9.9. For the $(\hat{\mathbf{d}}_n^k, \hat{\mathbf{d}}_p^k)$ pair depicted by Equations (9.32a) and (9.32b), show that

$$\left(\hat{\mathbf{d}}_n^k\right)^T \mathbf{Q}\hat{\mathbf{d}}_p^k = 0$$

$$\left(\hat{\mathbf{d}}_n^k\right)^T \mathbf{Q}\hat{\mathbf{d}}_n^k = \left(\hat{\mathbf{d}}_p^k\right)^T \mathbf{Q}\hat{\mathbf{d}}_p^k$$

$$\left(\mathbf{g}^k\right)^T \hat{\mathbf{d}}_p^k = 0$$

$$\left(\mathbf{g}^k\right)^T \hat{\mathbf{d}}_n^k = \left(\hat{\mathbf{d}}_n^k\right)^T \mathbf{Q}\hat{\mathbf{d}}_n^k$$

(These are often called the boundary conditions for the potential push for QP problems.)

9.10. For the $\mathbf{x}(t)$ given by Equation (9.31), show that

$$\lim_{\|\mathbf{Q}\| \to 0} \mathbf{x}(t) = \mathbf{x}^k + \rho_k t \mathbf{v}^k$$

such that $\mathbf{c}^T\mathbf{v}^k = 0$, $\mathbf{A}\mathbf{v}^k = 0$, and ρ_k is a positive constant. (*Note:* This is the potential push equation for the LP case.) [*Hint:* Note that θ in Equation (9.31) can be chosen as the spectral norm of $\mathbf{Q}$.]

9.11. Show that $\mathbf{x}(t)$ is a geodesic in the transformed space (where the constant objective surfaces are spherical), and hence the locus of $\mathbf{x}(t)$ represents a great circle. [Hint: Show that

$$\left[\frac{d^2\mathbf{x}'(t)}{dt^2}\right]^T \left[\frac{d\mathbf{x}'(t)}{dt}\right] = 0 \qquad \text{where } \mathbf{x}'(t) = \mathbf{L}^T\mathbf{x}(t) \text{ and } \mathbf{L}\mathbf{L}^T = \mathbf{Q}$$

which implies zero acceleration in the tangent space. This is the striking property of geodesics.]

9.12. Assume that the vector $\mathbf{y}^0 = \mathbf{e}$. Generate a sequence of vectors $\{\mathbf{y}_k\}$ by using the relationship

$$\mathbf{y}^{k+1} = \mathbf{Q}\mathbf{y}_k$$

Show that the ratio

$$\Lambda_k = \frac{\left(\mathbf{y}^k\right)^T \mathbf{y}^{k+1}}{\left(\mathbf{y}^k\right)^T \mathbf{y}^k}$$

approaches the spectral radius (i.e., the largest eigenvalue) of $\mathbf{Q}$. [*Note:* Λ_k for sufficiently large values of k can be considered as a practical choice of θ for the potential push in Equation (9.31).]

9.13. Carry out one more iteration of Example 9.1.

9.14. Carry out one more iteration of Example 9.2.

10

Implementation of Interior-Point Algorithms

In recent years the interior-point algorithms have shown their efficiency in solving large-scale linear and quadratic programming problems with a wide variety of successful applications. As a matter of fact, some large-scale problems became solvable owing to the invention of these techniques. However, it is important to understand that implementation techniques play a key role in the claimed efficiency of these methods. For example, we can easily implement the primal affine scaling algorithm for linear programming in APL language in less than an hour, involving less than twenty lines of coding, but the performance of such an implementation could be far from satisfactory. Many implementation issues need to be carefully addressed in order to achieve the expected performance. Nowadays, with the advent of vector/parallel processing capabilities of modern computers, implementation skills are much more involved than ever before. This is particularly true for any serious commercial software package.

In this chapter, we point out the computational bottleneck of interior-point algorithms and focus on some implementation techniques, including the Cholesky factorization, conjugate gradient, and LQ factorization methods, to tackle the bottleneck problem. By no means does this chapter provide a complete treatment; it only touches the tip of an iceberg.

10.1 THE COMPUTATIONAL BOTTLENECK

So far we have studied the Karmarkar's projective scaling algorithm, primal affine scaling algorithm, dual affine scaling algorithm, primal-dual algorithm, affine scaling with logarithmic barrier function method, affine scaling with power-series method, and extended affine scaling algorithms for linearly constrained quadratic and convex programming

problems. For all these interior-point algorithms, as discussed in previous chapters, most computational time is spent in inverting a fundamental matrix $\mathbf{M}$ to find a moving direction at each iteration—for example, $\mathbf{M} = \mathbf{A}\mathbf{X}_k^2\mathbf{A}^T$ in the primal affine scaling, $\mathbf{M} = \mathbf{A}\mathbf{S}_k^{-2}\mathbf{A}^T$ in the dual affine scaling, $\mathbf{M} = \mathbf{A}\mathbf{X}_k\mathbf{S}_k^{-1}\mathbf{A}^T$ in the primal-dual algorithm, and $\mathbf{M} = \mathbf{A}(\mathbf{Q} + \mathbf{X}_k^2)\mathbf{A}^T$ in the quadratic programming affine scaling algorithm.

Note that this time-consuming task is equivalent to solving a system of linear equations

$$\mathbf{M}\mathbf{u} = \mathbf{v} \tag{10.1}$$

where $\mathbf{M}$ is an $m \times m$ positive definite symmetric matrix and $\mathbf{u}, \mathbf{v} \in R^m$ are m-dimensional column vectors. Therefore, it is a crucial challenge to solve system (10.1) in a most efficient manner. Actually, solving a system of linear equations is not a new problem. Many books have been written for this purpose. Here we only focus on the three most popular methods, namely, Cholesky factorization, conjugate gradient, and LQ factorization, and discuss related implementation issues.

10.2 THE CHOLESKY FACTORIZATION METHOD

The idea of the Cholesky factorization method is quite simple. Instead of solving system (10.1) directly, since the fundamental matrix $\mathbf{M}$ in (10.1) is symmetric and positive definite, based on Cholesky, we first factorize it as a matrix product of an $m \times m$ lower triangular matrix $\mathbf{L}$ and its transpose matrix $\mathbf{L}^T$, i.e., $\mathbf{M} = \mathbf{L}\mathbf{L}^T$. In this way, (10.1) becomes

$$\mathbf{L}\mathbf{L}^T\mathbf{u} = \mathbf{v} \tag{10.2}$$

We further define $\mathbf{z} = \mathbf{L}^T\mathbf{u}$. By solving

$$\mathbf{L}\mathbf{z} = \mathbf{v} \tag{10.3}$$

for $\mathbf{z}$ first and then solving

$$\mathbf{L}^T\mathbf{u} = \mathbf{z} \tag{10.4}$$

for $\mathbf{u}$, we find a solution to system (10.1) in two stages. Since $\mathbf{L}$ is a lower triangular matrix, we can easily identify z_1 first, then $z_2, z_3, \ldots, z_m$ by simple arithmetic operations. Usually this process is called *forward solve*. Similarly, because $\mathbf{L}^T$ is an upper triangular matrix, we can easily identify u_m first, then $u_{m-1}, u_{m-2}, \ldots, u_1$ by simple arithmetic operations. Therefore, it is often called *backward solve*.

It is easy to understand the advantage of forward solve and backward solve. But the key to success is to find the *Cholesky factor* $\mathbf{L}$ in an efficient manner. Before we introduce potential factorization algorithms, let us study a fundamental theorem of Cholesky factorization.

Theorem 10.1. If $\mathbf{M}$ is an $(m \times m)$-dimensional symmetric positive definite matrix, then there is a unique lower triangular matrix $\mathbf{L}$ with positive diagonal elements such that $\mathbf{M} = \mathbf{L}\mathbf{L}^T$.

Proof. We prove this result by the induction method.

The result is obviously true for $m = 1$. Now, assuming the result is true for $m = n - 1$, we have to show it holds true for $m = n$, where n is a positive integer. Since $\mathbf{M}$ is symmetric and positive definite, we can partition it as

$$\mathbf{M} = \begin{bmatrix} d & \mathbf{u}^T \\ \mathbf{u} & \mathbf{M}_1 \end{bmatrix} \tag{10.5}$$

where $d > 0$, $\mathbf{u}$ is an $n - 1$ vector, and $\mathbf{M}_1$ is an $(n-1) \times (n-1)$ submatrix. It can be further written as

$$\mathbf{M} = \begin{bmatrix} \sqrt{d} & \mathbf{0} \\ \mathbf{u}/\sqrt{d} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \mathbf{0} & \overline{\mathbf{M}}_1 \end{bmatrix} \begin{bmatrix} \sqrt{d} & \mathbf{u}^T/\sqrt{d} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{10.6}$$

where $\mathbf{I}$ is the $(n-1) \times (n-1)$ identity matrix and $\overline{\mathbf{M}}_1 = \mathbf{M}_1 - \mathbf{u}\mathbf{u}^T/d$.

