Chapter 5

Linear Systems: Direct Methods

In this chapter we consider numerical methods for solving a system of linear equations $Ax = b$. We assume that the given matrix $A$ is real, $n \times n$, and nonsingular and that $b$ is a given real vector in $\mathbb{R}^n$, and we seek a solution $x$ that is necessarily also a vector in $\mathbb{R}^n$. Such problems arise frequently in virtually any branch of science, engineering, economics, or finance.

There is really no single technique that is best for all cases. Nonetheless, the many available numerical methods can generally be divided into two classes: direct methods and iterative methods. The present chapter is devoted to methods of the first type. In the absence of roundoff error, such methods would yield the exact solution within a finite number of steps.

The basic direct method for solving linear systems of equations is Gaussian elimination, and its various aspects and variants occupy the first seven sections of this chapter. Section 5.1 presents the method in simple terms.

The bulk of the algorithm involves only the matrix $A$ and amounts to its decomposition into a product of two matrices that have a simpler form. This is called an LU decomposition, developed in Section 5.2. Such an alternative view is useful, for instance, when there are several right-hand-side vectors $b$ each requiring a solution, as the LU decomposition can then be shared.

The simple algorithm of Section 5.1 is not guaranteed to be stable or even well-defined in general. In Section 5.3 we modify it using pivoting strategies to make the algorithm of Gaussian elimination with pivoting practically stable in general. Then in Section 5.4 we discuss efficient methods for implementing these algorithms.

The following three sections are devoted to variants of the same basic algorithm for special classes of matrices. Important simplifications can be made for symmetric positive definite matrices, as discussed in Section 5.5. If the matrix $A$ contains only a few nonzero elements, then it is called sparse. Techniques for storing such matrices and for solving linear systems for the special case where all elements are zero outside a narrow band along the main diagonal are presented in Section 5.6. Direct methods for handling more general sparse matrices are briefly considered in Section 5.7.

In Section 5.8 we consider not the numerical solution of $Ax = b$ but rather its assessment. Since there are roundoff errors, the result of any of our algorithms is approximate, and so the question of how close the approximate solution is to the exact one arises. This in turn depends on the condition number of the matrix, a concept that is defined and further discussed here.
5.1 Gaussian elimination and backward substitution

In this section we show the following:

- How to solve linear equations when $A$ is in upper triangular form. The algorithm is called **backward substitution**.
- How to transform a general system of linear equations into an upper triangular form, to which backward substitution can be applied. The algorithm is called **Gaussian elimination**.

**Note:** Experience suggests that many of our students have already seen much of the material in Section 5.1. As the section count in this chapter increases, though, there may be fewer people to whom this is all old news, so pick your spot for jumping in.

**Backward substitution**

Occasionally, $A$ has a special structure that makes the solution process simple. For instance, if $A$ is **diagonal**, written as

$$
A = \begin{pmatrix}
  a_{11} & 0 & \cdots & 0 \\
  a_{22} & a_{22} & \ddots & \vdots \\
  \vdots & \ddots & \ddots & 0 \\
  a_{nn} & \cdots & a_{nn} & a_{nn}
\end{pmatrix}
$$

(by this notation we mean that the off-diagonal elements are all 0), then the linear equations are uncoupled, reading $a_{ii}x_i = b_i$, and the solution is obviously

$$
x_i = \frac{b_i}{a_{ii}}, \quad i = 1, 2, \ldots, n.
$$

A more involved instance of a special structure is an **upper triangular matrix**

$$
A = \begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  0 & a_{22} & \cdots & a_{2n} \\
  \vdots & \ddots & \ddots & \vdots \\
  0 & \cdots & 0 & a_{nn}
\end{pmatrix}
$$

where all elements below the main diagonal are zero; $a_{ij} = 0$ for all $i > j$.

In this case each equation is possibly coupled only to those following it but not to those preceding it. Thus, we can solve an upper triangular system of equations backwards. The last row reads $a_{nn}x_n = b_n$, so $x_n = \frac{b_n}{a_{nn}}$. Next, now that we know $x_n$, the row before last can be written as $a_{n-1,n-1}x_{n-1} = b_{n-1} - a_{n-1,n}x_n$, so $x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n}}$. Next the previous row can be dealt with, yielding $x_{n-2}$, etc. We obtain the **backward substitution** algorithm given on the next page. In MATLAB this can be written as

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

Copyright © 2011 Society for Industrial and Applied Mathematics
Algorithm: Backward Substitution.

Given an upper triangular matrix $A$ and a right-hand-side $b$,

\[
\begin{align*}
\text{for } k &= n:-1:1 \\
x_k &= b_k - \sum_{j=k+1}^{n} \frac{a_{kj} x_j}{a_{kk}} \\
\end{align*}
\]

end

In Example 4.1 we performed one step of this backward substitution algorithm when obtaining $x_1$ once we knew $x_2$.

It is worth noting that the successful completion of the algorithm is possible only if the diagonal elements of $A$, $a_{ii}$, are nonzero. If any of them is zero, then the algorithm experiences what is known as a breakdown. However, for an upper triangular matrix such as $A$, a zero diagonal term will occur only if the matrix is singular, which violates our assumption from the beginning of this chapter. So we do not need to worry about breakdowns here.

Cost of backward substitution

What is the cost of this algorithm? In a simplistic way we just count each floating point operation (such as $+$ and $*$) as a flop. The number of flops required here is

\[
1 + \sum_{k=1}^{n-1} ((n-k) + (n-k) + 1) = \sum_{k=1}^{n} (2(n-k) + 1) \approx n^2.
\]

In general, this way of evaluating complexity of algorithms considers only part of the picture. It does not take into account data movement between elements of the computer’s memory hierarchy. In fact, concerns of data locality can be crucial to the execution of an algorithm. The situation is even more complex on multiprocessor machines. In MATLAB, the same backward substitution algorithm implemented as

\[
\begin{align*}
\text{for } k &= n:-1:1 \\
x(k) &= b(k); \\
\text{for } j &= k+1:n \\
x(k) &= x(k) - A(k,j) \cdot x(j); \\
\text{end} \\
x(k) &= x(k) / A(k,k); \\
\text{end}
\end{align*}
\]

may take much longer to execute compared to the vectorized script presented before, especially for large matrices, because no advantage is taken of the way in which the matrix and vector elements are stored. A general rule for generating efficient MATLAB implementations is to avoid for- and while-loops whenever possible.

Still, noting that the run time of this algorithm is $O(n^2)$ and not $O(n^3)$ is often meaningful in practice.
Forward substitution

Another instance of a special matrix structure is a lower triangular matrix

\[
A = \begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \ddots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix},
\]

where all elements above the main diagonal are zero: \( a_{ij} = 0 \) for all \( i < j \). The situation is similar to the upper triangular case, except that we apply forward substitution to compute a solution. The algorithm is given on this page.

**Algorithm: Forward Substitution.**

Given a lower triangular matrix \( A \) and a right-hand-side \( b \),

\[
\begin{align*}
&\text{for } k = 1 : n \\
&\quad x_k = \frac{b_k - \sum_{j=1}^{k-1} a_{kj} x_j}{a_{kk}} \\
&\text{end}
\end{align*}
\]

**Example 5.1.** Consider solving the equations

\[
\begin{align*}
5x_1 &= 15, \\
x_1 + 2x_2 &= 7, \\
-x_1 + 3x_2 + 2x_3 &= 5.
\end{align*}
\]

The matrix \( A \) is lower triangular, and in our general notation we have

\[
A = \begin{pmatrix}
5 & 0 & 0 \\
1 & 2 & 0 \\
-1 & 3 & 2
\end{pmatrix}, \quad b = \begin{pmatrix}
15 \\
7 \\
5
\end{pmatrix}.
\]

Applying the forward substitution algorithm we get \( x_1 = \frac{15}{5} = 3 \), then \( x_2 = \frac{7 - 3}{2} = 2 \), then \( x_3 = \frac{-6}{2} = 1 \).

Gaussian elimination

Next, assume that the matrix \( A \) has no special zero-structure. Gaussian elimination is a generalization of the procedure used in Example 4.1. Thus, we use elementary row transformations to reduce the given linear system to an equivalent upper triangular form. Then we solve the resulting system using backward substitution.

The reason this works is that the solution to our problem \( Ax = b \) is invariant to

- multiplication of a row by a constant,
- subtraction of a multiple of one row from another, and
- row interchanges.
5.1. Gaussian elimination and backward substitution

These claims can be easily verified directly. Simply recall that $A\mathbf{x} = \mathbf{b}$ is a concise form for the system of equations

\[
\begin{align*}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\
\vdots &= \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n,
\end{align*}
\]

and check what each of these operations amounts to.

Performing each of these operations can be recorded directly on the augmented matrix

\[
(A|b) = \begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\
a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn} & b_n
\end{pmatrix}.
\]

**Eliminating one column at a time**

1. **Eliminate the first column elements below the main diagonal:**
   - Subtract $\frac{a_{21}}{a_{11}} \times$ the first row from the second row.
   - Subtract $\frac{a_{31}}{a_{11}} \times$ the first row from the third row.
   - \[\vdots\]
   - Subtract $\frac{a_{n1}}{a_{11}} \times$ the first row from the $n$th row.

   This produces

   \[
   (A^{(1)}|b^{(1)}) = \begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} & b_1^{(1)} \\
0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} & b_n^{(1)}
\end{pmatrix}.
\]

2. Next, consider the $(n - 1) \times n$ submatrix of $(A^{(1)}|b^{(1)})$ obtained by ignoring the first row and column. These are the elements that have been modified by the first stage of the process, described above, but not set to 0. Apply exactly the same step as before to this submatrix; i.e., subtract $\frac{a_{22}}{a_{22}^{(1)}} \times$ the second row from the third row, etc.

   The reason we are “allowed” to do this is that the first row remains untouched by these operations, and the first column for the other rows is all zeros, so nothing changes in the first column during subsequent operations.
Chapter 5. Linear Systems: Direct Methods

After this second stage we have an augmented matrix of the form

\[
\begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\
0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} & b_n^{(2)} \\
0 & 0 & 0 & \cdots & a_{nn}^{(2)} & b_n^{(2)}
\end{pmatrix}
\]

The superscripts in the above expression show for each element at which stage it has last been modified.

3. Repeat the process. After \( n - 1 \) such stages we obtain an upper triangular system

\[
\begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\
0 & a_{22}^{(n-1)} & \cdots & a_{2n}^{(n-1)} & b_2^{(n-1)} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & a_{n3}^{(n-1)} & \cdots & a_{nn}^{(n-1)} & b_n^{(n-1)}
\end{pmatrix}
\]

This procedure leads to the Gaussian elimination algorithm in its simplest form given on the next page. It is depicted pictorially for \( n = 4 \) in Figure 5.1.

In stating the Gaussian elimination algorithm we have not bothered to zero out the elements below the main diagonal in the modified matrix \( A \): this part of the matrix is not considered in the ensuing backward substitution.

Of course, for this algorithm to work we need to assume that all those pivotal elements \( a_{kk}^{(k-1)} \) by which we divide are nonzero and in fact not very close to 0 either. We deal with the question of ensuring this later, in Section 5.3. For now, let’s assume there is simply no such issue.

Example 5.2. For Example 4.1 we have \( n = 2 \); hence the loops in the general algorithm collapse into defining just one update for \( k = 1, i = 2, j = 2 \):

\[
(A|b) = \begin{pmatrix}
1 & -4 & -10 \\
0.5 & -1 & -2
\end{pmatrix},
\]

\[
l_{21} = \frac{a_{21}}{a_{11}} = 0.5,
\]

\[
a_{22}^{(1)} = -1 - 0.5 \cdot (-4) = 1,
\]

\[
b_2^{(1)} = -2 - 0.5 \cdot (-10) = 3,
\]

\[
(A^{(1)}|b^{(1)}) = \begin{pmatrix}
1 & -4 & -10 \\
0 & 1 & 3
\end{pmatrix}.
\]

Note that in the algorithm we never actually define \( a_{22}^{(1)} \) or \( b_2^{(1)} \). Instead, we simply overwrite the original value \( a_{22} = -1 \) with the new value \( a_{22} = 1 \) and the original value \( b_2 = -2 \) with the new value \( b_2 = 3 \).
5.1. Gaussian elimination and backward substitution

**Algorithm: Gaussian Elimination.**

Given a real, nonsingular $n \times n$ matrix $A$ and a vector $b$ of size $n$, first transform into upper triangular form.

\[
\text{for } k = 1 : n - 1 \\
\text{for } i = k + 1 : n \\
\quad l_{ik} = a_{ik} / a_{kk} \\
\text{for } j = k + 1 : n \\
\quad a_{ij} = a_{ij} - l_{ik} a_{kj} \\
\text{end} \\
\text{end} \\
b_i = b_i - l_{ik} b_k \\
\text{end} \\
\text{end}
\]

Next, apply the algorithm of backward substitution.

**Figure 5.1.** Gaussian elimination for the case $n = 4$. Only areas of potentially nonzero entries are shown.

The cost of the Gaussian elimination algorithm in terms of operation count is approximately

\[
\sum_{k=1}^{n-1} ((n-k) + 2(n-k)(n-k+1)) \approx 2 \sum_{k=1}^{n-1} (n-k)^2 \\
= 2((n-1)^2 + \cdots + 1) = \frac{2}{3} n^3 + \Theta(n^2)
\]
flops. In particular, as the size of the matrix $A$ increases the cost of Gaussian elimination rises cubically. Comparing this to the cost of backward substitution we see that the cost of the elimination phase dominates for all but very small problems.

Specific exercises for this section: Exercises 1–2.

