4. SYSTEMS OF LINEAR ALGEBRAIC EQUATIONS (Direct Methods)

4.1 INTRODUCTION

"The problem of solving a linear system A·x = b is central in scientific computation" ([p. 87] Golub and van Loan 1997).

"It is important to note that some methods which are perfectly acceptable for theoretical use, may be useless for the numerical solution. A prime example is the explicit determinant formula (Cramer's rule) for the inverse matrix and for the solution of linear systems of equations." ([p. 1] Dahlquist & Björck 2008).

"The solution of systems of linear equations forms the cornerstone of many numerical methods for solving a wide variety of practical computational problems, and, consequently, it is imperative that we be able to solve such systems accurately and efficiently" ([p. 50] Heath 2002).

Solving systems of *n* linear algebraic equations with *n* unknowns is a *core* topic in computational physics and engineering. Linear systems are found in all areas of computational science. Discrete systems, such as static structures or electric circuits, can be investigated *directly* by systems of linear equations; continuous systems, represented by differential equations, in turn can modeled by finite-difference or finite-element techniques which themselves can be expressed as systems of linear equations ([p. 31], Kiusalaas 2005).

Assume that the linear system A·x = b, represents a linear system where the matrix A is real (versus complex), nonsingular and square (n-by-n).

Two types of algorithms exist for solving systems of linear equations, direct and iterative methods. Direct methods in the absence of floating-point round-off error would supply an *exact* solution to A·x = b after a finite number of steps; these algorithms are all variants on Gaussian elimination ([p. 31] Demmel 1997). Iterative methods begin with a trial solution, x_0 and then compute a sequence of solution vectors, x_1, x_2, x_3, of ever more accurate estimates to A·x = b, until the solution meets a predefined convergence criteria ([p. 31] Demmel 1997). Depending on the structure of the particular matrix *A* and the rate at which an iterative solution converges either a direct method or iterative approach may be faster or more accurate, consequently, in the absence of special knowledge of the matrix *A*, Demmel (1997) recommends that a direct method be the method of choice. Similarly, Golub and van Loan ([p. 87] 1997) observe that Gaussian elimination is the algorithm of choice when A is square, dense and unstructured. Consequently, direct methods, based on variants of Gaussian elimination, will be addressed next.

Creating an efficient Gaussian elimination algorithm comes from the following concept. A linear system can be solved by transforming it to a similar system that possesses a solution identical to that of the original, but the solution of the transformed system is much easier to calculate ([p. 63] Heath, 2002). Well what linear systems are easy to solve?

4.2 SOLVING SIMPLE LINEAR SYSTEMS

"We motivate the discussion of Gaussian elimination by discussing the ease with which triangular systems can be solved" ([p. 87] Golub and van Loan 1997).

4.2.1 Upper Triangular Matrix

Triangular linear systems play a fundamental role in matrix computation. Many solution algorithms are built on the idea of reducing a problem to a solution of one or more triangular systems, including *virtually* all direct methods for solving general linear systems ([p. 140] Higham 2002).

$$
\begin{aligned}
+x_1 + 2x_2 + 2x_3 + 3x_4 &= 3\\
+ 4x_2 + 6x_3 + 5x_4 &= 6\\
+ 1x_3 + 7x_4 &= 1\\
+ 9x_4 &= 9
\end{aligned}
$$

$$U \bullet x = b \Rightarrow$$

$$U \bullet x = \begin{bmatrix} 1 & 2 & 2 & 3 \\ 0 & 4 & 6 & 5 \\ 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 9 \end{bmatrix} \bullet \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 1 \\ 9 \end{bmatrix}$$

```
>> U = [1 2 2 3; 0 4 6 5; 0 0 1 7; 0 0 0 9]
U =     1      2      2      3
        0      4      6      5
        0      0      1      7
        0      0      0      9
>> b = [3; 6; 1; 9]
b =     3
        6
        1
        9
>> x = zeros(size(b));    % Use later
>> n = length(b);
```

One solves such systems via a method, termed *back substitution*, starting with the last row *first* and working backwards to the first row.

4th row:

$$+9x_4 = +9$$
$$x_4 = +1$$
$$x_4 = b_4/U_{44}$$

```
>> x(4)  = b(4)/U(4,4);
>> x(4)
ans = 1
```

3rd row:

$$+1x_3 + 7x_4 = +1,$$
$$x_3 = (-7x_4 + 1)/1 = (+1 - 7)/1 = -6$$
$$x_3 = (b_3 - U_{34}x_4)/U_{33}$$

```
>> x(3)  =  (b(3)  - U(3,  4)*x(4))/U(3,  3);
>> x(3)
ans =   -6
```

2nd row:

$4x_2 + 6x_3 + 5x_4 = +6,$

$x_2 = (+6 - 6x_3 - 5x_4)/4 = (+6 + 36 - 5)/4 = 9.25$

$x_2 = (b_3 - U_{23}x_3 - U_{24}x_4)/U_{22}$

```
>> x(2) = (b(2) -U(2,4)*x(4) -U(2, 3)*x(3))/U(2, 2);
>> x(2)
ans =  9.25
```

1st row:

$+1x_1 + 2x_2 + 2x_3 + 3x_4 = 3,$

$x_1 = (3 - 2x_2 - 2x_3 - 3x_4)/1,$

$x_1 = (3 - 2(9.25) - 2(-6) - 3(1))/1 = -6.5$

$x_1 = (b_1 - U_{12}x_2 - U_{13}x_3 - U_{14}u_4)/U_{11}$

```
>> x(1) = (b(1) -U(1, 2)*x(2) -U(1,3)*x(3) -U(1,4)*x(4))/U(1,1);
>> x(1)
ans = -6.5
```

Collecting all the x calculations -

```
>> x(4) = (b(4))/U(4,4);
>> x(3) = (b(3) -U(3, 4)*x(4))/U(3, 3);
>> x(2) = (b(2) -U(2, 3)*x(3) -U(2, 4)*x(4))/U(2,2);
>> x(1) = (b(1) -U(1, 2)*x(2) -U(1, 3)*x(3) -U(1, 4)*x(4))/U(1,1);
```

Inspect **`x(1).`** Setting i = 1, the contribution of U(i, j)·x(j), sums over j = 2 to 4. For **`x(2),`** setting i = 2, U(i, j)·x(j), sums over j = 3 to 4. Consequently, with j equal to i + 1 to n, U(i, j)·x(j) terms can be expressed as either a for-loop construct,

```
partial_sum = 0;
 for j = (i + 1):n
    partial_sum = partial_sum + U(i, j)*x(j);
 end
```

or employing an array section, `U(i, i + 1:n)*x(i + 1:n),` which leads to the solution algorithm below, **`back_sub.m`**, which uses an array section and a *single* for loop -

```
function x = back_sub(U, b)

n     = size(U, 1);
x     = zeros(size(b));

x(n)  = b(n)/U(n, n);
  for i = n - 1 :-1:1
    if U(i, i) == 0.0;
      x(i) = inf; disp('back_sub: U is singular');
    end
    x(i) = (b(i) - U(i, i + 1:n)*x(i + 1:n) )/U(i, i);
  end
```

```
>> U = [1 2 2 3; 0 4 6 5; 0 0 1 7; 0 0 0 9];
>> b = [3; 6; 1; 9];
>> x = back_sub(U, b)
x = -6.5000
    +9.2500
```

```
        -6.0000
        +1.0000
```

Observe that the above implementation of **`back_sub.m`** follows that of algorithm 3.1.2 of Golub and van Loan ([p. 89] 1996). Note, if a diagonal element of U is zero, U must be singular (e.g. [p. 83] Heath 2002), and the execution of **`back_sub`** is halted and an error message is displayed. The solution to an upper triangular linear system requires n^2 arithmetic operations (e.g. [p. 89] Golub and van Loan 1996)

4.2.2 Lower Triangular Matrix

$$L \bullet x = b,$$

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

```
>> L = [2 0 0; 1 5 0; 7 9 8]
L =  2     0     0
     1     5     0
     7     9     8
>> b = [6; 2; 5]
b =  6
     2
     5
>> n = length(b);
>> x = zeros(size(b));
```

One solves these systems via a method, termed *forward substitution*, similar to back substitution, discussed above. Starting with the *first* row and working to the last row.

1st row:

$2x_1 = 6,$

$x_1 = 3.$

$x_1 = b_1/L_{11}$

```
>> x(1) = b(1)/L(1, 1);
>> x(1)
ans = 3
```

2nd row:

$1x_1 + 5x_2 = 2,$

$x_2 = (2 - 1x_1)/5 = (2 - 13)/5 = -0.2.$

$x_2 = (b_2 - L_{12}x_1)/L_{22}$

```
>> x(2) = (b(2) - L(2, 1)*x(1))/L(2, 2);
>> x(2)
ans = -0.2
```

3rd row:

$7x_1 + 9x_2 + 8x_3 = 5,$

$x_3 = (5 - 7x_1 - 9x_2)/8$

$x_3 = (5 - 7*3 - 9(-0.2))/8 = -1.775$

$x_3 = (b_3 - L_{31}x_1 - L_{32}x_2)/L_{33}$

```
>> x(3) = (b(3) - L(3, 1)*x(1) - L(3, 2)*x(2))/L(3, 3);
>> x(3)
ans =  -1.775
```

Collecting all solutions -

```
>> x(1) = b(1)/L(1, 1);
>> x(2) = (b(2) - L(2, 1)*x(1))/L(2, 2);
>> x(3) = (b(3) - L(3, 1)*x(1) - L(3, 2)*x(2))/L(3, 3);
```

Inspect **x(3).** If one sets i = 3, the contribution of the L(i, j)·x(j) terms sums over j from 1 to 2, or j from 1 to (i - 1), Thus the contribution of these L·x terms can be expressed as for-loop construct

```
partial_sum = 0;
  for j = 1:(i - 1)
      partial_sum = partial_sum + L(i, j)*x(j);
   end
```

or as an array section, `L(i, 1:(i - 1))*x(1:(i - 1)).` This leads to a simple *forward-substitution* algorithm -

```
function x = forward_sub(L, b)

n     = size(L, 1);
x     = zeros(size(b));

x(1) = b(1)/L(1, 1);

  for i = 2:n
      if L(i, i) == 0.0;
        x(i) = inf; disp('forward_sub: L is singular');
     end
    x(i) = (b(i) - L(i, 1:i - 1)*x(1:i - 1))/L(i, i);
  end
```

```
>> L = [2 0 0; 1 5 0; 7 9 8];
>> b = [6; 2; 5];
>> x = forward_sub(L, b)
x =    +3.0000
       -0.2000
       -1.7750
```

Observe that the above implementation of **forward_sub.m** follows that of algorithm 3.1.1 of Golub and van Loan ([p. 89] 1996). If a diagonal element of L is zero, L must be singular (e.g. [p. 83] Heath 2002), and the execution of **forward_sub** is halted and an error message is displayed. The solution to an lower triangular linear system requires n^2 arithmetic operations (e.g. [p. 89] Golub and van Loan 1996).