$\overline{\mathbf{M}}_1$ is clearly symmetric. It is also positive definite, since for any nonzero vector $\mathbf{x} \in R^{n-1}$,

$$\mathbf{x}^T\overline{\mathbf{M}}_1\mathbf{x} = \begin{bmatrix} -\mathbf{x}^T\mathbf{u}/d \mid \mathbf{x}^T \end{bmatrix} \begin{bmatrix} d & \mathbf{u}^T \\ \mathbf{u} & \mathbf{M}_1 \end{bmatrix} \begin{bmatrix} -\mathbf{x}^T\mathbf{u}/d \\ \mathbf{x} \end{bmatrix} = \mathbf{z}^T\mathbf{M}\mathbf{z} > 0$$

where $\mathbf{z} = [-\mathbf{x}^T\mathbf{u}/d \mid \mathbf{x}^T]^T \in R^n$. Therefore, by our assumption, $\overline{\mathbf{M}}_1 = \mathbf{M}_1 - \mathbf{u}\mathbf{u}^T/d$ has a unique triangular factorization with positive diagonals, say $\overline{\mathbf{M}}_1 = \mathbf{L}_1\mathbf{L}_1^T$. Thus $\mathbf{M}$ may be expressed as

$$\begin{aligned} \mathbf{M} &= \begin{bmatrix} \sqrt{d} & \mathbf{0} \\ \mathbf{u}/\sqrt{d} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{L}_1 \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{L}_1^T \end{bmatrix} \begin{bmatrix} \sqrt{d} & \mathbf{u}^T/\sqrt{d} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \\ &= \begin{bmatrix} \sqrt{d} & \mathbf{0} \\ \mathbf{u}/\sqrt{d} & \mathbf{L}_1 \end{bmatrix} \begin{bmatrix} \sqrt{d} & \mathbf{u}^T/\sqrt{d} \\ \mathbf{0} & \mathbf{L}_1^T \end{bmatrix} \end{aligned} \tag{10.7}$$

Since $\mathbf{L}_1$ is unique, it is clear that

$$\mathbf{L} = \begin{bmatrix} \sqrt{d} & \mathbf{0} \\ \mathbf{u}/\sqrt{d} & \mathbf{L}_1 \end{bmatrix}$$

is also unique, and the proof is complete.

10.2.1 Computing the Cholesky Factor

We now focus on computing techniques for finding the Cholesky factor $\mathbf{L}$ of a symmetric positive definite matrix $\mathbf{M}$. Let m_{ij} and l_{ij} be the (i, j)th element of matrices $\mathbf{M}$ and $\mathbf{L}$, respectively, for $i, j = 1, 2, \ldots, m$. Since $\mathbf{M} = \mathbf{L}\mathbf{L}^T$, by matrix multiplication, we know that

$$m_{ij} = \sum_{k=1}^{j} l_{ik}l_{jk} \tag{10.8}$$

Because $\mathbf{L}$ is a lower triangular matrix, we need only consider the elements l_{ij} with $i \geq j$. In this case, (10.8) implies that, for $j = 1$,

$$l_{11} = (m_{11})^{1/2} \tag{10.9a}$$

and

$$l_{ij} = m_{ij}/l_{11}, \qquad i = 2, \ldots, m \tag{10.9b}$$

Moreover, for $j = 2, 3, \ldots, m$, we can first compute

$$l_{jj} = \left(m_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2} \tag{10.10a}$$

and then compute

$$l_{ij} = \left(m_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) / l_{jj}, \qquad \text{for } i = j+1, j+2, \ldots, m \tag{10.10b}$$

In this scheme, the columns of $\mathbf{L}$ are computed one by one, but the part of the matrix remaining to be factored is not accessed during the scheme. Also because the inner product of subrows of $\mathbf{L}$ is calculated in (10.10), this scheme is called an *inner-product form*. The inner-product form certainly is not the only way of computing the Cholesky factor. As a matter of fact, the proof of Theorem 10.1 itself is a constructive proof. It suggests a scheme called *outer-product form* of computing the rows of $\mathbf{L}$ one by one. Details of this new scheme will be provided in the exercises. As to the detailed implementation, the inner-product form scheme can be easily coded as follows:

Algorithm C-1

```
l11 ← √m11
for i = 2 to m
  li1 ← mi1/l11
end
for j = 2 to m
  for i = j to m
    s ← mij
    for k = 1 to j − 1
        s ← s−lik ljk
    end
    if i = j
        ljj ← √s
    else
        lij ← s/ljj
    end
  end
end
```

Note that Algorithm C-1 is based on the fact that the inner products between the subrows of $\mathbf{L}$ and the lower triangular portion of $\mathbf{M}$ can be overwritten by the corresponding elements of $\mathbf{L}$ (once $\mathbf{L}$ is known, we do not need to $\mathbf{M}$ any longer). To speed up the performance, we may consider using a computer with *vector processing* capability. In this environment, for $i = 1, \ldots, m$ and $j = 1, \ldots, i$, let $\mathbf{m}_{ij} = (m_{i1}, \ldots, m_{ij})^T$ be a column vector and consider the following coding, which overwrites m_{ij} for l_{ij}:

Algorithm C-2

```
l11 ← √m11
for i = 2 to m
  li1 ← mi1/l11
end
for j = 2 to m
  for i = j to m
    s ← mij − m^T_{i(j−1)} m_{i(j−1)}
    if i = i
        mjj ← √s
    else
        mij ← s/ljj
    end
  end
end
```

The difference between Algorithm C-1 and Algorithm C-2 may look subtle, but it clearly illustrates that detailed implementation on the coding level could gain speed and reduce the memory requirement. Another interesting issue to note here is that in Algorithms C-1 and C-2, the operations describing the vector inner products are on the subrows of a matrix, which may perform less efficiently if the matrix elements are stored columnwise (as in the case of FORTRAN). Therefore, if we choose to implement the algorithm in FORTRAN, a *code reorganizing* has to be done to allow the operations to access contiguous memory locations and thereby cut down memory access time. As to C programming, since the elements are stored rowwise, Algorithms C-1 and C-2 can be implemented without any degradation in performance due to memory access.

Newer FORTRAN and C compilers also allow the so-called *recursive* functions and subroutines. This is an interesting feature, where a function or subroutine can call itself. This feature can be effectively used in Cholesky factorization. The way recursion can be invoked varies from compiler to compiler for particular applications, and hence is beyond our scope. The important message to a serious program developer is to study the compiler before implementing any interior-point algorithm.

Another important aspect one should not leave out in this discussion is the *block Cholesky factorization*.

10.2.2 Block Cholesky Factorization

Knowing that matrix operations can be highly parallelizable (several row or column operations can be done simultaneously), when we are dealing with large-scale problems with special block structure in the constraint matrix, we should further study the Cholesky factorization algorithm.

Given an $(m \times m)$-dimensional symmetric positive definite matrix $\mathbf{M}$ partitioned into p^2 subblocks:

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_{11} & \cdots & \mathbf{M}_{1p} \\ \vdots & & \vdots \\ \mathbf{M}_{p1} & \cdots & \mathbf{M}_{pp} \end{bmatrix}$$

such that $m = pr$, where r is referred to as the *block size*. The Cholesky factor of $\mathbf{M}$ can be partitioned accordingly as

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_{11} & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{L}_{p1} & \cdots & \mathbf{L}_{pp} \end{bmatrix}$$

By directly equating $\mathbf{L}\mathbf{L}^T = \mathbf{M}$ with block structure, we see that

$$\mathbf{L}_{11}\mathbf{L}_{11}^T = \mathbf{M}_{11} \tag{10.11a}$$

$$\mathbf{L}_{i1}\mathbf{L}_{11}^T = \mathbf{M}_{i1}, \qquad \text{for } i = 2, \ldots, p \tag{10.11b}$$

Moreover, by matrix multiplication, for $p \geq i \geq j \geq 2$,

$$\mathbf{M}_{ij} = \sum_{k=1}^{j} \mathbf{L}_{ik}\mathbf{L}_{jk}^T$$

and hence

$$\mathbf{L}_{ij}\mathbf{L}_{jj}^T = \mathbf{M}_{ij} - \sum_{k=1}^{j-1} \mathbf{L}_{ik}\mathbf{L}_{jk}^T \tag{10.12}$$

If we denote

$$\mathbf{S}_{ij} = \mathbf{M}_{ij} - \sum_{k=1}^{j-1} \mathbf{L}_{ik}\mathbf{L}_{jk}^T$$

then, for $p \geq i \geq j \geq 2$, $\mathbf{L}_{jj}$ is the Cholesky factor of $\mathbf{S}_{jj}$ and $\mathbf{L}_{ij}$ is the solution of the matrix equation $\mathbf{Z}\mathbf{L}_{jj}^T = \mathbf{S}_{ij}$. Hence a block Cholesky factorization scheme is obtained:

Algorithm C-3

compute the Cholesky factor of $\mathbf{M}_{11}$ for $\mathbf{L}_{11}$
for $i = 2$ to p
  solve $\mathbf{Z}\mathbf{L}_{11}^T = \mathbf{M}_{i1}$ for $\mathbf{L}_{i1}$
end
for $j = 2$ to p
  for $i = j$ to p
    $\mathbf{S} \leftarrow \mathbf{M}_{ij}$
    for $k = 1$ to $j - 1$
      $\mathbf{S} \leftarrow \mathbf{S} - \mathbf{L}_{ik}\mathbf{L}_{jk}^T$
    end
    if $i = j$
      compute the Cholesky factor of $\mathbf{S}$ for $\mathbf{L}_{jj}$
    else
      solve $\mathbf{Z}\mathbf{L}_{jj}^T = \mathbf{S}$ for $\mathbf{L}_{ij}$
    end
  end
end

Note that Algorithm C-3 may use Algorithm C-1 or C-2 to find the Cholesky factor for block submatrices. Also note that since $\mathbf{L}_{jj}^T$ is upper triangular, solving $\mathbf{Z}\mathbf{L}_{jj}^T = \mathbf{S}$ is relatively simple. The recursive subroutines can be used here quite efficiently, if the compiler supports this feature. One key factor that affects the performance of block Cholesky factorization is the choice of the block size r, which often needs careful thinking and experimentation. The development of block Cholesky factorization algorithms and their implementations, especially on vector/parallel processors, is an active research area.