5.2 LU decomposition

In this section we show that the process of Gaussian elimination in fact decomposes the matrix $A$ into a product $L \times U$ of a lower triangular matrix $L$ and an upper triangular matrix $U$.

Let us continue to assume that the elements $a_{k-1}^{(k-1)}$ encountered in the Gaussian elimination process are all bounded safely away from $0$. We will remove this assumption in the next section.

Elementary lower triangular matrices

Consider the first stage of Gaussian elimination described above. These are elementary row operations applied to the matrix $A$ to zero out its first column below the $(1,1)$ entry. These operations are also applied to the right-hand-side vector $b$. Note, however, that the operations on $b$ can actually be done at a later time because they do not affect those done on $A$.

We can capture the effect of this first stage by defining the elementary $n \times n$ lower triangular matrix

$$M^{(1)} = \begin{bmatrix}
1 \\
-\ell_{21} & 1 \\
-\ell_{31} & \ddots & 1 \\
\vdots & \ddots & \ddots & \ddots \\
-\ell_{n1} & \cdots & \cdots & \cdots & 1
\end{bmatrix}.$$

(This matrix has zeros everywhere except in the main diagonal and in the first column: when describing matrices, blank does not mean nothing—it means zero.) Then the effect of this first stage is seen to produce

$$A^{(1)} = M^{(1)} A \quad \text{and} \quad b^{(1)} = M^{(1)} b.$$

Likewise, the effect of the second stage of Gaussian elimination, which is to zero out the second column of $A^{(1)}$ below the main diagonal, can be written as

$$A^{(2)} = M^{(2)} A^{(1)} = M^{(2)} M^{(1)} A \quad \text{and} \quad b^{(2)} = M^{(2)} b^{(1)} = M^{(2)} M^{(1)} b,$$

where

$$M^{(2)} = \begin{bmatrix}
1 \\
1 & 1 \\
\ell_{32} & \ddots & 1 \\
\vdots & \ddots & \ddots & \ddots \\
\ell_{n2} & \cdots & \cdots & \cdots & 1
\end{bmatrix}.$$

18For those readers who remember and appreciate the usefulness of Riemann sums, the final result can be approximated by recognizing that for large enough $n$ this is nothing but an approximation of the integral $\int_0^n x^2 \, dx$. 

Copyright © 2011 Society for Industrial and Applied Mathematics
5.2. LU decomposition

Obtaining the LU decomposition

This procedure is continued, and after \( n - 1 \) such stages it transforms the matrix \( A \) into an upper triangular matrix, which we call \( U \), by

\[
U = A^{(n-1)} = M^{(n-1)} \cdots M^{(2)} M^{(1)} A.
\]

Likewise, \( b^{(n-1)} = M^{(n-1)} \cdots M^{(2)} M^{(1)} b \). Multiplying \( U \) by \( [M^{(n-1)}]^{-1} \), then by \( [M^{(n-2)}]^{-1} \), etc., we obtain

\[
A = L U,
\]

where \( L = [M^{(1)}]^{-1} \cdots [M^{(n-2)}]^{-1} [M^{(n-1)}]^{-1} \).

To see what \( L \) looks like, note first that \( [M^{(k)}]^{-1} \) has exactly the same form as \( M^{(k)} \), i.e., it’s the identity matrix plus nonzero elements in the \( k \)th column below the main diagonal, except that these elements are without the minus sign, viz., \( l_{k+1,k}, l_{k+2,k}, \ldots, l_{n,k} \). For instance, you can verify directly that the matrix \( M^{(2)} \) defined above, multiplied by

\[
[M^{(2)}]^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ l_{32} & 1 & 1 \\ \vdots & \vdots & \vdots \\ l_{n2} & \cdots & 1 \end{pmatrix}
\]

yields the identity matrix.

So, \( [M^{(k)}]^{-1} \) are also elementary lower triangular matrices. Now, you can verify directly that the product of such matrices yields

\[
L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \\ \vdots & \vdots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{n,n-1} & 1 \end{pmatrix}
\]

The elements of \( L \) are therefore obtained during the Gaussian elimination process!

The LU decomposition is depicted pictorially for \( n = 10 \) in Figure 5.2.

Example 5.3. Let us verify the above claims for the matrix

\[
A = \begin{pmatrix} 1 & -1 & 3 \\ 1 & 1 & 0 \\ 3 & -2 & 1 \end{pmatrix}.
\]
1. The Gaussian elimination process for the first column yields \( l_{21} = \frac{1}{1} = 1, \ l_{31} = \frac{3}{1} = 3 \), so
\[
M^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -3 & 0 & 1 \end{pmatrix}, \quad A^{(1)} = M^{(1)} A = \begin{pmatrix} 1 & -1 & 3 \\ 0 & 2 & -3 \\ 0 & 1 & -8 \end{pmatrix}.
\]

2. The Gaussian elimination process for the second column yields \( l_{32} = \frac{1}{2} \), so
\[
M^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -0.5 & 1 \end{pmatrix}, \quad U = A^{(2)} = M^{(2)} A^{(1)} = \begin{pmatrix} 1 & -1 & 3 \\ 0 & 2 & -3 \\ 0 & 0 & -6.5 \end{pmatrix}.
\]

3. Note that
\[
[M^{(1)}]^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix}, \quad [M^{(2)}]^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.5 & 1 \end{pmatrix},
\]

\[
[M^{(1)}]^{-1}[M^{(2)}]^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 3 & 0.5 & 1 \end{pmatrix} = L,
\]

and
\[
LU = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 3 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} 1 & -1 & 3 \\ 0 & 2 & -3 \\ 0 & 0 & -6.5 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 3 \\ 1 & 1 & 0 \\ 3 & -2 & 1 \end{pmatrix} = A.
\]

This explicitly specifies the LU decomposition of the given matrix \( A \).  

We therefore conclude that the Gaussian elimination procedure of the previous section decomposes \( A \) into a product of a unit lower triangular matrix \( L \) and an upper triangular matrix \( U \).  

\[\text{A unit lower triangular matrix is a lower triangular matrix with 1's on the main diagonal.}\]
5.2. LU decomposition

This is the famous LU decomposition. Together with the ensuing backward substitution the entire solution algorithm for \( Ax = b \) can therefore be described in three steps, as depicted in the algorithm given on this page.

**Algorithm: Solving \( Ax = b \) by LU Decomposition.**

Given a real nonsingular matrix \( A \), apply LU decomposition first:

\[ A = LU. \]

Given also a right-hand-side vector \( b \):

1. **Forward substitution**: solve \( Ly = b \).
2. **Backward substitution**: solve \( Ux = y \).

**Separating the decomposition step**

Note that decomposing \( A \) and solving \( Ly = b \) simultaneously is identical to applying the Gaussian elimination algorithm given on page 99. Note also that the decomposition step costs \( O(n^3) \) operations, whereas the forward and backward substitutions each cost about \( n^2 \) operations. Thus, the decomposition step dominates the total cost. Moreover, since we do not need \( b \) at all to perform the decomposition step, we may perform the LU decomposition once and then solve different linear systems of equations for different right-hand-sides \( b \) at an \( O(n^2) \) cost for each system solve.

**Example 5.4.** Continuing Example 5.3, suppose we are to solve \( Ax = b \) for a new right-hand-side vector

\[
  b = \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix}.
\]

The LU decomposition phase having already been accomplished, we proceed with the forward substitution phase, solving \( Ly = b \) and obtaining \( y_1 = 2 \), \( y_2 = 4 - y_1 = 2 \), \( y_3 = 1 - 3y_1 - 0.5y_2 = -6 \). Then solving \( Ux = y \) yields \( x_3 = \frac{6}{-3} = \frac{12}{13} \), \( x_2 = \frac{1}{2}(2 + 3\frac{12}{13}) = \frac{31}{13} \), \( x_1 = 2 + \frac{31}{13} - 3\frac{12}{13} = \frac{21}{13} \), so

\[
  x = \begin{pmatrix} 21 \\ 31 \\ 12 \end{pmatrix}
\]

is the sought solution.

**Example 5.5.** Let us denote the inverse of a nonsingular matrix \( A \) by \( G = A^{-1} \). If we are given \( G \), then the solution of a linear system \( Ax = b \) can be calculated as \( Gb \), which takes \( O(n^2) \) flops to evaluate in general.
But first we have to find $G$. One way, as good as any, is to use the LU decomposition of $A$. Thus, we calculate $L$ and $U$ first. Now, the identity matrix is composed of the unit vectors $e_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}$ (with the value 1 in the $i$th location, $i = 1, \ldots, n$). For each of these unit vectors we then solve $Ax_i = e_i$ by writing $L(Ux_i) = e_i$ and performing forward and then backward substitutions. Thus $x_i$ is the $i$th column of $G$ and

$$G = A^{-1} = [x_1, \ldots, x_n].$$

Using again a simple operation count, each forward or backward substitution costs about $n^2$ flops, so for $n$ right-hand sides we have $n \cdot 2n^2$ flops added to the cost of the LU decomposition. Discounting lower order terms in $n$ this approximately yields $2\frac{2}{3}n^3 + 2n^3 = \frac{8}{3}n^3$ flops.

Forming $A^{-1}$ explicitly and multiplying by $b$ is generally not a recommended way to solve linear systems of equations for several reasons. For one, it can be wasteful in storage, a point that will become much clearer when we discuss banded matrices in Section 5.6. Moreover, it is more computationally expensive than going by the LU decomposition route, though by less than an order of magnitude. Also, it may give rise to a more pronounced presence of roundoff errors. Finally, it simply has no advantage over the methods presented here to offset the disadvantages mentioned above.

Example 5.6. Another simple, yet general, by-product of LU decomposition is a tool for computing the determinant of a matrix. Recall that the determinant of a product of matrices equals the product of the matrix determinants. Moreover, the determinant of an upper triangular or a lower triangular matrix is the product of the elements on the main diagonal of such a matrix. Thus, for a general square matrix $A = LU$, we have

$$\det(A) = \det(L) \cdot \det(U) = 1 \cdot 1 \cdot u_{11}u_{22} \cdots u_{nn} = u_{11}u_{22} \cdots u_{nn} = \prod_{k=1}^{n} u_{kk},$$

In particular, $A$ is nonsingular if and only if all $u_{kk}$ are nonzero, $k = 1, \ldots, n$.

Note that storage can be handled very efficiently for the LU decomposition. Assuming that the input matrix $A$ can be discarded, its storage locations can be reused to store both $U$ and $L$ in the
The main diagonal of \( L \) need not be stored since all its values are equal to 1.

Specific exercises for this section: Exercises 3–4.
In the above examples the process broke down because a zero pivotal element \( a_{kk}^{(k-1)} \) was encountered. But \( a_{kk}^{(k-1)} \) does not have to exactly equal 0 for trouble to arise: undesirable accumulation of roundoff error may arise also when \( a_{kk}^{(k-1)} \) is near 0.

**Example 5.8.** Consider a perturbation of Example 5.7 that reads

\[
\begin{align*}
x_1 + x_2 + x_3 &= 1, \\
x_1 + 1.0001x_2 + 2x_3 &= 2, \\
x_1 + 2x_2 + 2x_3 &= 1.
\end{align*}
\]

The exact solution, correct to 5 digits, is \( x \approx (1, -1.0001, 1.0001)^T \). Now, Gaussian elimination in exact arithmetic can be completed and yields

\[
\begin{pmatrix}
1 & 1 & 1 \\
1 & 1 & 2 \\
1 & 2 & 2
\end{pmatrix} \Rightarrow \begin{pmatrix}
1 & 1 & 1 \\
0 & 0.0001 & 1 \\
0 & 0 & 0
\end{pmatrix} \Rightarrow \begin{pmatrix}
1 & 1 & 1 \\
0 & 1 & 0 \\
0 & 0 & 0
\end{pmatrix} \Rightarrow \begin{pmatrix}
1 & 1 & 1 \\
0 & 1 & 0 \\
0 & 0 & 0.9999
\end{pmatrix}.
\]

Assume next that we are using floating point arithmetic with base \( \beta = 10 \) and precision \( t = 3 \). Then backward substitution gives

\[
x_3 = 1, \quad x_2 = 0, \quad x_1 = 0.
\]

On the other hand, if we interchange the second and third rows we obtain

\[
\begin{pmatrix}
1 & 1 & 1 \\
1 & 2 & 2 \\
1 & 1.0001 & 2
\end{pmatrix} \Rightarrow \begin{pmatrix}
1 & 1 & 1 \\
0 & 1 & 0 \\
0 & 0 & 0
\end{pmatrix} \Rightarrow \begin{pmatrix}
1 & 1 & 1 \\
0 & 1 & 0 \\
0 & 0 & 0.9999
\end{pmatrix}.
\]

Now backward substitution with 3 decimal digits gives

\[
x_3 = 1.000, \quad x_2 = -1.000, \quad x_1 = 1.000,
\]

which is correct, in that the difference between this solution and the exact solution is less than the rounding unit.

**Partial pivoting**

The problem highlighted in these simple examples is real and general. Recall from Chapter 2 that roundoff errors may be unduly magnified if divided by a small value. Here, if a pivotal element \( a_{kk}^{(k-1)} \) is small in magnitude, then roundoff errors are amplified, both in subsequent stages of the Gaussian elimination process and (even more apparently) in the backward substitution phase.