4.3 SOLVING LINEAR SYSTEMS VIA GAUSSIAN ELMINATION (GE)

The strategy of the Gaussian elimination (GE) algorithm is to reduce a problem we cannot solve (a full linear system) to one which we can solve (a triangular system) employing elementary row operations ([p.

158] Higham 2002). The row[1] transformation operations which leave solutions unchanged are (Higham 2002):

- Interchange any two rows of the coefficient matrix, *A*.
- Multiply a row of the coefficient matrix by a nonzero constant.
- Adding a multiple of one row of the coefficient matrix to another row of the same coefficient matrix.

A simple Gaussian elimination algorithm will be derived employing the following concrete example:

```
>> A = [2   1  1  0; 4  3  3  1; 8  7  9  5; 6  7  9   8]
A =    2     1     1     0
       4     3     3     1
       8     7     9     5
       6     7     9     8
>> b = [1; 4; 6; 1]';
>> n = size(A, 1);
```

To start one creates an *augmented* matrix with dimensions (n, n + 1), where the first *n* columns are the *A* matrix and the right hand side, *b*, forms the last column:

```
>> A = [A b]
A =     2     1     1     0     1
        4     3     3     1     4
        8     7     9     5     6
        6     7     9     8     1
```

In detail, to zero out the first column below the diagonal requires *t*hree operations:

Add to row$_2$ m times row$_1$;   `% row 1 → k = 1`

```
>> m      = +A(2, 1)/A(1, 1);     % j = 2
>> A(2,:) = +A(2,:) - m*A(1,:);
```

Add to row$_3$ m times row$_1$:

```
>> m      = +A(3, 1)/A(1, 1);     % j = 3
>> A(3,:) = +A(3,:) - m*A(1,:);
```

Add to row$_4$ m times row$_1$:

```
>> m      = +A(4, 1)/A(1, 1);     % j = 4
>> A(4,:) = +A(4,:) - m*A(1,:);
A =     2     1     1     0     1
        0     1     1     1     2
        0     3     5     5     2
        0     4     6     8    -2

% These three operations can be performed by:
k = 1;
  for j = 2:n
    m      = +A(j, k)/A(k, k);
    A(j, :) = A(j,:) - m*A(k, :);
  End
```

To zero out the second column below the diagonal, requires two similar operations:

[1] Columns of *A* can be exchanged provided the corresponding elements of the solution vector, *x*, are exchanged ([p. 40] Dahlquist & Björck 2008).

Add to row_3 m times row_2:   % row_2 k = 2

```
>>m        = +A(3, 2)/A(2, 2);      % j = 3
>>A(3,:) = +A(3,:) - m*A(2,:);
```

Add to row_4 m times row_2:

```
>> m      = +A(4, 2)/A(2, 2);      % j = 4
>> A(4,:)= +A(4,:) - m*A(2,:);
>> A
A =     2     1     1     0     1
        0     1     1     1     2
        0     0     2     2    -4
        0     0     2     4   -10

% These two operations can be performed by:
k = 2;
  for j = 3:n
     m       = +A(j, k)/A(k, k);
     A(j, :) = A(j, :) - m*A(k, :);
  end
```

To zero out the third column below the diagonal, requires only *one* operation:

Add to row_4 m times row_3;   % row_3 $\rightarrow$ k = 3

```
>> m      = +A(4, 3)/A(3, 3);   % j = 4
>> A(4,:) = +A(4,:) - m*A(3,:);
>> A
A =     2     1     1     0     1
        0     1     1     1     2
        0     0     2     2    -4
        0     0     0     2    -6

% This operations can be performed by:
k = 3;
 for j = 4:n
   m       = +A(j, k)/A(k, k);
   A(j, :) = +A(j, :) - m*A(k, :);
 end
```

One extracts A and b from the augmented array:

```
>> b = A(:, n + 1)
b =      1
         2
        -4
        -6
>> A(:, n + 1) = []
A =   2     1     1     0
      0     1     1     1
      0     0     2     2
      0     0     0     2
```

Now one employs `back_sub.m` with the modified A to solve for x:

```
>> x = back_sub(A, b)
x =     -2
        +4
        +1
        -3
```

One encapsulates all these commands below in **`simple_gauss_elim.m`** –

```
function x = simple_gauss_elim(A, b)

n = size(A, 1);
% Augmented matrix
A = [A b];

  for k = 1:n - 1
     for j = k + 1:n
        if A(k, k) == 0.0;
            disp('simple_gauss_elim: A is singular');
            x = nan(size(b)); % return dummy x values
            return;
          end

        m       = +A(j, k)/A(k, k);
        A(j, :) = A(j, :) - m*A(k, :);
     end
  end

% Extract A and from the augmented matrix
b            = A(:, n + 1);
A(:, n + 1) = [];

x            = back_sub(A, b);
```

```
>> A = [2   1  1  0; 4  3  3  1; 8  7  9  5; 6  7  9   8];
>> b = [1; 4; 6; 1];
>> x = simple_gauss_elim(A, b)
x = -2
     4
     1
    -3
>> A*x - b
ans =0
     0
     0
     0
```

If a diagonal element of *A* is zero, then *A* is *viewed* as singular, and execution of **`simple_gauss_elim.m`** is halted and an error message is displayed. Whether or not the matrix A is actually singular will be addressed in the next section.

In solving A·x = b the bulk of the numerical effort is expended in factoring *A* into upper triangular format. This factorization requires ~$2n^3/3$ arithmetic operations; the subsequent solution of A·x = b by back substitution requires only n^2 floating-point operations. However, in modern computations the GE algorithm is employed in a format different from the one developed above. In this GE algorithm the matrix *A* is factored into two coupled triangular matrices - one a lower triangular matrix, *L*, with a unit diagonal (ones along its diagonal) and the other, an upper triangular matrix, *U*.

4.4 SOLVING LINEAR SYSTEMS WITH LU DECOMPOSITION

The LU factorization is a key example of matrix decomposition in numerical linear algebra which has revolutionized the field of matrix computations; matrix decompositional approach has been identified as one the ten most influential algorithms in science and engineering in the 20th century (Stewart 2000; [p. 45] Dahlquist & Björck. 2008). Although, as noted by Dahlquist & Björck, LU factorization is just a different interpretation of GE it offers important benefits (e.g. Stewart 2000):

- A matrix decomposition, which is usually expensive to calculate, can be used again to solve new problems involving the original matrix.
- The decompositional approach facilitates investigations of floating-point round-off errors.
- The decompositional approach often shows that apparently different algorithms are actually calculating the same object.
- Focusing a few decompositional algorithms, instead of a host of specific problems, have allowed software developers to create more efficient matrix packages.

This section follows the presentation of the factorization of a square matrix A into a product of two triangular matrices from chapter 20 of Trefethen and Bau (1997).

Since triangular systems are simple to solve, as seen above, the LU algorithm factors a general square matrix, *A*, into a product of a *lower* triangular matrix, *L*, and an upper triangular matrix, *U*. Then solving the composite system, A·x = L·U·x = b is simple -

$$\begin{aligned} A\cdot x &= b, \\ A &= L\cdot U, \\ L\cdot U\cdot x &= b, \\ L\cdot y &= b, \quad \leftarrow \text{solve via forward substitution,} \\ U\cdot x &= y, \quad \leftarrow \text{solve by backwards substitution.} \end{aligned}$$

Can one factor a general matrix *A* into a product of *L* and *U*? Yes, it turns out that the elements of *L* have *already* been determined in the Gaussian elimination process; these *L* elements are simply the *m* ratios calculated in the Gaussian transformation of *A* into an upper triangular matrix. This is demonstrated by re-examining the previous example:

```
>> A = [2   1  1  0; 4  3  3  1; 8  7  9  5; 6  7  9   8];
>> n = size(A, 1);
>> U = A;                                  % Initialize U
>> L = eye(n, n);                          % Initialize L
```

As noted previously to zero out the first column of *U* below the diagonal requires three operations:

Add to row$_2$ L$_{2,1}$ times row$_1$;

```
                                           % row 1 → k = 1
>> L(2,1) = +U(2, 1)/U(1, 1);              % j = 2
>> U(2,:) = +U(2,:) - L(2,1)*U(1,:);
```

Add to row$_3$ L$_{3,1}$ times row$_1$:

```
>> L(3,1) = +U(3, 1)/U(1, 1);              % j = 3
>> U(3,:) = +U(3,:) - L(3,1)*U(1,:);
```

