1.5 BANDED POSITIVE DEFINITE SYSTEMS

Large systems of equations occur frequently in applications, and large systems are usually sparse. In this section we will study a simple yet very effective scheme for applying Cholesky's method to large, positive definite systems of equations that are banded or have an envelope structure. This method is less popular than it once was, as more efficient methods have become widely available [14] (and even implemented in MATLAB [38]). It is worth studying nevertheless, as it shows via very simple arguments that enormous savings in computer time and storage space can be achieved by exploiting sparseness. More sophisticated sparse matrix methods are discussed in Section 1.6. For extremely large systems, iterative methods are preferred. These are discussed in Chapter 8.

A matrix A is *banded* if there is a narrow band around the main diagonal such that all of the entries of A outside of the band are zero, as shown in Figure 1.10. More precisely, if A is $n \times n$, and there is an $s \ll n$ such that $a_{ij} = 0$ whenever

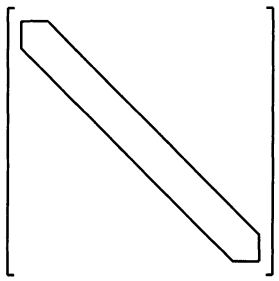


Figure 1.10 Banded matrix: All entries outside of the band are zero.

$|i - j| > s$, then all of the nonzero entries of A are confined to a band of $2s + 1$ diagonals centered on the main diagonal. We say that A is *banded* with *band width* $2s+1$. Since we are concerned with symmetric matrices in this section, we only need half of the band. Since $a_{ij} = 0$ whenever $i - j > s$, there is a band of s diagonals above the main diagonal that, together with the main diagonal, contains all of the nonzero entries of A. We say that A has *semiband width* s.

Example 1.5.1 Consider the mass-spring system depicted in Figure 1.11. This is

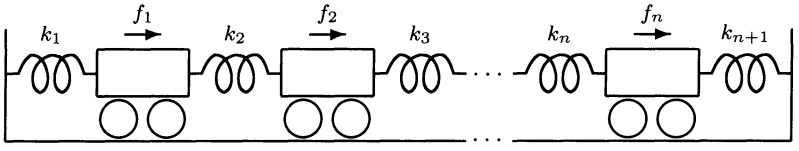


Figure 1.11 System of n masses

exactly the system that we discussed in Exercise 1.2.20. There are n carts attached

by springs. If forces are applied to the carts, we can calculate their displacements x_i by solving a system $Ax = b$ of n equations in n unknowns. Since the ith cart is directly attached only to the two adjacent carts, the ith equation involves only the unknowns x_{i-1}, x_i, and x_{i+1}. Thus its form is

$$a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1} = b_i,$$

and $a_{ij} = 0$ whenever $|i - j| > 1$. This is an extreme example of a banded coefficient matrix. The band width is 3 and the semiband width is 1. Such matrices are called *tridiagonal.* □

Example 1.5.2 Consider a 100×100 system of equations $Ax = b$ associated with the grid depicted in Figure 1.12. The ith grid point has one equation and one

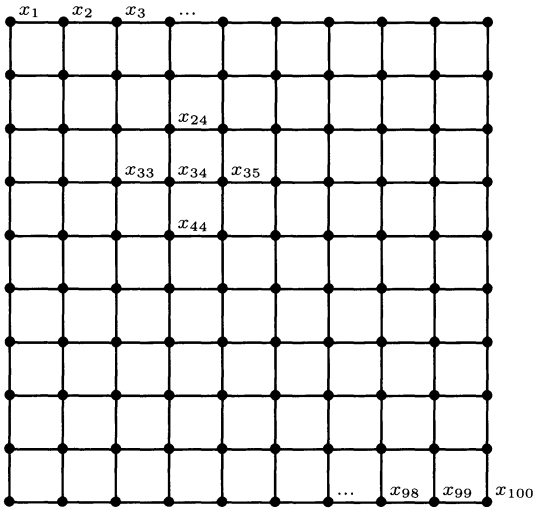


Figure 1.12 "Large" grid

unknown associated with it. For example, the unknown could be a nodal voltage (or a displacement, a pressure, a temperature, a hydraulic head, ...). Assume that the ith equation involves only the unknowns associated with the ith grid point and the grid points that are directly connected to it. For example, the 34th equation involves only the unknowns x_{24}, x_{33}, x_{34}, x_{35}, and x_{44}. This means that in the 34th row of A, only the entries $a_{34,24}$, $a_{34,33}$, $a_{34,34}$, $a_{34,35}$, and $a_{34,44}$ are nonzero. The other 95 are zero. Clearly the same is true for every other equation; no row of A contains more than five nonzero entries. Thus the matrix is very sparse. It is also banded, if the equations and unknowns are numbered as shown in the figure. Clearly $a_{ij} = 0$ if $|i - j| > 10$. Thus the system is 100×100 with a semiband width of 10. □

Exercise 1.5.3 Make a rough sketch of the matrix A of Example 1.5.2, noting where the zeros and nonzeros lie. Notice that even within the band, most of the entries are zero. Most large application problems have this feature. □

Exercise 1.5.4 Modern applications usually involve matrices that are much larger than the one discussed in Example 1.5.2. Figure 1.12 depicts a 10×10 network of nodes. Imagine an $m \times m$ network with $m \gg 10$. How many equations does the resulting system have? How many nonzeros does each row of the matrix have? What is the bandwidth of the system, assuming that the equations and unknowns are numbered as depicted in Figure 1.12? Answer these questions in general and also in the specific cases (a) $m = 100$, (b) $m = 1000$. □

Notice that the bandedness depends on how the nodes are numbered. If, for example, nodes 2 and 100 are interchanged in Figure 1.12, the resulting matrix is not banded, since $a_{100,1}$ and $a_{1,100}$ are nonzero. However it is still sparse; the number of nonzero entries in the matrix does not depend on how the nodes are ordered.