These simple examples also suggest a strategy to resolve the difficulty, called *partial pivoting*: as the elimination proceeds for \( k = 1, \ldots, n-1 \), at each stage \( k \) choose \( q = q(k) \) as the smallest integer for which

\[
|a_{qk}^{(k-1)}| = \max_{k \leq i \leq n} |a_{ik}^{(k-1)}|,
\]

and interchange rows \( k \) and \( q \). (Recall from page 97 that this operation does not change the solution, provided that the corresponding entries of \( b^{(k-1)} \) are exchanged as well.) Then proceed with the elimination process.
5.3. Pivoting strategies

This Gaussian elimination with partial pivoting (GEPP) strategy certainly resolves the difficulty in all the above examples. In Examples 5.7 and 5.8, in particular, for \( k = 1 \) we have \( q = 1 \), i.e., no row interchange is necessary, but for \( k = 2 \) we get \( q = 3 \), so the second and third rows are interchanged.

How does partial pivoting affect the LU decomposition? Note first that for the matrix of Example 5.7 it really is not possible to write \( A = LU \) with \( L \) unit lower triangular and \( U \) nonsingular upper triangular (try it!). However, the operation of row interchange is captured in general by a permutation matrix \( P \). We start with \( P = I \) and proceed with GEPP; at each stage \( k \) where a row interchange is mandated, we record this fact by interchanging rows \( k \) and \( q \) of \( P \). For Examples 5.7 and 5.8 this yields

\[
PA = LU,
\]

where

\[
P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}
\]

**Forming \( PA = LU \)**

But does this really work in general? Recall from Section 5.2 that \( L \) is obtained as the inverse of a product of elementary matrices, \( L^{-1} = M^{(n-1)} \cdots M^{(2)} M^{(1)} \). Now the process is captured instead by the sequence

\[
B = M^{(n-1)} P^{(n-1)} \cdots M^{(2)} P^{(2)} M^{(1)} p^{(1)},
\]

where each \( P^{(i)} \) is an elementary permutation matrix encoding one row interchange. What is implied, then, is that we can pull all the permutations together and write

\[
B = L^{-1} P,
\]

where \( P = P^{(n-1)} \cdots P^{(2)} P^{(1)} \).

Such a claim is no longer elementary, dear Watson! Indeed, it is by far the most subtle point in Sections 5.1–5.3. But it turns out to hold in general, namely, the straightforward process of partial pivoting that results in \( U = BA \) being upper triangular can be written as \( PA = LU \) for an appropriate unit lower triangular matrix \( L \).

Let us show that this is indeed so. Suppose we have a 4 \( \times \) 4 matrix, and thus the process terminates after three steps. (Everything we discuss here straightforwardly extends to an \( n \times n \) matrix, and the only reason we opt for the specific case of \( n = 4 \) is for ease of exposition.) We have

\[
U = M^{(3)} P^{(3)} M^{(2)} P^{(2)} M^{(1)} P^{(1)} A
\]

Define \( \tilde{M}^{(3)} = M^{(3)} \), \( \tilde{M}^{(2)} = P^{(3)} M^{(2)} (P^{(3)})^T \), and \( \tilde{M}^{(1)} = P^{(3)} P^{(2)} M^{(1)} (P^{(2)})^T (P^{(3)})^T \). Then, by using the fact that the inverse of an elementary permutation matrix is its transpose, we have

\[
\tilde{U} = \tilde{M}^{(3)} \tilde{M}^{(2)} \tilde{M}^{(1)} P^{(3)} P^{(2)} P^{(1)} A
\]

It is tempting to think that we are done, because we have managed to have the \( \tilde{M}^{(j)} \) and the \( \tilde{P}^{(j)} \) comfortably bundled with their own specimen. Not so fast! The order is right, and \( P \) is certainly a valid permutation matrix, but to declare victory we need to be convinced that \( \tilde{M} \) (and hence its inverse) is indeed a unit lower triangular matrix.

In general, a symmetric permutation of a lower triangular matrix does not maintain this structure. The only reason it works in this case is because for a given \( j \), the unit lower triangular matrix \( M^{(j)} \) is basically an identity matrix with the conveniently limited “correction” of having only the
$n - j$ entries in the $j$th column below the main diagonal possibly nonzero. A symmetric permutation of the form we are discussing corresponds to multiplying on the left by an elementary permutation matrix $P^{(i)}$ with $i > j$ and on the right by its transpose, and amounts to swapping rows and columns in a way that merely affects nonzero off-diagonal entries in the strictly lower triangular part of $M^{(j)}$. Indeed, zeros that change places are of no interest, and neither are swapped diagonal entries, which are all equal to 1 to begin with! In Exercises 7 and 8 you are given an opportunity to see all this up close, by working on a $4 \times 4$ matrix with actual numbers and by looking into a similar procedure for symmetric indefinite matrices.

In general, note that the permutation matrix $P$ is nonsingular, being just a permutation of the rows of the identity matrix. Moreover, $P$ is orthogonal, i.e., $P^{-1} = P^T$, so we can write

$$A = (P^T L)U,$$

where the rows and columns of $P^T$ are just unit vectors.

For GEPP it follows immediately that the multipliers $l_{ik}$ (the elements of the unit lower triangular matrix $L$ in the decomposition $PA = LU$) are all bounded in magnitude by 1, written explicitly as

$$|l_{ik}| \leq 1, \quad 1 \leq i, k \leq n.$$

In an actual implementation we will not store a matrix $P$, most of which consists of zeros, nor do we have to physically interchange matrix rows in memory (although we may end up doing the latter for reasons of easier memory access). We can simply keep track of what row interchanges are implied by the algorithm in a one-dimensional array $p$. The elements of $p$ are used for indexing the correct rows, as specified in algorithmic form later on. For example, the array $p$ that corresponds to the matrix $P$ in Examples 5.7 and 5.8 is $p = [1, 3, 2]$.

**GEPP stability**

Does the incorporation of this partial pivoting procedure guarantee stability of the Gaussian elimination algorithm, in the sense that roundoff errors do not get amplified by factors that grow unboundedly as the matrix size $n$ increases? As far as $P^T L$ is concerned the answer is affirmative, but we have to check $U$, too. We need to be assured that

$$g_n(A) = \max_{i,j,k} |a_{ik}|$$

(in words, the maximum magnitude of all elements that arise in the course of the elimination process) does not grow too fast, i.e., does not grow exponentially as a function of $n$.

Unfortunately, it is not possible in general to guarantee stability of GEPP. Indeed, this algorithm is not even invariant to scaling of the rows: in Example 5.8 if the third row were originally multiplied by $10^{-5}$, then no row interchanges would have resulted upon applying the GEPP strategy. But the difficulty highlighted in that example would persist.

We could modify the partial pivoting strategy into one called **scaled partial pivoting**. Thus, define initially for each row $i$ of $A$ a size

$$s_i = \max_{1 \leq j \leq n} |a_{ij}|.$$

Then, at each stage $k$ of the Gaussian elimination procedure, choose $q = q(k)$ as the smallest integer for which

$$\frac{|a_{qk}^{(k-1)}|}{s_q} = \max_{k \leq i \leq n} \frac{|a_{ik}^{(k-1)}|}{s_i},$$

and interchange rows $k$ and $q$. 

Copyright © 2011 Society for Industrial and Applied Mathematics
5.3. Pivoting strategies

This latter strategy entails a computational overhead and cannot completely eliminate potential instability, as we see below in Example 5.9. Still, we should not underestimate scaling as an important tool for enhancing the numerical stability of the process in general. At the very least, we should ensure that the linear system to be solved has rows that all have comparable norm. In certain situations it may be worth paying the extra computational cost of ensuring reasonable scaling throughout the whole elimination process. See Example 5.11 for a case in point.

Example 5.9. Let us give a rare instance where Gaussian elimination with scaled partial pivoting is unstable. Applying it to the matrix

\[
A = \begin{pmatrix}
1 & 0 & \cdots & 0 & 0 \\
-1 & 1 & \cdots & 0 & 0 \\
-1 & -1 & \ddots & \vdots & \vdots \\
-1 & -1 & \ddots & 1 & \vdots \\
-1 & \cdots & \cdots & -1 & 1 \\
\end{pmatrix}
\]

produces no row interchanges, and yet after the \(k\)th step, \(k = 1, 2, \ldots, n - 1\), the elements in the last column satisfy \(a_{n,n}^{(k)} = 2^k, i = k + 1, \ldots, n\) (please check!), so \(g_n(A) = a_{n,n}^{(n-1)} = 2^{n-1}\). Such a growth rate of \(g_n\) is unacceptable.

A strategy that does guarantee stability is complete pivoting: at each stage choose \(q\) and \(r\) as the smallest integers for which

\[
|a_{qr}^{(k-1)}| = \max_{k \leq i, j \leq n} |a_{ij}^{(k-1)}|,
\]

and interchange both row \(q\) and column \(r\) with row \(i\) and column \(j\), respectively. However, this strategy is significantly more expensive than partial pivoting, and instances in which (scaled) partial pivoting really fails in a big way appear to be extremely rare in practice. For this reason, most of the commonly used computer codes for directly solving linear systems use a GEPP approach.

Matrices requiring no pivoting

We end this section by mentioning that there are certain families of matrices for which it is known that pivoting is not required (at least theoretically). One such family is symmetric positive matrices, to the decomposition of which we will devote a whole section soon. Another class are diagonally dominant matrices. An \(n \times n\) matrix \(A\) is diagonally dominant if

\[
|a_{ii}| \geq \sum_{\substack{j=1 \atop j \neq i}}^{n} |a_{ij}|.
\]

In words, for each row the diagonal element is at least as large in magnitude as the sum of magnitudes of the rest of the elements combined. It is possible to show that if a nonsingular matrix is diagonally dominant, Gaussian elimination does not require pivoting.

Specific exercises for this section: Exercises 5–9.
5.4 Efficient implementation

In general, the performance of an algorithm depends not only on the number of arithmetic operations that are carried out but also on the frequency of memory access. So, if the algorithms we have seen thus far in this chapter can be arranged to work on chunks of vectors and matrices, then their efficiency increases. In this section we explore this important practical issue. Specifically, we briefly discuss the efficient implementation of LU decomposition and other basic linear algebra operations.

Vectorization

MATLAB, in particular, used to implement if-, for-, and while-loops in a relatively inefficient way, while doing operations on chunks of matrices very efficiently. MathWorks, the commercial company that owns the MATLAB product, has put considerable effort in the last few years into closing the gaps in performance, and these are now much smaller compared to the past. In any case, different variants of the algorithms we have seen in this chapter which have precisely the same operation count may perform very differently, depending on the implementation. We have already had a glimpse of this in Section 5.1, where the two for-loops which the backward substitution algorithm uses were implemented in a MATLAB script using only one for-loop, with the other replaced by an inner product of the form $\mathbf{A}(k,k+1:n) \cdot \mathbf{x}(k+1:n)$.

Even more important, this also can be done for the LU decomposition algorithm. Let us assume no pivoting at first, and consider the first algorithm we saw on page 99, ignoring the operations on the right-hand-side $\mathbf{b}$. Of the three loops involving $k$, $i$, and $j$, the loop on $k$ appears to be particularly sequential (i.e., it is not easy to couple together or perform more than one stage at a time). The results of the altered matrix at stage $k$ are used at stage $k+1$, and later on. But this is not so for the other two loops. The determination of the multipliers $l_{ik}$, done in the $i$-loop for a given $k$, can be done all at once, as a vector

$$\mathbf{l}_k = \begin{pmatrix} l_{k+1,k} \\ l_{k+2,k} \\ \vdots \\ l_{n,k} \end{pmatrix}.$$ 

Similarly for the subsequent update (the $j$-loop) of the rows of the submatrix of $\mathbf{A}$: each row $i$ is updated by a multiple $l_{ik}$ of the row $\mathbf{a}_{k,k+1:n}$.

For two vectors $\mathbf{y}$ and $\mathbf{z}$ of the same length, the inner product is defined to recall as the scalar

$$\mathbf{y}^T \mathbf{z} = \sum_{i=1}^{m} y_i z_i.$$ 

This is used in the forward and backward substitution algorithms. The outer product, on the other hand, is defined as

$$\mathbf{y} \mathbf{z}^T,$$

i.e., it is the matrix whose $(i, j)$th element is $y_i z_j$.

Example 5.10. For the two vectors

$$\mathbf{y} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}, \quad \mathbf{z} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix},$$
there is no inner product because the lengths don’t match, but there are two outer products, given by

\[ yz^T = \begin{pmatrix} 3 & 6 & 9 \\ 2 & 4 & 6 \end{pmatrix} \quad \text{and} \quad zy^T = \begin{pmatrix} 3 & 2 \\ 6 & 4 \\ 9 & 6 \end{pmatrix}. \]

MATLAB carries out these operations quietly, when issuing instructions such as \( y \ast z' \). So you have to be careful about whether the array you have there corresponds to a row or a column vector, as unwelcome surprises may otherwise result.

Now, the \( i \) and \( j \) for-loops in the LU decomposition algorithm can be expressed in terms of an outer product:

\[
\begin{align*}
\text{for } k &= 1:n-1 \\
\mathbf{l}_k &= (\frac{a_{k+1,1}}{a_{kk}}, \ldots, \frac{a_{nk,1}}{a_{kk}})^T \\
A_{k+1:n, k+1:n} &= A_{k+1:n, k+1:n} - \mathbf{l}_k \ast \mathbf{a}_{k+1:n} \\
\end{align*}
\]

This outer product form allows us to express the bulk of Gaussian elimination as operations on consecutive elements in chunks of matrices. If we now incorporate partial pivoting, then to retain the ability to relate to chunks of \( A \) we will actually execute row interchanges if necessary, rather than use indirect indexing.