Add to row$_4$ L$_{4,1}$ times row$_1$:

```
>> L(4,1) = +U(4, 1)/U(1, 1);              % j = 4
>> U(4,:) = +U(4,:) - L(4,1)*U(1,:);
```

```
k = 1; % These three operations can be performed by:
>> for j = 2:n
     L(j, k) = U(j, k)/ U(k, k);
     U(j, :) = U(j, :) - L(j, k)*U(k, :);
   end
```

To zero out the second column of *U* below the diagonal requires two operations:

Add to row$_3$ L$_{3,2}$ times row$_2$: % row$_2$ k = 2

```
>>L(3,2) = +U(3, 2)/U(2, 2);      % j = 3
>>U(3,:) = +U(3,:) -L(3,2)*U(2,:);
```

Add to row$_4$ L$_{4,2}$ times row$_2$:

```
>> L(4,2) = +U(4, 2)/U(2, 2);     % j = 4
>> U(4,:) = +U(4,:) - L(4,2)*U(2,:);
```

```
k = 2;  % These two operations can be performed by:
  for j = 3:n
     L(j, k) = +U(j, k)/U(k, k);
     U(j, :) = U(j, :) - L(j, k)*U(k, :);
  end
```

To zero out the third column of U below the diagonal, requires only *one* operation:

a) Add to row$_4$ L$_{4,3}$ times row$_3$; % row$_3$ → k = 3

```
>> L(4,3) = +U(4, 3)/U(3, 3);   % j = 4
>> U(4,:) = +U(4,:) -L(4,3)*U(3,:);
```

```
k = 3;  % This operations can be performed by:
 for j = 4:n
   L(j, k) = +U(j, k)/U(k, k);
   U(j, :) = +U(j, :) - L(j, k)*U(k, :);
 end
```

A short algorithm can be constructed to encapsulate the above operations -

```
function [L, U] = simple_lu(A)

   n = size(A, 1); L = eye(n, n); U = A;
     for k = 1:n - 1
        for j = k + 1:n
           L(j, k) = U(j, k)/ U(k, k);
           U(j, :) = U(j, :) - L(j, k)*U(k, :);
        end
    end
```

It can be simplified even further using array sections to replace the jth for- loop computations -

```
function [L, U] = lu_fact_no_pivot(A)

n      = size(A, 1);
U      = A;
L      = eye(n);

  for k = 1:n - 1
     L(k + 1:n, k)    = U(k + 1:n, k)/U(k, k);
     U(k + 1:n, k:n)  = U(k + 1:n, k:n) - L(k + 1:n, k)*U(k, k:n);
  end
```

```
>> [L, U] = lu_fact_no_pivot(A)
L =  1     0     0     0
     2     1     0     0
     4     3     1     0
     3     4     1     1
U =  2     1     1     0
     0     1     1     1
     0     0     2     2
     0     0     0     2
>> >> max(max(abs(L*U - A)))  % maximum error
ans = 0
```

Observe that the above implementation of **`lu_fact_no_pivot.m`** follows that of the algorithm 3.2.1 of Golub and van Loan ([p. 98] 1996) and the algorithm 20.1 of Trefethen and Bau ([p. 151] 1997).

Another example from Golub & van Loan is listed below:

```
C = [3 5; 6 7];
>> C                   % example ([p. 94] Golub & van Loan 1997)
C =  3     5
     6     7
>> [L, U] = lu_fact_no_pivot(C)
L =  1     0
     2     1
U =  3     5
     0    -3
```

The function below employs the above LU factorization to *solve* a simple A·x = b system,

```
function x = solution_lu_no_pivot(A, b)

 [L, U]    = lu_fact_no_pivot(A);
  y        = forward_sub(L, b);
  x        = back_sub(U, y);
```

```
>> A = [2   1  1  0; 4  3  3  1; 8  7  9  5; 6  7  9   8]; b = [1; 4; 6; 1];
>> x = solution_lu_no_pivot(A, b)
x = -2
     4
     1
    -3
>> max(max(abs(A*x - b)))
ans = 0
```

Still yet another example is the following:

```
 >> A = [ 1 4 7; 2 5 8; 3 6 10]  % example ([p. 99] Golub & van Loan 1997)
A =  1     4     7
     2     5     8
     3     6    10
>> b = [1; 1; 1];
>> [L, U] = lu_fact_no_pivot(A)
L =  1     0     0
     2     1     0
     3     2     1
U =  1     4     7
     0    -3    -6
     0     0     1
```

```
>> y = forward_sub(L, b)
y =  1
    -1
     0
>> x = back_sub(U, y)
x =  -0.33333
      0.33333
            0
```

As noted in the previous section the factorization process of *A* into L·U requires ~$2n^3/3$ floating point operations. The calculation of each column of x after LU factorization requires a total of ~$2n^2$ floating-point operations.

Observe that critical issue of division by zero-valued U(k, k) remains. For example, in the following matrix, *C*, zeroing out the sub-diagonal elements in the first column creates a zero-valued pivot, U(2, 2). Consequently, the LU factorization must halt.

```
>> C = [1 1 1; 1 1 2; 1 2 2]
C =  1     1     1
     1     1     2
     1     2     2
>> n = size(C, 1);
>> U = C; I = eye(n, n); k  = 1;
>>  for j = k + 1:n  % code fragment from lu_fact_no_pivot.m
       L(j, k) = U(j, k)/ U(k, k);
       U(j, :) = U(j, :) - L(j, k)*U(k, :);
    end
>> U
U =  1     1     1
     0     0     1
     0     1     1
```

LU factorization could be continued if the second and third rows of *U* were exchanged. Furthermore, if a diagonal element U(k, k) is very small, but not zero, the subsequent division, U(j, k)/U(k, k), increases the loss of numerical significance due to round-off errors; consequently, row exchanges are necessary in order to ensure numerical stability of Gaussian elimination ([p. 158] Higham 2002; [p. 46] Dahlquist & Björck 2008). The inclusion of row exchange into the LU decomposition algorithm, in a process termed *pivoting*, is addressed in the next section.

4.5 PIVOTING – GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING (GEPP)

"Thus Gaussian elimination can give arbitrarily poor results even for well-conditioned problems. The method is unstable." ([p. 107] Golub and van Loan 1997).

"In order to repair this short coming of the algorithm, it is necessary to introduce row and/or column interchanges during the elimination process with the intention of keeping the numbers that arise during the calculation suitably bounded." ([p. 107] Golub and van Loan 1997).

"GEPP is the most common way to implement GE in practice" ([p. 40] Demmel 1997)

This section follows the discussion of the stabilization of LU factorization via pivoting from chapter 21 of Trefethen and Bau (1997).

The standard method for the stabilization of Gaussian elimination is termed *partial pivoting*, where rows of *A* are exchanged. Note that the diagonal entry, A(k, k), is termed the *pivot*. In partial pivoting the pivot at each kth step is identified as the largest magnitude element of |A(k:n, k)|, found `in` the sub-diagonal entries in the kth column. Then this row containing the largest magnitude element is swapped with the kth row. To keep track of the row interchanges a permutation matrix, *P*, is introduced. This matrix is initialized as **`eye(n, n)`**. The rows of *P* are exchanged corresponding to the interchanges of the rows of *A* before the *L* and *U* elements are computed. The *L* and *U* matrices computed in this fashion are the same triangular matrices one would find if one first applied row interchanges to the original matrix *A* to obtain P·A and then implemented LU decomposition on P·A *without* row exchanges (e.g. [p. 46] Dahlquist & Björck 2008).

O(n^2) comparisons are performed in pivot searches ([p. 112] Golub & van Loan 1996). Row exchanges do not require any floating-point arithmetic operations, however, such exchanges often incorporate irregular movements of data which in turn can result in a significant computational overheads ([p. 110] Golub & van Loan 1996). In the absence of such data exchange effects, the overall solution process remains dominated by the ~ $2n^3/3$ floating-point operations required to factor an n-by-n square matrix *A* into triangular *L* and *U* matrices ([p. 157] Bau & Trefethen 1997; [p. 112] Golub & van Loan 1996). In certain *rare* situations the LU decomposition with partial pivoting is unstable.

An alternative, more expensive pivoting strategy, termed *complete pivoting* exists where both rows and columns are exchanged ([p. 117] Golub & van Loan 1996; [p. 158] Higham 2002). However, this pivot search entails a significant overhead of O(n^3) comparisons associated with the two-dimensional searches at each kth stage in addition to expected $2n^3/3$ floating-point operations required to factor an n-by-n square matrix *A* into triangular *L* and *U* matrices,. Golub & van Loan [p. 119] conclude although *"Gaussian elimination with complete pivoting is stable … there appears to be no practical justification for choosing complete pivoting over partial pivoting except in cases where rank determination is an issue."* Similarly, Higham [p. 158] states "*because partial pivoting works so well, compete pivoting is used only in special circumstances."* Stewart makes even a stronger case for GEPP; he declares that *"Certain contrived examples show that Gaussian elimination with partial pivoting can be unstable. Nonetheless, it works well for the overwhelming majority of real-life problems."*

Below I illustrate LU factorization process employing row exchanges:

```
>> A = [2   1  1  0; 4  3  3  1; 8  7  9  5; 6  7  9   8];  A0 = A;
>> n = size(A, 1);
>> p = eye(n, n);          % Permutation matrix
>> A  % Exchange row3 with row1to zero out column1 sub-diagonal
A =  2     1     1     0
     4     3     3     1
     8     7     9     5
     6     7     9     8

>> P([3, 1], :)    = P([1, 3], :);
>> A([3, 1], :) = A([1, 3], :) % Exchange row3 and row1
A =  8     7     9     5
     4     3     3     1
     2     1     1     0
     6     7     9     8
```