10.2.3 Sparse Cholesky Factorization

For large-scale problems, it is quite possible that most elements of matrix $\mathbf{M}$ have zero value. The sparsity is measured by the ratio between the number of nonzero elements and the total number of elements in a matrix. When the sparsity ratio is relatively low, say 0.01 or even smaller, we say the matrix is a *sparse matrix*. Otherwise, we have a *dense matrix*. However, there is no clear-cut threshold sparsity ratio.

When sparse matrices are involved, it is no longer necessary to keep track of their every element. Most attention needs to be focused on the "position and value" of nonzero elements only. The techniques which help us manipulate the sparse matrix operations are often called the *sparse matrix techniques*. Many books have been written on this topic.

As to applying Cholesky factorization methods to a symmetric positive definite sparse matrix $\mathbf{M}$, the key concern is to prevent the Cholesky factor $\mathbf{L}$ from being dense. The following example shows that a relatively sparse matrix $\mathbf{M}$ could have a relatively dense Cholesky factor $\mathbf{L}$.

Example 10.1

(from A. George and J. W. Liu): Let

$$\mathbf{M} = \begin{bmatrix} 4 & 1 & 2 & 0.5 & 2 \\ 1 & 0.5 & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 \\ 0.5 & 0 & 0 & 0.625 & 0 \\ 2 & 0 & 0 & 0 & 16 \end{bmatrix}$$

Applying Algorithm C-1, we have

$$\mathbf{L} = \begin{bmatrix} 2 & & & & \mathbf{0} \\ 0.5 & 0.5 & & & \\ 1 & -1.0 & 1 & & \\ 0.25 & -0.25 & -0.50 & 0.50 & \\ 1 & -1.0 & -2 & -3 & 1 \end{bmatrix}$$

Observe that, although $\mathbf{M}$ is relatively sparse, the corresponding Cholesky factor $\mathbf{L}$ is really dense. This phenomenon of increasing the number of nonzero elements is called *fill-in*, which is a direct consequence of the structure of $\mathbf{M}$. Fill-in no doubt increases the computational burden, since more nonzero elements need to be taken care of. In addition, the storage requirements obviously increase with the fill-in phenomenon.

A commonly adopted technique to reduce fill-ins is to permute the rows and columns of $\mathbf{M}$ such that it attains a structure which in turn makes the Cholesky factor sparse. To be more specific, we choose $\mathbf{P}$ to be an appropriate *permutation matrix* which permutes the rows of $\mathbf{M}$ into a desirable structure. Then, instead of handling the system (10.1), we consider an equivalent system

$$[\mathbf{PMP}^T][\mathbf{Pu}] = \mathbf{Pv} \tag{10.13}$$

Note that for any permutation matrix $\mathbf{P}$, the matrix $\mathbf{PMP}^T$ is still symmetric and positive definite. It is also interesting to note that, in $\mathbf{PMP}^T$, $\mathbf{P}$ permutes the rows of $\mathbf{M}$ and $\mathbf{P}^T$ permutes the columns of $\mathbf{M}$.

Example 10.2

For Example 10.1, if we choose a permutation matrix

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

then the rows and columns of $\mathbf{M}$ are permuted as

$$\mathbf{PMP}^T = \begin{bmatrix} 16 & 0 & 0 & 0 & 2 \\ 0 & 0.625 & 0 & 0 & 0.5 \\ 0 & 0 & 3 & 0 & 2 \\ 0 & 0 & 0 & 0.5 & 1 \\ 2 & 0.5 & 2 & 1 & 4 \end{bmatrix}$$

In this way, the Cholesky factor of $\mathbf{PMP}^T$ becomes

$$\mathbf{L} = \begin{bmatrix} 4 & & & & \mathbf{0} \\ 0 & 0.791 & & & \\ 0 & 0 & 1.73 & & \\ 0 & 0 & 0 & 0.707 & \\ 0.5 & 0.632 & 1.15 & 1.41 & 0.129 \end{bmatrix}$$

Compared to the Cholesky factor in Example 10.1, the new Cholesky factor is relatively sparse. Consequently, solving system (10.13) is more efficient than solving system (10.1).

With the abovementioned concept, we understand the key issue in sparse Cholesky factorization is to find an appropriate permutation $\mathbf{P}$ for a given symmetric positive definite matrix $\mathbf{M}$ such that the number of fill-ins is minimized. Unfortunately, minimizing fill-ins is not a simple problem in general. So far, only heuristics have been proposed by various researchers to provide acceptable, but not necessarily optimal, results.

The most popular fill-in reduction scheme is the so-called *minimum degree reordering* algorithm. Here we briefly outline the algorithm, leaving the reader to find detailed explanations and theoretical insights in other books.

The algorithm may be best understood by graphical illustrations. First of all, let us establish a relationship between graphs and matrices. For an $(m \times m)$-dimensional matrix $\mathbf{M}$, we define an *ordered graph* of $\mathbf{M}$ and denoted it by G^M. In G^M, there are m nodes, and a node i is connected to node j (where $i \neq j$) by a link if the (i, j)th element of $\mathbf{M}$ is not zero, i.e., $m_{ij} \neq 0$. Figure 10.1 illustrates this situation with the help of an example, where the off-diagonal nonzeros of $\mathbf{M}$ are depicted by asterisks.

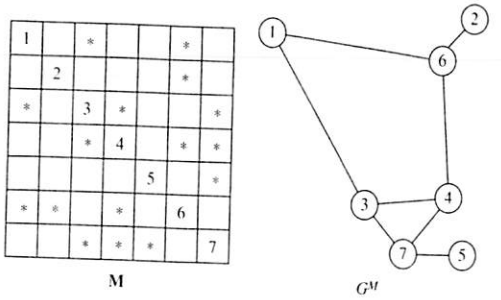


Figure 10.1

Two nodes i and j are *adjacent* if they are connected by a link. The *degree of a node* i, denoted by Deg (i), is defined to be the number of adjacent nodes of i, in other words, the number of links connected to node i. For example, in Figure 10.1, Deg (1) = 2, Deg (2) = 1, and Deg (7) = 3.

The idea of the minimum degree reordering algorithm is to eliminate a node with the minimum degree from the graph, one at a time, until every node is eliminated. Once a node is eliminated, the degree of nodes may change in the remaining graph and hence needs to be updated. The node elimination sequence eventually suggest us a candidate of the desired permutation matrix $\mathbf{P}$.

We now describe the minimum degree reordering algorithm in terms of the *graph elimination* model, where an *elimination graph* is defined as a graph which is subjected to the elimination of selected nodes. Here we eliminate the nodes of G^M one by one in a systematic order. At each step the resulting graph is labeled as G_k^M, where the subscript k denotes the step number.

MINIMUM DEGREE REORDERING ALGORITHM

Step 1 (initialization): Set $G_0^M \leftarrow G^M$ and $k = 1$.

Step 2 (minimum degree selection): In the elimination graph G_{k-1}^M, choose a node i of minimum degree.

Step 3 (graph elimination): Form an elimination graph G_k^M by eliminating the node i from G_{k-1}^M.

Step 4 (loop or stop): Set $k \leftarrow k+1$. If $k > m$ (the number of nodes of G_0^M), then stop. Otherwise go to Step 2.

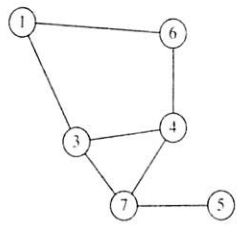
At the end, a permuted graph is obtained by swapping the step number k and the node number i. Moreover, a permutation matrix $\mathbf{P}$ is generated by assigning $p_{ki} = 1$, for $k = 1, \ldots, m$, and other elements being zero.

Notice that more than one node can assume the minimum degree in Step 2. Different heuristics of node selection give different versions of the minimum degree reordering algorithm. In the simple case without any further information, we may break ties arbitrarily.

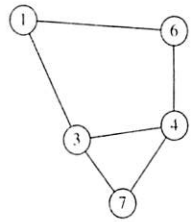
The following example illustrates the minimum degree algorithm applied to the example of Figure 10.1 with an arbitrary tie-breaker.