**Our code for solving linear systems of equations**

We are now in position to write an efficient linear system solver. It is not as robust, general and efficient as MATLAB’s backslash instruction \( x = A \backslash b \), but on the other hand we can see what is going on. Our code uses the algorithm given on page 103, and you should be able to just read it.

```matlab
function x = ainvb(A,b)
    % function x = ainvb(A,b)
    % solve Ax = b
    [p,LU] = plu (A);
    y = forsub (LU,b,p);
    x = backsub (LU,y);
end

function [p,A] = plu (A)
    % function [p,A] = plu (A)
    % Perform LU decomposition with partial pivoting.
    % Upon return the coefficients of L and U replace those
    % of the input n by n nonsingular matrix A. The row interchanges
    % performed are recorded in the 1D array p.
    n = size(A,1);
```
% initialize permutation vector p
p = 1:n;

% LU decomposition with partial pivoting
for k = 1:n-1
    % find row index of relative maximum in column k
    [val, q] = max(abs(A(k:n,k)));
    q = q + k-1;

    % interchange rows k and q and record this in p
    A([k,q],:) = A([q,k],:);
    p([k,q]) = p([q,k]);

    % compute the corresponding column of L
    J = k+1:n;
    A(J,k) = A(J,k) / A(k,k);

    % update submatrix by outer product
    A(J,J) = A(J,J) - A(J,k) * A(k,J);
end

function y = forsub (A,b,p)
    % Given a unit lower triangular, nonsingular n by n matrix A,
    % an n-vector b, and a permutation p,
    % return vector y which solves Ay = Pb

    n = length(b);

    % permute b according to p
    b = b(p);

    % forward substitution
    y(1) = b(1);
    for k = 2:n
        y(k) = b(k) - A(k,1:k-1) * y(1:k-1);
    end

function x = backsub (A,b)
    % Given an upper triangular, nonsingular n by n matrix A and
    % an n-vector b, return vector x which solves Ax = b

    n = length(b); x = b;
    x(n) = b(n) / A(n,n);
    for k = n-1:-1:1
        x(k) = ( b(k) - A(k,k+1:n) * x(k+1:n) ) / A(k,k);
    end
5.4. Efficient implementation

In the MATLAB version we have used a function like `plu` which actually utilizes the three for-loops on `i`, `j`, and `k` would take about three times longer to obtain the LU decomposition of a $100 \times 100$ matrix. Note that these two implementations have the same operation (or flop) count. Other examples can be easily constructed where two algorithms require the same operation count to achieve the same output, differing only by the amount of memory access, with vastly different CPU times. How can they differ so drastically in execution time, then?

Fast memory access and BLAS

Indeed, the simple flop count that was given in Section 5.1 for Gaussian elimination and for forward and backward substitutions is useful for a coarse (“first cut”) indication of which algorithm is expected to be faster. However, these operation counts completely ignore the relative cost of logical operations and the important question of memory access.

Computer memories are built as hierarchies, ranging from very fast, expensive, and small to slow, cheap, and large. In decreasing order of speed, such a hierarchy would typically include

- registers,
- cache,
- memory, and
- disk.

Arithmetic and logical operations are typically done only at the registers, at the top of the hierarchy, so data stored elsewhere have to be moved into the registers first. Such data movement, especially if it does not start at the cache, can be much more expensive than the cost of an arithmetic operation. Moving data in chunks, rather than using many independent data item moves, makes a big difference in overall performance.

Computer manufacturers, especially of high-performance machines, have standardized basic matrix operations such as matrix-matrix and matrix-vector multiplications into basic linear algebra subroutines (BLAS). The actual implementation is machine-dependent, but the user can see a library of BLAS which are machine-independent, so one is not required to modify code because the laptop battery has died. The BLAS are organized into levels: the higher the level, the larger the number of flops per memory reference that can be achieved. The first level (BLAS1) includes operations such as multiplying a vector by a scalar and adding another vector, as in

$$z = a \times x + y \quad \text{(SAXPY)},$$

as well as inner products. The mysterious word `SAXPY` is actually a widely used term in numerical linear algebra, standing for “scalar alpha X plus Y”. The second level (BLAS2) are matrix-vector multiplications. These include triangular systems and matrix updates by vector outer products. Thus, the LU decomposition utilizing outer products uses level-2 rather than level-1 BLAS. The third-level routines (BLAS3) are for matrix-matrix multiplications and include triangular systems with many right-hand sides. The details become complex, involving, for instance, the possibility of LU decomposition by blocks. This may be important in special circumstances for high-performance computing, but it goes well beyond the scope of this chapter.

20 The S in this acronym is not a plural but rather is a remnant of days when the Fortran language reigned, to the fury of computer scientists.
Scaling the rows of the given problem

Now that we have introduced the function \texttt{plu} let us return to the question of scaling. Note that the partial pivoting strategy implemented here does not take scaling into account, so we should ensure ahead of time that the matrix \( A \) is not scaled too poorly, i.e., replace the problem \( Ax = b \) if necessary by a scaled version \( S Ax = Sb \), with \( S \) a diagonal matrix.

Example 5.11. Let us define a linear system to be solved by the script

\begin{verbatim}
 n = 100; h = 1/(n-1); K = 100;
 A = zeros(n,n);
 for i = 2:n-1
   A(i,i) = -2/h^2 - K;
   A(i,i-1) = -1/h^2; A(i,i+1) = -1/h^2;
 end
 A(1,1) = 1; A(n,n) = 1; % end definition of A
 xe = ones(n,1); % exact solution of 1's
 b = A*xe; % corresponding right hand side
\end{verbatim}

Having defined the exact solution to be \( x = (1, 1, \ldots, 1)^T \), we can calculate the errors that result upon applying our linear system solver introduced above and those for MATLAB's staple "backslash" command. The obtained errors are 1.92e-11 and 5.68e-11, respectively.

Such errors can often be attributed to the conditioning of the problem; see Section 5.8. However, here the source for the loss of accuracy is different. Dividing each \( b_i \) and row \( i \) of \( A \) by \( s_i = \max_{1 \leq j \leq n} |a_{ij}| \) (notice that \( s_1 = s_n = 1 \) and \( s_i \approx 2h^{-2} \) otherwise), and solving the scaled systems by the same two codes, the obtained errors are 1.52e-14 and 1.38e-14, respectively. This is an improvement by three orders of magnitude!

The scaling does induce a different pivoting strategy in \texttt{plu}. Without it the first row is used for pivoting only in the 76th step, whereas with scaling no row permutation is found necessary.

Specific exercises for this section: Exercises 10–11.

5.5 The Cholesky decomposition

Recall from Chapter 4 that a matrix \( A \) is symmetric if \( A^T = A \) and positive definite if

\[ x^T Ax > 0 \quad \forall \ x \neq 0. \]

Such matrices extend in important ways the concept of a positive scalar. In this section we see how solving linear systems for such matrices through the LU decomposition correspondingly simplifies, leading to the Cholesky decomposition.

No need for pivoting

It is easy to see that if \( A \) is symmetric positive definite, then all principal minors \( A_{j,k} \) are also symmetric positive definite (simply consider all vectors \( x \) that have zeros in the first \( j-1 \) and the last \( n-k \) places). The determinants of these principal minors are all positive as well, and it transpires
that the pivots $u_{kk}$ resulting from LU decomposition are also all positive. Indeed, it can be shown that Gaussian elimination without pivoting is a stable algorithm for symmetric positive definite matrices.

Symmetrizing the LU decomposition

Consider the LU decomposition of a symmetric positive definite matrix, written as

\[ A = LU. \]

Recall that $L$ is unit lower triangular, whereas $U$ is upper triangular with diagonal elements $u_{kk} > 0$. We can write

\[ U = \begin{bmatrix} u_{11} & \cdots & \cdots & u_{1n} \\
\vdots & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \vdots \\
u_{n1} & \cdots & \cdots & u_{nn} \end{bmatrix}, \]

so the LU decomposition reads

\[ A = LD\tilde{U}. \]

But transposing it we have

\[ A = A^T = \tilde{U}^T D L^T. \]

Since this decomposition is unique we must have $\tilde{U}^T = L$, i.e., the decomposition reads

\[ A = LDL^T, \]

where $L$ is unit lower triangular and $D$ is diagonal with positive elements $u_{kk}$.

It is also possible to write $D = D^{1/2} D^{1/2}$ with

\[ D^{1/2} = \text{diag}(\sqrt{u_{11}}, \ldots, \sqrt{u_{nn}}), \]

whence the LU decomposition is written as

\[ A = GG^T \]

with $G = LD^{1/2}$ a lower triangular matrix. This is called the Cholesky decomposition. It is possible to derive an algorithm for its implementation which requires not only roughly half the space (due to symmetry, of course) but also half the flops of the general LU decomposition algorithm, viz., $\frac{1}{3}n^3 + O(n^2)$. This can be done by imposing the equality $A = GG^T$ elementwise. As naive as this may sound at first, it is something that one cannot do for Gaussian elimination algorithms for general matrices without being potentially forced to employ complex arithmetic.

Example 5.12. In the $2 \times 2$ case we can require

\[ A = \begin{bmatrix} d_{11} & d_{12} \\
d_{12} & d_{22} \end{bmatrix} = \begin{bmatrix} g_{11} & 0 \\
g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} \\
g_{21} & g_{22} \end{bmatrix}. \]
Now we have precisely three conditions for the three unknown quantities $g_{11}$, $g_{12}$, and $g_{22}$. For $a_{11}$ we get $a_{11} = g_{11}^2$, from which it follows that $g_{11} = \sqrt{a_{11}}$. Next, we have for $a_{12}$ the condition $g_{21} = \frac{a_{12}}{g_{11}}$, where $g_{11}$ is already known. Finally, using the equation for $a_{22}$ we get $g_{22} = \sqrt{a_{22} - g_{21}^2}$.

This algorithm, given below, can be straightforwardly applied to larger matrices in the same fashion. In this algorithm the entries of $A$ are overwritten by the entries of $G$. A key here, and the reason attempting to follow the same strategy for general matrices (that is, not necessarily symmetric positive definite) will fail, is that the arguments of the square root are guaranteed to be positive if $A$ is symmetric positive definite. (This is not entirely trivial to show, but please trust us: it is true.)

**Algorithm: Cholesky Decomposition.**

Given a symmetric positive definite $n \times n$ matrix $A$, this algorithm overwrites its lower part with its Cholesky factor.

```
for k = 1 : n - 1
    a_kk = sqrt(a_kk)
for i = k + 1 : n
    a_ik = a_ik / a_kk
end
for j = k + 1 : n
    for i = j : n
        a_ij = a_ij - a_ik * a_jk
    end
end
a_ii = sqrt(a_ii)
```

In MATLAB, the reliable built-in function `chol` calculates the so-called *Cholesky factor* of a matrix. Notice, though, that $R = chol(A)$ gives the matrix $R = G^T$ such that $A = R^T R$. It does not make a difference whether $R$ or $G$ is used, so long as it is done consistently.

**Example 5.13.** Let us create a symmetric positive definite matrix as follows. For a given $n$, fill an $n \times n$ array $C$ by random numbers. With a little luck, the corresponding matrix $C$ will be nonsingular, and thus $A = C^T C$ will be symmetric positive definite (check this!).

We now create also a random exact solution $x$ and define the right-hand-side $b = Ax$. Next we “forget” that we know $x$ and try to find it, once through the Cholesky decomposition (involving no pivoting) and once through the general LU decomposition procedure with pivoting. The following script carries all this out and records relative errors:

```matlab
C = randn(n,n); A = C'*C;
xe = randn(n,1); % the exact solution
b = A*xe; % generate right-hand-side data
R = chol(A); % Cholesky factor
% the following line is for compatibility with our forsub
```
5.6. Sparse matrices

D = diag(diag(R)); L = D \ R'; bb = D \ b; p = 1:n;
y = forsub(L,bb,p); % forward substitution R'y = b
x = backsub(R,y); % backward substitution Rx = y
rerx = norm(x-xe)/norm(xe) % error by Cholesky

xd = ainvb(A,b); % ignores spd use partial pivoting
rerxd = norm(xd-xe)/norm(xe) % error by general routine

Running this script with \( n = 500 \), say, the routine ainvb uses very few (and, as it turns out, unnecessary!) row permutations. When using the more efficient Cholesky decomposition, errors of at least a similar quality and occasionally better are produced. Of course, no partial pivoting is utilized with Cholesky.

We return to this script in Example 5.20.

Specific exercises for this section: Exercises 12–14.

5.6 Sparse matrices

There are many situations in practice where the matrix is large and sparse. What this means is that most of its elements are equal to zero. For an \( n \times n \) matrix, it is convenient to think of a sparse matrix as having \( O(n) \) nonzero elements.

Frequently, the sparsity of a matrix originates from the nature of the underlying operator. The latter often represents local operations, where only a few components of a given grid or network of nodes “interact” with each other. For instance, the second tridiagonal matrix that we have seen in Example 4.17 in the context of discretization of a one-dimensional second derivative operator is \( N \times N \), but of its \( N^2 \) elements only \( 3N-2 \) are nonzero. For \( h = 0.01 \) we have \( N = 100 \), so this translates into 298 out of 10,000 elements of \( A \) being nonzero. The discretized second derivative is indeed an example of a local operator: its value for a certain point on a grid depends only on the point itself and its two immediate neighbors.

For very large sparse matrices, simply ignoring the sparsity property is not a practical option, because memory requirements may become prohibitive. For instance, think of the above-mentioned tridiagonal matrix, with \( N = 1,000,000 \). Storing the approximately 3 million nonzero elements in double precision, and other data such as row and column indices, consumes a fraction of computer memory in today’s terms: a mere few tens of megabytes. But if we try to store this matrix on all of its 1 trillion entries in memory, then we would be seriously testing (or, in fact, wildly exceeding) the limits of our computer’s fast memory resources, and for no good reason at all: there is no need to waste space on storing zero entries—we just need to know where they are.