Observe the subtraction process, which creates U starts at k + 1 so that L is not overwritten -

```
>> k = 1;                     % From Gaussian elimination section 3
   for j = 2:n
     A(j, k)     = A(j, k)/ A(k, k);                 % Creates L elements
     A(j, k+1:n) = A(j, k+1:n) - A(j, k)*A(k, k+1:n); % Creates U elements
   end
```

```
>> A
A = 8.0000    7.0000    9.0000    5.0000
    0.5000   -0.5000   -1.5000   -1.5000
    0.2500   -0.7500   -1.2500   -1.2500
    0.7500    1.7500    2.2500    4.2500

% Exchange row_4 with row_2 to zero column_2 sub diagonals elements
P([4, 2], :) = P([2, 4], :);
A([4, 2], :) = A([2, 4], :)   % Exchange row_4 and row
A = 8.0000    7.0000    9.0000    5.0000
    0.7500    1.7500    2.2500    4.2500
    0.2500   -0.7500   -1.2500   -1.2500
    0.5000   -0.5000   -1.5000   -1.5000

>> k = 2;                          % From Gaussian elimination section 3


 >> for j = 3:n
       A(j, k) = +A(j, k)/A(k, k);                     % Creates L elements
       A(j, :) = +A(j, k+1:n) - A(j, k)*A(k, k+1:n); % Creates U elements
    end

>> A
    8.0000    7.0000    9.0000    5.0000
    0.7500    1.7500    2.2500    4.2500
    0.2500   -0.4286   -0.2857    0.5714
    0.5000   -0.2857   -0.8571   -0.2857

% Exchange row_4 with row_3 to zero column_3 sub-diagonal
A([4, 3], :) = A([3, 4], :); P([4, 3], :) = P([3, 4], :);

>> k = 3;                     % From Gaussian elimination section 3
>> for j = 4:n
     A(j, k)     = +A(j, k)/A(k, k);                     % Creates L elements
     A(j, k+1:n) = +A(j, k+1:n) - A(j, k)*A(k, k+1:n); % Creates U elements
   end

% final results for the heavily modified A matrix
A = 8.0000    7.0000    9.0000    5.0000
    0.7500    1.7500    2.2500    4.2500
    0.5000   -0.2857   -0.8571   -0.2857
    0.2500   -0.4286    0.3333    0.6667

>> U = triu(A)
>> L = tril(A, -1) + eye(n, n);

U = 8.0000    7.0000    9.0000    5.0000
         0    1.7500    2.2500    4.2500
         0         0   -0.8571   -0.2857
         0         0         0    0.6667

L = 1.0000         0         0         0
    0.7500    1.0000         0         0
    0.5000   -0.2857    1.0000         0
    0.2500   -0.4286    0.3333    1.0000

P  = 0     0     1     0
     0     0     0     1
     0     1     0     0
     1     0     0     0
max(max(abs(P*A0 - L*U)))
ans = 0
```

Above the largest magnitude elements of |A(k:n, k)|, the pivot at each kth step, is identified by inspection. To automate this process the largest magnitude element of |A(k:n, k)|, the sub-diagonal entries in the kth column is identified by the following line of code -

```
[maxx, kmax] = max(abs(A(k:n, k)),
```

where kmax identifies the row where the largest magnitude is found. However, note in MATLAB's **max** function kmax identifies the row location *relative* to k:n, thus, to identify row location relative to 1:n one must employ the following code fragment -

```
kmax = kmax + k – 1;
```

Consequently, the above two lines of code

```
>> p([3, 1])    = p([1, 3]);
>> A([3, 1], :) = A([1, 3], :) % Exchange row3 and row1
```

becomes

```
k = 1;
[maxx, kmax] = max(abs(A(k:n, k)),   kmax = kmax + k – 1;
     if k ~= kmax
        A([kmax, k], :) = A([k, kmax], :);
        P([kmax, k], :) = P([k, kmax], :);
     end
```

This code is encapsulated in

```
  function [L, U, P] = lu_basic_pivot(A)
% function [L, U, P] = lu_basic_pivot(A)
% Gaussian elimination with partial pivoting
% INPUT:
%                  A = n-by-n matrix to be factored
% OUTPUT:
%                  L = n-by-n unit [L(k, k) = 1] lower triangular matrix
%                  U = n-by-n upper matrix
%                  P = n-by-n permutation matrix
%
%  Trefethen, L N & Bau, D, Numerical Linear Algebra,, p. 151
%         Algorithm 20.1  =>  PA = L*U

n      = size(A, 1);
P      = eye(n, n);

      for k = 1:n - 1
        [maxx, kmax] = max(abs(A(k:n, k)));   % Determines kmax relative to k
        kmax         = kmax + k - 1;          % Determines kmax relative to row 1
         if k ~= kmax
            A([kmax, k], :) = A([k, kmax], :);
            P([kmax, k], :) = P([k, kmax], :);
         end
        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

U      = triu(A);
L      = tril(A, -1) + eye(n, n);
```

Note that MATLAB possesses a built-in LU (GEPP) factorization function, `lu`, which employs LAPACK routines DGETRF and ZGETRF to calculate LU factorizations for real and complex matrices respectively,

```
>> help lu                                        % only partial listing
 lu     lu factorization.
    [L,U] = lu(A) stores an upper triangular matrix in U and a
    "psychologically lower triangular matrix" (i.e. a product of lower
    triangular and permutation matrices) in L, so that A = L*U. A can be
    rectangular.

    [L,U,P] = lu(A) returns unit lower triangular matrix L, upper
    triangular matrix U, and permutation matrix P so that P*A = L*U.

    [L,U,p] = lu(A,'vector') returns the permutation information as a
    vector instead of a matrix.  That is, p is a row vector such that
    A(p,:) = L*U.  Similarly, [L, U, P] = lu(A,'matrix') returns a
    permutation matrix P.  This is the default behavior.

[L2, U2, P2] = lu(A);  % lu is the built-in MATLAB function
>> max(max(L - L2))
ans =  0
>> max(max(U - U2))
ans =  0
>> max(max(P - P2))
ans =  0
```

Be careful employing MATLAB's `lu` function *without* a third output variable (above defined as P2) since –

Without a *P* return variable, [L, U] = lu(A) returns L = P^T*L (Remember P^T = inv(P)!]

```
>> [L3, U3] = lu(A);
>> L3
L3 =    0.2500   -0.4286    0.3333    1.0000
        0.5000   -0.2857    1.0000         0
        1.0000         0         0         0
        0.7500    1.0000         0         0
>> max(max(abs(P'*L2 - L3)))
ans = 0
```

To solve such the linear system, A·x = b, with *row exchange* is straightforward:

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

Thus

```
  function x = solution_lu_pivot(A, b)
% function x = solution_lu_pivot(A, b)
% Solves Ax = b for x via LU elimination & PARTIAL pivoting
% INPUT:
%       A = n x n matrix
%       b = n-sized vector
% OUTPUT:
%       x = n-sized vector

  [L, U, P]  = lu_basic_pivot(A);
  y          = forward_sub(L, P*b);
  x          = back_sub(U, y);
```

```
>> b = [3 5 6 1]';
>> x = solution_lu_pivot(A, b)
x = 1.7500
     0.5000
    -1.0000
    -0.5000
>> max(max(abs(A*x - b)))
ans =  1.7764e-015
```

MATLAB has several built-in linear –system solvers. e.g. `linsolve`,

```
>> help linsolve   % only a partial listing
linsolve Solve linear system A*X=B.
X = linsolve(A, B) solves the linear system A*X=B using
LU factorization with partial pivoting when A is square,

>> z = linsolve(A, b)
z = 1.7500
     0.5000
    -1.0000
    -0.5000
>> max(max(abs(A*z - b)))
ans = 1.7764e-015
```

Observe that it is *safe* not to employ row exchanges (pivots) for certain classes of matrices ([p.166] Higham 2002):

- Row or column diagonally[2] dominant.
- Totally nonnegative.
- Symmetric positive definite.
- Complex symmetric with positive definite Hermitian and skew-Hermitian parts.

4.6 INVERSES AND DETERMINANTS

4.6.1 Inverses

"To most numerical analysts, matrix inversion is a sin." ([p. 260] Higham 2002).

The best example of when a inverse should *not* be calculated is of course, A·x = b ([p. 260] Higham 2002).Calculating the solution, x, column by column, setting b = I_k, requires ~$2n^3$ floating-point operations using GEPP, where I = `eye(n, n).` First, as frequently noted, the inverse computation is three times as expensive as solving the linear system directly, using `linsolve`. Second, such an inverse computation is *much less stable* than a GEPP calculation due to roundoff error ([p. 260] Higham 2002). Third, using inverses wastes storage space ([p. 106] Ascher & Greif 2011).

There are two somewhat esoteric reasons for computing inverses ([p. 260] Higham 2002). First, there exists occasions where inverses need to computed: statistics, calculation of certain matrix functions (roots and logarithms), and in numerical integrations. Second, and more significantly, matrix inversions are used in a wide variety of stability studies and error analyses.

[2] Diagonally dominant means that the diagonal element is *at least* as large in magnitude as the sum of magnitudes of the remaining elements in the row combined ([p 109] Ascher & Greif 2011).

The most frequently described method for calculating A^{-1} of an n-by-n matrix, *A* (e.g. [p. 267] Higham 2002) is the somewhat obvious approach:

```
for j =1:n
  solve $Ax_j = e_j$      [ $e_j$ is jth column of an n-by-n identity matrix, $I$]
end
```

```
>> A = randi(999, 5, 5)
A =163   810   777   332   154
   114   187   511   174   357
   912   247    28   626   144
   482    55   990   575   850
   851   609   501   751   338
>> n   = size(A, 1);
>> I   = eye(n, n);
>> Ai  = zeros(n, n);
>>    for k = 1:n
          Ai(:, k) = linsolve(A, I(:, k));
      end
>> Ai
Ai= 0.0028344     0.0084036     0.0099781   -0.0016586    -0.0102480
   -0.0023304     0.0051084    -0.0029255   -0.0030716     0.0046369
    0.0052072    -0.0018120     0.0067275    0.0023812    -0.0093130
   -0.0020282    -0.0159770    -0.0106890    0.0042955     0.0115510
   -0.0061493     0.0078225    -0.0060737   -0.0033634     0.0085441
>> isequal(Ai*A, I)
ans = 0
>> max(max(abs(Ai*A - I)))
ans =  2.6645e-015   % round off error ~ 12*eps
```

4.6.2 Determinants

"Like the matrix inverse, the determinant is a quantity that rarely needs to be computed" ([p. 279] Higham 2002).