Example 10.3

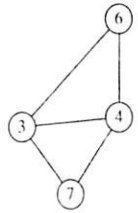
$k = 1$, node selected = 2, min. degree = 1



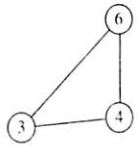
$k = 2$, node selected = 5, min. degree = 1



$k = 3$, node selected = 1, min. degree = 2



$k = 4$, node selected = 7, min. degree = 2

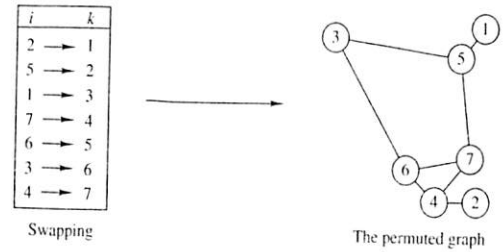


$k = 5$, node selected = 6, min. degree = 2

$k = 6$, node selected = 3, min. degree = 1

Finally,

$k = 7$, node selected = 4, min. degree = 0



Swapping

The permuted graph

The resulting permutation matrix becomes

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

10.2.4 Symbolic Cholesky Factorization

When the Cholesky factorization method is applied to find moving directions at each iteration of the previously mentioned interior-point algorithms, we need to factorize $\mathbf{M} = \mathbf{A}\mathbf{D}_k\mathbf{A}^T$ repeatedly, where $\mathbf{D}_k$ is a diagonal matrix with positive diagonal elements for each k. It will be awfully tedious, if we have to permute every $\mathbf{A}\mathbf{D}_k\mathbf{A}^T$ in order to reduce fill-ins.

Fortunately, closer observation indicates that although $\mathbf{A}\mathbf{D}_k\mathbf{A}^T$ changes along with the value of $\mathbf{D}_k$ at each iteration, the positions of nonzero elements remain intact. This means the sparsity structure is preserved as in $\mathbf{A}\mathbf{A}^T$. Therefore, in the implementation of an interior-point algorithm for large-scale problems, it is advantageous to perform a *symbolic factorization* first. In this phase, we focus on $\mathbf{A}\mathbf{A}^T$ to analyze the positions in which the nonzero entries of the computational result would occur. The minimum degree reordering algorithm could be applied to reduce the fill-ins. Once this work is done, we record the positions of nonzero elements as a template. Then, at each iteration, since the positions of nonzero elements are known, we need only find the numerical value of each nonzero element. Correspondingly, we may call it a *numerical factorization* phase. Figure 10.2 illustrates this two-phase procedure using block diagrams.

10.2.5 Solving Triangular Systems

Once the Cholesky factor of matrix $\mathbf{M}$ is computed, solving system (10.1) is equivalent to solving the triangular systems (10.3) and (10.4).

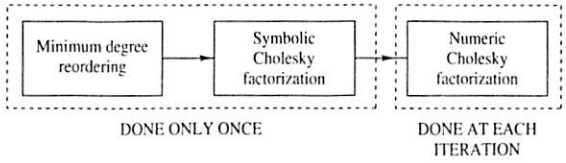


Figure 10.2

Forward solve. Since $\mathbf{L}$ is a lower triangular matrix, system (10.3) can be solved by getting z_1 from the first equation, z_2 from the second, ..., and z_m from the last. Therefore we call it a *forward solve* procedure. More specifically, we have

$$z_1 = v_1/l_{11} \tag{10.14a}$$

and

$$z_i = \left(v_i - \sum_{k=1}^{i-1} l_{ik} z_k\right) / l_{ii} \qquad \text{for } i = 2, \ldots, m \tag{10.14b}$$

It is easy to code as follows:

Algorithm F-1

```
z1 = v1/l11
for i = 2 to m
    s = 0
    for k = 1 to i - 1
        s ← s+lik zk
    end
    zi = (vi - s)/lii
end
```

Similar to Algorithm C-1, since we access matrix $\mathbf{L}$ row by row, and inner products of row vectors are involved in (10.14b), Algorithm F-1 can be modified for vector processing. This scheme is certainly more appropriate, if matrix $\mathbf{L}$ is stored rowwise (like C programming). We leave this to the reader.

If matrix $\mathbf{L}$ is stored column by column and sparsity of the solution vector is being considered, the following coding scheme is more efficient:

Algorithm F-2

```
for i = 1 to m
    zi = vi/lii
    for k = (i + 1) to n
        vk ← vk - zi lki
    end
end
```


The reader is asked to verify that Algorithm F-2 solves system (10.3) and accesses the matrix $\mathbf{L}$ column by column. Note that if v_i turns out to be zero at the beginning of the ith step, then z_i must be zero and the inner loop can be completely skipped. Hence the sparsity issue is exploited. When a columnwise storage scheme (for example, FORTRAN) is used, Algorithm F-2 is more efficient.

Backward solve. If, on the other hand, $\mathbf{L}^T$ is an upper triangular matrix, system (10.4) can be solved by getting u_m from the last equation, u_{m-1} from the second last, ..., and u_1 from the first. This forms a *backward solve* procedure. To be more specific, we have

$$u_m = z_m/l_{mm} \tag{10.15a}$$

and

$$u_{m-i} = \left(z_{m-i} - \sum_{k=m-i+1}^{m} l_{k(m-i)} u_k\right) / l_{(m-i)(m-i)} \qquad \text{for } i = 1, \ldots, m-1 \tag{10.15b}$$

It is easy to code as follows:

Algorithm B-1

```
um = zm/lmm
for i = 1 to m - 1
    s = 0
    for k = m - i + 1 to m
        s ← s+lk(m-i) uk
    end
    um-i = (zm-i - s)/l(m-i)(m-i)
end
```

Similar to Algorithm F-1, other coding scheme are available for further consideration. Algorithm B-1 is only one of the simple implementations.

The forward solve and backward solve together with Cholesky factorization have become the most popular method used by many interior-point algorithms for solving system (10.1). Other methods, including the conjugate gradient method and the LQ factorization method, will be introduced in subsequent sections.

10.3 THE CONJUGATE GRADIENT METHOD

In addition to the Cholesky factorization method, the conjugate gradient method can also be applied to solve system (10.1) with a symmetric positive definite matrix $\mathbf{M}$. The method was originally suggested by M. R. Hestenes and E. Stiefel in 1952. Like the steepest descent method, it is classified as an *error correction method*, which means that the algorithm starts with an approximated solution (say $\mathbf{u}^k$), evaluates an *error*

function, and then iterates along a direction (say $\mathbf{d}^k$) with an appropriate step-length to reduce the error. Instead of moving directly along the negative gradient direction for a maximum reduction of the error function, the moving directions are required to be *mutually conjugate* with respect to the matrix $\mathbf{M}$, i.e.,

$$(\mathbf{d}^k)^T \mathbf{M} \mathbf{d}^j = 0, \qquad \text{for } k \neq j \tag{10.16}$$

Therefore, this method carries the name "conjugate gradient."

For an $(m \times m)$-dimensional matrix $\mathbf{M}$, the conjugate gradient theory guarantees that the mth iterate $\mathbf{u}^m$ is an exact solution of $\mathbf{Mu} = \mathbf{v}$. Of course, it is quite possible that the method produces a sufficiently accurate solution prior to reaching $\mathbf{u}^m$. Moreover, as shown later, the most complicated arithmetic required by the method is merely the matrix-to-vector multiplications (in computing $\mathbf{Mu}^k$). Hence the method is well suited for solving large sparse systems.

Now let us introduce the basic ideas of the conjugate gradient method. Suppose that $\mathbf{u}^k$ is a current approximate of the system $\mathbf{Mu} = \mathbf{v}$. We define

$$\mathbf{r}^k = \mathbf{v} - \mathbf{M}\mathbf{u}^k \tag{10.17}$$

to be a corresponding *error vector* (or *residual vector*), and

$$h_k = h\left(\mathbf{r}^k\right) = \left(\mathbf{r}^k\right)^T \mathbf{M}^{-1} \mathbf{r}^k \tag{10.18}$$

to be an *error function* at $\mathbf{u}^k$. Remembering that $\mathbf{M}$ is symmetric, by combining (10.17) and (10.18) we obtain that

$$h_k = \left(\mathbf{v} - \mathbf{M}\mathbf{u}^k\right)^T \mathbf{M}^{-1} \left(\mathbf{v} - \mathbf{M}\mathbf{u}^k\right) = \left(\mathbf{u}^k\right)^T \mathbf{M}\mathbf{u}^k - 2\mathbf{v}^T\mathbf{u}^k + \mathbf{v}^T\mathbf{M}^{-1}\mathbf{v} \tag{10.19}$$

Assume that the next iterate $\mathbf{u}^{k+1}$ is determined by

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \alpha_k \mathbf{d}^k \tag{10.20}$$

where α_k is an appropriate step-length and $\mathbf{d}^k$ is an appropriate direction of translation such that the value of the error function is reduced by this translation. Figure 10.3 illustrates a two-dimensional example, in which $\mathbf{u}^{k+1}$ is on the straight line passing $\mathbf{u}^k$. The slope of the line is determined by $\mathbf{d}^k$ and the step-size α_k is proportional to the distance between $\mathbf{u}^k$ and $\mathbf{u}^{k+1}$. Our objective here is to find $\mathbf{d}^k$ and α_k. Suppose that $\mathbf{d}^k$ is known. In order to achieve a maximum reduction in h_{k+1}, we try to find a local minimum of h_{k+1} along $\mathbf{d}^k$ by plugging (10.20) into (10.19) and setting the derivative of h_{k+1} with respect to α_k to be zero. This yields the result

$$\alpha_k = \frac{\left(\mathbf{d}^k\right)^T \mathbf{r}^k}{\left(\mathbf{d}^k\right)^T \mathbf{M}\mathbf{d}^k} = \frac{\left(\mathbf{d}^k\right)^T \mathbf{r}^k}{\left(\mathbf{d}^k\right)^T \mathbf{p}^k}, \qquad \text{where } \mathbf{p}^k = \mathbf{M}\mathbf{d}^k \tag{10.21}$$

