u(1) = d(1);
for i=2:n
   l(i) = e(i)/u(i-1);
   u(i) = d(i) - l(i)*f(i-1);
end
```

This process requires $3n$ flops to carry out and is defined as long as $u_1, \ldots, u_{n-1}$ are nonzero.

As mentioned previously, to solve $Ax = b$ we must solve

$$Ly = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_2 & 1 & 0 & 0 \\ 0 & \ell_3 & 1 & 0 \\ 0 & 0 & \ell_4 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = b$$

for y, and

$$Ux = \begin{bmatrix} u_1 & f_1 & 0 & 0 \\ 0 & u_2 & f_2 & 0 \\ 0 & 0 & u_3 & f_3 \\ 0 & 0 & 0 & u_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = y$$

for x. These *bidiagonal* systems can be solved very simply. Looking at $Ly = b$, we see by comparing components that

$$\begin{array}{lcl} y_1 = b_1 & \Rightarrow & y_1 = b_1 \\ \ell_2 y_1 + y_2 = b_2 & \Rightarrow & y_2 = b_2 - \ell_2 y_1 \\ \ell_3 y_2 + y_3 = b_3 & \Rightarrow & y_3 = b_3 - \ell_3 y_2 \\ \ell_4 y_3 + y_4 = b_4 & \Rightarrow & y_4 = b_4 - \ell_4 y_3 \\ \ell_5 y_4 + y_5 = b_5 & \Rightarrow & y_5 = b_5 - \ell_5 y_4 \end{array}$$

From this we conclude

```
  function x = LBiDiSol(l,b)
% x = LBiDiSol(l,b)
% Solves the n-by-n unit lower bidiagonal system  Lx = b
% where l and b are n-by-1 and L = I + diag(l(2:n),-1).
n = length(b); x = zeros(n,1);
x(1) = b(1);
for i=2:n
   x(i) = b(i) - l(i)*x(i-1);
end
```

This requires $2n$ flops. Likewise, the upper bidiagonal system $Ux = y$ can be solved as follows:

```
  function x = UBiDiSol(u,f,b)
% x = UBiDiSol(u,f,b)
% Solves the n-by-n nonsingular upper bidiagonal system  Ux = b
% where u, f, and b are n-by-1 and U = diag(u) + diag(f(1:n-1),1).
n = length(b); x = zeros(n,1);
x(n) = b(n)/u(n);
for i=n-1:-1:1
   x(i) = (b(i) - f(i)*x(i+1))/u(i);
end
```

Summarizing the overall solution process, the script

```
[l,u] = TriDiLU(d,e,f);
y = LBiDiSol(l,b);
x = UBiDiSol(u,f,y);
```

solves the tridiagonal system $Ax = b$, assuming that `d`, `e`, and `f` house the diagonal, subdiagonal, and superdiagonal of A. The following table indicates the amount of arithmetic required:

Operation	Procedure	Flops
$A = LU$	`TriDiLU`	$3n$
$Ly = b$	`LBiDiSol`	$2n$
$Ux = y$	`UBiDiSol`	$3n$

Run the script file `ShowTriD`, which illustrates some of the key ideas behind tridiagonal system solving.

6.2.2 Hessenberg Systems

We derived the algorithm for tridiagonal LU by equating coefficients in $A = LU$. We could use this same strategy for Hessenberg LU. However, in anticipation of the general LU computation, we proceed in "elimination terms."

Presented with a Hessenberg system

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix},$$

we notice that we can get the first unknown to "drop out" of the second equation by multiplying the first equation by a_{21}/a_{11} and subtracting from the second equation. This transforms the system to

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix}.$$

Then we notice that we can eliminate the second unknown from equation 3 by multiplying the (new) second equation by a_{32}/a_{22} and subtracting from the third equation:

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix}.$$

The pattern emerges regarding what happens to A during this process:

```
for k=1:n-1
   Set v(k) = A(k+1,k)/A(k,k).
   Update row k+1 by subtracting from it, v(k) times row k.
end
```

The modification of row $k+1$ involves the following computations:

```
for j=1:n
   A(k+1,j) = A(k+1,j) - v(k)*A(k,j)
end
```

Note that since rows k and $k+1$ have zeros in their first $k-1$ positions, it makes sense to modify the loop range to `j=k:n`. Incorporating this change and vectorizing, we obtain

```
  function [v,U] = HessLU(A)
% [v,U] = HessLU(A)
% Computes the factorization H = LU where H is an n-by-n upper Hessenberg
% and L is an n-by-n lower unit triangular and U is an n-by-n upper triangular
% matrix.
% v is a column n-by-1 vector with the property that L = I + diag(v(2:n),-1).
[n,n] = size(A);
v = zeros(n,1);
for k=1:n-1
   v(k+1) = A(k+1,k)/A(k,k);
   A(k+1,k:n) = A(k+1,k:n) - v(k+1)*A(k,k:n);
end
U = triu(A);
```

It can be shown that this procedure requires n^2 flops. The connection between the vector v of multipliers and the lower triangular matrix L needs to be explained.

In the $n = 6$ case, steps 1 through 5 in `HessLU` involve the premultiplication of the matrix A by the matrices

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -v_2 & 1 & 0 & 0 & 0 & \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \ldots, M_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -v_6 & 1 \end{bmatrix}$$

respectively. Run the script file `ShowHessLU` for an illustration of the reduction. From this we conclude that `HessLU` basically finds *multiplier matrices* $M_1, \ldots, M_{n-1}$ so that

$$M_{n-1} \cdots M_1 A = U$$

is upper triangular. The multiplier matrices are nonsingular, and it is easy to verify (for example) that

$$M_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -v_4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & v_4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Thus,

$$A = LU,$$

where

$$L = M_1^{-1} \cdots M_{n-1}^{-1}.$$

The product of the M_i^{-1} is lower triangular, and it can be shown that

$$M_1^{-1} M_2^{-1} M_3^{-1} M_4^{-1} M_5^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ v_2 & 1 & 0 & 0 & 0 & 0 \\ 0 & v_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & v_4 & 1 & 0 & 0 \\ 0 & 0 & 0 & v_5 & 1 & 0 \\ 0 & 0 & 0 & 0 & v_6 & 1 \end{bmatrix}$$

is a lower bidiagonal matrix. Thus the script