If a network is regular, it is easy to see how to number the nodes to obtain a narrow band. Irregular networks can also lead to banded systems, but it will usually be more difficult to decide how the nodes should be numbered.

Banded positive definite systems can be solved economically because it is possible to ignore the entries that lie outside of the band. For this it is crucial that the Cholesky factor inherits the band structure of the original matrix. Thus we can save storage space by using a data structure that stores only the semiband of A. R can be stored over A. Just as importantly, computer time is saved because all operations involving entries outside of the band can be skipped. As we shall soon see, these savings are substantial.

Instead of analyzing banded systems, we will introduce a more general idea, that of the envelope of a matrix. This will increase the generality of the discussion while simplifying the analysis. The *envelope* of a symmetric or upper-triangular matrix A is a set of ordered pairs (i, j), $i < j$, representing element locations in the upper triangle of A, defined as follows: (i, j) is in the envelope of A if and only if $a_{kj} \neq 0$ for some $k \leq i$. Thus if the first nonzero entry of the jth column is a_{mj} and $m < j$, then (m, j), $(m + 1, j)$, $\ldots$, $(j - 1, j)$ are the members of the envelope of A from the jth column.

The crucial theorem about envelopes (Theorem 1.5.7) states that if R is the Cholesky factor of A, then R has the same envelope as A. Thus A can be stored in a data structure that stores only its main diagonal and the entries in its envelope, and R can be stored over A. All operations involving the off-diagonal entries lying outside of the envelope can be skipped. If the envelope is small, substantial savings in computer time and storage space are realized. Banded matrices have small envelopes.

A simple example of an unbanded matrix with a small envelope is

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 & 0 & -1 \\ -1 & 2 & -1 & \ddots & & 0 & 0 \\ 0 & -1 & 2 & \ddots & & & 0 \\ \vdots & \ddots & \ddots & \ddots & & & \vdots \\ 0 & & & & 2 & -1 & 0 \\ 0 & 0 & & & -1 & 2 & -1 \\ -1 & 0 & 0 & \cdots & 0 & -1 & 2 \end{bmatrix}, \tag{1.5.5}$$

which was obtained from discretization of an ordinary differential equation with a periodic boundary condition.

Exercise 1.5.6 Identify the envelope of the matrix in (1.5.5). Assuming the matrix is $n \times n$, approximately what fraction of the upper triangle of the matrix lies within the envelope? □

Like the band width, the envelope of a matrix depends on the order in which the equations and unknowns are numbered. Often it is easy to see how to number the nodes to obtain a reasonably small envelope. For those cases in which it is hard to tell how the nodes should be ordered, there exist algorithms that attempt to minimize the envelope in some sense. For example, see the discussion of the reverse Cuthill-McKee algorithm in [37].

Theorem 1.5.7 *Let A be positive definite, and let R be the Cholesky factor of A. Then R and A have the same envelope.*

Proof. Consider the bordered form of Cholesky's method. At the jth step we solve a system

$$R_{j-1}^T h = c,$$

where $c \in \mathbb{R}^{j-1}$ is the portion of the jth column of A lying above the main diagonal, and h is the corresponding portion of R. (See equations (1.4.36) and (1.4.37).) Let $\hat{c} \in \mathbb{R}^{s_j}$ be the portion of c that lies in the envelope of A. Then

$$c = \begin{bmatrix} 0 \\ \hat{c} \end{bmatrix}.$$

In Section 1.3 we observed that if c has leading zeros, then so does h:

$$h = \begin{bmatrix} 0 \\ \hat{h} \end{bmatrix},$$

where $\hat{h} \in \mathbb{R}^{s_j}$. See (1.3.6) and the accompanying discussion. It follows immediately that the envelope of R is contained in the envelope of A. Furthermore it is not hard

to show that the first entry of $\hat{h}$ is nonzero. Thus the envelope of R is exactly the envelope of A. □

Corollary 1.5.8 *Let A be a banded, positive definite matrix with semiband width s. Then its Cholesky factor R also has semiband width s.*

Referring to the notation in the proof of Theorem 1.5.7, if $\hat{c} \in \mathbb{R}^{s_j}$, then the cost of the arithmetic in the jth step of Cholesky's method is essentially equal to that of solving an $s_j \times s_j$ lower-triangular system, that is, s_j^2 flops. (See the discussion following (1.3.6).) If the envelope is not exploited, the cost of the jth step is j^2 flops. To get an idea of the savings that can be realized by exploiting the envelope structure of a matrix, consider the banded case. If A has semiband width s, then the portion of the jth row that lies in the envelope has at most s entries, so the flop count for the jth step is about s^2. Since there are n steps in the algorithm, the total flop count is about ns^2.