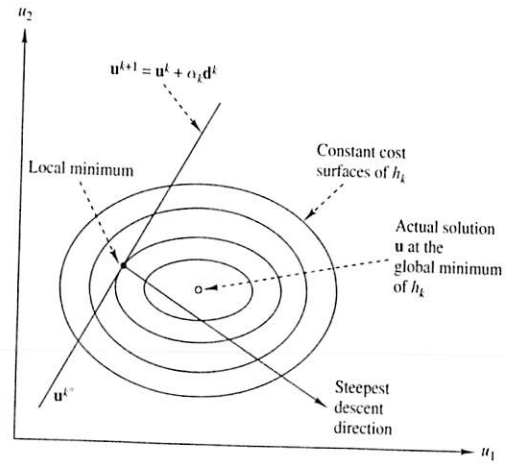


Figure 10.3

Note that the steepest descent method suggests that we consider using the negative gradient of h_k with respect to $\mathbf{u}^k$ as $\mathbf{d}^k$. In this case, we have

$$-\frac{dh_k}{d\mathbf{u}^k} = -2\left(\mathbf{M}\mathbf{u}^k - \mathbf{v}\right) = 2\mathbf{r}^k \tag{10.22}$$

which means the negative gradient of h_k is proportional to the residual $\mathbf{r}^k$. Therefore, the residual vector can be used in place of $\mathbf{d}^k$. With this in mind, the conjugate gradient method intends to stay close to $\mathbf{r}^k$ while satisfying the conjugacy requirements (10.16). Therefore, when $\mathbf{u}^0$ is arbitrarily chosen, we can take $\mathbf{d}^0 = \mathbf{r}^0$. After that, we may consider taking $\mathbf{d}^k$ as the component of $\mathbf{r}^k$ orthogonal to $\mathbf{M}\mathbf{d}^{k-1}$, for $k \geq 1$. In this way, we define

$$\mathbf{d}^k = \mathbf{r}^k - \beta_k \mathbf{M}\mathbf{d}^{k-1} \tag{10.23}$$

where β_k is a scalar to be fixed by the conjugacy condition $\left(\mathbf{M}\mathbf{d}^{k-1}\right)^T \mathbf{d}^k = 0$. Consequently, we know that

$$\beta_k = \frac{\left(\mathbf{M}\mathbf{d}^{k-1}\right)^T \mathbf{r}^k}{\left(\mathbf{M}\mathbf{d}^{k-1}\right)^T \mathbf{p}^{k-1}} = \frac{\left(\mathbf{p}^{k-1}\right)^T \mathbf{r}^k}{\left(\mathbf{p}^{k-1}\right)^T \mathbf{p}^{k-1}}, \qquad \text{where } \mathbf{p}^{k-1} = \mathbf{M}\mathbf{d}^{k-1} \tag{10.24}$$

Also, it is interesting to note that

$$\begin{aligned} \mathbf{r}^{k+1} &= \mathbf{v} - \mathbf{M}\mathbf{u}^{k+1} = \mathbf{v} - \mathbf{M}\left[\mathbf{u}^k + \alpha_k \mathbf{d}^k\right] \\ &= \left(\mathbf{v} - \mathbf{M}\mathbf{u}^k\right) - \alpha_k \mathbf{M}\mathbf{d}^k = \mathbf{r}^k - \alpha_k \mathbf{p}^k \end{aligned} \tag{10.25}$$

Based on this idea, the gradient algorithm can be stated as follows:

Algorithm CG: Set $\mathbf{u}^0$ to be arbitrary, $k = 0$, and $\epsilon > 0$ sufficiently small. Compute $\mathbf{d}^0 = \mathbf{r}^0 = \mathbf{v} - \mathbf{M}\mathbf{u}^0$. Repeat:

$$\mathbf{p}^k = \mathbf{M}\mathbf{d}^k$$

$$\alpha_k = \left[\left(\mathbf{d}^k\right)^T \mathbf{r}^k\right] / \left[\left(\mathbf{d}^k\right)^T \mathbf{p}^k\right]$$

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \alpha_k \mathbf{d}^k$$

$$\mathbf{r}^{k+1} = \mathbf{r}^k - \alpha_k \mathbf{p}^k$$

$$\beta_{k+1} = \left[\left(\mathbf{p}^k\right)^T \mathbf{r}^{k+1}\right] / \left[\left(\mathbf{p}^k\right)^T \mathbf{p}^k\right]$$

$$\mathbf{d}^{k+1} = \mathbf{r}^{k+1} - \beta_{k+1}\mathbf{p}^k$$

$$k \leftarrow k + 1$$

until $||\mathbf{r}^{k+1}|| \leq \epsilon$, output $\mathbf{u}^{k+1}$ as the solution.

As mentioned earlier, owing to the orthogonal relationship, the conjugate gradient method in theory generates an exact solution of the system (10.1) in at most m iterations. Therefore, strictly speaking, this method is a finite algorithm. However, owing to numerical round-off and/or truncation errors, it is highly possible for the algorithm to take more than m steps to reach a satisfying result. On the other hand, it is also possible for the algorithm to terminate in less than m steps with acceptable accuracy.

In connection with our application to the interior-point algorithms, as a primal algorithm converges, the dual estimate varies little from one iteration to another. Therefore, we can set $\mathbf{u}^0$ to be the dual estimate of the previous iteration and expect Algorithm CG to terminate quickly. For example, in the primal affine scaling algorithm, we set $\mathbf{M} = \mathbf{A}\mathbf{X}_k^2\mathbf{A}^T$ and $\mathbf{v} = \mathbf{A}\mathbf{X}_k^2\mathbf{c}$; then, solving $\mathbf{M}\mathbf{u} = \mathbf{v}$ provides the dual estimate $\mathbf{w}^k$ at the kth iteration.

Also note that the most intensive computation at each iteration of the conjugate gradient algorithm is the multiplication of matrix $\mathbf{M}$ to a vector $\mathbf{d}^k$. The sparse matrix techniques can be implemented effectively to perform this operation. In general, if $\mathbf{M}$ has, on the average, γ nonzero elements per row, then each iteration of Algorithm CG requires of the order of $(\gamma + 5)m$ multiplications. Hence, an exact solution can be generated in the order of $(\gamma + 5)m^2$ multiplications. If we start with a good estimate, a satisfactory solution is expected to be obtained in $k < m$ iterations; then the algorithm requires only of the order of $(\gamma + 5)km$ multiplications.

Several ad-hoc enhancements have been suggested by various researchers for the conjugate gradient method. A detailed discussion of these techniques is beyond the scope of this book.

10.4 THE LQ FACTORIZATION METHOD

The idea of LQ factorization is to utilize the power-series expansion in matrix form. Given that the symmetric positive definite matrix $\mathbf{M}$ in system (10.1) has the form

$$\mathbf{M} = \mathbf{I} - \mathbf{B} \tag{10.26}$$

where $\mathbf{B}$ is symmetric positive definite with all eigenvalues being positive and less than one, then the series

$$\sum_{k=0}^{\infty} \mathbf{B}^k = \mathbf{I} + \mathbf{B} + \mathbf{B}^2 + \ldots \tag{10.27}$$

is convergent and

$$\mathbf{M}^{-1} = (\mathbf{I} - \mathbf{B})^{-1} = \sum_{k=0}^{\infty} \mathbf{B}^k \tag{10.28}$$

In this way, a solution of system (10.1) is provided by

$$\mathbf{u} = \mathbf{M}^{-1}\mathbf{v} = \left[\mathbf{I} + \mathbf{B} + \mathbf{B}^2 + \cdots\right]\mathbf{v} \tag{10.29}$$

Therefore the required matrix inversion can be replaced by matrix multiplications and additions.

However, for a general linear programming problem, the fundamental matrix $\mathbf{M} = \mathbf{A}\mathbf{D}_k^2\mathbf{A}^T$ does not necessarily fit this scheme, unless the constraint matrix $\mathbf{A}$ is properly manipulated. In this section we introduce the LQ factorization method to achieve this purpose. For simplicity, we focus on the implementation of the primal affine scaling algorithm and leave other algorithms to the reader.

To begin with, we define an $(m \times n)$-dimensional matrix $\mathbf{Q}$ to be *orthonormal* if $\mathbf{Q}\mathbf{Q}^T = \mathbf{I}$. Moreover, the norm of matrix $\mathbf{Q}$ is defined by

$$||\mathbf{Q}|| = \max_{||\mathbf{x}||=1} ||\mathbf{Q}\mathbf{x}||$$

Similarly,

$$\left|\left|\mathbf{Q}^T\right|\right| = \max_{||\mathbf{y}||=1} \left|\left|\mathbf{Q}^T\mathbf{y}\right|\right|$$

For an orthonormal matrix $\mathbf{Q}$ with full row-rank $m < n$, we have the following special property of its matrix norm.

Theorem 10.2. Let $\mathbf{Q}$ be an $(m \times n)$-dimensional orthonormal matrix with full row-rank $m < n$; then $||\mathbf{Q}|| \leq 1$ and $\left|\left|\mathbf{Q}^T\right|\right| = 1$.