Storing sparse matrices

So, a problem has been identified, but how do we solve it? When a matrix is sparse it is not necessarily best to store it as a two-dimensional array. Rather, we could store it in the form of three nonzero one-dimensional arrays, which contain row indices, column indices, and the corresponding numerical values. Notice that the first two are integer arrays.

The triplet form (or coordinate form) is the simplest format: an \( n \times n \) matrix that has \( m \) nonzero entries is represented by vectors \( \{i,j,v\} \), which are \( m \)-long each. For an integer \( 1 \leq k \leq m \), the corresponding \( k \)th entries of \( i_k, j_k, v_k \) represent \( v_k = a_{i_k,j_k} \). This is the format used in MATLAB’s sparse matrix representation.

Other commonly used formats are the compressed sparse row form and the compressed sparse column form. These forms are in fact slightly more economical to work with (in terms of performing matrix operations) than the triplet form. The idea here is that if we go in an orderly fashion by rows
or by columns, one of the integer arrays could simply contain pointers to the start of every row or column in the other integer array, saving some storage along the way. For instance, in sparse row form, the vector $i$ has $n + 1$ entries, whereas $j$ and $v$ have $m$ entries. Column indices of the entries in row $k$ are stored in $j_k$ through $j_{k+1} - 1$.

**Example 5.14.** The matrix

$$
\begin{bmatrix}
2.4 & -3 & 0 \\
0 & 0 & 1.3 \\
0 & 2 & -6
\end{bmatrix}
$$

is represented in MATLAB’s triplet form as follows:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1)</td>
<td>2.4000</td>
<td></td>
</tr>
<tr>
<td>(1, 2)</td>
<td>3.0000</td>
<td></td>
</tr>
<tr>
<td>(3, 2)</td>
<td>2.0000</td>
<td></td>
</tr>
<tr>
<td>(2, 3)</td>
<td>1.3000</td>
<td></td>
</tr>
<tr>
<td>(3, 3)</td>
<td>-6.0000</td>
<td></td>
</tr>
</tbody>
</table>

Note that the entries are presented column by column.

Alternatively, Figure 5.3 illustrates the compressed sparse row form of the matrix.

![Figure 5.3. Compressed row form of the matrix in Example 5.14. Shown are the vectors $i, j, v$. The arrows and the "eof" node are intended merely for illustration of where the elements of the vector $i$ point to in $j$ and correspondingly in $v$.](image)

It is impossible to appreciate the merits of sparse storage schemes for such a small matrix, but these concepts become useful for larger sparse matrices.

We also mention data structures that are based on storing matrices by their diagonals. The MATLAB command `spdiags` does so. This often comes in handy, as many matrices in applications can be characterized in terms of a few nonzero diagonals. The matrices of Example 4.17 are in such a form.
5.6. Sparse matrices

Check out the commands \texttt{sparse} and \texttt{full}. The help documentation for these commands provides the names of a few additional MATLAB commands that deal with sparse matrices.

When representing matrices in sparse form, part of the programming effort is related to writing functions for matrix operations. For example, matrix addition typically requires inserting new nonzero entries, unless the matrices share identical nonzero patterns. It is not difficult to program, but is not as trivial and seamless as matrix addition in the dense case. Instances of basic operations that entail writing specialized routines are the transpose operation, matrix-vector product, matrix multiply, and several other operations.

Banded matrices and their LU decomposition

When solving a linear system of equations, \( Ax = b \), the effectiveness of Gaussian elimination and its variants greatly depends on the particular sparsity pattern of the matrix \( A \). In Section 5.7 we briefly mention ways of increasing the efficiency by performing effective permutations. In Chapter 7 we discuss in detail alternatives to direct methods for situations where forming the LU decomposition results in a prohibitive amount of extra floating point operations and extra storage. But there are special cases of sparse matrices for which the LU decomposition works very effectively. One such instance is the class of banded matrices.

The matrix \( A \) is \textit{banded} if all its elements are zero outside a band of diagonals around the main diagonal. Thus, there are positive integers \( p \) and \( q \), \( 1 \leq p, q \leq n \), such that

\[
 a_{ij} = 0 \text{ if } i \geq j + p \text{ or if } j \geq i + q. \]

The \textit{bandwidth} is defined as the number of consecutive diagonals that may contain nonzero elements, i.e., \( p + q - 1 \). The \textit{upper bandwidth} is \( q \) and the \textit{lower bandwidth} is \( p \).

A particular banded matrix is depicted pictorially for \( n = 10 \), \( p = 2 \), and \( q = 3 \) in Figure 5.4. Elementwise we have

\[
 A = \begin{pmatrix}
     a_{11} & \cdots & a_{1q} \\
     \vdots & \ddots & \vdots \\
     a_{p1} & \cdots & a_{pq} \\
     \vdots & \ddots & \vdots \\
     & \ddots & \ddots \\
     & \vdots & \ddots & \ddots \\
     a_{n-q+1,n} & \cdots & a_{n,n} \\
     a_{n,n-p+1} & \cdots & a_{nn}
  \end{pmatrix}.
\]

For a diagonal matrix, \( p = q = 1 \); for a full (or dense) matrix, \( p = q = n \). More interestingly, in Example 4.17 we see \textit{tridiagonal} matrices, where \( p = q = 2 \).

\footnote{If \( p = q \), then \( p - 1 = q - 1 \) is often referred to as the \textit{semibandwidth} of the matrix.}
Figure 5.4. A banded matrix for the case $n = 10, p = 2, q = 3$. Only areas of potentially nonzero entries are shaded in green.

LU decomposition for banded matrices

Let us consider the LU decomposition without pivoting first. In the loops on $i$ and on $j$ we simply don’t need to zero out elements that are already zero or to update rows with that part of the pivot’s row which is zero. For a banded matrix the algorithm is therefore directly modified and is given on the current page.

Algorithm: LU Decomposition for Banded Matrices.

Given a real, banded, $n \times n$ matrix $A$ which requires no pivoting, the following overwrites the upper triangular part of $A$ by $U$:

```
for $k = 1 : n - 1$
    for $i = k + 1 : k + p - 1$
        $l_{ik} = \frac{a_{ik}}{a_{kk}}$
        for $j = k + 1 : k + q - 1$
            $a_{ij} = a_{ij} - l_{ik} a_{kj}$
        end
    end
end
```
The resulting $L$ and $U$ factors then inherit the band form and can be written as

\[
L = \begin{bmatrix}
1 & &  &  \\
& 1 & &  \\
&  & \ddots & \\
&  & & 1
\end{bmatrix},
\]

\[
U = \begin{bmatrix}
\ell_{p1} & \cdots & \ell_{q1} \\
\ell_{p2} & \cdots & \ell_{q2} \\
\vdots & \ddots & \vdots \\
\ell_{n,n-p+1} & \cdots & \ell_{n,n-1}
\end{bmatrix}.
\]

(Combine Figures 5.2 and 5.4 to see it in color.)

Of course, none of these zero triangles is ever stored. If the band sizes $p$ and $q$ remain small and fixed as the matrix size $n$ grows, then we obtain an algorithm with $O(n)$ storage and $O(n)$ execution time—a great improvement. Applying higher level BLAS is more complicated, though.

For the case of a tridiagonal matrix, in particular, the $i$ and $j$ loops simplify into single statements. The resulting algorithm is occasionally referred to as the **Thomas algorithm**; see Exercise 17.

### Pivoting

Just because a matrix is banded does not mean that it no longer requires pivoting! If the usual row partial pivoting is applied, then the band structure may be altered due to row interchanges. Using such pivoting, the upper bandwidth $q$ of $U$ may increase to $q + p - 1$. Moreover, although the number of nonzeros in each column of the unit lower triangular $L$ remains as without pivoting, they may no longer be tucked next to the main diagonal in a neat band. (Please check this statement, e.g., by constructing a suitable example and running MATLAB’s routine `lu`.)

In the above discussion we did not assume anything about the sparsity pattern within the band. In general, those locations in $A$ where an original zero is overwritten by a nonzero element during LU decomposition are referred to as a **fill-in**. Banded matrices for which no pivoting is required and which do not have zeros within the band yield no fill-in, as we have seen. Row permutations used during partial pivoting do introduce some fill-in into $L$, but not overwhelmingly so when the band is narrow.

However, significant sparsity within a not-very-narrow band is harder to take care of. Banded matrices that have many zeros also within the band serve as a motivating example for the class of...
iterative solution methods, discussed in Chapter 7. Techniques for minimizing the damage caused by fill-in when using direct methods, with which the present chapter is concerned, are discussed next.

Specific exercises for this section: Exercises 15–19.

5.7 Permutations and ordering strategies

We have already noticed that the effectiveness of Gaussian elimination may strongly depend on the sparsity pattern of the matrix; banded matrices introduced in the previous section provide a convincing example that great advances can be made with LU decomposition if no crippling fill-in results. In this section we discuss ways of making the Gaussian elimination procedure for more general sparse matrices as effective as possible by permutations and reordering strategies.

Note: The topic of Section 5.7 is more advanced and more current than those of other sections in this chapter.

The effect of row and column permutations on fill-in

Let us start with a long but motivating example which shows that in fact reordering may result in a significant performance difference in operation count and storage requirements.

Example 5.15. Consider an \( n \times n \) matrix \( A \) whose elements satisfy

\[
a_{ij} = \begin{cases} 
\neq 0 & \text{if } i = j \text{ or } i = 1 \text{ or } j = 1, \\
0 & \text{otherwise.} 
\end{cases}
\]

An illustration of \( A \) for a 5 \( \times \) 5 case is given below, where \( \times \) stands for a possibly nonzero matrix element.\(^{22}\) Consider

\[
A = \begin{pmatrix}
\times & \times & \times & \times & \times \\
\times & 0 & 0 & 0 & 0 \\
\times & 0 & \times & 0 & 0 \\
\times & 0 & 0 & \times & 0 \\
\times & 0 & 0 & 0 & \times 
\end{pmatrix}.
\]

Suppose now that \( C \) is a matrix obtained by swapping the first and the last rows of \( A \), so

\[
C = \begin{pmatrix}
\times & 0 & 0 & 0 & \times \\
\times & \times & 0 & 0 & 0 \\
\times & 0 & \times & 0 & 0 \\
\times & 0 & 0 & \times & 0 \\
\times & \times & \times & \times & \times 
\end{pmatrix}.
\]

\(^{22}\)To fully appreciate the effect of sparsity and fill-in you really should think of a large matrix, say, \( n = 500 \) or \( n = 5000 \).
If \( A \) represents a matrix associated with a linear system \( Ax = b \), then the first and the last elements of \( b \) are also exchanged. This corresponds to simply writing down the equations of the system in a slightly different order, just like we did when applying row partial pivoting in Section 5.3.

However, if \( A \) is symmetric, which can be important, as we saw in Section 5.5, then we lose this property in \( C \). Therefore, we apply the same swapping also to the columns of \( A \). This corresponds to swapping the first and last unknowns and yields a matrix \( B \) whose elements are given by

\[
\begin{align*}
    b_{ij} &= \begin{cases} 
        \neq 0 & \text{if } i = j \text{ or } i = n \text{ or } j = n, \\
        0 & \text{otherwise}.
    \end{cases}
\end{align*}
\]

An illustration of the sparsity pattern of \( B \) for the \( 5 \times 5 \) case is given by

\[
B = \begin{bmatrix}
    \times & 0 & 0 & 0 & \times \\
    0 & \times & \times & \times & \times \\
    0 & \times & 0 & \times & \times \\
    0 & 0 & 0 & \times & \times \\
    \times & \times & \times & \times & \times
\end{bmatrix}.
\]

The matrices \( A \) and \( B \) are often referred to as arrow matrices, for obvious reasons. If \( A \) is symmetric, then so is \( B \).

Suppose that no pivoting is required (for example, if \( A \) and hence \( B \) are diagonally dominant), and let us denote by \( A = L_A U_A \) the LU decomposition of \( A \) and by \( B = L_B U_B \) the LU decomposition of \( B \). What are the nonzero patterns of \( L_A, U_A, L_B, U_B \) and the overall computational costs entailed in solving the system with \( A \) and in solving the system with \( B \)?

For \( A \) the first stage of Gaussian elimination produces a disastrous fill-in: when zeroing out the first column below the (1,1) element, we subtract scalar multiples of the first row from each of the other rows, and this yields

\[
A^{(1)} = \begin{bmatrix}
    \times & \times & \times & \times & \times \\
    0 & \times & \times & \times & \times \\
    0 & \times & \times & \times & \times \\
    0 & \times & \times & \times & \times \\
    0 & \times & \times & \times & \times
\end{bmatrix}.
\]

At this point the original, attractive sparsity pattern of the matrix is gone! The resulting \( A^{(1)} \) looks like what one would get by applying LU decomposition to any generic dense matrix, and subsequently the factors \( L_A \) and \( U_A \) are fully dense.

The situation with the matrix \( B \) is much more encouraging. It is straightforward to see that thanks to the fact that the first row of \( B \) has only nonzero elements in the first and last columns, we get no fill-in, so

\[
B^{(1)} = \begin{bmatrix}
    \times & 0 & 0 & 0 & \times \\
    0 & \times & 0 & \times & \times \\
    0 & \times & \times & \times & \times \\
    0 & 0 & \times & \times & \times \\
    0 & 0 & \times & \times & \times
\end{bmatrix}.
\]
The same happens in the next stages of the LU decomposition, yielding

\[ L_B = \begin{pmatrix}
\times & 0 & 0 & 0 \\
0 & \times & 0 & 0 \\
0 & 0 & \times & 0 \\
0 & 0 & 0 & \times \\
\times & \times & \times & \times
\end{pmatrix}, \quad U_B = \begin{pmatrix}
\times & 0 & 0 & 0 & \times \\
0 & \times & 0 & 0 & \times \\
0 & 0 & \times & 0 & \times \\
0 & 0 & 0 & \times & \times \\
0 & 0 & 0 & 0 & \times
\end{pmatrix}. \]

Thus, the overall storage requirements as well as the overall computational work are very modest for \( B \).