Exercise 1.5.9 Let R be an $n \times n$ upper-triangular matrix with semiband width s. Show that the system $Rx = y$ can be solved by back substitution in about $2ns$ flops. An analogous result holds for lower-triangular systems. □

Example 1.5.10 The matrix of Example 1.5.2 has $n = 100$ and $s = 10$. If we perform a Cholesky decomposition using a program that does not exploit the band structure of the matrix, the cost of the arithmetic is about $\frac{1}{3}n^3 \approx 3.3 \times 10^5$ flops. In contrast, if we do exploit the band structure, the cost is about $ns^2 = 10^4$ flops, which is about 3% of the previous figure. In the forward and back substitution steps, substantial but less spectacular savings are achieved. The combined arithmetic cost of forward and back substitution without exploiting the band structure is about $2n^2 = 2 \times 10^4$ flops. If the band structure is exploited, the flop count is about $4ns = 4 \times 10^3$, which is 20% of the previous figure.

If the matrix is stored naively, space for $n^2 = 10,000$ numbers is needed. If only the semiband is stored, space for not more than $n(s+1) = 1100$ numbers is required. □

The results of Example 1.5.10, especially the savings in flops in the Cholesky decomposition, are already impressive, even though the matrix is not particularly large. Much more impressive results are obtained if larger matrices are considered, as the following exercise shows.

Exercise 1.5.11 As in Exercise 1.5.4, consider the banded system of equations arising from an $m \times m$ network of nodes like Figure 1.12 but larger, with the nodes numbered by rows, as in Figure 1.12.

(a) For the case $m = 100$ (for which $n = 10^4$) calculate the cost of solving the system $Ax = b$ (Cholesky decomposition plus forward and back substitution) with and without exploiting the band structure. Show that exploiting the band structure cuts the flop count by a factor of several thousand. Show that if only

the semiband is stored, the storage space required is only about 1% of what would be required to store the matrix naively.

(b) Repeat part (a) with $m = 1000$.

□

Savings such as these can make the difference between being able to solve a large problem and not being able to solve it.

The following exercises illustrate another important feature of banded and envelope matrices: The envelope structure of A is *not* inherited by A^{-1}. In fact, it is typical of sparse matrices that the inverse is not sparse. Thus it is highly uneconomical to solve a sparse system $Ax = b$ by finding the inverse and computing $x = A^{-1}b$.

Exercise 1.5.12 Consider a mass-spring system as in Figure 1.11 with six carts. Suppose each spring has a stiffness $k_i = 1$ newton/meter.

(a) Set up the tridiagonal, positive definite, coefficient matrix A associated with this problem.

(b) Use the MATLAB `chol` command to calculate the Cholesky factor R. Notice that R inherits the band structure of A. (To learn an easy way to enter this particular A, type `help toeplitz`.)

(c) Use the MATLAB `inv` command to calculate A^{-1}. Notice that A^{-1} does not inherit the band structure of A.

□

Exercise 1.5.13 In the previous exercise, the matrix A^{-1} is full; none of its entries are zero.

(a) What is the physical significance of this fact? (Think of the equation $x = A^{-1}b$, especially in the case where only one entry, say the jth, of b is nonzero. If the (i, j) entry of A^{-1} were zero, what would this imply? Does this make physical sense?)

(b) The entries of A^{-1} decrease in magnitude as we move away from the main diagonal? What does this mean physically?

□

Exercise 1.5.14 Consider the linear system $Ax = b$ from Exercise 1.2.19. The matrix A is banded and positive definite.

(a) Use MATLAB to compute the Cholesky factor R. Observe that the envelope is preserved.

(b) Use MATLAB to calculate A^{-1}. Observe that A^{-1} is full. (What is the physical significance of this?)

□

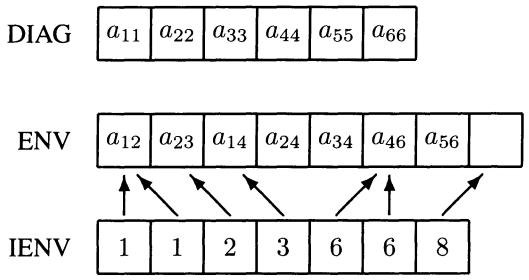
Envelope Storage Scheme

A fairly simple data structure can be used to store the envelope of a coefficient matrix. We will describe the scheme from [37]. A one-dimensional real array DIAG of length n is used to store the main diagonal of the matrix. A second one-dimensional real array ENV is used to store the envelope by columns, one after the other. A third array IENV, an integer array of length $n + 1$, is used to store pointers to ENV. Usually IENV(J) names the position in ENV of the first (nonzero) entry of column J of the matrix. However, if column J contains no nonzero entries above the main diagonal, then IENV(J) points to column $J + 1$ instead. Thus the absence of nonzero entries in column J above the main diagonal is signaled by IENV(J) = IENV($J + 1$). IENV($n + 1$) points to the first storage location after the envelope. These rules can be expressed more succinctly (and more accurately) as follows: IENV(1) = 1 and IENV($J + 1$) - IENV(J) equals the number of elements from column J of the matrix that lie in the envelope.