Proof. Since $\mathbf{Q}$ is orthonormal with row-rank $m < n$, by the Gram-Schmidt orthogonalization process, there exists an $[(n - m) \times n]$-dimensional matrix $\mathbf{R}$ such that

$$\overline{\mathbf{Q}} = \begin{bmatrix} \mathbf{Q} \\ \mathbf{R} \end{bmatrix}$$

becomes an $(n \times n)$-dimensional orthonormal matrix. It can be readily shown that $\overline{\mathbf{Q}}^{-1} = \overline{\mathbf{Q}}^T$ and $||\mathbf{Q}\mathbf{x}|| = ||\mathbf{x}||$ for all $\mathbf{x}$ measured in the $2 - norm$. Hence, we have

$$1 = ||\overline{\mathbf{Q}}|| = \max_{||\mathbf{x}||=1} ||\overline{\mathbf{Q}}\mathbf{x}||$$

$$= \max_{||\mathbf{x}||=1} \left|\left|\begin{bmatrix}\mathbf{Q}\mathbf{x}\\ \mathbf{R}\mathbf{x}\end{bmatrix}\right|\right|$$

$$\geq \max_{||\mathbf{x}||=1} ||\mathbf{Q}\mathbf{x}|| = ||\mathbf{Q}||$$

This completes the first part of the proof. Now, for the second part,

$$||\mathbf{Q}^T|| = \max_{||\mathbf{y}||=1} ||\mathbf{Q}^T\mathbf{y}||$$

$$= \max_{||\mathbf{y}||=1} ||\mathbf{Q}^T\mathbf{y} + \mathbf{R}^T\mathbf{0}||$$

$$= \max_{||\mathbf{y}||=1} \left|\left|\overline{\mathbf{Q}}^T \begin{pmatrix}\mathbf{y}\\ \mathbf{0}\end{pmatrix}\right|\right|$$

$$= \max_{||\mathbf{y}||=1} \left|\left|\begin{pmatrix}\mathbf{y}\\ \mathbf{0}\end{pmatrix}\right|\right| = 1$$

and we are done.

The LQ factorization method is based upon the following fundamental theorem.

Theorem 10.3 (fundamental theorem for LQ factorization). Let $\mathbf{A}$ be an $(m \times n)$-dimensional matrix with full row-rank $m < n$; then there exists an $(m \times m)$-dimensional lower triangular matrix $\mathbf{L}$ and an $(m \times n)$-dimensional orthonormal matrix $\mathbf{Q}$ such that $\mathbf{A} = \mathbf{LQ}$.

Proof. Since $\mathbf{A}$ has full row-rank with $m < n$, by Theorem 10.1, we know that $\mathbf{AA}^T = \mathbf{LL}^T$ for a lower triangular matrix $\mathbf{L}$ with positive diagonal elements. Hence $\mathbf{L}^{-1}$ exists and $\mathbf{Q} = \mathbf{L}^{-1}\mathbf{A}$ is well defined. Moreover,

$$\mathbf{QQ}^T = \mathbf{L}^{-1}\mathbf{AA}^T\left(\mathbf{L}^{-1}\right)^T = \mathbf{L}^{-1}\mathbf{LL}^T\left(\mathbf{L}^T\right)^{-1} = \mathbf{I}$$

Hence we know that $\mathbf{Q}$ is orthonormal and $\mathbf{A} = \mathbf{LQ}$.

Note that for a linear programming problem in its standard form, i.e.,

$$\text{Minimize} \quad \mathbf{c}^T\mathbf{x}$$

$$\text{subject to} \quad \mathbf{Ax} = \mathbf{b}, \qquad \mathbf{x} \geq \mathbf{0}$$

if the constraint matrix $\mathbf{A}$ is $(m \times n)$-dimensional with full row-rank $m < n$, then $\mathbf{A}$ can be factorized as the product of $\mathbf{L}$ and $\mathbf{Q}$ according to Theorem 10.3. In this case we have $\mathbf{Q} = \mathbf{L}^{-1}\mathbf{A}$, where $\mathbf{L}$ is obtained by Cholesky factorization (say, Algorithm C-1).

However, computing $\mathbf{Q}$ does not require $\mathbf{L}^{-1}$ explicitly. If we let $\mathbf{Q}_j$ and $\mathbf{A}_j$ be the jth column of $\mathbf{Q}$ and $\mathbf{A}$, respectively, for $j = 1, \ldots, n$, then $\mathbf{Q}_j$ can be obtained by applying a "forward solve" (say, Algorithm F-1) to the system $\mathbf{LQ}_j = \mathbf{A}_j$, for $j = 1, \ldots, n$. In summary, we have the following algorithm for LQ factorization:

Algorithm Q

Step 1: Compute the Cholesky factor $\mathbf{L}$ for $\mathbf{AA}^T$.

Step 2: For $j = 1$ to n, use forward solve to find $\mathbf{Q}_j$ for the system

$$\mathbf{LQ}_j = \mathbf{A}_j$$

End.

Once the LQ factorization is done, the original linear programming problem can be expressed as

$$\text{Minimize} \quad \mathbf{c}^T\mathbf{x} \tag{10.30a}$$

$$\text{subject to} \quad \mathbf{Qx} = \mathbf{b}', \qquad \mathbf{x} \geq \mathbf{0} \tag{10.30b}$$

where $\mathbf{b}' = \mathbf{L}^{-1}\mathbf{b}$ (which can be obtained by applying a forward solve to the system $\mathbf{Lb}' = \mathbf{b}$).

As mentioned earlier, we shall focus on the primal affine scaling algorithm and leave similar development of other interior-point algorithms to the reader. Remember that in each iteration of the primal affine scaling algorithm, most computational time is spent on finding the dual estimate $\mathbf{w}^k$ for the moving direction $\mathbf{d}^k$. To be more precise, for the problem (10.30), we need to compute

$$\mathbf{w}^k = \left(\mathbf{QX}_k^2\mathbf{Q}^T\right)^{-1}\mathbf{QX}_k^2\mathbf{c} \tag{10.31}$$

where $\mathbf{X}_k$ is a diagonal matrix formed by the current primal solution $\mathbf{x}^k$ at the kth iteration. Or, equivalently, $\mathbf{w}^k$ is a solution vector to the system

$$\mathbf{Mu} = \mathbf{v} \tag{10.32}$$

where $\mathbf{M} = (\mathbf{QX}_k^2\mathbf{Q}^T)$ and $\mathbf{v} = \mathbf{QX}_k^2\mathbf{c}$.

Now, since $\mathbf{Q}$ is an orthonormal matrix, we would like to show that $(\mathbf{QX}_k^2\mathbf{Q}^T)^{-1}$ can be represented as a convergent power series in matrix form. Then we can compute $\mathbf{w}^k$ according to the basic idea mentioned at the beginning of this section. To achieve this objective, let us focus on the kth solution $\mathbf{x}^k > \mathbf{0}$ and choose

$$\lambda_{\max} = \max_i \left(x_i^k\right)^2 \quad \text{and} \quad \lambda_{\min} = \min_i \left(x_i^k\right)^2 \tag{10.33}$$

In this way, $\lambda_{\max}, \lambda_{\min} > 0$. Moreover, for any $\alpha > \lambda_{\max}$, we have

$$\mathbf{QX}_k^2\mathbf{Q}^T = \alpha\left[\mathbf{I} - \mathbf{Q}\overline{\mathbf{X}}\mathbf{Q}^T\right] \tag{10.34}$$

where $\overline{\mathbf{X}}$ is a diagonal matrix with its ith diagonal element being

$$\overline{x}_i = \frac{\alpha - (x_i^k)^2}{\alpha} < 1 \qquad \text{for } i = 1, \ldots, n$$

We now denote matrix $\mathbf{Q}\overline{\mathbf{X}}\mathbf{Q}^T$ by $\mathbf{B}$. It is clear that $\mathbf{B}$ is symmetric and positive definite with all eigenvalues being positive. Moreover, if $\kappa_{\max}$ is the largest eigenvalue

of $\mathbf{B}$ (often called the *spectral radius* of $\mathbf{B}$ and denoted by $\rho(\mathbf{B})$), we see that

$$\kappa_{\max} = ||\mathbf{B}|| = \left|\left|\mathbf{Q}\overline{\mathbf{X}}\mathbf{Q}^T\right|\right|$$

$$\leq ||\mathbf{Q}||\left|\left|\overline{\mathbf{X}}\right|\right|\left|\left|\mathbf{Q}^T\right|\right| \leq \frac{\alpha - \lambda_{\min}}{\alpha} < 1 \tag{10.35}$$

This further implies that each eigenvalue of $\mathbf{B}$ is less than 1. Therefore, the power series (10.27) is convergent and

$$\left(\mathbf{Q}\mathbf{X}_i^2\mathbf{Q}^T\right)^{-1} = \frac{1}{\alpha}(\mathbf{I} - \mathbf{B})^{-1} = \frac{1}{\alpha}\left[\mathbf{I} + \mathbf{B} + \mathbf{B}^2 + \mathbf{B}^3 + \cdots\right] \tag{10.36}$$