There is nothing special about the size \( n = 5 \); everything above is applicable to a general \( n \). It is worth your time doing Exercise 21: the difference in storage and flop count estimates between handling \( A \) and \( B \) is intriguing, showing that swapping equations and unknowns in the original linear system results in a substantially improved situation.

**Permutation strategies**

To understand things in a more general way, let us revisit the notion of permutation matrices. We have defined those in the context of partial pivoting in Section 5.3 and have observed that a permutation matrix \( P \) satisfies \( P^T = P^{-1} \). Now, suppose that \( P \) is a given permutation matrix, and \( p \) is the corresponding one-dimensional array that represents the same permutation. Then \( Ax = b \) and \( PAx = Pb \) have the same solution. Using \( P^T P = I \), let us write

\[(PA^TP)(Px) = Pb.\]

Doing this incorporates mathematically the fact that we are now looking at \( Px \) rather than \( x \), as per the performed permutation. Note that if \( A \) is symmetric, then so is \( B = PA^TP \). We can rewrite the linear system as

\[By = c,\]

where \( y = Px \) and \( c = Pb \). In Example 5.15, \( P \) is a permutation matrix associated with the array \( p = [n, 2, 3, 4, \ldots, n-2, n-1, 1] \).

In modern direct solvers much attention is devoted to the question of what permutation matrix should be selected. Based on the discussion so far, we may intuitively think of two possible ways of aiming to reduce the storage and computational work:

- Reduce the bandwidth of the matrix.
- Reduce the expected fill-in in the decomposition stage.

The question is how to accomplish the above mentioned goals in practice. One danger to be aware of is the combinatorial nature of the issue at hand. Suffice it to observe that for a linear system with \( n \) unknowns there are \( n! \) different orderings. We certainly do not want to sort through such a number of possibilities when the solution algorithm itself involves \( O(n^3) \) operations; the latter all of a sudden seems small in comparison. Therefore, we need to be assured that the amount of work for determining the ordering does not dominate the computation, and hence seeking an “optimal” graph may turn out to be too costly an adventure. The art of finding effective orderings within a reasonable cost must rely on good heuristics and does not always have a solid theory to back it up.

Commonly used approaches for deriving effective ordering strategies utilize the matrix graph, discussed next.
Matrix graph

Suppose a matrix is structurally symmetric. This means that if $a_{ij} = 0$, then also $a_{ji} = 0$. For instance, the matrices $A$ and $B$ in Example 5.15 are structurally symmetric, whereas $C$ is not. The graph of our matrix $A$ comprises vertices and edges as follows: a row/column index is represented as a vertex, and if $a_{ij} \neq 0$, then there is an edge that connects vertex $i$ to vertex $j$. Since we assume structural symmetry we need not worry about distinguishing between $a_{ij}$ and $a_{ji}$.

The graphs of the matrices $A$ and $B$ of Example 5.15 are depicted in Figure 5.5. The graphs of any matrix $A$ and a symmetric permutation $PAP^T$ are identical in terms of the edges; the only difference is in the labeling of the vertices. In particular, vertex $i$ is replaced by vertex $p_i$, where $p$ is the permutation array.

Figure 5.5. Graphs of the matrices $A$ (left) and $B$ (right) of Example 5.15.

In the process of Gaussian elimination, each stage can be described in terms of the matrix graph as follows: upon zeroing out the $(i, j)$ entry of the matrix with entry $(j, j)$ being the current pivot, the vertices that are the neighbors of vertex $j$ will cause the creation of an edge connecting them to vertex $i$, if such an edge does not already exist. For instance, in Example 5.15, when attempting to zero out entry $(5, 1)$ using entry $(1, 1)$ as pivot, in the graph of $A^{(1)}$ all vertices $j$ connected to vertex 1 generate an edge $(5, j)$, and hence new edges $(2, 5)$, $(3, 5)$, and $(4, 5)$ appear. In contrast, for $B$ no new edge is generated because there are no edges connected to vertex 1 other than vertex 5 itself.

The above underlines the importance of the degree of a vertex, i.e., the number of edges emanating from it. Generally speaking, we should postpone dealing with vertices of a high degree as much as possible. For the matrix $A$ in Example 5.15 all the vertices except vertex 1 have degree 1, but vertex 1 has degree 4 and we start off the Gaussian elimination by eliminating it; this results in disastrous fill-in. On the other hand, for the matrix $B$ all vertices except vertex 5 have degree 1, and vertex 5 is the last vertex we deal with. Until we hit vertex 5 there is no fill because the latter is the only vertex that is connected to the other vertices. When we deal with vertex 5 we identify vertices that should hypothetically be generated, but they are already in existence to begin with, so we end up with no fill whatsoever.

Two vertex labeling strategies

Ordering strategies in abundance have been derived throughout the years. As tempting as it is, we will not describe them in detail.

Many of these algorithms rely on heuristics related to the degrees of the vertices of the matrix graphs and use a good amount of auxiliary objects from graph theory. The reverse Cuthill McKee (RCM) algorithm is one that is effective at reducing bandwidth, and the minimum degree or approximate minimum degree (AMD) algorithm aims at minimizing the expected fill-in.

Copyright © 2011 Society for Industrial and Applied Mathematics
Example 5.16. Let us illustrate how intriguing the nonzero structure of these matrices could be.

The leftmost plot in Figure 5.6 shows the sparsity pattern of a certain well-known symmetric positive definite matrix $A$. This matrix is derived in Example 7.1 on page 168, but all that concerns us here is its zero pattern.

The middle plot of Figure 5.6 shows the effect of RCM ordering. It can be believed from the picture that the goal was to minimize the bandwidth. The pattern is pretty elegant, at least in the eyes of those who appreciate the contemporary art of matrix sparsity plots (or, alternatively, wind surfing). The rightmost plot depicts the result of applying the AMD algorithm. Here it is harder to identify a unifying theme. But note that, as expected, rows with more elements (corresponding to vertices with a higher degree) appear closer to the bottom of the matrix, i.e., their index or label value is higher.

![Figure 5.6. Sparsity pattern of a certain symmetric positive definite matrix $A$ (left), its RCM ordering (middle), and approximate minimum degree ordering (right).](image)

Notice the numbers $nz$ at the bottom of each plot. This stands for “nonzeros” and states how many nonzero entries the matrix has. Since we only permute rows and columns, $nz$ does not change here. But it will change following the procedure in Example 5.17.

The MATLAB code for generating the plots of Figure 5.6 (omitting the generation of $A$ and a couple of output commands) is

```matlab
p = symrcm(A); spy(A(p,p));
p = symamd(A); spy(A(p,p));
```

The actual work is encoded in the two built-in functions with the suggestive names (other than `spy`).

Example 5.17. Figure 5.7 shows the sparsity patterns of the Cholesky factors of the same matrices as in Example 5.16. There is fill in the original matrix as well as for the RCM matrix, but since the latter has a lower bandwidth the overall number of nonzeros is smaller, so we see an improvement. The fill-in is yet smaller for the AMD algorithm, which as expected (by its name, at least) produces the Cholesky factor with approximately the least number of nonzeros.

Try to figure out the corresponding MATLAB instructions for Figure 5.7 that are additional to those displayed in Example 5.16.

Specific exercises for this section: Exercises 20–21.
5.8 Estimating errors and the condition number

Thus far in this chapter we have discussed at length solution methods for solving

\[ Ax = b, \]

and Chapter 7 will add several more. In this section we take a short break from methods and ponder the question of how accurate we may expect our numerical solutions to be.

The error in the numerical solution

Suppose that, using some algorithm, we have computed an approximate solution \( \hat{x} \). With \( x \) denoting as usual the exact, unknown solution, we would like to be able to evaluate the absolute error \( \| x - \hat{x} \| \) or the relative error

\[ \frac{\| x - \hat{x} \|}{\| x \|}, \]

in some norm such as those defined in Section 4.2. Needless to say, an exact expression for these errors is not something we can get our hands on: if the error were known, then the error-free solution \( x \) would also be known! So, we seek an upper bound on the error and rely on quantities that we can directly calculate.

One such quantity is the residual

\[ \hat{r} = b - A\hat{x}. \]

In the ideal world we have \( r = b - Ax = 0 \), but in practice there will always be some nonzero residual \( \hat{r} \) because of roundoff errors. When using iterative methods in Chapter 7 there is also convergence errors to reckon with, as in Chapter 3, but let’s not worry about that now. A stable Gaussian elimination algorithm variant will deliver a residual with a small norm. The question is, what can be concluded from this about the error in \( \hat{x} \)?

Example 5.18. Let

\[ A = \begin{pmatrix} 1.2969 & .8648 \\ .2161 & .1441 \end{pmatrix}, \quad b = \begin{pmatrix} .8642 \\ .1440 \end{pmatrix}. \]
Suppose that, using some algorithm, the approximate solution
\[ \hat{x} = \begin{pmatrix} .9911 \\ - .4870 \end{pmatrix} \]
was calculated. Then
\[ \hat{r} = b - A\hat{x} = \begin{pmatrix} -10^{-8} \\ 10^{-8} \end{pmatrix}, \]
so \( \|\hat{r}\|_\infty = 10^{-8} \). This is a sensibly small residual for many practical purposes. However, the exact solution is
\[ x = \begin{pmatrix} 2 \\ -2 \end{pmatrix}, \] so \( \|x - \hat{x}\|_\infty = 1.513. \)
Thus, the error is of the same size as the solution itself, roughly \( 10^8 \) times that of the residual!

It is important to understand that we cannot expect our algorithms to deliver much more than a small residual. We return to this in Section 7.4.

**Condition number and a relative error estimate**

How does the residual \( \hat{r} \) generally relate to the error in \( \hat{x} \)? We can write
\[ \hat{r} = b - A\hat{x} = Ax - A\hat{x} = A(x - \hat{x}). \]
so
\[ x - \hat{x} = A^{-1}\hat{r}. \]
Let us choose a vector norm (say, the maximum norm), and its corresponding matrix norm.\(^{23}\) Then
\[ \| \hat{r} \| \leq \| A^{-1} \| \| r \|. \]
This gives a bound on the absolute error in \( \hat{x} \) in terms of \( \| A^{-1} \| \). But usually the relative error is more meaningful. Since \( \| b \| \leq \| A \| \| x \| \) implies \( \frac{\| x \|}{\| b \|} \leq \frac{\| A \|}{\| A^{-1} \|} \), we have
\[ \frac{\| x - \hat{x} \|}{\| x \|} \leq \| A^{-1} \| \| \hat{r} \| \| A \| \| b \|. \]
We therefore define the **condition number** of the matrix \( A \) as
\[ \kappa(A) = \| A \| \| A^{-1} \| \]
and write the bound obtained on the relative error as
\[ \frac{\| x - \hat{x} \|}{\| x \|} \leq \kappa(A) \frac{\| \hat{r} \|}{\| b \|}. \] (5.1)

\(^{23}\) Whereas the choice of which vector norm to use is less important for our current purposes, it does fix the choice of the matrix norm as the corresponding induced matrix norm.
In words, the relative error in the solution is bounded by the condition number of the matrix $A$ times
the relative error in the residual. Memorizing this estimate would be worth your while.

The notion of **conditioning** of a problem is fundamental and has been briefly discussed in
Section 1.3. Here it can be expressed in terms of a quantity relating to the matrix $A$ alone. Note that
$\kappa(A)$ does depend on the norm in which it is defined. To make sure it is clear what norm is used
(when it is important to make the distinction) we will use notation such as $\kappa_2(A)$, $\kappa_\infty(A)$, and so on.
Regardless of norm, we have

$$
1 = \|I\| = \|A^{-1}A\| \leq \kappa(A),
$$

at one end of the scale, so a matrix is ideally conditioned if its condition number equals 1, and
$\kappa(A) = \infty$ for a singular matrix at the other end of the scale. In fact, it can be shown that if $A$ is
nonsingular and $E$ is the matrix of minimum $\ell_2$-norm such that $A+E$ is singular, then

$$
\frac{\|E\|_2}{\|A\|_2} = \frac{1}{\kappa_2(A)}.
$$

So, the condition number measures how close to singularity in the relative sense a matrix is.

**Example 5.19.** Continuing with Example 5.18, we have

$$
A^{-1} = 10^8 \begin{pmatrix}
.1441 & -.8648 \\
-.2161 & 1.2969
\end{pmatrix}.
$$

This yields

$$
\|A^{-1}\|_\infty = 1.513 \times 10^8, \quad \text{hence } \kappa_\infty(A) = 2.1617 \times 10^8 \approx 3.27 \times 10^8.
$$

Regarding the actual computed $\hat{x}$, note that $\frac{1.513}{2} < \frac{3.27}{3.8642}$. So, indeed

$$
\frac{\|x - \hat{x}\|_\infty}{\|x\|_\infty} < \kappa_\infty(A) \frac{\|\hat{r}\|_\infty}{\|b\|_\infty}.
$$

However, the mere knowledge that the bound (5.1) holds would not make the computed solution
necessarily satisfactory for all purposes.