Example 1.5.15 The matrix

$$\begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ & a_{22} & a_{23} & a_{24} & 0 & 0 \\ & & a_{33} & a_{34} & 0 & 0 \\ & & & a_{44} & 0 & a_{46} \\ & & & & a_{55} & a_{56} \\ & & & & & a_{66} \end{bmatrix}$$

is stored as follows using the envelope scheme:



□

When the envelope storage scheme is used, certain formulations of the Cholesky decomposition, forward-substitution, and back-substitution algorithms are much more appropriate than others. For example, we would not want to use the outer-product formulation of Cholesky's method, because that algorithm operates on (the upper triangle of) A by rows. In the envelope storage scheme A is stored by columns; rows are hard to access. The inner-product formulation is inappropriate for the same reason.[6] From the proof of Theorem 1.5.7 it is clear that the bordered form of

[6]The inner-product formulation accesses A by both rows and columns.

Cholesky's method is appropriate. At each step virtually all of the work goes into solving a lower-triangular system $R_{i-1}^T h = c$, where

$$c = \begin{bmatrix} 0 \\ \hat{c} \end{bmatrix} \qquad \text{and} \qquad h = \begin{bmatrix} 0 \\ \hat{h} \end{bmatrix}.$$

If we partition R_{i-1}^T conformably,

$$R_{i-1}^T = \begin{bmatrix} H_{11} & 0 \\ H_{21} & H_{22} \end{bmatrix},$$

the equation $R_{i-1}^T h = c$ reduces to $H_{22}\hat{h} = \hat{c}$. H_{22} is a lower-triangular matrix consisting of rows and columns $i - s_i$ through $i - 1$ of R^T. A subroutine can be used to solve $H_{22}\hat{h} = \hat{c}$. What is needed is a forward-substitution routine that solves systems of the form $\hat{R}^T\hat{h} = \hat{c}$, where $\hat{R}$ is a submatrix of R consisting of rows and columns j through k, where j and k can be any integers satisfying $1 \leq j \leq k \leq n$. Since R^T, hence $\hat{R}^T$, is stored by rows, the appropriate formulation of forward substitution is the row-oriented version. This subroutine can also be used with $j = 1$ and $k = n$ to perform the forward-substitution step ($R^T y = b$) after the Cholesky decomposition has been completed. Finally, a back-substitution routine is needed to solve $Rx = y$. Since R is stored by columns, we use the column-oriented version.

Exercise 1.5.16 Write a set of three Fortran subroutines to solve positive definite systems, using the envelope storage scheme:

(a) Row-oriented forward-substitution routine, capable of solving systems $\hat{R}^T\hat{h} = \hat{c}$, where $\hat{R}$ is a submatrix of R consisting of rows and columns j through k, $1 \leq j \leq k \leq n$.

(b) Cholesky decomposition routine, bordered form, which calls the forward-substitution routine to do most of the work.

(c) column-oriented back-substitution routine.

Write a main program that allocates storage, handles input and output, and calls the subroutines to solve positive definite systems $Ax = b$. Test your programs using the test problems given below.

For storage you will need the arrays DIAG, ENV, and IENV discussed above and one additional real array of length n that holds b initially, gets changed to y during the forward-substitution step, and finally gets changed to x during the back-substitution step. The arrays DIAG and ENV contain A initially and get changed to R as the Cholesky decomposition is computed. This is all the storage space that is needed (except, of course, for the space occupied by the program itself). Test problems:

(a) $A = \begin{bmatrix} 4 & 0 & 6 & 0 & 2 \\ 0 & 1 & 3 & 0 & 2 \\ 6 & 3 & 19 & 2 & 6 \\ 0 & 0 & 2 & 5 & -5 \\ 2 & 2 & 6 & -5 & 16 \end{bmatrix}$ (i) $b = \begin{bmatrix} 12 \\ 6 \\ 36 \\ 2 \\ 21 \end{bmatrix}$ (ii) $b = \begin{bmatrix} 12 \\ 4 \\ 26 \\ -8 \\ 27 \end{bmatrix}$

The Cholesky decomposition only has to be done once. *Solution:*

$$R = \begin{bmatrix} 2 & 0 & 3 & 0 & 1 \\ & 1 & 3 & 0 & 2 \\ & & 1 & 2 & -3 \\ & & & 1 & 1 \\ & & & & 1 \end{bmatrix}$$

(i) $y = \begin{bmatrix} 6 \\ 6 \\ 0 \\ 2 \\ 1 \end{bmatrix}$ $x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$ (ii) $y = \begin{bmatrix} 6 \\ 4 \\ -4 \\ 0 \\ 1 \end{bmatrix}$ $x = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$

(b) $A = \begin{bmatrix} 16 & 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 5 & -3 & 6 & 0 & 0 & 0 & 10 & 0 & 0 \\ 4 & -3 & 6 & -8 & 0 & 2 & 0 & -9 & 0 & 0 \\ 0 & 6 & -8 & 17 & 8 & -2 & 0 & 13 & 0 & 0 \\ 0 & 0 & 0 & 8 & 17 & 3 & 2 & -1 & 0 & 0 \\ 0 & 0 & 2 & -2 & 3 & 15 & 1 & -3 & 3 & 0 \\ 0 & 0 & 0 & 0 & 2 & 1 & 21 & -4 & -7 & 0 \\ 0 & 10 & -9 & 13 & -1 & -3 & -4 & 32 & -1 & 4 \\ 0 & 0 & 0 & 0 & 0 & 3 & -7 & -1 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 24 \end{bmatrix}$ $b = \begin{bmatrix} 36 \\ 109 \\ -76 \\ 188 \\ 141 \\ 113 \\ 68 \\ 281 \\ 51 \\ 272 \end{bmatrix}$

(c) Use your program to show that

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 13 & 16 \\ 4 & 16 & 10 \end{bmatrix}$$

is not positive definite.