Observe that the matrix power series (10.27) can be approximated by a matrix polynomial $P_r(\mathbf{B})$ of degree $r > 0$, i.e.,

$$(\mathbf{I} - \mathbf{B})^{-1} = P_r(\mathbf{B}) + E_r(\mathbf{B}) \tag{10.37}$$

where

$$P_r(\mathbf{B}) = \sum_{k=0}^{r} \mathbf{B}^k \tag{10.38}$$

and

$$E_r(\mathbf{B}) = \sum_{k=r+1}^{\infty} \mathbf{B}^k \tag{10.39}$$

By (10.37), we have

$$||E_r(\mathbf{B})|| = \left|\left|(\mathbf{I} - \mathbf{B})^{-1} - P_r(\mathbf{B})\right|\right| \leq \sum_{k=r+1}^{\infty} ||\mathbf{B}||^k = \frac{||\mathbf{B}||^{r+1}}{(1 - ||\mathbf{B}||)} \tag{10.40}$$

Since $||\mathbf{B}|| = \rho(\mathbf{B})$, (10.40) implies that

$$||E_r(\mathbf{B})|| \leq \frac{[\rho(\mathbf{B})]^{r+1}}{[1 - \rho(\mathbf{B})]} \tag{10.41}$$

Consequently, for an arbitrary small tolerance level $\epsilon > 0$, in order to obtain a good matrix approximation with $||E_r(\mathbf{B})|| \leq \epsilon$, we simply have to choose a large r such that

$$r \geq \left\lceil \frac{\log\left([1 - \rho(\mathbf{B})]\epsilon\right)}{\log[\rho(\mathbf{B})]} - 1 \right\rceil \tag{10.42}$$

i.e., the smallest integer larger than

$$\frac{\log\left([1 - \rho(\mathbf{B})]\epsilon\right)}{\log[\rho(\mathbf{B})]} - 1$$

Observe that, from Equation (10.42), the value of r increases as ϵ gets smaller or $\rho(\mathbf{B})$ approaches unity. This factor directly affects the computational effort.

Summarizing what we have discussed, we can devise an algorithm, based on the LQ factorization, to solve the computational bottleneck (10.30) at the kth iteration for the primal affine scaling algorithm.

Algorithm LQ-1

Step 1: Choose $\epsilon > 0$ to be sufficiently small. Set

$$\lambda_{\min} = \min_i \left(x_i^k\right)^2 \quad \text{and} \quad \alpha > \lambda_{\max} = \max_i \left(x_i^k\right)^2$$

Set

$$\rho(\mathbf{B}) = 1 - \frac{\lambda_{\min}}{\alpha} \quad \text{and} \quad r = \left\lceil \frac{\log\left([1 - \rho(\mathbf{B})]\epsilon\right)}{\log[\rho(\mathbf{B})]} - 1 \right\rceil$$

Step 2: Set $\mathbf{s}^{(0)} = \mathbf{w}^{(0)} = \mathbf{Q}\mathbf{X}_i^2\mathbf{c}$ and $j = 1$. Set $\overline{x}_i = (\alpha - (x_i^k)^2)/\alpha$, for $i = 1, \ldots, n$. Compute $\mathbf{B} = \mathbf{Q}\overline{\mathbf{X}}\mathbf{Q}^T$, where $\overline{\mathbf{X}}$ is a diagonal matrix with $\overline{x}_i$ being its ith diagonal element.

Step 3: Repeat

$$\mathbf{s}^{(j)} \leftarrow \mathbf{B}\mathbf{s}^{(j-1)}$$

$$\mathbf{w}^{(j)} \leftarrow \mathbf{w}^{(j-1)} + \mathbf{s}^{(j)}$$

$$j \leftarrow j + 1$$

until $j = r + 1$.

Step 4: Set $\mathbf{w}^k \leftarrow \frac{1}{\alpha}\mathbf{w}^{(r)}$.

One potential drawback of applying Algorithm LQ-1 to solve large-scale problems is due to sparsity considerations. For a given sparse matrix $\mathbf{A}$, after factorization, $\mathbf{Q}$ may become very dense and difficult to manipulate. Fortunately, this potential problem can be overcome. The idea is to express $\mathbf{Q}\overline{\mathbf{X}}\mathbf{Q}^T$ as $\mathbf{L}^{-1}\mathbf{A}\overline{\mathbf{X}}\mathbf{A}^T(\mathbf{L}^T)^{-1}$. Remember that the sparsity issue of the Cholesky factor $\mathbf{L}$ has been handled in a previous section. Also note that the operation of premultiplying $\mathbf{L}^{-1}$ is equivalent to applying a "forward solve" and postmultiplying $(\mathbf{L}^T)^{-1}$ is equivalent to a "backward solve." Therefore, we can replace Step 3 in Algorithm LQ-1 by the following procedure:

Repeat:

Use "backward solve" to compute $\mathbf{u}^{(j-1)}$ for the system of equations

$$\mathbf{L}^T\mathbf{u}^{(j-1)} = \mathbf{s}^{(j-1)}$$

$$\mathbf{u}^{(j)} = \mathbf{A}\overline{\mathbf{X}}\mathbf{A}^T\mathbf{u}^{(j-1)}$$

Use "forward solve" to compute $\mathbf{s}^{(j)}$ for the system of equations

$$\mathbf{L}\mathbf{s}^{(j)} = \mathbf{u}^{(j)}$$

$$\mathbf{w}^{(j)} = \mathbf{w}^{(j-1)} + \mathbf{s}^{(j)}$$

$$j \leftarrow j + 1$$

until $j = r + 1$.

With this modification, since $\mathbf{A}$ and $\mathbf{L}$ are sparse, the sparsity issue of $\mathbf{Q}$ is bypassed. Any sparse matrix multiplication technique can be used here. Also note that, in Step 2, we no longer have to compute $\mathbf{B}$, and $\mathbf{s}^{(0)} = \mathbf{w}^{(0)} = \mathbf{L}^{-1}\mathbf{A}\mathbf{X}_k^2\mathbf{c}$.

Another potential drawback of Algorithm LQ-1 is due to the large value of r required by the algorithm. In theory, we know that as $\lambda_{\min}/\lambda_{\max} \to 0$ or, equivalently, as $\rho(\mathbf{B}) \to 1$, the value of r approaches infinity. This means that more and more iterations of Step 3 are needed, which results in inefficient computation. This inevitably happens in the interior-point methods, because when we approach an optimal vertex, even from interior, $\lambda_{\min}/\lambda_{\max}$ still approaches 0.

To overcome this potential problem, we may consider a fixed-point scheme. From (10.31)–(10.34), we know that

$$\begin{aligned}\mathbf{w}^k &= \left(\mathbf{Q}\mathbf{X}_k^2\mathbf{Q}^T\right)^{-1}\mathbf{Q}\mathbf{X}_k^2\mathbf{c} \\ &= \frac{1}{\alpha}(\mathbf{I}-\mathbf{B})^{-1}\mathbf{v}\end{aligned}$$

where $\mathbf{B} = \mathbf{Q}\overline{\mathbf{X}}\mathbf{Q}^T$ and $\mathbf{v} = \mathbf{Q}\mathbf{X}_k^2\mathbf{c}$. If we further denote $\mathbf{w} = \alpha\mathbf{w}^k$, then $\mathbf{w} = (\mathbf{I}-\mathbf{B})^{-1}\mathbf{v}$. This implies that

$$\mathbf{w} = \mathbf{B}\mathbf{w} + \mathbf{v} \tag{10.43}$$

In other words, $\mathbf{w}$ is a *fixed-point* solution of (10.43). Hence we can replace Algorithm LQ-1 by the following iterative scheme:

Algorithm LQ-2

Step 1: Choose $\epsilon > 0$ to be sufficiently small. Choose

$$\alpha > \lambda_{\max} = \max_i \left(x_i^k\right)^2$$

Set $\mathbf{v} = \mathbf{Q}\mathbf{X}_k^2\mathbf{c}$. Set

$$\overline{x}_i = \left(\alpha - \left(x_i^k\right)^2\right)/\alpha, \qquad \text{for } i = 1, \ldots, n$$

Compute $\mathbf{B} = \mathbf{Q}\overline{\mathbf{X}}\mathbf{Q}^T$.

Step 2: Set $j = 1$ and select an arbitrary $\mathbf{w}^{(0)}$.

Repeat:

$$\mathbf{w}^{(j)} = \mathbf{B}\mathbf{w}^{(j-1)} + \mathbf{v}$$

$$j \leftarrow j + 1$$

until $\|\mathbf{w}^{(j)} - \mathbf{w}^{(j-1)}\| \le \epsilon$.

Step 3: Assign

$$\mathbf{w}^k = (1/\alpha)\mathbf{w}^{(j)}$$

Notice that Algorithm LQ-2 allows an arbitrary starting point $\mathbf{w}^{(0)}$. In practice, when the primal affine scaling algorithm converges, the dual solution $\mathbf{w}^k$ varies little and

$\rho(\mathbf{B}) \to 1$. Hence, a considerable advantage can be gained by setting $\mathbf{w}^{(0)}$ as the previous dual estimate $\mathbf{w}^{k-1}$. In this way, though the convergence may slow down, we are close to a solution from the beginning.

Unlike Algorithm LQ-1, being a finite algorithm, Algorithm LQ-2 produces a sequence $\mathbf{w}^{(j)}$: $j = 0, 1, 2, \ldots$. We need to show that the sequence indeed converges and produces a solution to system (10.32).