```
[v,U] = HessLU(A);
y = LBiDiSol(v,b);
x = UTriSol(U,y);
```

solves the upper Hessenberg system $Hx = b$ and $n^2 + 2n + n^2 \approx 2n^2$ flops are required.

Problems

P6.2.1 Write a MATLAB function `x = HessTrans(A,b)` that solves the linear system $A^T x = b$, where A is upper Hessenberg. Hint: If $A = LU$, then $A^T = U^T L^T$. In the Hessenberg case, U^T is lower triangular and L^T is upper bidiagonal.

P6.2.2 Incorporate the tridiagonal system solving codes into `CubicSpline` of Chapter 3. Quantify through benchmarks the improvement in efficiency. Notice also the reduction in memory requirements.

P6.2.3 Suppose $H \in \mathbb{R}^{m \times m}$ and $T \in \mathbb{R}^{n \times n}$ are given matrices with H upper Hessenberg and T upper triangular. Assume that $B \in \mathbb{R}^{m \times n}$. Write a MATLAB function `X = SylvesterH(H,T,B)` that solves the matrix equation $HX - XT = B$ for X. (See P6.1.3 on page 214.)

P6.2.4 Suppose $W \in \mathbb{R}^{n\times n}$ is tridiagonal and $T = W + \alpha e_n e_1^T + \beta e_1 e_n^T$ where α and β are scalars and e_1 and e_n are the first and last columns of the n-by-n identity matrix. This is just a fancy way of defining a tridiagonal matrix "with corners":

$$T = \begin{bmatrix} w_{11} & w_{12} & 0 & 0 & 0 & \beta \\ w_{21} & w_{22} & w_{23} & 0 & 0 & 0 \\ 0 & w_{32} & w_{33} & w_{34} & 0 & 0 \\ 0 & 0 & w_{43} & w_{44} & w_{45} & 0 \\ 0 & 0 & 0 & w_{54} & w_{55} & w_{56} \\ \alpha & 0 & 0 & 0 & w_{65} & w_{66} \end{bmatrix} \qquad n = 6.$$

This kind of matrix arises in the periodic spline problem.

It can be shown that the solution to the linear system $(W + \alpha e_n e_1^T + \beta e_1 e_n^T)x = b$ is given by

$$x = z - Y \begin{bmatrix} 1+y_{11} & y_{12} \\ y_{n1} & 1+y_{n2} \end{bmatrix}^{-1} \begin{bmatrix} z_1 \\ z_n \end{bmatrix},$$

where $z = W^{-1}b \in \mathbb{R}^n$ and $Y = W^{-1}[\alpha e_n \;\; \beta e_1] \in \mathbb{R}^{n\times 2}$. Write a function `x = TriCorner(d,e,f,alpha,beta,b)` that solves $(W + \alpha e_n e_1^T + \beta e_1 e_n^T)x = b$ where `d`, `e`, and `f` are linear arrays that encode W as in `TriDiLU`. Make effective use of that function as well as `LBiDiSol` and `UBiDiSol`. You may assume that pivoting is not necessary.

6.3 General Problems

We are now ready to develop a general linear equation solver. Again, the goal is to find a lower triangular L and an upper triangular U such that $A = LU$.

6.3.1 The $n = 3$ Case

The method of *Gaussian elimination* proceeds by systematically removing unknowns from equations. The core calculation is the multiplication of an equation by a scalar and its subtraction from another equation. For example, if we are given the system

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 13 \\ -4x_1 + 6x_2 - 5x_3 &= -28 \\ 6x_1 + 13x_2 + 16x_3 &= 37 \end{aligned} \tag{6.1}$$

then we start by multiplying the first equation by $-4/2 = -2$ and subtracting it from the second equation. This removes x_1 from the second equation. Likewise we can remove x_1 from the third equation by subtracting from it $6/2 = 3$ times the first equation. With these two reductions we obtain

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 13 \\ 4x_2 + x_3 &= -2 \\ 16x_2 + 7x_3 &= -2 \end{aligned}$$

We then multiply the (new) second equation by $16/4 = 4$ and subtract it from the (new) third equation, obtaining

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 13 \\ 4x_2 + x_3 &= -2 \\ 3x_3 &= 6 \end{aligned} \tag{6.2}$$

Thus, the elimination transforms the given square system into an equivalent upper triangular system that has the same solution. The solution of triangular systems is discussed in §6.1. In

our 3-by-3 example we proceed as follows:

$$\begin{array}{rcll} x_3 &=& 6/3 & = & 2 \\ x_2 &=& (-2 - x_3)/4 & = & -1 \\ x_1 &=& (3 - 3x_3 + x_2)/2 & = & 3 \end{array}$$

This description of Gaussian elimination can be succinctly described in matrix terms. In particular, the process finds a lower triangular matrix L and an upper triangular matrix U so $A = LU$. In the preceding example, we have

$$A = \begin{bmatrix} 2 & -1 & 3 \\ -4 & 6 & -5 \\ 6 & 13 & 16 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 3 \\ 0 & 4 & 1 \\ 0 & 0 & 3 \end{bmatrix} \equiv LU.$$

Notice that the subdiagonal entries in L are made up of the multipliers that arise during the elimination process. The diagonal elements of L are all equal to one. Lower triangular matrices with this property are called *unit lower triangular*.

In matrix computations, the language of "matrix factorizations" has assumed a role of great importance. It enables one to reason about algorithms at a high level, which in turn facilitates generalization and implementation on advanced machines. Thus, we regard Gaussian elimination as a procedure for computing the LU factorization of a matrix. Once this factorization is obtained, then as we have discussed, the solution to $Ax = b$ requires a pair of triangular system solves: $Ly = b$, $Ux = y$. There are practical reasons why it is important to decouple the right-hand side from the elimination process. But the curious reader will note that the transformed right-hand side in (6.2) is the solution to the lower triangular system

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 13 \\ -28 \\ 37 \end{bmatrix}.$$

After this build-up for the LU factorization, it is disturbing to note that the elimination process on which it is based can break down in some very simple examples. For example, if we modify (6.1) by changing the $(1,1)$ coefficient from 2 to 0,

$$\begin{array}{rcrcrcr} & & x_2 & + & 3x_3 & = & 13 \\ -4x_1 & + & 6x_2 & - & 5x_3 & = & -28 \\ 6x_1 & + & 13x_2 & + & 16x_3 & = & 37 \end{array}$$

then the elimination process defined previously cannot get off the ground because we cannot use the first equation to get rid of x_1 in the second and third equations. A simple fix is proposed in §6.3.4 on page 227. Until then, we assume that the matrices under discussion submit quietly to the LU factorization process without any numerical difficulty.

6.3.2 General n

We now turn our attention to the LU factorization of a general matrix. In looking at the system

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix},$$

we see that we can eliminate the unknown x_1 from equation i by subtracting from it a_{i1}/a_{11} times equation 1. If we do this for $i = 2{:}6$, then the given linear system transforms to

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix}.$$

To remove x_2 from equation i, we scale (the new) 2nd equation by a_{i2}/a_{22} and subtract from row i. Doing this for $i = 3{:}6$ gives

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \end{bmatrix} \cdot \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix}.$$

The pattern should now be clear regarding the operations that must be performed on A:

```
for k=1:n-1
   Compute the multipliers required to eliminate x(k) from equations
      k+1 through n and store in v(k+1:n).
   Update equations k+1 through n.
end
```

The multipliers required in the kth step are specified as follows:

```
for i=k+1:n
   v(i) = A(i,k)/A(k,k);
end
```

That is, `v(k+1:n) = A(k+1:n,k)/A(k,k)`. The act of multiplying row k by $v(i)$ and subtracting from row i can be implemented with `A(i,k:n) = A(i,k:n) - v(i)*A(k,k:n)`. The column range begins at k because the first $k-1$ entries in both rows k and i are zero. Incorporating these ideas, we get the following procedure for upper triangularizing A:

```
for k=1:n-1
   v(k+1:n) = A(k+1:n,k)/A(k,k);
   for i=k+1:n                                                    (6.3)
      A(i,k+1:n) = A(i,k+1:n) - v(i)*A(k,k+1:n);
   end
end
U = triu(A)
```

The i-loop oversees a collection of row-oriented saxpy operations. For example, if $n = 6$ and $k = 3$, then the three row saxpys

$$\begin{aligned} A(4,4{:}6) &\leftarrow A(4,4{:}6) - v(4)A(3,4{:}6) \\ A(5,4{:}6) &\leftarrow A(5,4{:}6) - v(5)A(3,4{:}6) \\ A(6,4{:}6) &\leftarrow A(6,4{:}6) - v(6)A(3,4{:}6) \end{aligned}$$

are equivalent to

$$A(4{:}6,4{:}6) = \begin{bmatrix} A(4,4{:}6) \\ A(5,4{:}6) \\ A(6,4{:}6) \end{bmatrix} \leftarrow \begin{bmatrix} A(4,4{:}6) \\ A(5,4{:}6) \\ A(6,4{:}6) \end{bmatrix} - \begin{bmatrix} v_4 \\ v_5 \\ v_6 \end{bmatrix} A(3,4{:}6).$$

Thus, the i-loop in (6.3) can be replaced by a single outer product update:

```
for k=1:n-1
   v(k+1:n) = A(k+1:n,k)/A(k,k)
   A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - v(k+1:n)*A(k,k+1:n);          (6.4)
end
U = triu(A);
```

So much for the production of U. To compute L, we proceed as in the Hessenberg case and show that it is made up of the multipliers. In particular, during the kth pass through the loop in (6.3), the current A matrix is premultiplied by a multiplier matrix of the form

$$M_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -v_4 & 1 & 0 & 0 \\ 0 & 0 & -v_5 & 0 & 1 & 0 \\ 0 & 0 & -v_6 & 0 & 0 & 1 \end{bmatrix}, \qquad (n = 6, k = 3).$$

After $n - 1$ steps,

$$M_{n-1} \cdots M_2 M_1 A = U$$

is upper triangular and so

$$A = \left(M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1}\right) U.$$

The inverse of a multiplier matrix has a particularly simple form. For example,

$$M_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & v_4 & 1 & 0 & 0 \\ 0 & 0 & v_5 & 0 & 1 & 0 \\ 0 & 0 & v_6 & 0 & 0 & 1 \end{bmatrix}.$$

Moreover, $L = M_1^{-1}M_2^{-1}\cdots M_{n-1}^{-1}$ is lower triangular with the property that $L(:,k)$ is the kth column of M_k^{-1}. Thus, in the $n = 6$ case we have

$$L = M_1^{-1}M_2^{-1}M_3^{-1}M_4^{-1}M_5^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ v_2^{(1)} & 1 & 0 & 0 & 0 & 0 \\ v_3^{(1)} & v_3^{(2)} & 1 & 0 & 0 & 0 \\ v_4^{(1)} & v_4^{(2)} & v_4^{(3)} & 1 & 0 & 0 \\ v_5^{(1)} & v_5^{(2)} & v_5^{(3)} & v_5^{(4)} & 1 & 0 \\ v_6^{(1)} & v_6^{(2)} & v_6^{(3)} & v_6^{(4)} & v_6^{(5)} & 1 \end{bmatrix},$$

where the superscripts are used to indicate the step associated with the multiplier. This suggests that the multipliers can be stored in the locations that they are designed to zero. For example, $v_5^{(2)}$ is computed during the second step in order to zero a_{52} and it can be stored in location $(5, 2)$. This leads to the following implementation of Gaussian elimination:

```
  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,n] = size(A);
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);
```

It can be shown that this calculation requires $2n^3/3$ flops. The script `ShowGE` steps through this factorization process for the matrix

$$A = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix},$$

and terminates with

$$L = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 0 \\ 1.3529 & 1.0000 & 0 & 0 & 0 \\ 0.2353 & -0.0128 & 1.0000 & 0 & 0 \\ 0.5882 & 0.0771 & 1.4003 & 1.0000 & 0 \\ 0.6471 & -0.0899 & 1.9366 & 4.0578 & 1.0000 \end{bmatrix}$$

and

$$U = \begin{bmatrix} 17.0000 & 24.0000 & 1.0000 & 8.0000 & 15.0000 \\ 0 & -27.4706 & 5.6471 & 3.1765 & -4.2941 \\ 0 & 0 & 12.8373 & 18.1585 & 18.4154 \\ 0 & 0 & 0 & -9.3786 & -31.2802 \\ 0 & 0 & 0 & 0 & 90.1734 \end{bmatrix}.$$

Together with `LTriSOl` and `UTriSol`, `GE` can be used to solve a linear system $Ax = b$:

```
[L,U] = GE(A);
y = LTriSol(L,b);
x = UTriSol(U,y)
```

But this assumes that no zero divides arise during the execution of `GE`.

6.3.3 Stability

The time has come to address the issue of breakdown in the Gaussian elimination process. We start with the grim fact that a matrix need not have an LU factorization. To see this, equate coefficients in

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \ell_{21} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}.$$

Equality in the $(1,1)$ position implies that $u_{11} = 0$. But then it is impossible to have agreement in the $(2,1)$ position since we must have $\ell_{21}u_{11} = 1$. Note that there is nothing "abnormal" about an $Ax = b$ problem in which $a_{11} = 0$. For example,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

has solution $x = [1\ 1]^T$. It looks like corrective measures are needed to handle the undefined multiplier situation.

But problems also arise if the multipliers are large:

$$A = \begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1/\delta & 1 \end{bmatrix} \begin{bmatrix} \delta & 1 \\ 0 & 1 - \frac{1}{\delta} \end{bmatrix} = LU, \qquad (\delta \neq 0).$$

The following script solves

$$Ax = \begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1+\delta \\ 2 \end{bmatrix} = b$$

by computing $A = LU$ and then solving $Ly = b$ and $Ux = y$ in the usual fashion:

```
% Script File: NoPivot
% Examines solution to [delta 1 ; 1 1][x1;x2] = [1+delta;2]
% for a sequence of diminishing delta values.
clc
disp(' Delta           x(1)                  x(2)  ' )
disp('-----------------------------------------------------')
for delta = logspace(-2,-18,9)
   A = [delta 1; 1 1];
   b = [1+delta; 2];
   L = [ 1 0; A(2,1)/A(1,1) 1];
   U = [ A(1,1) A(1,2) ; 0 A(2,2)-L(2,1)*A(1,2)];
   y(1) = b(1);
   y(2) = b(2) - L(2,1)*y(1);
   x(2) = y(2)/U(2,2);
   x(1) = (y(1) - U(1,2)*x(2))/U(1,1);
   disp(sprintf(' %5.0e   %20.15f  %20.15f',delta,x(1),x(2)))
end
```

Here are the results:

```
 Delta           x(1)                  x(2)
-----------------------------------------------------
 1e-02     1.000000000000001     1.000000000000000
 1e-04     0.999999999999890     1.000000000000000
 1e-06     1.000000000028756     1.000000000000000
 1e-08     0.999999993922529     1.000000000000000
 1e-10     1.000000082740371     1.000000000000000
 1e-12     0.999866855977416     1.000000000000000
 1e-14     0.999200722162641     1.000000000000000
 1e-16     2.220446049250313     1.000000000000000
 1e-18     0.000000000000000     1.000000000000000
```

(You might want to deduce why $\hat{x}_2$ is exact.) A simple way to avoid this degradation is to introduce *row interchanges*. In the preceding example this means we apply Gaussian elimination to compute the LU factorization of A with its rows reversed:

$$\begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1+\delta \end{bmatrix}.$$

A full precision answer is obtained.

6.3.4 Pivoting

To anticipate the row interchange process in the general case, we consider the third step in the $n = 6$ case. At the beginning of the step we face a partially reduced A that has the following

form:

$$A = \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & a_{33} & \times & \times & \times \\ 0 & 0 & a_{43} & \times & \times & \times \\ 0 & 0 & a_{53} & \times & \times & \times \\ 0 & 0 & a_{63} & \times & \times & \times \end{bmatrix}.$$

(Note that the displayed a_{ij} are not the original a_{ij}; they have been updated twice.) Ordinarily, we would use multipliers a_{43}/a_{33}, a_{53}/a_{33}, and a_{63}/a_{33} to zero entries $(4,3)$, $(5,3)$, and $(6,3)$ respectively. *Wouldn't it be nice if* $|a_{33}|$ *was the largest entry in* $A(3{:}6,3)$? That would ensure that all the multipliers are less than or equal to 1. This suggests that at the beginning of the kth step we swap row k and row q, where it is assumed that a_{qk} has the largest absolute value of any entry in $A(k{:}n,k)$. When we emerge from this process we will have in hand the LU factorization of a row permuted version of A:

```
  function [L,U,piv] = GEpiv(A)
% [L,U,piv] = GE(A)
% The LU factorization with partial pivoting. If A is n-by-n, then
% piv is a permutation of the vector 1:n and L is unit lower triangular
% and U is upper triangular so A(piv,:) = LU. |L(i,j)|<=1 for all i and j.
[n,n] = size(A);
piv = 1:n;
for k=1:n-1
   [maxv,r] = max(abs(A(k:n,k)));
   q = r+k-1;
   piv([k q]) = piv([q k]);
   A([k q],:) = A([q k],:);
   if A(k,k) ~= 0
      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
end
L = eye(n,n) + tril(A,-1);
U = triu(A);
```

A number of details need to be discussed. Applied to a vector, the MATLAB `max` function returns the largest entry and its index. Thus, the preceding max computation assigns the largest of the numbers $|a_{kk}|,\ldots,|a_{nk}|$ to `maxv` and its index to `r`. However, the rth index of the length $n-k+1$ *vector* $A(k{:}n,n)$ identifies an entry from row $r+k-1$ of A. Thus, to pick up the right row index, we need the adjustment `q = r+k-1`. With q so defined, rows q and k are swapped. Note that by swapping *all* of these two rows, some of the earlier multipliers are swapped. This ensures that the multipliers used in the elimination of unknowns from a given equation "stay with" that equation as it is reindexed.

The last issue to address concerns the recording of the interchanges. An integer vector `piv(1:n)` is used. It is initialized to 1:n. Every time rows are swapped in `A`, the corresponding entries in `piv` are swapped. Upon termination, `piv(k)` is the index of the equation that is now

the kth equation in the permuted system. The integer vector `piv` is a representation of a *permutation matrix* P. A permutation matrix is obtained by reordering the rows of the identity matrix. If `piv = [ 3 1 5 4 2]`, then it represents the permutation

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Observe that

$$P \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} = \begin{bmatrix} b_3 \\ b_1 \\ b_5 \\ b_4 \\ b_2 \end{bmatrix} = b(piv).$$

Likewise, $PA = A(piv, :)$. Thus, `GEpiv` computes a factorization of the form $PA = LU$. To solve a linear system $Ax = b$ using `GEpiv`, we notice that x also satisfies $(PA)x = (Pb)$. Thus, if $PA = LU$, $Ly = Pb$, and $Ux = y$, then $Ax = b$. Using the `piv` representation, the three-step process takes the following form