(d) Think about how your subroutines could be used to calculate the inverse of a positive definite matrix. Calculate A^{-1}, where

$$A = \begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix}.$$

It turns out that the entries of A^{-1} are all integers. Notice that your computed solution suffers from significant roundoff errors. This is because A is (mildly) ill conditioned. This is the 3×3 member of a famous family of ill-conditioned

matrices called *Hilbert matrices*; the condition gets rapidly worse as the size of the matrix increases. We will discuss ill-conditioned matrices in Chapter 2.

□

1.6 SPARSE POSITIVE DEFINITE SYSTEMS

If one compares a sparse matrix A with its Cholesky factor R, one normally finds that R has many more nonzero entries than the upper half of A does. The "new" nonzero entries are called *fill* or *fill-in*. How much fill one gets depends on how the equations are ordered.

Example 1.6.1 An *arrowhead matrix* like

$$A = \begin{bmatrix} 7 & 0 & 0 & 1 \\ 0 & 3 & 0 & 1 \\ 0 & 0 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

suffers no fill-in during the Cholesky decomposition. Its Cholesky factor is

$$R = \begin{bmatrix} \sqrt{7} & 0 & 0 & 1/\sqrt{7} \\ & \sqrt{3} & 0 & 1/\sqrt{3} \\ & & \sqrt{2} & 1/\sqrt{2} \\ & & & 1/\sqrt{42} \end{bmatrix},$$

which has as many zeros above the main diagonal as A has. Now consider the matrix

$$\hat{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 0 & 3 & 0 \\ 1 & 0 & 0 & 7 \end{bmatrix},$$

obtained from A by reversing the order of the rows and columns. This reversed arrowhead matrix has the Cholesky factor

$$\hat{R} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ & 1 & -1 & -1 \\ & & 1 & -2 \\ & & & 1 \end{bmatrix},$$

which is completely filled in. Obviously the same will happen to any arrowhead matrix; the pattern of nonzeros is what matters, not their exact values. Furthermore, there is nothing special about the case $n = 4$; we can build arbitrarily large arrowhead matrices. Notice that A has a small envelope that contains no nonzero entries, whereas $\hat{A}$ has a large envelope that includes many zeros. Theorem 1.5.7 guarantees that the

envelope of R will be no bigger than the envelope of A, but it says nothing about the fate of zeros within the envelope. Usually zeros within the envelope of A will turn into nonzeros in R. □

The key to making the sparse Cholesky decomposition economical is to keep the fill under control. Ideally one would like to find the reordering that minimizes fill. As it turns out, this is a difficult problem (there are $n!$ orderings to consider). The solution seems to be beyond reach. Fortunately there are several practical algorithms that do a reasonable job of keeping the fill-in under control [14, 37]. Two very effective methods are implemented in MATLAB. The *reverse Cuthill-McKee* algorithm attempts to make the envelope small. The *approximate minimum-degree* algorithm tries to minimize fill-in by analyzing the sparsity pattern of the matrix using graph-theoretic methods. Bandwidth is ignored. For descriptions of these two methods see [14]. We will explore their performance by means of some MATLAB examples.

MATLAB has considerable support for sparse matrix computations. There is an easy-to-use sparse matrix data structure. Most of the operations that are available for full (i.e. non-sparse) matrices can be applied to sparse matrices as well. For example, if A is a positive-definite matrix stored in the sparse format, the command `R = chol(A)` gives the Cholesky factor of A, also stored in the sparse format. There are also numerous commands that apply strictly to sparse matrices. Type `help sparfun` in MATLAB for a list of these, or look in MATLAB's help browser.

Example 1.6.2 A nice example of a sparse matrix that's not too big is `bucky`, which is the incidence matrix of the Bucky Ball (soccer ball). The Bucky Ball is a polyhedron with 60 vertices, 32 faces (12 pentagons and 20 hexagons), and 90 edges. An incidence matrix is obtained by numbering the vertices and building a 60×60 matrix whose (i, j) entry is 1 if vertices i and j are connected by an edge and 0 otherwise. Since each vertex is connected to exactly three others, each row of the Bucky Ball matrix has exactly three nonzero entries. This matrix is not positive definite; its main diagonal entries are all zero. Figure 1.13 shows the pattern of nonzeros in the Bucky Ball matrix in the "original" ordering specified by MATLAB and in three reorderings. Plots of this type are called *spy plots* in MATLAB and are generated by the command `spy(A)` . Each of the four plots in Figure 1.13 has 180 dots, corresponding to the 180 nonzero entries of the matrix. We note that the reverse Cuthill-McKee ordering gathers the nonzeros into a band, whereas the minimum-degree ordering does not. □

Example 1.6.3 We now move on to a positive-definite example. We used the following commands to generate a very small scale 3d discrete Laplacian matrix A.

```
m = 5;
r = zeros(1,m); r(1) = 2; r(2) = -1;
B = sparse(toeplitz(r));
C = speye(m);
A = kron(kron(B,C),C) + ...
    kron(kron(C,B),C) + ...
    kron(kron(C,C),B);
```