An n-by-n sized matrix *A* is decomposed into three matrices, *P*, *L*, and *U* via LU factorization. Then the determinant of *A* is computed from the product rule, det(L·U) = det(L)·det(U), the fact that the determinant of a *triangular* matrix is the product of the elements of its main diagonal (det(L) ≡ 1),

$$\det(A) = (-1)^s \prod_{m=1}^{n} U_{mm}$$

where ΠU_{mm} represents the product of the series, U_{mm} for m = 1:n, s is the number of row and column interchanges ([p. 18] Dahlquist & Björck 2008). To perform this computation on makes a simple addition to **lu_basic_pivot.m** to ascertain the number of times the rows are exchanged.

```
function [L, U, P, d] = lu_basic_pivotD(A)
n      = size(A, 1);
P      = eye(n, n);
s      = 0;    % addition
      for k = 1:n - 1
        [maxx, kmax] = max(abs(A(k:n, k)));
        kmax         = kmax + k - 1;
          if k ~= kmax
             A([kmax, k], :)  = A([k, kmax], :);
             P([kmax, k], :)  = P([k, kmax], :);
             s                = s  + 1; % addition
```

```
                end
              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
      U      = triu(A);
      L      = tril(A, -1) + eye(n, n);
      d      = ((-1.0)^s)*prod(diag(U));  % addition
```

```
>> help det                % built-in MATLAB function
 DET    Determinant.
    DET(X) is the determinant of the square matrix X.
```

```
>> n = 4; A = pascal(n);  [L, U, P, d] = lu_basic_pivotD(A); disp([det(A), d])
            1             1
>> n = 8; A = pascal(n);  [L, U, P, d] = lu_basic_pivotD(A); disp([det(A), d])
            1             1
>> n = 8; A = rand(n);    [L, U, P, d] = lu_basic_pivotD(A); disp([det(A), d])
     0.082898     0.082898
>> n = 11; A = rand(n);   [L, U, P, d] = lu_basic_pivotD(A); disp([det(A), d])
   -0.0013551   -0.0013551
>> n = 5; A = invhilb(n); [L, U, P, d] = lu_basic_pivotD(A); disp([det(A), d])
  2.6672e+011  2.6672e+011
>> n = 4; A = hilb(n);    [L, U, P, d] = lu_basic_pivotD(A); disp([det(A), d])
  1.6534e-007  1.6534e-007
```

4.7 SYMMETRIC POSITIVE DEFINITE SYSTEMS

If the square matrix *A* is symmetric and positive definite then an LU factorization can be simplified since U = L^T, Thus A ≡ L·U = L^T·L, where L has positive entries (usually not unit values) along the diagonal. This method is known as Cholesky factorization of the matrix, A. This factorization is popular ([p. 85] Heath, 2002):

- No pivoting is required for purposes of stability.
- Only the lower triangle of parent matrix need be stored.
- Only ~n^3/6 multiplications are required to factor symmetric positive-definite matrix.

Consequently, the Cholesky factorization requires half the effort as well as half the storage of an equivalent LU factorization ([p. 85] Heath, 2002).

Any of the following tests is a sufficient condition for a real symmetric matrix, A, to be positive definite ([p. 331] Strang 1988):

- x^T·A·x > 0 for all nonzero x vectors.
- All eigenvalues of A, λ_m, are positive.
- All upper left submatrices, A_k, have positive determinants.
- All the pivots, m_k (without row exchanges), are positive.

Built-in[3] MATLAB symmetric positive-definite matrices are ([p. 51] Higham & Higham 2005): **`hilb`, `invhilb`, `ipjfact`, `kms`, `lehmer`, `minij`, `moler`, `pascal`, `pei`, `poisson`, `prolate`, `randcorr`, `randsvd`, `toeppd`, `tridiag`**, and, **`wathen`**.

The solution algorithm is predicated by equating the corresponding entries of A and $L \cdot L^T$. In a simple 2-by-2 matrix, one finds that -

$$\begin{bmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix} \bullet \begin{bmatrix} L_{11} & L_{21} \\ L_{21} & L_{22} \end{bmatrix} = \begin{bmatrix} L_{11}^2 & L_{11}L_{21} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 \end{bmatrix}$$

$$\Downarrow$$

$$A_{11} = L_{11}^2, \quad A_{21} = A_{12} = L_{11}L_{21}, \quad A_{22} = L_{21}^2 + L_{22}^2.$$

which leads to

$$L_{11} = \sqrt{A_{11}}, \quad L_{21} = A_{21}/L_{11}, \quad L_{22} = \sqrt{A_{22} - L_{21}^2}$$

Similarly, a 4-by-4 positive-definite matrix can be factored in the following way -

$$\begin{bmatrix} A_{11} & A_{21} & A_{31} & A_{41} \\ A_{21} & A_{22} & A_{32} & A_{42} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \bullet \begin{bmatrix} L_{11} & L_{21} & L_{31} & L_{41} \\ 0 & L_{22} & L_{32} & L_{42} \\ 0 & 0 & L_{33} & L_{43} \\ 0 & 0 & 0 & L_{44} \end{bmatrix} =$$

$$\begin{bmatrix} L_{11}^2 & L_{11}L_{21} & L_{11}L_{31} & L_{11}L_{41} \\ L_{11}L_{21} & L_{21}^2+L_{22}^2 & L_{21}L_{31}+L_{22}L_{32} & L_{21}L_{41}+L_{22}L_{42} \\ L_{11}L_{31} & L_{21}L_{31}+L_{22}L_{32} & L_{31}^2+L_{32}^2+L_{33}^2 & L_{31}L_{41}+L_{32}L_{42}+L_{33}L_{43} \\ L_{11}L_{41} & L_{21}L_{41}+L_{22}L_{42} & L_{31}L_{41}+L_{32}L_{42}+L_{33}L_{43} & L_{41}^2+L_{42}^2+L_{43}^2+L_{44}^2 \end{bmatrix}.$$

Inspecting the above off-diagonal L_{43} term, and overwriting the elements of the A matrix, one finds that

$$L_{43} = \frac{A_{43} - L_{31}L_{41} - L_{32}L_{42}}{L_{33}} \Rightarrow A_{jk} = \frac{A_{jk} - \sum_{m=1}^{m=k-1} A_{km}A_{jm}}{A_{kk}}, \text{ for j = 4 and k = 3.}$$

Now examining the diagonal L_{44} term, and again overwriting the elements of the A matrix, one discovers that

$$L_{44} = \sqrt{A_{44} - L_{41}^2 - L_{42}^2 - L_{43}^2} \Rightarrow A_{jj} = \sqrt{A_{jj} - \sum_{m=1}^{m=j-1} A_{jm}^2}, \text{ for j = 4.}$$

Building on these results one creates the following algorithm for factoring a positive definite matrix into the matrix product, $L \cdot L^T$. Given a symmetric positive definite square matrix of size n-by-n, this scheme overwrites lower part of A with its Cholesky factorization -

[3] Most of these matrices are accessed through MATLAB's **`gallery`** function.

```
for j = 1:n
    for k = 1:j - 1  % off diagonal terms
      for m = 1:k - 1
         $A_{jk} = A_{jk} - A_{km}*A_{jm}$
      end
   $A_{jk} = A_{jk}/A_{kk}$
      for m = 1:j - 1 % diagonal terms
        $A_{jj} = A_{jj} - A_{jm}*A_{jm}$
     end
   $A_{jj} = \sqrt{A_{jj}}$
end
```

```
  function L = chol_factorization(A)
% function L = chol_factorization(A)
% INPUT:
%        A = positive definite n x n matrix
% OUTPUT:
%        L = n x n matrix (lower triangle)
%        where   A = L*L'

% Check to see if A is a square matrix
   [nr, nc] = size(A);
   if nr ~= nc; error('chol_factorization.m: A is not square'); end
   n  = nr;
   for j = 1:n
         for k = 1:j - 1
           A(j, k) = (A(j, k) - sum(A(k, 1:k - 1)*A(j, 1:k - 1)'))/A(k, k);
         end
      A(j, j) = sqrt( A(j, j) - A(j, 1:j - 1)*A(j, 1:j - 1)');
   end
L = tril(A);
```

This algorithm, `chol_factorization.m`, is applied now to a positive-definite matrix -

```
>> A = pascal(7)
A =  1     1     1     1     1     1     1
     1     2     3     4     5     6     7
     1     3     6    10    15    21    28
     1     4    10    20    35    56    84
     1     5    15    35    70   126   210
     1     6    21    56   126   252   462
     1     7    28    84   210   462   924

>> L = chol_factorization(A)
L =  1     0     0     0     0     0     0
     1     1     0     0     0     0     0
     1     2     1     0     0     0     0
     1     3     3     1     0     0     0
     1     4     6     4     1     0     0
     1     5    10    10     5     1     0
     1     6    15    20    15     6     1

>> max(max(abs(A - L*transpose(L))))
ans = 0
```