```
[L,U,piv] = GEpiv(A);
y = LTriSol(L,b(piv));
x = UTriSol(U,y);
```

and $2n^3/3 + n^2 + n^2 \approx 2n^3/3$ flops are required. Thus, the factorization dominates the overall computation for large n. Moreover, the pivoting amounts to an $O(n^2)$ overhead and so the stabilization purchased by the row swapping does not seriously affect flop-efficiency.

The script `ShowGEpiv` steps through the factorization applied to the matrix

$$A = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix},$$

producing

$$L = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 0 \\ 0.7391 & 1.0000 & 0 & 0 & 0 \\ 0.1739 & 0.2527 & 1.0000 & 0 & 0 \\ 0.4348 & 0.4839 & 0.7231 & 1.0000 & 0 \\ 0.4783 & 0.7687 & 0.5164 & 0.9231 & 1.0000 \end{bmatrix}$$

$$U = \begin{bmatrix} 23.0000 & 5.0000 & 7.0000 & 14.0000 & 16.0000 \\ 0 & 20.3043 & -4.1739 & -2.3478 & 3.1739 \\ 0 & 0 & 24.8608 & -2.8908 & -1.0921 \\ 0 & 0 & 0 & 19.6512 & 18.9793 \\ 0 & 0 & 0 & 0 & -22.2222 \end{bmatrix}$$

and $piv = \begin{bmatrix} 2 & 1 & 5 & 3 & 4 \end{bmatrix}$. Notice that entries in L are all ≤ 1 in absolute value.[1]

6.3.5 The LU Mentality

It is important to interpret formulas that involve the inverse of a matrix in terms of the LU factorization. Consider the problem of computing

$$\alpha = c^T A^{-1} d,$$

where $A \in \mathbb{R}^{n \times n}$ is nonsingular and $c, d \in \mathbb{R}^n$. Note that α is a scalar because it is the consequence of a dot product between the vectors c and $A^{-1}d$. At first glance, it looks like we actually need to compute the inverse of A. But the preferred approach is to recognize that $A^{-1}d$ is the solution to a linear system $Ax = d$ and so

```
[L,U,piv] = GEpiv(A);
y = LTriSol(L,d(piv));
x = UTriSol(U,y);
alpha = c'*x;
```

This is more efficient by a factor of 2 because the explicit formation of A^{-1} requires about $4n^3/3$ flops.

But a more dramatic payoff for "thinking LU" arises when several linear systems must be solved that involve the same matrix of coefficients. For example, suppose $v^{(0)}$ is a given n-vector and that for $j = 1{:}k$ we want to compute the solution $v^{(j)}$ to the system $Av = v^{(j-1)}$. For example, if $k = 4$ we need to solve the systems

$$\begin{aligned} Av^{(1)} &= v^{(0)} \\ Av^{(2)} &= v^{(1)} \\ Av^{(3)} &= v^{(2)} \\ Av^{(4)} &= v^{(3)} \end{aligned}$$

If we agree to assemble these solutions column by column in a matrix `V`, we obtain

```
b = v0;
V = zeros(n,k);
for j=1:k
   [L,U,piv] = GEpiv(A);
   y = LTriSol(L,b(piv));
   V(:,k) = UTriSol(U,y);
   b = V(:,k);
end
```

[1] The pivoting strategy that we have outlined does not *guarantee* that the entries in U are small. However, in practice the algorithm is extremely reliable.

This approach requires $k(2n^3/3)$ flops since there are k applications of `GEpiv`. On the other hand, why repeat all the Gaussian elimination operations on A every time through the loop, since they are independent of k? A much more efficient approach is to factor A once and then "live off factors" as the linear systems come in the door:

```
b = v0;
V = zeros(n,k);
[L,U,piv] = GEpiv(A);
for j=1:k
   y = LTriSol(L,b(piv));
   V(:,k) = UTriSol(U,y);
   b = V(:,k);
end
```

This implementation involves $(2n^3/3 + O(kn^2))$ flops.

6.3.6 The MATLAB Linear System Tools

The built-in function `LU` can also be used to compute the $PA = LU$ factorization. A call of the form

```
[L,U,P] = LU(A)
```

returns the lower triangular factor in `L`, the upper triangular factor in `U`, and an explicit representation of the permutation matrix in `P`. The command `piv = P*(1:n)'` assigns to `piv` the same integer vector representation of P that is used in `GEpiv`. Regarding the \ operation, `x = A\b` produces the same `x` as

```
[L,U,P] = LU(A);
y = LTriSol(L,P*b);
x = UTriSol(U,y);
```

The \ operator is "smart enough" to exploit triangular structure and so the last two steps can be carried out with equal efficiency via

```
y = L\(P*b);
x = U\y;
```

Moreover, the \ operator "honors" general sparsity if the matrix of coefficients is established as a sparse array. Suppose A is tridiagonal but with nonzeros in the first columnn and row. Here is a script that reports the number of flops required to solve $Ax = b$ via \ for the cases when A is a regular array and when A is a sparse array:

```
% Script File: ShowSparseSolve
% Illustrates how the \ operator can exploit sparsity
clc
disp('        n     Flops Full    Flops Sparse ')
disp('-----------------------------------------')
```