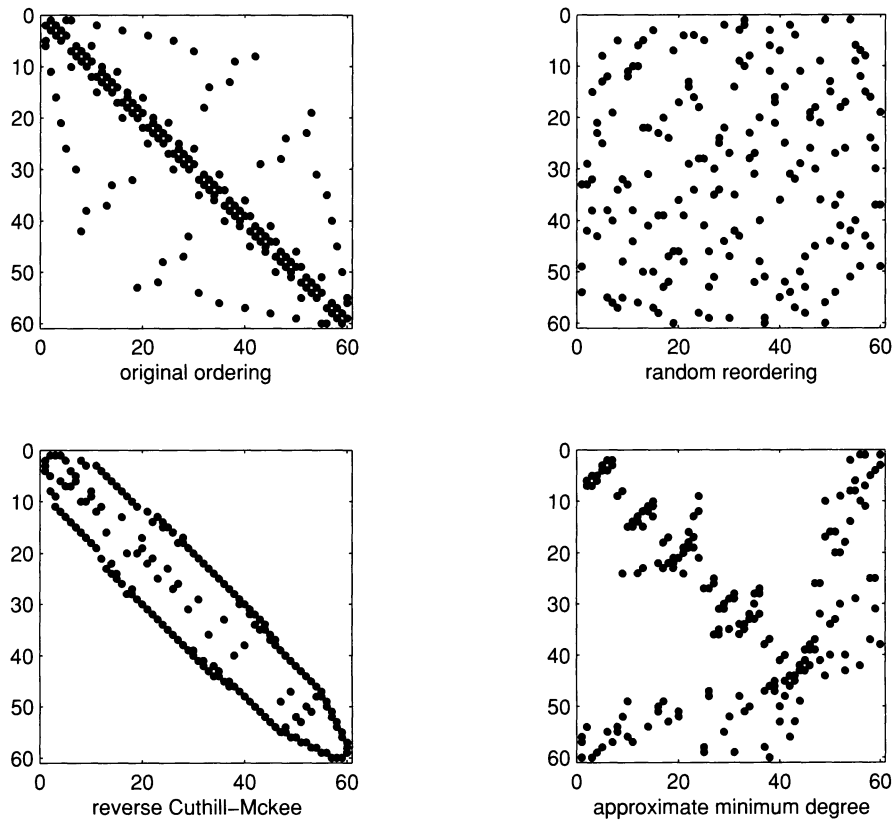


Figure 1.13 Spy plots of several orderings of the Bucky Ball matrix

This is a 125×125 positive-definite matrix. Figure 1.14 shows the spy plot of A and three of its reorderings. Notice that in its original ordering A is already banded. The reverse Cuthill-McKee ordering makes the envelope a bit narrower, but the approximate minimum-degree ordering destroys the band structure entirely.

Figure 1.15 shows spy plots of the Cholesky factors of A and each of the reorderings. In each case the number of nonzero entries is shown. Notice that the random ordering has much worse fill-in than the others. The reverse Cuthill-McKee ordering has slightly less fill than the original banded ordering, but the approximate minimum-degree ordering is much better. □

One small example proves nothing, but extensive tests on larger matrices have confirmed that the approximate minimum-degree algorithm does significantly better than reverse Cuthill-McKee on a wide variety of problems. However, effective

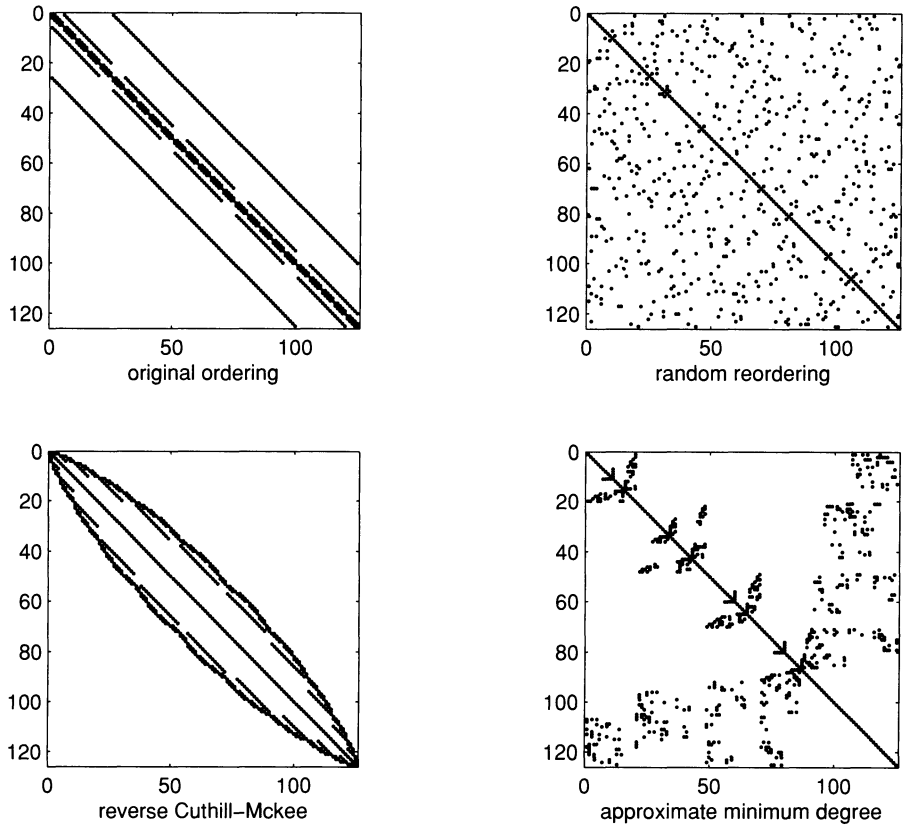


Figure 1.14 Spy plots of several orderings of a discrete Laplacian matrix

exploitation of the good fill properties of the approximate minimum-degree algorithm requires use of a more flexible data structure for sparse matrices since the fill is not restricted to a narrow band. In contrast, if we use the reverse Cuthill-McKee algorithm, we can use a simple band or envelope scheme that accommodates the fill automatically.