Note that MATLAB possesses a built-in Cholesky factorization function, `chol`, which employs LAPACK routines DPOTRF and ZPOTRF to calculate Cholesky factorizations for real and complex matrices respectively,

```
>> help chol
CHOL   Cholesky factorization.          % only a partial listing
CHOL(A) uses only the diagonal and upper triangle of A. The lower
triangle is assumed to be the (complex conjugate) transpose of the
upper triangle.  If A is positive definite, then R = CHOL(A) produces
an upper triangular R so that R'*R = A. If A is not positive definite,
an error message is printed.

L = CHOL(A,'lower') uses only the diagonal and the lower triangle of A
to produce a lower triangular L so that L*L' = A.  If A is not positive
definite, an error message is printed.

>> L2 = chol(A,'lower')
L2 = 1      0      0      0      0      0      0
     1      1      0      0      0      0      0
     1      2      1      0      0      0      0
     1      3      3      1      0      0      0
     1      4      6      4      1      0      0
     1      5     10     10      5      1      0
     1      6     15     20     15      6      1
>> isequal(L, L2)
ans = 1
                          % another positive-define matrix
>>  A = gallery('pei',6, 0.2)
A = 1.2000    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0000    1.2000    1.0000    1.0000    1.0000    1.0000
    1.0000    1.0000    1.2000    1.0000    1.0000    1.0000
    1.0000    1.0000    1.0000    1.2000    1.0000    1.0000
    1.0000    1.0000    1.0000    1.0000    1.2000    1.0000
    1.0000    1.0000    1.0000    1.0000    1.0000    1.2000

>>  L = chol_factorization(A)
L = 1.0954         0         0         0         0         0
    0.9129    0.6055         0         0         0         0
    0.9129    0.2752    0.5394         0         0         0
    0.9129    0.2752    0.1685    0.5123         0         0
    0.9129    0.2752    0.1685    0.1220    0.4976         0
    0.9129    0.2752    0.1685    0.1220    0.0957    0.4883

>> L2 = chol(A, 'lower')
L2 =1.0954         0         0         0         0         0
    0.9129    0.6055         0         0         0         0
    0.9129    0.2752    0.5394         0         0         0
    0.9129    0.2752    0.1685    0.5123         0         0
    0.9129    0.2752    0.1685    0.1220    0.4976         0
    0.9129    0.2752    0.1685    0.1220    0.0957    0.4883

>> isequal(L, L2)                 % NOT equal
ans = 0
>> max(max(abs(L - L2)))
ans =   1.2490e-016               % Differ on order of ~0.5*eps

>> isequal(L2*L2', A)              % ALSO NOTE
ans =  0
>> max(max(abs(L2*L2' - A)))
ans =2.2204e-016


>> [L, U, P] = lu(A);              % ALSO NOTE
>> L
L = 1.0000         0         0         0         0         0
    0.8333    1.0000         0         0         0         0
    0.8333    0.4545    1.0000         0         0         0
    0.8333    0.4545    0.3125    1.0000         0         0
    0.8333    0.4545    0.3125    0.2381    1.0000         0
    0.8333    0.4545    0.3125    0.2381    0.1923    1.0000
>> U
U = 1.2000    1.0000    1.0000    1.0000    1.0000    1.0000
         0    0.3667    0.1667    0.1667    0.1667    0.1667
         0         0    0.2909    0.0909    0.0909    0.0909
         0         0         0    0.2625    0.0625    0.0625
         0         0         0         0    0.2476    0.0476
         0         0         0         0         0    0.2385
>> P
P =  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     0     1
>> isequal(L*U, A)
ans =  0
>> max(max(abs(L*U - A)))
ans =  1.1102e-016

>> A = randi(9999, 4,4)  % NOTE A is not a symmetric positive definite array
A =     4218        6557        6787        6555
        9157         358        7577        1712
        7922        8491        7431        7060
        9594        9339        3922         319
>> L = chol(A, 'lower');
??? Error using ==> chol
Matrix must be positive definite.
```

The following is *one* approach of how `linsolve` can be used with positive definite matrices -

```
>> n = 7;
>> A = pascal(n)
A =  1     1     1     1     1     1     1
     1     2     3     4     5     6     7
     1     3     6    10    15    21    28
     1     4    10    20    35    56    84
     1     5    15    35    70   126   210
     1     6    21    56   126   252   462
     1     7    28    84   210   462   924
>> b = A*ones(n, 1)
b =        7
          28
          84
         210
         462
         924
        1716
>> opts.POSDEF = true;
>> opts.SYM    = true;
>> x           = linsolve(A, b, opts)
x =  1
     1
     1
     1
     1
     1
     1
```

4.8 BANDED LINEAR SYSTEMS

4.8.1 Basic Properties

A matrix *A* is defined to have an upper bandwidth, *r*, and a lower bandwidth, *s*, if

$$A_{ij} = 0, j > i + r,$$
$$A_{ij} = 0, i > j + s,$$

If all the nonzero elements are found in these diagonals, A is termed a *banded* matrix ([p. 82] Dahlquist, & Björck 2008). Of interest here (LU factorization) is the fact that if two matrices, *A* and *B* both possess an upper bandwidth, *r*, and a lower bandwidth, *s*, then their product, A·B, has lower (upper) bandwidth, *r* + *s*. Moreover, if pivoting is not necessary for purposes of stability, (e.g. A is diagonally dominant), band structure is maintained in GE factorizations. Consequently, if a matrix *A* possesses an upper bandwidth *r* and a lower bandwidth *s*, then a pivot-free LU factorization of A generates an upper triangular matrix, *U*, with an upper bandwidth, *r*, and a lower triangular matrix, *L*, with a lower bandwidth, *s* ([p. 84] Dahlquist, & Björck 2008). One could create a GE algorithm *without* pivoting, equivalent to **`lu_fact_no_pivot.`** Consequently`,` one has the following simple, but perhaps inefficient, algorithm -

```
function [L, U] = lu_fact_band(A, r, s)
% Variant on Banded Gaussian Elimination, Algorithm 7.7
%  Dahlquist, G & Bjorck, A 2008  p. 84 [SIAM]
% Numerical Methods in Scientific Computing, 2

n       = size(A, 1);
U       = A;
L       = eye(n);

  for k = 1:n - 1
       for i = k + 1:min(k + s, n)
          L(i, k)     = U(i, k)/U(k, k);
             for j = k:min(k + r, n)
               U(i, j) = U(i, j) - L(i, k)*U(k, j);
             end % end jth loop
       end % end ith loop
  end
```

```
% Create following tridiagonal matrix:  s = r = 1
>> n = 6;
>> a = 2.0*ones(n, 1); b = -ones(n, 1); c = -ones(n, 1);
>> A = diag(a) + diag(b(2:end), -1) + diag(c(2:end), + 1)
A =  2    -1     0     0     0     0
    -1     2    -1     0     0     0
     0    -1     2    -1     0     0
     0     0    -1     2    -1     0
     0     0     0    -1     2    -1
     0     0     0     0    -1     2
>> [L, U, P] = lu(A);
>> isequal(P, eye(n,n)) % No pivoting occurs!
ans = 1
```

```
>> r = 1; s = 1; [L1, U1] = lu_fact_band(A, r, s);
>> L1
L1 =
            1            0            0            0            0            0
         -0.5            1            0            0            0            0
            0     -0.66667            1            0            0            0
            0            0        -0.75            1            0            0
            0            0            0         -0.8            1            0
            0            0            0            0     -0.83333            1
>> U1
U1 =
            2           -1            0            0            0            0
            0          1.5           -1            0            0            0
            0            0       1.3333           -1            0            0
            0            0            0         1.25           -1            0
            0            0            0            0          1.2           -1
            0            0            0            0            0       1.1667
>> isequal(L, L1)
ans = 1
>> isequal(U, U1) % Results from lu_fact_band agree exactly with lu output!
ans = 1
```

4.8.2 Tridiagonal Systems

Banded linear systems occur in computation most frequently in the form of tridiagonal systems (r = 1, s = 1). Developing an algorithm, specific to tridiagonal systems, permits the user to create a *very efficient* solver. The following discussion is based on ([p. 82] Dahlquist, & Björck 2008).

$$A = \begin{bmatrix} a_1 & c_2 & & & \\ b_2 & a_2 & c_3 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-1} & a_{n-1} & c_n \\ & & & b_n & a_n \end{bmatrix}$$

$$A = L \bullet U = \begin{bmatrix} 1 & & & & \\ \gamma_2 & 1 & & & \\ & \gamma_3 & \ddots & & \\ & & \ddots & 1 & \\ & & & \gamma_2 & 1 \end{bmatrix} \bullet \begin{bmatrix} \alpha_1 & c_2 & & & \\ & \alpha_1 & c_3 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{n-1} & c_n \\ & & & & \alpha_n \end{bmatrix}$$

Equating term by term the elements of *A* and L·U on ascertains the upper diagonals in *A* and *U* are the same and the remaining elements in L and U are derived via recursion –

$$\alpha_1 = a_1,$$
$$\gamma_k = b_k / \alpha_{k-1}, \quad \alpha_k = a_k - \gamma_k c_k, \text{ for } k = 2{:}n.$$

Employing recursion again, one finds that for A·x = f,

$$L \bullet y = f$$
$$y_1 = f_1,$$
$$y_k = f_k - \gamma_k \cdot y_{k-1} \qquad k = 2{:}n \;\Rightarrow$$

$$U \bullet x = y,$$
$$x_n = y_n / \alpha_n,$$
$$x_j = (y_j - c_{j+1} \cdot x_{j+1}) / \alpha_j \quad j = n - 1 :\!-1{:}1.$$