```
for n=[25 50 100 200 400 800]
   T = randn(n,n)+1000*eye(n,n);
   T = triu(tril(T,1),-1); T(:,1) = .1; T(1,:) = .1;
   b = randn(n,1);
   flops(0); x = T\b; fullFlops   = flops;
   T_sparse = sparse(T);
   flops(0); x = T_sparse\b; sparseFlops = flops;
   disp(sprintf('%10d  %10d  %10d ',n,fullFlops,sparseFlops))
end
```

The results show the number of flops required grows linearly with n if A is a sparse array and cubically with n otherwise:

n	Flops Full	Flops Sparse
25	14950	927
50	101150	1878
100	737300	3797
200	5614600	7652
400	43789200	15328
800	345818400	30697

Problems

P6.3.1 Use `GEpiv`, `LTriSol`, and `UTriSol` to compute $X \in \mathbb{R}^{n\times p}$ so that $AX = B$, where $A \in \mathbb{R}^{n\times n}$ is nonsingular and $B \in \mathbb{R}^{n\times p}$.

P6.3.2 Go into `ShowGEpiv` and modify the interchange strategy so that in the kth step, no interchange is performed if the current $|A(k,k)|$ is greater than or equal to largest value in $\alpha|A(k{:}n,k)|$, where $0 < \alpha \leq 1$. If this is not the case, the largest value in $A(k{:}n,k)$ should be swapped into the (k,k) position. Try different values of α and observe the effect.

P6.3.3 How could `GEpiv`, `LTriSol`, and `UTriSol` be used to compute the (i,i) entry of A^{-1}?

P6.3.4 Assume that A and C are given n-by-n matrices and that A is nonsingular. Assume that g and h are given n-by-1 vectors. Write a MATLAB script that computes n-vectors y and z so that both of the following equations hold:

$$\begin{aligned} A^T y + Cz &= g \\ Az &= h. \end{aligned}$$

You may use the \ operator. Efficiency matters.

P6.3.5 Assume that A, B, and C are nonsingular n-by-n matrices and that f is an n-by-1 vector. Write an efficient MATLAB fragment that computes a vector x so that $ABCx = f$.

P6.3.6 Assume that `A` is a given n-by-n nonsingular matrix and that `b` and `c` are given column n-vectors. Write a MATLAB script that computes a scalar α (if possible) so that the solution to $Ax = b + \alpha c$ satisfies $x(1) = 0$. Make effective use of `GEpiv`, `LTriSol` and `UTriSol`. If it is not possible to chose α so that $x(1) = 0$, then the script should print the message "impossible".

P6.3.7 Write a MATLAB function `[U,L] = UL(A)` that computes the "UL" factorization, e.g.,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ 0 & \ell_{32} & \ell_{33} \end{bmatrix}.$$

You may ignore the possibility of division by zero.

P6.3.8 Complete the following function

```
  function t = Intersect(P,Q,R,S)
% t = Intersect(P,Q,R,S)
% P, Q, R, and S are column 3-vectors.
% and t is a scalar with the property that P + t*Q is a linear combination
% of R and S.
```

You may assume that such a t exists and you may use the \ operator.

P6.3.9 Implement the following function:

```
  function [u,v] = Lines(a,b,c,d)
%
% a,b,c, and d are column n-vectors with the property that for i=1:n, the lines
%
%             x(t1) =  a(i) + t1(b(i) - a(i))       -inf < t1 < inf
%
%             y(t2) =  c(i) + t2(d(i) - c(i))       -inf < t2 < inf
%
% intersect. The points of intersection (u(i),v(i)) are returned in
% column n-vectors u and v.
%
```

Your implementation should use Gaussian Elimination with partial pivoting to solve for the intersection points. It should be fully vectorized. (No loops necessary.)

P6.3.10 Suppose a point on the unit sphere has latitude ϕ and longitude θ. Then the point has Cartesian coordinates $(\cos(\theta)\cos(\phi), -\sin(\theta)\cos(\phi), \sin(\phi)$ and the tangent plane to the sphere at that point is given by

$$\cos(\theta)\cos(\phi)x \; - \; \sin(\theta)\cos(\phi)y \; + \; \sin(\phi)z = 1.$$

Implement the following function