MATLAB's sparse matrix data structure is quite simple. Three numbers, m, n, and nz specify the number of rows, columns, and nonzero entries of the array. The information about the matrix entries and their locations is stored in three lists (one-dimensional arrays), i, j, and s, of length (at least) nz. Arrays i and j have integer entries, and s has (double precision) real entries. For each $k \leq nz$, $s(k)$ is the

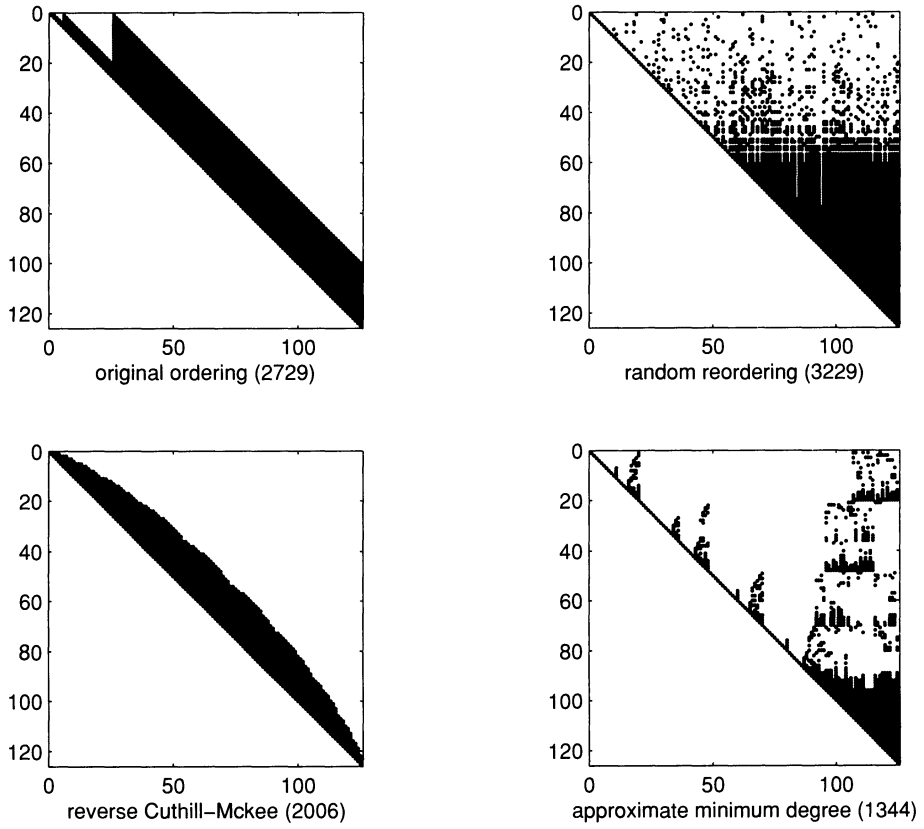


Figure 1.15 Spy plots of Cholesky factors of reorderings of a discrete Laplacian matrix. For each ordering, the number of nonzeros is given in parentheses.

value of a nonzero entry of A. Its position in the matrix is $(i(k), j(k))$.[7] When the MATLAB command `R = chol(A)` is used to compute the Cholesky factor of the sparse matrix A, MATLAB has to allocate space for the sparse matrix R. In doing so it must take into account not only the amount of space that A occupies but also the fill that occurs as R is computed. This all looks easy to the user because MATLAB does it all automatically.

[7] If more than one entry is assigned to a given (i, j) location, entries assigned to the same location are added together. For example, if sparse matrix A has $(i(3), j(3)) = (i(7), j(7)) = (i(50), j(50)) = (21, 36)$ (and all other (i, j) pairs differ from (21,36)), then $a_{21,36} = s(3) + s(7) + s(50)$.

Exercise 1.6.4 Investigate the MATLAB commands in Example 1.6.3, and figure out how the 3D discrete Laplacian matrix was constructed. □

Exercise 1.6.5 Build a larger version of the 3D discrete Laplacian matrix of Example 1.6.3 by increasing the value of m in the code shown there. The dimension of the matrix is m^3, so if you take, for example, $m = 20$, you will get an 8000×8000 matrix. Once you have generated your matrix, type `size(A)` to confirm the dimensions and `issparse(A)` to find out whether A is stored as a sparse or a full matrix. The answer 1 indicates sparse and 0 indicates full. Type `spy(A)` to get a picture of the nonzero structure of A. (Type simply `A` to get a long list of the nonzero entries of A.) The label on the spy plot tells you how many nonzero entries A has. (In the case $m = 20$ it is 53,600.) This information can also be obtained by typing `nnz(A)` .