Finally, if we modify $a_{22} \leftarrow .8648 \cdot .2161 \cdot 1.2969$, then obviously the matrix becomes singular, because
the columns will be a scalar multiple of one another. So, the matrix $A+E$ is singular, where

$$
E = \begin{pmatrix}
0 & 0 \\
0 & 8648.2161 - .1444
\end{pmatrix} \approx \begin{pmatrix}
0 & 0 \\
0 & -7.7 \times 10^{-9}
\end{pmatrix}.
$$

Indeed, for this particular perturbation we have

$$
\frac{\|E\|}{\|A\|} \approx \frac{1}{\kappa(A)}
$$
in both $\ell_2$- and $\ell_\infty$-norms.

In MATLAB use $\text{cond}(A, \infty)$, or $\text{cond}(A, 2)$, etc., to find the condition number in the
specified norm. But usually there is really no reason, other than for deductive purposes, why we
would want to find $\kappa(A)$ exactly. Typically, we want just a cheap estimate, especially when the condition number may be large. A cheap estimate of a lower bound for $\kappa_1(A)$ is obtained by the instruction `condest(A)`. Assuming we have such an estimate, let us now tie this to an actual computation.

The error when using a direct method

There are two separate issues. The more practical one is the actual estimation, or bounding, of the relative error in the result, given a calculated residual $\hat{r} = b - A\hat{x}$. This has already been answered in Equation (5.1), regardless of how the residual $\hat{r}$ arises.24

The other issue is tying the above to our direct methods for solving linear equations. Consider Gaussian elimination with pivoting, and assume no error other than the accumulation of roundoff errors. The accumulated error in the LU decomposition can be viewed as the exact decomposition of a slightly perturbed matrix $\tilde{A} = A + \delta A$. Then the forward and backward substitutions add more small perturbations, each of the order of the rounding unit times $n$, so the entire algorithm may be considered as producing the exact solution of a perturbed problem

$$(A + \delta A)\hat{x} = b + \delta b.$$ 

This is called backward error analysis. A simple substitution shows that

$$\hat{r} = b - A\hat{x} = (\delta A)\hat{x} - \delta b.$$ 

How large are $\delta A$ and $\delta b$, and thus $\hat{r}$? An LU decomposition procedure with pivoting can be shown to yield a perturbation bounded by

$$\|\delta A\|_\infty \leq \eta \phi(n) g_n(A),$$

where $\phi(n)$ is a low order polynomial in $n$ (cubic at most) and $\eta$ is the rounding unit. The extra perturbations which the forward and backward substitutions yield, in particular the bound on $\|\delta b\|_\infty$, are significantly smaller. Thus, as long as the pivoting keeps $g_n(A)$ growing only moderately in $n$ (like a low order polynomial, but not exponentially), and assuming $n$ not to be gigantic, the overall perturbations $\delta A$ and $\delta b$ are not larger than a few orders of magnitude times $\eta$.

Plugging the expression for $\hat{r}$ into Equation (5.1) we finally obtain

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \kappa(A) \left( \frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right).$$

This can be further beautified, provided $\|\delta A\| < 1/\|A^{-1}\|$, to read

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \kappa(A)\|\delta A\|/\|A\|} \left( \frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right).$$

In summary, a stable algorithm is responsible for producing a small residual. This yields acceptably small error in the solution if the problem is well-conditioned, i.e., if $\kappa(A)$ is not too large. Just what is “small” and what is “too large” may depend on the particular circumstances, but the qualitative concept is general.

---

24The residual could actually arise as a result of measurement errors in data, for instance.
5.8. Estimating errors and the condition number

Example 5.20. Let us return to Example 5.13. If you invoke its script with $n = 500$ several times, you’ll get different results, as befits an unseeded random number generator. But a typical run (unless you are incredibly lucky) would yield errors that are well above the rounding unit.

Using the Cholesky decomposition, in one instance tried we had a relative error of $2.6 \times 10^{-11}$ and in another the relative error was $8.0 \times 10^{-10}$. The corresponding errors using \texttt{ainvb} (which required two and three theoretically unnecessary row permutations) were fairly close and slightly worse. Recall that the default rounding unit is $\eta \approx 10^{-16}$.

In search for the lost digits of accuracy we computed the condition numbers $\kappa_2(A)$ of the constructed random matrices $A$ and these were, respectively, $8.8 \times 10^5$ and $3.9 \times 10^8$. These condition numbers, then, account for the loss of accuracy in the computed solutions.

Indeed, the relative residuals were $8.1 \times 10^{-16}$ and $9.2 \times 10^{-16}$, respectively, which shows not only that both our variants of Gaussian elimination performed very well but also that the bound (5.1) is respected and indicative of actual error behavior.

More on the condition number

The condition number is of fundamental importance in numerical computing. Note that we cannot use the determinant of $A$ to indicate closeness to singularity, as the following example demonstrates.

Example 5.21. Consider $A = 0.1I$, explicitly written as

$$
A = \begin{pmatrix}
0.1 & & & \\
& 0.1 & & \\
& & \ddots & \\
& & & 0.1
\end{pmatrix}
$$

This matrix is perfectly conditioned: $\kappa(A) = 1$. Yet, $\det(A) = 0.1^n$. For $n = 16$ the determinant is already of the order of our standard rounding unit.

The opposite can also hold, namely, a matrix $A$ can be nearly singular and yet its determinant is not small. Such is the case for

$$
A = \begin{pmatrix}
1 & -1 & & & \\
& 1 & -1 & & \\
& & \ddots & \ddots & \\
& & & 1 & -1 \\
& & & & 1
\end{pmatrix}
$$

whose inverse has the element $2^{n-2}$ in the upper right corner. The condition number grows exponentially in $n$ while $\det(A) = 1$.

Conditioning of orthogonal matrices

An important class of matrices that are perfectly conditioned in the induced $\ell_2$-norm are orthogonal matrices, introduced in Section 4.3. Recall that an $n \times n$, real matrix $Q$ is orthogonal if $Q^T = Q^{-1}$. (For instance, a permutation matrix is orthogonal.) Then for any vector $x$ satisfying $\|x\|_2 = 1$ we have

$$
\|Qx\|_2^2 = x^T Q^T Qx = x^T x = 1,
$$
so \( \| Q \|_2 = 1 \). Likewise, \( \| Q^{-1} \|_2 = \| Q^T \|_2 = 1 \), hence \( \kappa_2(Q) = 1 \). Orthogonal transformations are therefore particularly stable and thus a long-time favorite in numerical linear algebra.

**Note:** Orthogonal matrices are ideally conditioned. On the other hand, there is nothing in principle to prevent a symmetric positive definite matrix from having a large condition number.

### Conditioning of symmetric positive definite matrices

Let us consider another important class of matrices, the symmetric positive definite ones. Recall from Section 4.3 that the eigenvalues of such matrices are all real and positive, so let’s write them as

\[
\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n > 0.
\]

Thus, \( \| A \|_2 = \sqrt{\rho(A^T A)} = \sqrt{\lambda_1^2} = \lambda_1 \). Also, the inverse of \( A \) is symmetric positive definite with eigenvalues

\[
0 < \frac{1}{\lambda_1} \leq \frac{1}{\lambda_2} \leq \cdots \leq \frac{1}{\lambda_n}.
\]

Thus, \( \| A^{-1} \|_2 = \frac{1}{\lambda_n} \), and this yields the compact expression

\[
\kappa_2(A) = \frac{\lambda_1}{\lambda_n}.
\]

**Example 5.22.** Recall from Example 4.9 that the “unit circle”, i.e., all the vectors with unit norm, actually corresponds to our usual notion of a circle in \( \mathbb{R}^2 \) when using the \( \ell_2 \)-norm. In Figure 5.8 we plot the mapping of this unit circle by the symmetric positive definite matrix

\[
A = \begin{pmatrix}
1 & .75 \\
.75 & 1
\end{pmatrix}
\]

Producing such a plot is very simple in MATLAB:

```matlab
d = .75;
A = [1,d;d,1];
t = 0:.01:10; m = length(t);
x(1,1:m) = sin(t);
x(2,1:m) = cos(t);
y = A*x;
plot (y(1,:),y(2,:),’b’, x(1,:),x(2,:),’r:’)
xlabel(’x_1’)
ylabel(’x_2’)
```

The eigenvalues of \( A \) are .25 and 1.75, so \( \kappa(A) = \frac{1.75}{.25} = 7 \). These eigenvalues are the lengths of the semi-axes of the ellipse in Figure 5.8. The larger the difference between the eigenvalues (i.e., the larger the condition number), the more “long and skinny” the unit circle becomes under this transformation.

**Specific exercises for this section:** Exercises 22–24.
5.9. Exercises

0. Review questions

(a) What is the overall cost in flops of applying a forward or backward solve?

(b) What is the overall cost in flops of decomposing a matrix using LU?

(c) During the course of Gaussian elimination without pivoting a zero pivot has been encountered. Is the matrix singular?

(d) State three disadvantages of computing the inverse of a matrix to solve a linear system rather than using the LU decomposition approach.

(e) The complete pivoting strategy is numerically stable, whereas partial pivoting is not always stable. Why is the latter approach preferred in practice in spite of this?

(f) Show that for any symmetric positive definite matrix $A$ there is a nonsingular matrix $H$ such that $A = H^T H$.

(g) Give three examples of data structures for storing a sparse matrix. Why aren’t they used for full (dense) matrices, too?

(h) How are the storage requirements and overall computational work different for banded matrices compared with general dense matrices?

(i) State two possible strategies for reordering the equations and unknowns in a way that improves the efficiency of Gaussian elimination for sparse matrices.

(j) What is the fundamental difference between error and residual in terms of computability?

(k) Suppose we compute an approximate solution $\tilde{x}$ for $Ax = b$ and get $r = b - A\tilde{x}$ whose norm is very small. Can we conclude in this case that the error $x - \tilde{x}$ must also be small?

(l) The condition number is an important concept, but it is rarely computed exactly. Why?
1. Consider the problem

\[
\begin{align*}
  x_1 - x_2 + 3x_3 &= 2, \\
  x_1 + x_2 &= 4, \\
  3x_1 - 2x_2 + x_3 &= 1.
\end{align*}
\]

Carry out Gaussian elimination in its simplest form for this problem. What is the resulting upper triangular matrix? Proceed to find the solution by backward substitution.

2. The Gauss–Jordan method used to solve the prototype linear system can be described as follows. Augment \( A \) by the right-hand-side vector \( b \) and proceed as in Gaussian elimination, except use the pivot element \( a_{kk}^{(k-1)} \) to eliminate not only \( a_{ik}^{(k-1)} \) for \( i = k + 1, \ldots, n \) but also the elements \( a_{ik}^{(k-1)} \) for \( i = 1, \ldots, k - 1 \), i.e., all elements in the \( k \)th column other than the pivot. Upon reducing \( (A|b) \) into

\[
\begin{bmatrix}
  a_{11}^{(n-1)} & 0 & \cdots & 0 & b_1^{(n-1)} \\
  0 & a_{22}^{(n-1)} & \ddots & \vdots & b_2^{(n-1)} \\
  \vdots & \ddots & \ddots & 0 & \vdots \\
  0 & \cdots & 0 & a_{nn}^{(n-1)} & b_n^{(n-1)}
\end{bmatrix},
\]

the solution is obtained by setting

\[
x_k = \frac{b_k^{(n-1)}}{a_{kk}^{(n-1)}}, \quad k = 1, \ldots, n.
\]

This procedure circumvents the backward substitution part necessary for the Gaussian elimination algorithm.

(a) Write a pseudocode for this Gauss–Jordan procedure using, e.g., the same format as for the one appearing in Section 5.2 for Gaussian elimination. You may assume that no pivoting (i.e., no row interchanging) is required.

(b) Show that the Gauss–Jordan method requires \( n^3 + \Theta(n^2) \) floating point operations for one right-hand-side vector \( b \)—roughly 50% more than what’s needed for Gaussian elimination.

3. Let \( A \) and \( T \) be two nonsingular, \( n \times n \) real matrices. Furthermore, suppose we are given two matrices \( L \) and \( U \) such that \( L \) is unit lower triangular, \( U \) is upper triangular, and

\[
TA = LU.
\]

Write an algorithm that will solve the problem

\[
Ax = b
\]

for any given vector \( b \) in \( \Theta(n^2) \) complexity. First explain briefly yet clearly why your algorithm requires only \( \Theta(n^2) \) flops (you may assume without proof that solving an upper triangular or a lower triangular system requires only \( \Theta(n^2) \) flops). Then specify your algorithm in detail (including the details for lower and upper triangular systems) using pseudocode or a MATLAB script.
4. The classical way to invert a matrix $A$ in a basic linear algebra course augments $A$ by the $n \times n$ identity matrix $I$ and applies the Gauss–Jordan algorithm of Exercise 2 to this augmented matrix (including the solution part, i.e., the division by the pivots $a_{kk}^{(n-1)}$). Then $A^{-1}$ shows up where $I$ initially was.

How many floating point operations are required for this method? Compare this to the operation count of $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ required for the same task using LU decomposition (see Example 5.5).

5. Let

$$A = \begin{pmatrix} 5 & 6 & 7 & 8 \\ 0 & 4 & 3 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & -2 \end{pmatrix}.$$ \(\text{(a) The matrix } A \text{ can be decomposed using partial pivoting as}

$$PA = LU,$$

\(\text{where } U \text{ is upper triangular, } L \text{ is unit lower triangular, and } P \text{ is a permutation matrix. Find the } 4 \times 4 \text{ matrices } U, L, \text{ and } P.\)

\(\text{(b) Given the right-hand-side vector } b = (26, 9, 1, -3)^T, \text{ find } x \text{ that satisfies } Ax = b. \text{ (Show your method: do not just guess.)}\)

6. Let $B$ be any real, nonsingular $n \times n$ matrix, where $n$ is even, and set $A = B - B^T$. Show that $A$ does not admit an LU decomposition (i.e., some pivoting must be applied, even if $A$ is nonsingular).