```
  function P = Tetra(Lat,Long)
%
% Lat and Long are column 4-vectors that specify the latitude and longitude
% (in degrees) of four distinct points Q1, Q2, Q3, Q4 on the unit sphere.
%
% Let T1, T2, T3 and T4  be the tangent planes to the unit sphere at these points.
%
% Assume that each possible triplet of these planes intersects at a point.
% P is a 4-by-3 matrix  whose rows are these intersection points.
```

You may use the backslash operator "\". Hint: You should think about how to solve a linear system whose matrix of coefficients is a permutation matrix.

P6.3.11 Suppose `n` is a given positive integer and that `d` is a given n-by-1 vector, `B` is a given n-by-n matrix, and `A` is a given nonsingular n-by-n matrix. Write an efficient MATLAB script that computes the scalar z defined by

$$z = d^T B^T A^{-1} B d.$$

Make effective use of the Matlab function `[P,L,U] = LU(A)` and the triangular system solvers `UTriSol(U,b)` and `LTriSol(L,b)`.

6.4 Analysis

We now turn our attention to the quality of the computed solution produced by Gaussian elimination with partial pivoting.

6.4.1 Residual Versus Error

Consider the following innocuous linear system:

$$\begin{bmatrix} .780 & .563 \\ .913 & .659 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} .217 \\ .254 \end{bmatrix}.$$

Suppose we apply two different methods and get two different solutions:

$$x^{(1)} = \begin{bmatrix} .341 \\ -.087 \end{bmatrix} \qquad x^{(2)} = \begin{bmatrix} .999 \\ -1.00 \end{bmatrix}.$$

Which is preferred? An obvious way to compare the two solutions is to compute the associated *residuals*:

$$b - Ax^{(1)} = \begin{bmatrix} .000001 \\ 0 \end{bmatrix} \qquad b - Ax^{(2)} = \begin{bmatrix} .000780 \\ .000913 \end{bmatrix}.$$

On the basis of residuals, it is clear that $x^{(1)}$ is preferred. However, it is easy to verify that the exact solution is given by

$$x^{(exact)} = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

and this creates a dilemma. We see that $x^{(1)}$ renders a small residual while $x^{(2)}$ is much more accurate.

Reasoning in the face of such a dichotomy requires an understanding about the context in which the linear system arises. We may be in a situation where how well Ax predicts b is paramount. In this case small residuals are important. In other settings, accuracy is critical and the focus is on nearness to the true solution.

It is clear from the discussion that (1) the notion of a "good" solution can be ambiguous and (2) the intelligent appraisal of algorithms requires a sharper understanding of the mathematics behind the $Ax = b$ problem.

6.4.2 Problem Sensitivity and Nearness

In the preceding 2-by-2 problem, the matrix A is very close to singular in the sense that

$$\tilde{A} = \begin{bmatrix} .780 & .563001095\ldots \\ .913 & .659 \end{bmatrix}$$

is exactly singular. Thus an $O(10^{-6})$ perturbation of the data renders our problem insoluble. Our intuition tells us that difficulties should arise if our given $Ax = b$ problem is "near" to a singular $Ax = b$ problem. In that case we suspect that small changes in the problem data (i.e., A and b) will induce relatively large changes in the solution. It is clear that we need to quantify such notions as "nearness to singularity" and "problem sensitivity."

We remark that these issues have *nothing* to do with the underlying algorithms. They are mathematical concepts associated with the $Ax = b$ problem. However, these concepts do clarify what we can expect from an algorithm in light of rounding errors.

To appreciate this point, consider the hypothetical situation in which there is *no* roundoff during the entire solution process except when A and b are stored. This means from Theorem 5 that the computed solution $\hat{x}$ satisfies the perturbed linear system

$$(A + E)\hat{x} = b + f, \tag{6.5}$$

where $\| E \|_1 \leq \mathtt{eps} \| A \|_1$ and $\| f \|_1 \leq \mathtt{eps} \| b \|_1$. Two fundamental questions arise: How can we guarantee that $A + E$ is nonsingular, and how close is $\hat{x}$ to the true solution x? The answer to both of these questions involves the quantity

$$\kappa_1(A) \;=\; \| A \|_1 \| A^{-1} \|_1$$

which is called the *condition number* of A (in the 1-norm). It can be shown that

- $\kappa_1(A) \geq 1$.
- $\kappa_1(\alpha A) = \kappa_1(A)$.
- $\kappa_1(A)$ is large if A is close to singular.

The last point is affirmed by the preceding 2-by-2 example.

$$A = \begin{bmatrix} .780 & .563 \\ .913 & .659 \end{bmatrix} \;\Rightarrow\; A^{-1} = 10^6 \begin{bmatrix} .659 & -.563 \\ -.913 & .780 \end{bmatrix} \;\Rightarrow\; \kappa_1(A) \approx 10^6$$

The following theorem uses the condition number to describe properties of the stored linear system (6.5).

Theorem 6 *Suppose $A \in \mathbb{R}^{n\times n}$ is nonsingular and that $Ax = b$, where $x, b \in \mathbb{R}^n$. If*

$$\mathtt{eps}\;\; \kappa_1(A) < 1,$$

then the stored linear system $\mathrm{fl}(A)\hat{x} = \mathrm{fl}(b)$ *is nonsingular and*

$$\frac{\| \hat{x} - x \|_1}{\| x \|_1} \leq \mathtt{eps}\;\; \kappa_1(A) \left(1 + \frac{\| \hat{x} \|_1}{\| x \|_1} \right).$$

Proof From Theorem 5 we know that $\mathrm{fl}(A) = A+E$ and $\mathrm{fl}(b) = b+f$, where $\| E \|_1 \leq \;\mathtt{eps}\;\; \| A \|_1$ and $\| f \|_1 \leq \mathtt{eps}\;\; \| b \|_1$. If $A+E$ is singular, then there exists a nonzero vector z so $(A+E)z = 0$. This implies that $z = -A^{-1}Ez$, and so

$$\| z \|_1 = \| A^{-1}Ez \|_1 \leq \| A^{-1} \|_1 \| E \|_1 \| z \|_1.$$

But this contradicts the assumption that $\mathtt{eps}\cdot\kappa_1(A) < 1$ and shows that $(A + E)\hat{x} = b + f$ is nonsingular system. Since $Ax = b$, it follows that $A(\hat{x}-x) = f-E\hat{x}$ and so $\hat{x}-x = A^{-1}f-A^{-1}E\hat{x}$. Thus,

$$\begin{aligned} \| \hat{x} - x \|_1 &\leq \| A^{-1} \|_1 \| f \|_1 + \| A^{-1} \|_1 \| E \|_1 \| \hat{x} \|_1 \\ &\leq \mathtt{eps}\;\; \| A^{-1} \|_1 \| b \|_1 + \;\mathtt{eps}\;\; \| A \|_1 \| A^{-1} \|_1 \| \hat{x} \|_1. \end{aligned}$$

The theorem is established by dividing both sides by $\| x \|_1$ and observing that

$$\frac{\| b \|_1}{\| x \|_1} = \frac{\| Ax \|_1}{\| x \|_1} \leq \frac{\| A \|_1 \| x \|_1}{\| x \|_1} = \| A \|_1. \quad \square$$

There are a number of things to say about this result. To begin with, the factor $\tau = \texttt{eps} \cdot \kappa_1(A)$ has a key role to play. If this quantity is close to 1, then it is appropriate to think of A as *numerically singular*. If A is not numerically singular, then it is possible to show that the quotient $\| \hat{x} \|_1 / \| x \|_1$ contributes little to the upper bound. The main contribution of the theorem is thus to say that the 1-norm relative error in $\hat{x}$ is essentially bounded by τ.

In our discussion of $Ax = b$ sensitivity, we have been using the 1-norm. Condition numbers can also be defined in terms of the other norms mentioned in §5.2.4 on page 184. The 2-norm condition number can be computed, at some cost, using the function `cond`. A cheap estimate of $\kappa_1(A)$ can be obtained with `condest`.

6.4.3 Backward Stability

It turns out that Gaussian elimination with pivoting produces a computed solution $\hat{x}$ that satisfies

$$(A + E)\hat{x} = b,$$

where

$$\| E \| \approx \texttt{eps} \| A \|.$$

(It does not really matter which of the preceding norms we use, so the subscript on the norm symbol has been deleted.) This says that the Gaussian elimination solution is essentially as good as the "ideal" solution that we discussed earlier (i.e., $\hat{x}$ solves a nearby problem exactly).

Two important heuristics follow from this:

$$\| A\hat{x} - b \| \approx \texttt{eps} \ \| A \| \ \| \hat{x} \| \tag{6.6}$$

$$\frac{\| \hat{x} - x \|}{\| x \|} \approx \texttt{eps} \, \kappa(A) \tag{6.7}$$

The first essentially says that the algorithm produces small residuals compared to the size of A and $\hat{x}$. The second heuristic says that if the unit roundoff and condition satisfy $\texttt{eps} \approx 10^{-t}$, and $\kappa_1(A) \approx 10^p$, then (roughly) $\tilde{x}$ has approximately $t - p$ correct digits.

The function `GE2` can be used to illustrate these results. It solves a given 2-by-2 linear system in simulated three-digit arithmetic. Here is a sample result:

```
Stored A   =   .981x10^0   .726x10^0
               .529x10^0   .384x10^0

Computed L =   .100x10^1   .000x10^0
               .539x10^0   .100x10^1

Computed U =   .981x10^0   .726x10^0
               .000x10^0  -.700x10^-2
```

```
Exact b           =  .255x10^0
                     .145x10^0

Exact Solution    =  .100x10^1
                    -.100x10^1

Computed Solution =  .110x10^1
                    -.114x10^1

cond(A) = 2.608e+02

Computed solution solves (A+E)*x = b where

               E =      0.001552    -0.001608
                        0.000377    -0.000391
```

Note that in this environment, the unit roundoff `eps` is approximately 10^{-3}. The script file `ShowGE2` can be used to examine further examples.

The Pascal matrices provide another interesting source of test problems. `Pascal(n)` returns the n-by-n Pascal matrix, which has integer entries and is increasingly ill conditioned as n grows:

n	Condition of P_n
4	6.9e+02
8	2.0e+07
12	8.7e+11
16	4.2e+16

The script file `CondEgs` examines what happens when Gaussian elimination with pivoting is applied to a sequence of Pascal linear systems that are set up to have the vector of all ones as solution. Here is the output for $n = 12$:

```
cond(pascal(12)) = 8.7639e+11
True solution is vector of ones.
x =
   0.99999998079317
   1.00000022061699
   0.99999887633133
   1.00000337593939
   0.99999331477863       Relative error = 4.7636e-06
   1.00000919709849       Predicted value = EPS*cond(A) = 1.9460e-04
   0.99999100757082
   1.00000625939438
   0.99999695671209
   1.00000098506337
   0.99999980884814
   1.00000001685318
```

Notice that $-\log_{10}(\texttt{eps}\kappa(P_n))$ provides a reasonable upper bound for the number of correct significant digits in the computed solution.

Another nice feature of the Pascal matrices is that they have determinant 1 for any n. The preceding discussion confirms the *irrelevance* of the determinant in matters of linear system sensitivity and accuracy. The determinant usually figures quite heavily in any introduction to linear algebra. For example,

$$\det(A) = 0 \iff A \text{ singular.}$$

It is natural to think that

$$\det(A) \approx 0 \iff A \text{ approximately singular.}$$

However, this is not the case. The Pascal example shows that nearly singular matrices can have determinant 1. Diagonal matrices of the form $D = \alpha I_n$ have unit condition number but determinant α^n, further weakening the correlation between determinant size and condition.

Problems

P6.4.1 Gaussian elimination with pivoting is used to solve a 2-by-2 system $Ax = b$ on a computer with machine precision 10^{-16}. It is known that $\| A \|_1 \| A^{-1} \|_1 \approx 10^{10}$ and that the exact solution is given by

$$x = \begin{bmatrix} 1.234567890123456 \\ \\ .0000123456789012 \end{bmatrix}.$$

Underline the digits in x_1 and x_2 that can probably agree with the corresponding digits in the computed solution. Explain the heuristic assumptions used to answer the question.

P6.4.2 It is known that the components of the exact solution to a linear system $Ax = b$ range from 10^{-1} to 10^3, and that $cond(A)$ is about 10^4. What must the machine precision be in order to ensure that the smallest component of `x = A\b` has at least five significant digits of accuracy? No proof is necessary, just a reasonable heuristic argument.

P6.4.3 For n = 2:12, use `GEPiv`, `LTriSol`, and `UtriSol` to compute $d(1{:}n)$, the diagonal of the inverse of the n-by-n Hilbert matrix. The built-in Matlab function `hilb(n)` can be used to set up these matrices. The exact inverse is obtainable via `invhilb(n)`. Print a table that reports $\| d - d_{exact} \|_2 / \| d_{exact} \|_2$ for each n-value and the condition of `hilb(n)`. Submit output and the script used to produce it.

P6.4.4 On a computer with `EPS` = 10^{-6}, `x = A\b` has no correct digits. What can you say about the number of correct significant digits when the same calculation is carried out on a computer with `EPS` = 10^{-16}?

M-Files and References

Script Files

`ShowTri`	Uses `LTriSol` to get inverse of Forsythe Matrix.
`ShowTriD`	Illustrates tridiagonal system solving.
`ShowHessLU`	Illustrates Hessenberg LU factorization.
`ShowGE`	Illustrates Gaussian Elimination.
`ShowSparseSolve`	Examines \ with sparse arrays.
`NoPivot`	Illustrates the dangers of no pivoting.
`ShowGEPiv`	Illustrates `GEPiv`.
`ShowGE2`	Applies `GE2` to three different examples.
`CondEgs`	Accuracy of some ill-conditioned Pascal linear systems.

Function Files

`LTriSol`	Solves lower triangular system $Lx = b$.
`UTriSol`	Solves upper triangular system $Ux = b$.
`LTriSolM`	Solves multiple right-hand-side lower triangular systems.
`TriDiLU`	LU factorization of a tridiagonal matrix.
`LBiDiSol`	Solves lower bidiagonal systems.
`UBiDiSol`	Solves upper bidiagonal systems.
`HessLU`	Hessenberg LU factorization.
`GE`	General LU factorization without pivoting.
`GEpiv`	General LU factorization with pivoting.
`GE2`	Illustrates 2-by-2 GE in three-digit arithmetic.

References

G.E. Forsythe and C. Moler (1967). *Computer Solution of Linear Algebraic Systems,* Prentice-Hall, Englewood Cliffs, NJ.

W.W. Hager (1988). *Applied Numerical Linear Algebra,* Prentice-Hall, Englewood Cliffs, NJ.

G.W. Stewart (1973). *Introduction to Matrix Computations,* Academic Press, New York.

G.H. Golub and C.F. Van Loan (1996). *Matrix Computations, Third Edition*, Johns Hopkins University Press, Baltimore, MD.