Calculate the Cholesky factor of A and several reorderings of A:

(a) First consider A in its original ordering. Calculate the Cholesky factor of A, keeping track of how long it takes to do so. For example, you can use the commands

```
tic, R = chol(A); toc
```

or

```
t = cputime; R = chol(A); time = cputime - t
```

How many nonzeros does R have? Take a look at the spy plot of R.

(b) Now permute the rows/columns of A randomly and repeat part (a). This is achieved as follows, for example:

```
p = randperm(size(A,1));
arnd = A(p,p);
spy(arnd)
tic, rrnd = chol(arnd); toc
nz = nnz(rrnd)
spy(rrnd)
```

The first command returns a random permutation of the integers 1 through k, where k is the dimension of A. The second produces a new matrix `arnd` whose rows and columns have been permuted according to the random permutation.

(c) Repeat part (b) using the reverse Cuthill-McKee ordering instead of a random reordering. The correct permutation is obtained by replacing the random permutation by `p = symrcm(A);` . Thus

```
p = symrcm(A);
arcm = A(p,p);
spy(arcm)
tic, rrcm = chol(arcm); toc
```

```
nz = nnz(rrcm)
spy(rrcm)
```

The command `symrcm` is a mnemonic for "SYMmetric Reverse Cuthill-McKee."

(d) Repeat part (c) using the approximate minimum-degree ordering instead of reverse Cuthill-McKee. The approximate minimum-degree permutation is obtained by `p = symamd(A);` `symamd` is a mnemonic for "SYMmetric Approximate Minimum Degree."

(e) Comment on your results.

□

Exercise 1.6.6 Repeat Exercise 1.6.5 with a much larger 3D discrete Laplacian matrix, leaving out part (b). □

Exercise 1.6.7 MATLAB has a command `delsq`, which generates 2D discrete Laplacian ("del squared") matrices associated with various regions. For example, try

```
m = 92
A = delsq(numgrid('S',m));
issparse(A)
size(A)
```

for a discrete Laplacian on a square. This produces an 8100×8100 matrix. If this matrix is too small or too big, a larger or smaller version can be obtained by increasing or decreasing m. In general the matrix A has dimension $(m-2)^2$. Its structure is the same as that of the matrices discussed in Example 1.5.2 and Exercise 1.5.4. For more information on `delsq` type `help delsq` and `help numgrid` in MATLAB, or search in the help browser. Numerous variations can be obtained by replacing the `'S'` by other letters in the `numgrid` command.

Using the matrix A generated as shown above (using a larger m if your computer allows it), calculate the Cholesky factor of A and several reorderings of A. Use the MATLAB commands that you learned in Exercise 1.6.5.

(a) Make a spy plot of A. Notice that the original ordering already gives a narrow bandwidth. Calculate the Cholesky factor of A, noting the CPU time. How many nonzeros does the Cholesky factor have? Take a look at its spy plot.

(b) Repeat part (a) using a random reordering of the rows/columns of A. (`p = randperm((m-2)^2); arnd = A(p,p);`).

(c) Repeat part (a) using the reverse Cuthill-McKee ordering.

(d) Repeat part (a) using the approximate minimum-degree ordering.

(e) Another ordering that is available for this particular example is the *nested-dissection* ordering. Type `Anest = delsq(numgrid('N',m))`. This

gives the same matrix as before, except that the rows/columns are numbered according to a nested-dissection ordering [14, 37]. Repeat part (a) using the nested-dissection ordering.

(f) Discuss the results from parts (a)–(e).

□

Exercise 1.6.8 Another interesting example that is supplied with MATLAB is the "NASA airfoil." Run the NASA Airfoil command-line Demo by typing `airfoil` in MATLAB. This begins by plotting a grid of triangles that has been used for computation of the flow around an airplane wing by the finite element method. The grid has 4253 points or nodes, each having from three to nine neighbors. The Demo also builds a positive definite matrix A related to the adjacency matrix of the grid. This is a 4253×4253 matrix with a -1 in the (i, j) position if $i \neq j$ and node i is adjacent to node j (the nodes having been numbered beforehand). The main diagonal entries are made large enough that the matrix is positive definite. The exact recipe for A is given in the Demo. The Demo then proceeds to produce spy plots of A and several reorderings of A. Once you have run the Demo, the matrix A remains in memory for your use.

(a) Calculate the Cholesky factor of A, noting the CPU time. How many nonzeros does the Cholesky factor have? Take a look at its spy plot.

(b) Repeat part (a) using the reverse Cuthill-McKee ordering.

(c) Repeat part (a) using the approximate minimum-degree ordering.

(d) Discuss the results from parts (a)–(c).

□

MATLAB comes equipped with many other test matrices, both sparse and full. Type `help elmat` for a partial list. In particular, the `gallery` collection of N. J. Higham contains many good specimens. Type `help gallery` and follow the instructions.

Exercise 1.6.9 Repeat Exercise 1.6.7 (skipping part (e)) using a Wathen matrix from Higham's gallery. Type `A = gallery('wathen',70,60)`, for example. For a bigger matrix replace the 70 and 60 by larger numbers. Type `help private/wathen` for information about this matrix family. □

1.7 GAUSSIAN ELIMINATION AND THE LU DECOMPOSITION

In this and the next section, we will consider the problem of solving a system of n linear equations in n unknowns $Ax = b$ by Gaussian elimination. The algorithms developed here produce (in the absence of rounding errors) the unique solution