Note that the number of floating-point arithmetic operations required to solve a square n-by-n tridiagonal linear system is very modest, 3n and 2.5n, respectively, for the factorization and subsequent solution. As observed earlier, applying LU factorization, with or without pivoting, to a fully populated, filed n-by-n square array requires ~ $2n^3/3$ floating-point arithmetic operations! Please remember the fundamental issue with using tridiagonal solvers is that the parent sparse tridiagonal matrix must be diagonally dominant in order that pivoting can be disregarded!

```
function x = tridiagonal(a, b, c, rhs)
%function x = tridiagonal(a, b, c, rhs)
%Tridiagonal solver where A*X = rhs and
%  a = n-by-1 vector   main diagonal
%  b = n-by-1 vector   subdiagonal
%  c = n-by-1 vector   superdiagonal diagonal
%  Dahlquist, G & Bjorck, A 2008  [SIAM]
% Numerical Methods in Scientific Computing, 2

    n        = length(a);
    alpha    = zeros(size(a));
    gamma    = zeros(size(a));

    alpha(1) = a(1);
       for k = 2:n
         gamma(k) = b(k)/alpha(k - 1);
         alpha(k) = a(k) - gamma(k)*c(k);
       end

  % L*y      = rhs   => U*x = y
   y         = zeros(size(a));
   x         = zeros(size(a));

   y(1)      = rhs(1);
     for k = 2:n
        y(k) = rhs(k) - gamma(k)*y(k - 1);
     end

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

```
>> n = 12;
>> a = 2.0*ones(n, 1); b = -ones(n, 1); c = -ones(n, 1);
>> A = diag(a) + diag(b(2:end), -1) + diag(c(2:end), + 1);
>> rhs = A*ones(n, 1);    % expect x_tri to be unity
>> x_tri = tridiagonal(a, b, c, rhs);
>> x_tri
x_tri =     1
            1
            1
            1
            1
            1
            1
            1
            1
            1
            1
            1
```

4.9 ERROR ESTIMATES FOR LINEAR SYSTEMS

4.9.1 Condition Numbers

The concept of a matrix size, or norm, developed previously is now used to estimate errors or uncertainties in solving linear systems. The matrix norm is defined as

$$\|A\| \equiv \frac{\max}{x \neq 0} \frac{\|A \bullet x\|}{\|x\|}.$$

Intuitively, the matrix norm measures the maximum stretching the matrix performs on any vector ([p. 55] Heath 2002).

Assume that x is the solution to the linear system, $A \cdot x = b$, and $\hat{x}$ is the solution to the *perturbed* system, $A \bullet \hat{x} \equiv b + \Delta b$, where Δb represents perturbations in b. Define $\Delta x = \hat{x} - x$. Thus

$$A \bullet \hat{x} = b + \Delta b = A \bullet x + \Delta b \Rightarrow A \bullet \Delta x = \Delta b \Rightarrow \Delta x = A^{-1} \bullet \Delta b,$$

$$\|\Delta x\| \leq \|A^{-1}\| \|\Delta b\| \Rightarrow \|A\| \left[\frac{\|\Delta x\|}{\|A\| \|x\|} \leq \frac{\|A^{-1}\| \|\Delta b\|}{\|b\|} \right], \Rightarrow$$

$$\frac{\|\Delta x\|}{\|x\|} \leq \|A\| \|A^{-1}\| \frac{\|\Delta b\|}{\|b\|}.$$

Assume now that $\hat{x}$ is the solution to a different perturbed system, $(A + \Delta A) \bullet \hat{x} \equiv b$. Also $\Delta x = \hat{x} - x$.

$$\begin{aligned}
(A + \Delta A) \bullet \hat{x} &= b, \Rightarrow A \bullet \hat{x} = b - \Delta A \bullet \hat{x}, \\
\hat{x} &= A^{-1} \bullet (b - \Delta A \bullet \hat{x}), \\
\hat{x} - x &= A^{-1} \bullet (b - \Delta A \bullet \hat{x}) - A^{-1} \bullet b = A^{-1} \bullet \Delta A \bullet \hat{x}, \\
\Delta x &= A^{-1} \bullet \Delta A \bullet \hat{x}, \\
\frac{\|\Delta x\|}{\|\hat{x}\|} &\leq \|A\| \|A^{-1}\| \frac{\|\Delta A\|}{\|A\|}.
\end{aligned}$$

As can be seen here, the norms of the relative errors of the input uncertainties, Δb/b and ΔA/A, are amplified by a numerical factor, $||A||\cdot||A^{-1}||$ ([p. 60] Heath 2002). NOTE: in reality the simple numerical factor is not calculated directly from *A*, but is estimated or approximated from the detailed structure of A. This factor, a scalar, is called the *condition number* of the square nonsingular matrix, *A*,

$$\mathrm{cond}(A) \equiv \|A^{-1}\| \bullet \|A\|.$$

The condition number is a measure of how close a matrix is to being singular. A large condition number implies that your linear system is close to singular. Note that a *maximally* well-conditioned matrix possesses a condition number of unity.

```
>> help cond     % on-line MATLAB help
cond(X) Condition number with respect to inversion.

cond(X) returns the 2 norm condition number (the ratio of the largest
singular value of X to the smallest).  Large condition numbers indicate
a nearly singular matrix. cond(X, p) returns the condition number of X
in p-norm: norm(X,P)*norm(inv(X), p). where p = 1, 2, inf, or 'fro'.
```

```
>> n = 6;
>> A = hilb(n)    % Hilbert matrix poorly conditioned matrix
A =         1          0.5      0.33333         0.25          0.2      0.16667
          0.5      0.33333         0.25          0.2      0.16667      0.14286
      0.33333         0.25          0.2      0.16667      0.14286        0.125
         0.25          0.2      0.16667      0.14286        0.125      0.11111
          0.2      0.16667      0.14286        0.125      0.11111          0.1
      0.16667      0.14286        0.125      0.11111          0.1     0.090909
>> disp( [cond(A, 1) cond(A, 2) cond(A, inf)] )
   2.907e+007  1.4951e+007   2.907e+007
>>  disp( [norm(A, 1) norm(A, 2) norm(A, inf)] )
         2.45       1.6189         2.45
>> Ai = inv(A)
Ai =       36         -630         3360        -7560         7560        -2772
         -630        14700       -88200  2.1168e+005  -2.205e+005        83160
         3360       -88200  5.6448e+005 -1.4112e+006    1.512e+006 -5.8212e+005
        -7560  2.1168e+005 -1.4112e+006  3.6288e+006  -3.969e+006  1.5523e+006
         7560  -2.205e+005    1.512e+006  -3.969e+006    4.41e+006 -1.7464e+006
        -2772        83160 -5.8212e+005  1.5523e+006 -1.7464e+006  6.9854e+005
  >> disp( [norm(Ai, 1) norm(Ai, 2) norm(Ai, inf)] )
  1.1865e+007  9.2353e+006  1.1865e+007
```

```
condest 1-norm condition number estimate.
C = condest(A) computes a lower bound C for the 1-norm condition number of a
square matrix A.

>> condest(A)[4]
ans = 2.907e+007
```

Observe the condition number, in the form of its reciprocal, is returned when linsolve is employed to solve systems of linear equations –

[4] The MATLAB built-in condition *estimator* is based on an iterative search for $||A^{-1}||_1$ *without* actually calculating A^{-1} (Hager, W. W. 1984, SIAM J. Sci. Stat. Comput., 5, p. 311-316; Higham,N. J. & Tisseur, F. 2000, SIAM J. Matrix Anal. Appl., 21, 1185-2001.

```
   >> help linsolve    % only partial listing
    linsolve Solve linear system A*X=B.
    X = linsolve(A,B) solves the linear system A*X=B using
    LU factorization with partial pivoting when A is square.
    Warning is given if A is ill conditioned for square matrice.

    [X, R] = linsolve(A,B) suppresses these warnings and returns R
    the reciprocal of the condition number of A for square matrices.

>> b = A*ones(n, 1)    % Using the matrix A from above
b =      2.45          % Expect solution vector x of 1.0
       1.5929
       1.2179
      0.99563
      0.84563
      0.73654
>> [x, inR] = linsolve(A, b)
x =         1
            1
            1
            1
            1
            1
inR =  3.4399e-008
>> 1.0/inR
ans = 2.907e+007
```

If the relative uncertainties of the matrix parameters (*A* and *b*) are the order of the machine precision, ε_{mach}, the *absolute best* possible case, then the relative uncertainty in the solution vector is

$$\frac{\|\hat{x}-x\|}{\|x\|} \equiv \frac{\|\Delta x\|}{\|x\|} \cong \text{cond}(A)\varepsilon_{mach}.$$

An excellent way to interpret cond(A) is to note that the computed solution loses $\log_{10}(\text{cond}(A))$ digits of accuracy relative to the input accuracy ([p. 61] Heath 2002). Thus, if a condition number is greater than 1000, we lose *all* precision in a model if the input is measured only to three-digit accuracy!

Some properties of the condition number are:

- For any matrix, A, $\text{cond}(A) \geq 1$.
- For the identity matrix, $\text{cond}(I) = 1$.
- For any permutation matrix, $\text{cond}(P) = 1$.
- For any matrix with a nonzero scalar coefficient, γ, $\text{cond}(\gamma A) = \text{cond}(A)$.
- For any diagonal matrix, D, $\text{cond}(D) = (\max|d_i|)/(\min(|d_i|)$.

4.9.2 Residuals

An obvious, but *incorrect*, way to verify the accuracy of an approximate solution to a system of linear equations, *x*, is to substitute the solution into the equation and inspect the difference between the left side (A·x) and the right side (b); the term *residual* is used to quantify this difference -

$$r = b - A \bullet x.$$

The difference between the approximate solution vector, *x*, and the actual solution, x_e, is expressed as

$$\Delta x = x - x_e,$$

which in turn can be represented as

$$A \bullet \Delta x = A \bullet x - A \bullet x_e = A \bullet x - b \Rightarrow$$
$$\Delta x = A^{-1}(A \bullet x - b) = -A^{-1} \bullet r$$
$$\|\Delta x\| = \|-A^{-1} \bullet r\| \le \|A^{-1}\| \cdot \|r\|,$$

dividing this equation by ||x||,

$$\frac{\|\Delta x\|}{\|x\|} \le \frac{\|A^{-1}\| \cdot \|r\|}{\|x\|} = \frac{\|A\| \cdot \|A^{-1}\| \cdot \|r\|}{\|A\| \cdot \|x\|} \Rightarrow$$
$$\frac{\|\Delta x\|}{\|x\|} \le \text{cond}(A)\frac{\|r\|}{\|b\|},$$

Consequently, a small relative residual implies a corresponding small relative solution error only when the matrix A is *well conditioned.*

Examine the example linear system above, which illustrates well the effect of ill conditioning -