7. Let

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 \\ -5 & 0 & -2 & 1 \\ -5 & -3 & 1 & 0 \end{pmatrix}.$$ \(\text{Decompose } A \text{ using partial pivoting as}

$$PA = LU,$$

\(\text{where } U \text{ is upper triangular, } L \text{ is unit lower triangular and } P \text{ is a permutation matrix. Record all the elementary matrices, } P(i), \text{ and } M(i) \text{ along the way. (You can use MATLAB to help you with these small, annoying calculations.)}\)

\(\text{Show that } L \text{ relates to } [M(3)]^{-1} [M(2)]^{-1} [M(1)]^{-1} \text{ in the way explained in Section 5.3, and find } \tilde{M}(1), \tilde{M}(2), \text{ and } \tilde{M}(3).\)

8. Symmetric indefinite matrices require pivoting, and it can be shown that a numerically stable Gaussian elimination process has the form

$$PAP^T = LDL^T,$$
where \( P \) is a permutation matrix and \( D \) is a block diagonal matrix of a certain form. The MATLAB command \( \text{l dl} \) produces such a decomposition.

For simplicity, suppose that \( D \) is diagonal (rather than block diagonal), and suppose we are given a sequence of elementary permutation matrices interleaved with unit lower triangular matrices. Show that, similarly to GEPP in the general case, we can collect all elementary permutation matrices together into a single permutation matrix, bringing about the above defined \( LDL^T \) decomposition.

9. Can you find an example other than the one given on page 109 where scaled partial pivoting is unstable while complete pivoting does well?
   [If you have difficulties finding such an example, then the one in the text may look contrived.]

10. The MATLAB code given in Section 5.4 for solving linear systems of equations, using LU decomposition in outer form with partial pivoting, works well if the matrix \( A \) is nonsingular to a working precision. But if \( A \) is singular, then the exit is not graceful.
    Please fix this by modifying the functions \( \text{ainvb} \) and \( \text{plu} \) to include checks so that \( \text{ainvb}+\text{plu} \) will always return as follows:
    
    (i) In case of a nonsingular system, return silently with the solution \( x \).
    
    (ii) In case of a singular system, display a message regarding the singularity and return with \( x \) assigned the value NaN.

    Assume that MATLAB will complain if a division by a number smaller than \( \epsilon_p \) = 2.22 \( \times \) 10\(^{-16} \) is attempted. (You want to avoid precisely this sort of complaint.)

11. Apply the modified solver obtained in Exercise 10 to the following systems. In each case, check the difference between the computed solution \( x \) and the result of MATLAB’s built-in solver \( A \setminus b \).

    (a) 
    \[
    \begin{align*}
    x_1 + x_2 + x_4 &= 2, \\
    2x_1 + x_2 - x_3 + x_4 &= 1, \\
    4x_1 - x_2 - 2x_3 + 2x_4 &= 0, \\
    3x_1 - x_2 - x_3 + x_4 &= -3.
    \end{align*}
    \]

    (b) Same as the previous system, but with the coefficient of \( x_4 \) in the last equation set to \( a_{4,4} = 2 \).

    (c) 
    \[
    \begin{align*}
    x_1 + x_2 + x_3 &= 1, \\
    x_1 + (1 + 10^{-15})x_2 + 2x_3 &= 2, \\
    x_1 + 2x_2 + 2x_3 &= 1.
    \end{align*}
    \]

    (d) Same as the previous system, but with the second equation multiplied by \( 10^{20} \).

    (e) 
    \[
    A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \ b = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}.
    \]
5.9. Exercises

(f)

\[ A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 3 & 2 & 1 \end{pmatrix} \]

with the same right-hand side as before.

12. The Cholesky algorithm given on page 116 has all those wretched loops as in the Gaussian elimination algorithm in its simplest form. In view of Section 5.4 and the program `ainvb` we should be able to achieve also the Cholesky decomposition effect more efficiently.

Write a code implementing the Cholesky decomposition with only one loop (on \( k \)), utilizing outer products.

13. The script in Example 5.13 will give different answers each time you run it, because the random number generator starts at different spots.

Please fix this, making the program having the option to run on the same input when reinvoked. (Check `help randn` to see how to do this.)

Run the program for \( n = 200 \) and \( n = 500 \). What are your observations?

14. This exercise is for lovers of complex arithmetic.

Denote by \( x^H \) the conjugate transpose of a given complex-valued vector \( x \), and likewise for a matrix. The \( \ell_2 \)-norm is naturally extended to complex-valued vectors by

\[ \|x\|^2 = x^H x = \sum_{i=1}^{n} |x_i|^2, \]

which in particular is real, possibly unlike \( x^T x \).

A complex-valued \( n \times n \) matrix \( A \) is called Hermitian if \( A^H = A \). Further, \( A \) is positive definite if

\[ x^H A x > 0 \]

for all complex-valued vectors \( x \neq 0 \).

(a) Show that the \( 1 \times 1 \) matrix \( A = i \) is symmetric but not Hermitian, whereas \( B = \begin{pmatrix} 2 & i \\ -i & 2 \end{pmatrix} \)

is Hermitian positive definite but not symmetric. Find the eigenvalues of \( A \) and of \( B \).

[In general, all eigenvalues of a Hermitian positive definite matrix are real and positive.]

(b) Show that an \( n \times n \) Hermitian positive definite matrix \( A \) can be decomposed as \( A = LDL^H \), where \( D \) is a diagonal matrix with positive entries and \( L \) is a complex-valued unit lower triangular matrix.

(c) Extend the Cholesky decomposition algorithm to Hermitian positive definite matrices. Justify.

15. (a) Write a MATLAB script that stores a matrix in compressed column form.

(b) Write a symmetric version of your program, that is, a script for symmetric matrices, where if \( a_{i,j} \) is stored \( (i \neq j) \), then \( a_{j,i} \) is not.

Try your programs on a \( 5 \times 5 \) example of your choice.
16. Continuing Exercise 15, write a program that gets a symmetric positive definite matrix in compressed column form as input and performs a sparse Cholesky factorization. The program should generate the Cholesky factor in compressed column form too. To increase modularity, you may want to write a few separate routines that perform operations such as inserting a new nonzero entry into a matrix.

Test your program using the matrix \( A = B^T B \), where \( B \) is the matrix from Exercise 15. Invent additional data as needed for your test.

17. Write a MATLAB function that solves tridiagonal systems of equations of size \( n \). Assume that no pivoting is needed, but do not assume that the tridiagonal matrix \( A \) is symmetric. Your program should expect as input four vectors of size \( n \) (or \( n - 1 \)): one right-hand-side \( b \) and the three nonzero diagonals of \( A \). It should calculate and return \( x = A^{-1}b \) using a Gaussian elimination variant that requires \( \mathcal{O}(n) \) flops and consumes no additional space as a function of \( n \) (i.e., in total \( 5n \) storage locations are required).

Try your program on the matrix defined by \( A \) and for \( n = 10 \), \( a_{i-1,i} = a_{i+1,i} = -i \), and \( a_{i,i} = 3i \) for all \( i \) such that the relevant indices fall in the range 1 to \( n \). Invent a right-hand-side vector \( b \).

18. Apply your program from Exercise 17 to the problem described in Example 4.17 using the second set of boundary conditions, \( v(0) = v'(1) = 0 \), for \( g(t) = \left( \frac{\pi}{2} \right)^2 \sin(\frac{\pi}{2} t) \) and \( N = 100 \). Compare the results to the vector \( u \) composed of \( u(ih) = \sin(\frac{\pi}{2} ih), \ i = 1, \ldots, N \), by recording \( \|v - u\|_\infty \).

19. Suppose that we rewrite the problem of Example 4.17 and Exercise 18 as the system of two simple differential equations,

\[
\begin{align*}
v'(t) &= w(t), \quad w'(t) = -g(t), \\
v(0) &= 0, \quad w(1) = 0.
\end{align*}
\]

From this it is clear that we can solve first for \( w(t) \) and then for \( v(t) \), but let us ignore this and solve for the two unknown functions simultaneously.

On the same mesh as in Example 4.17, define the discretization

\[
\begin{align*}
\frac{v_{i+1} - v_i}{h} &= \frac{w_{i+1} + w_i}{2}, \quad i = 0, \ldots, N - 1, \quad v_0 = 0, \\
\frac{w_{i+1} - w_i}{h} &= -g(ih + h/2), \quad i = 0, \ldots, N - 1, \quad w_N = 0.
\end{align*}
\]

(The resulting values \( v \) in the notation of Exercise 18 should have comparable accuracy as an approximation to \( u \), but let us ignore that, too.)

Defining the vector of \( n = 2N \) unknowns \( x = (w_0, v_1, w_1, v_2, \ldots, v_{N-1}, w_{N-1}, v_N)^T \), construct the corresponding matrix problem. Show that the resulting matrix is banded: what is the bandwidth? How would you go about finding \( x \)?

20. Consider the LU decomposition of an upper Hessenberg (no, it’s not a place in Germany) matrix, defined on the facing page, assuming that no pivoting is needed: \( A = LU \).

(a) Provide an efficient algorithm for this LU decomposition (do not worry about questions of memory access and vectorization).

(b) What is the sparsity structure of the resulting matrix \( L \) (i.e., where are its nonzeros)?

(c) How many operations (to a leading order) does it take to solve a linear system \( Ax = b \), where \( A \) is upper Hessenberg?
5.10. Additional notes

(d) Suppose now that partial pivoting is applied. What are the sparsity patterns of the factors of \( A \)?

21. For the arrow matrices of Example 5.15 determine the overall storage and flop count requirements for solving the systems with \( A \) and with \( B \) in the general \( n \times n \) case.

22. Given that \( a \) and \( b \) are two real positive numbers, the eigenvalues of the symmetric tridiagonal matrix \( A = \text{triu} \begin{bmatrix} b, a, b \end{bmatrix} \) of size \( n \times n \) are \( \lambda_j = a + 2b \cos \left( \frac{\pi j}{n+1} \right), j = 1, \ldots, n \). (A nonsparse version of this matrix can be obtained in MATLAB with the instruction

\[
A = \text{diag}(\text{a} \ast \text{ones}(n,1),0) + \text{diag}(\text{b} \ast \text{ones}(\text{n-1},1),1) + \text{diag}(\text{b} \ast \text{ones}(\text{n-1},1),-1)
\]

Type \text{help spdiags} for the scoop on the sparse version.)

(a) Find \( \| A \|_\infty \).

(b) Show that if \( A \) is strictly diagonally dominant, then it is symmetric positive definite.

(c) Suppose \( a > 0 \) and \( b > 0 \) are such that \( A \) is symmetric positive definite. Find the condition number \( \kappa_2(A) \). (Assuming that \( n \) is large, an approximate value would suffice. You may also assume that \( a \neq 2b \).)

23. Let \( b + \delta b \) be a perturbation of a vector \( b \) (\( b \neq 0 \)), and let \( x \) and \( \delta x \) be such that \( Ax = b \) and \( A(x + \delta x) = b + \delta b \), where \( A \) is a given nonsingular matrix. Show that

\[
\frac{\| \delta x \|}{\| x \|} \leq \kappa(A) \frac{\| \delta b \|}{\| b \|}.
\]

24. Run the MATLAB script of Example 5.22, plugging in different values for \( d \). In particular, try \( d = .25 \), \( d = .85 \), and \( d = -.75 \). What do you observe? What will happen as \( d \downarrow (-1) \)?

\textbf{Hessenberg matrix}

An \( n \times n \) matrix \( A \) is said to be in \textit{Hessenberg} or \textit{upper Hessenberg} form if all its elements below the first subdiagonal are zero, so that

\[
a_{ij} = 0, \quad i > j + 1.
\]

5.10 Additional notes

This chapter is the first in our text to address and describe some important and nontrivial numerical methods, or at least major components thereof, that see an enormous amount of use in real-world applications. It is not for naught that the MATLAB command \( \backslash \) is both the shortest in the language’s repertoire and one that invokes a program rumored to contain over 100,000 lines of code.

Just about any introductory text on numerical methods addresses in various ways the material in Sections 5.1–5.3 and usually also Section 5.5, part of Section 5.6, and Section 5.8.

The analysis of stability for the various variants of Gaussian elimination with pivoting was done by J. Wilkinson in the 1960s and is considered a classic of numerical analysis. Several modern texts describe it in detail, e.g., Trefethen and Bau [70], Golub and van Loan [30], Watkins [74], and Demmel [21].
The Cholesky decomposition is loved for much more than the relatively modest gains in CPU time and storage that it offers over LU decomposition. It is a very clean decomposition that in essence corresponds to taking the square root of a positive scalar.

A much more thorough discussion than what is contained in Section 5.4 can be found in [21]. A modern reference discussing the important topic of direct methods for sparse matrices is Davis [19]. Occasionally, solution of many linear systems \( A x_j = b_j, \ j = 1, 2, 3, \ldots \), is required, where \( A \) is fixed, large, and sparse, and the given right-hand-side vectors \( b_j \) are not particularly close to one another. For such problems, the iterative methods discussed in Chapter 7 lose some of their attraction, because they must deal with each right-hand side essentially from scratch. Direct methods may be preferred then, because the LU decomposition of \( A \) is carried out just once and then only a pair of forward-backward substitutions is required for each \( j \). Lots of fast access memory may be required for the successful utilization of such direct methods, though, because of the fill-in that results from the matrix factorization even when the best ordering strategies are utilized. This consideration is central enough to influence in a major way the sort of computing equipment purchased for carrying out simulations for certain real-world applications.