```
>> n = 8; A = hilb(n); b = A*ones(n, 1);
>> x = linsolve(A, b);
>> r = A*x - b;
>> disp([x r])
            1           0
            1           0
            1  2.2204e-016
            1           0
            1           0
            1 -1.1102e-016
            1           0
            1           0
>> norm(r, 1)/norm(b, 1)
ans =   3.1404e-017          % relative residual is small ~ 0.15 eps
>> dx = x - ones(n, 1)       % actual error in solution x
dx =  -2.9008e-011
      +1.5723e-009
      -2.0688e-008
      +1.1258e-007
      -3.0436e-007
      +4.3208e-007
      -3.0831e-007
      +8.7177e-008
>> norm(dx, 1)               % norm of actual error in solution
ans =   1.2668e-006
>> cond(A, 1)
ans = 3.3873e+010            % condition number is large!!
>> cond(A,1)*(norm(r, 1)/norm(b,1))
ans =   1.0637e-006          % true estimate of error!
```

4.10 REFERENCES

Dahlquist, G. and Björck A. 2008, *Numerical Methods in Scientific Computing, Volume 2* (SIAM: Philadelphia, PA).

Demmel, J. W. 1997, *Applied Numerical Linear Algebra* (SIAM: Philadelphia, PA).

Golub, G. H. and van Loan, C. H. 1996, *Matrix Computations, 3rd Ed.* (Johns Hopkins: Baltimore)

Heath, M. T. 2002, *Scientific Computing: An Introductory Survey*, 2nd Ed. (McGraw-Hill: NY).

Higham, N. J. 2002, *Accuracy and Stability of Numerical Algorithms 2nd Ed.* (SIAM: Philadelphia, PA).

Higham, D. J., and Higham, N. J. 2002, *MATLAB Guide, 2nd Ed.* (SIAM: Philadelphia, PA).

Kiusalaas, J, 2005, *Numerical Methods in Engineering with MATLAB®* (Cambridge University Press, Cambridge, UK).

Meyers, C. 2001, *Matrix Analysis and Applied Linear Algebra* (SIAM: Philadelphia).

Moler, C. B.2004, *Numerical Computing with MATLAB* (SIAM: Philadelphia).

Stewart, G. W. 2000, *Computing in Science and Engineering*, 2, 50-59.

Strang, G. 1988, *Linear Algebra and its Applications 3rd Ed.* (Cengage Learning: Florence KY).

Trefethen, L. N., and Bau, D. 1997, *Numerical Linear Algebra* (SIAM: Philadelphia).

Watkins, D. S. 2002, *Fundamentals of Matrix Computations 2nd Ed.* (John Wiley: NY)

4.11 QUESTIONS

1. True or false: If matrix A is nonsingular, then the number of solutions to the linear system $A \cdot x = b$ depends on the particular choice of right-hand-side vector b.

2. True or false: If a matrix has a very small determinant, then the matrix is nearly singular.

3. True or false: If a triangular matrix has a zero entry on its main diagonal, then the matrix is nearly singular.

4. True or false: If a matrix has a zero entry on its main then the matrix is singular.

5. True or false: If a linear system is well conditioned, then pivoting is unnecessary in Gaussian elimination.

6. True or false: Gaussian elimination without pivoting fails only when the matrix is ill conditioned or singular.

7. True or false: Once the LU factorization of a matrix has been to solve a linear system, then subsequent linear systems with the same matrix but different right-hand-side vectors can be solved without refactoring the matrix.

8. True or false: In explicit matrix inversion LU factorization and triangular the majority of the work is due to the factorization.

9. True or false: If A is any n-by-n nonsingular matrix, then cond(A) = cond(A^{-1}).

10. True or false: In solving a nonsingular system of linear equations, Gaussian elimination with partial pivoting usually yields a small residual even if the matrix is ill-conditioned.

11. True or false. The multipliers in Gaussian elimination are bounded by 1 in magnitude, so the entries of the successive reduced magnitudes cannot grow in magnitude.

12. Suppose that both sides of a system of linear equations $A \cdot x = b$ are multiplied by a nonzero scalar α.
(a) Does this change the true solution *x?*
(b) Does this change the residual vector $r = b - Ax$ for a given *x?*
(c) What conclusion can be drawn about assessing the quality of a computed solution?

13. *(a)* What is the difference between partial pivoting and complete pivoting in Gaussian elimination? *(b)* State one advantage of each type of pivoting relative to the other.

14. Consider the following matrix *A,* whose LU factorization we wish to compute using Gaussian elimination:

$$A=\begin{bmatrix} +4 & -8 & +1 \\ +6 & +5 & +7 \\ +0 & -10 & -3 \end{bmatrix}$$

(a) No pivoting is used? *(b)* Partial pivoting is used? *(c)* Complete pivoting is used?

15. Give two reasons why pivoting is essential for a numerically stable implementation of Gaussian elimination.

16. If *A* is an ill-conditioned matrix, and its LU factorization is computed by Gaussian elimination with partial pivoting, would you expect the ill-conditioning to be reflected in *L,* in *U,* or both? Why?

17. How does the computational work in solving an *n* x *n* triangular system of linear equations compare with that for solving a general *n* x *n* system of linear equations?

18. Assume that you have already computed the LU factorization, *A* = *LU,* of the nonsingular matrix *A.* How would you use it to solve the linear system $A^T \cdot X = b$?

19. (a) What is the condition number of the following matrix using the I-norm?

$$\begin{bmatrix} +4 & 0 & 0 \\ 0 & -6 & 0 \\ 0 & 0 & +2 \end{bmatrix}$$

(*b)* Does your answer differ using the ∞-norm?

20. Suppose that the *n* x *n* matrix *A* is perfectly well-conditioned, i.e., cond(A) = 1. Which of the following matrices would then necessarily share this same property?
(a) cA, where c is any nonzero scalar
(b) DA, where *D* is a nonsingular diagonal matrix
(c) P *A,* where P is any permutation matrix
(d) BA, where *B* is any nonsingular matrix
(e) A^{-1}, the inverse of *A*

(f) A^T*,* the transpose of *A*

21. Let *A* = diag(1/2) be an *n* x *n* diagonal matrix with all its diagonal entries equal to 1/2.
(a) What is the value of *det(A)?*
(b) What is the value of cond(A)?
(c) What conclusion can you draw from these results?

22. Suppose that the *n* x *n* matrix *A* is exactly singular, but its floating-point representation is nonsingular In this case, what would you expect the order of magnitude of the condition number cond(fl(A)) to be?

23. Classify each of the following matrices as well-conditioned or ill-conditioned:

(a) $\begin{bmatrix} 10^{+10} & 0 \\ 0 & 10^{-10} \end{bmatrix}$

(b) $\begin{bmatrix} 10^{+10} & 0 \\ 0 & 10^{+10} \end{bmatrix}$

(c) $\begin{bmatrix} 10^{-10} & 0 \\ 0 & 10^{-10} \end{bmatrix}$

(d) $\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$

24. Which of the following are good indicators that a matrix is nearly singular?
(a) Its determinant is small.
(*b)* Its norm is small.
(c) Its norm is large.
(d) Its condition number is large.

25. (a) In solving a linear system $A{\cdot}x = b$, what is meant by the *residual* of an approximate solution*:* x_{ap}*?*
(b) Does a small relative residual always imply that the solution is accurate? Why?
(c) Does a large relative residual always imply that the solution is inaccurate? Why?

26. In a floating-point system having 10 decimal digits of precision, if Gaussian elimination with partial pivoting is used to solve a linear system whose matrix has a condition number of 10^3, and whose input data are accurate to full machine precision, about how many digits of accuracy would you expect in the solution?

27. Assume that you are solving a system of linear equations $Ax = b$ on a computer whose floating-point number system has 12 decimal digits of precision, and that the problem data are correct to full machine precision. About how large can the condition number of the matrix *A* be before the computed solution *x* will contain no significant digits?

28. Under what circumstances does a small residual vector $T = b - A{\cdot}x$ imply that *x* is an accurate solution to the linear system $A{\cdot}x = b$*?*

29. What two properties of a matrix *A* together imply that *A* has a Cholesky factorization?

30. List three advantages of Cholesky factorization compared with LU factorization.

31. How many square roots are required to compute the Cholesky factorization of an *n* x *n* symmetric positive definite matrix?

32. Let *A* be an *n xn* symmetric positive definite matrix.

(*a)* What is the (1, 1) entry of its Cholesky factor?
(b) What is the *(n,* 1) entry of its Cholesky factor?

33. What is a singular matrix?

34. Suppose $Ax = b \neq 0$ is a linear system and A is a square, singular matrix. How many solutions is it possible for the system to have?

35. Suppose we know that a matrix is nonsingular. How many solutions are there for the linear system $A \cdot x = 0$?

36. Find all the values of a and b for which the matrix $\begin{bmatrix} a & 1 & 1+b \\ 1 & a & 1 \\ 1-b^2 & 1 & a \end{bmatrix}$ is symmetric positive definite.

37. Let A = $\begin{bmatrix} 1 & 1+\varepsilon \\ 1-\varepsilon & 0 \end{bmatrix}$

a) What is the determinant of A?
b) What is the LU factorization of A?
c) In floating-point arithmetic for what values of ε will the computed value of U be singular?