Theorem 10.4. The sequence $\mathbf{w}^{(j)}$: $j = 0, 1, 2, \ldots$ generated by Algorithm LQ-2 is a Cauchy sequence and thus converges. Moreover, if we let $\mathbf{w}$ be its limit point, then $\mathbf{w}^k = (1/\alpha)\mathbf{w}$ solves the system (10.32).

Proof. Let $p \ge 1$, then

$$\begin{aligned}\left\|\mathbf{w}^{(j+p)} - \mathbf{w}^{(j)}\right\| &\le \left\|\mathbf{w}^{(j+p)} - \mathbf{w}^{(j+p-1)}\right\| + \cdots + \left\|\mathbf{w}^{(j+1)} - \mathbf{w}^{(j)}\right\| \\ &= \left\|\mathbf{B}^{j+p-1}\left(\mathbf{w}^{(1)} - \mathbf{w}^{(0)}\right)\right\| + \cdots + \left\|\mathbf{B}^{j}\left(\mathbf{w}^{(1)} - \mathbf{w}^{(0)}\right)\right\| \\ &\le [\rho(\mathbf{B})]^{j+p-1}\left\|\left(\mathbf{w}^{(1)} - \mathbf{w}^{(0)}\right)\right\| + \cdots + [\rho(\mathbf{B})]^{j}\left\|\left(\mathbf{w}^{(1)} - \mathbf{w}^{(0)}\right)\right\| \\ &\le \frac{[\rho(\mathbf{B})]^j}{[1-\rho(\mathbf{B})]}\left\|\mathbf{w}^{(1)} - \mathbf{w}^{(0)}\right\|\end{aligned}$$

which approaches 0 as j approaches $+\infty$. Hence $\mathbf{w}^{(j)}$ is a Cauchy sequence, and thus converges. From Step 2, we know its limit point $\mathbf{w}$ satisfies that

$$\mathbf{w} = \mathbf{B}\mathbf{w} + \mathbf{v}$$

Hence $\mathbf{w} = (\mathbf{I}-\mathbf{B})^{-1}\mathbf{v}$, and $\mathbf{w}^k$ solves (10.32).

The proof also shows that the rate of convergence of Algorithm LQ-2 is at least linear in $\rho(\mathbf{B})$. As a final remark, it is noteworthy that the iterative scheme presented in Algorithm LQ-2 may be accelerated. Let $0 < \theta < 1$ be arbitrary and consider Step 2 in Algorithm LQ-2 being modified as

$$\mathbf{w}^{(j)} = \theta\mathbf{w}^{(j-1)} + (1-\theta)\left[\mathbf{B}\mathbf{w}^{(j-1)} + \mathbf{v}\right]$$

It can be shown that the sequence $\{\mathbf{w}^{(j)}: j = 0, 1, 2, \ldots\}$ generated by the modified Algorithm LQ-2 is also a Cauchy sequence. Its convergence rate is at least linear in $\rho(\theta\mathbf{I} + (1-\theta)\mathbf{B})$. This could improve the rate of convergence. But it is not an easy problem to find an optimal θ for a general setting.

A more general and new model of treating the infinitely summable series by using Chebychev approximation to accelerate the convergence is also under current investigation.

10.5 CONCLUDING REMARKS

In this chapter we have introduced three methods to overcome the computational bottleneck of implementing interior-point algorithms. While all three methods (Cholesky, CG,

and LQ) appear promising, the most popular is the Cholesky factorization method. This is partially due to the fact that a vast amount of literature is available in this area and it is numerically stable. Moreover, any floating-point exceptions can be easily "tracked" and "trapped" in the implementations of Cholesky factorization.

Sparsity considerations are very important in solving large-scale problems. In addition to what has been introduced in this chapter, one idea is to split the constraint matrix into two parts, a dense part and a sparse part, and treat them separately. For the sparse part we may use sparse matrix techniques, such as the ones described previously, while a low rank updating scheme or the conjugate gradient method may be used for the dense part. This approach could result in substantial reduction in computer run-time for a number of cases. It is often referred to as the *column dropping technique*. However, it requires us to take extreme care to ensure the robustness of the implemented software. This has been recognized as a challenging task. The major problem is numerical instability. Even though the original system is nonsingular, the subsystem of equations to be solved with some columns deleted may turn out to be singular. Another idea is to solve dual problems as if they were primal. Under this situation, as long as the original problem does not contain both dense rows and dense columns, we may still be able to take advantage of one formulation.

Other techniques that have been tested for the implementation of interior-point methods include (1) matrix reduction via the elimination of singleton columns and corresponding variables, (2) fixing variables to their bounds, whenever applicable, (3) scaling the data to render numerical stability in computation, and (4) using various acceleration techniques applied to Cholesky factorization. The authors' limited experience with these techniques indicates that little improvement can be obtained for overall efficiency in solving real-life problems.

Finally, we want to mention that there have been attempts to apply *decomposition principles* in conjunction with interior-point methods to solve large-scale problems. Basically, a large-scale problem is decomposed into a *restricted master problem* and several *subproblems*, and then interior-point methods are applied. However, as far as the authors know, there is no evidence showing any significant improvement.

REFERENCES FOR FURTHER READING

10.1. Adler, I., Karmarkar, N., Resende, M. G. C., and Veiga, G., "An implementation of Karmarkar's algorithm for linear programming," *Mathematical Programming* 44, 297–335 (1989).

10.2. Adler, I., Karmarkar, N., Resende, M. G. C., and Veiga, G., "Data structures and programming techniques for the implementation of Karmarkar's algorithm," *ORSA Journal of Computing* 1, 84–106 (1989).

10.3. Cheng, Y-C., Houck, D. J., Jr., Meketon, M. S., Slutsman, L., Vanderbei, R. J., and Wang, P., "The AT&T KORBX® System," *AT&T Technical Journal* 68, No. 3, 7–19 (1989).

10.4. Duff, I. S., Erisman, A. M., and Reid, J. K., *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford (1986).

10.5. Gay, D., "Massive memory buys little speed for complete in-core sparse Cholesky factorizations," Technical Report, AT&T Bell Laboratories (1988).

10.6. George, A., and Liu, J. W., *Computer Solution of Large Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ (1981).

10.7. Gill, P. E., Murray, W., and Wright, M. H., *Numerical Linear Algebra and Optimization*, Vol. 1, Addison-Wesley, Redwood City, CA (1991).

10.8. Golub, G. H., and Van Loan, C. F., *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD (1983).

10.9. Hestenes, M. R., and Stiefel, E., "Methods of conjugate gradients for solving linear systems," *Journal of Researches National Bureau of Standards* 49, 409–436 (1952).

10.10. Housos, E. C., Huang, C. C., and Liu, J. M., "Parallel algorithms for the AT&T KORBX® System," *AT&T Technical Journal* 68, No. 3, 37–47 (1989).

10.11. Markowitz, H. M., "The elimination form of the inverse and its application to linear programming," *Management Science* 3, 255–269 (1957).

10.12. Pan, V., "How can we speed up matrix multiplications?", *SIAM Review* 26, 393–415 (1984).

10.13. Pissanetzky, S., *Sparse Matrix Technology*, Academic Press, New York (1984).

10.14. Puthenpura, S., Saigal, R., and Sinha, L. P., "Application of LQ factorization in implementing the Karmarkar algorithm and its variants," Technical Memorandum, No. 51173-900205-01TM, AT&T Bell Laboratories (1990).

10.15. Saigal, R., "An infinitely summable series implementation of interior point methods," Technical Report 92-37, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, May (1992).

10.16. Saigal, R., "Matrix partitioning methods for interior point algorithms," Technical Report 92-39, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, June (1992).

10.17. Vanderbei, R. J., "An implementation of the minimum-degree algorithm using simple data structures," Technical Memorandum, No. 11212-900115-02TM, AT&T Bell Laboratories (1990).

10.18. Vanderbei, R. J., "ALPO: Another linear program solver," Technical Memorandum, No. 11212-900522-18TM, AT&T Bell Laboratories (1990).

10.19. Van Loan, C., "A survey of matrix computations," Technical Report, Cornell University (1990).

10.20. Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press (1965).

EXERCISES

10.1. Let $\mathbf{M}$ be an $n \times n$ symmetric positive definite matrix. Show that

(a) Any principal submatrix of $\mathbf{M}$ is positive definite.

(b) $\mathbf{M}$ is nonsingular and its inverse matrix $\mathbf{M}^{-1}$ is also positive definite.

(c) $\mathbf{B}^T\mathbf{M}\mathbf{B}$ is positive definite if and only if $\mathbf{B}$ is nonsingular.

(d) $m_{ii} > 0$, for $i = 1, \ldots, n$.

(e) $\max_{1\le i\le n} m_{ii} \ge \max_{1\le i,j\le n} |m_{ij}|$.

(f) $m_{ii}m_{jj} \ge m_{ij}^2$, for all i, j.

10.2. Let $\mathbf{P}$ be a permutation matrix that interchanges only two columns or two rows.

(a) Show that $\mathbf{P}$ is symmetric.

(b) Is the product of two such permutations symmetric? Why?

(c) Suppose that R is a product of a finite number of such permutations and $\mathbf{M}$ is an $n \times n$ symmetric matrix; show that any diagonal element of $\mathbf{R}^T\mathbf{M}\mathbf{R}$ must be a diagonal element of $\mathbf{M}$.

10.3. Is the product of two orthogonal matrices orthogonal? Why?

10.4. Let $\mathbf{M}$ be an $n \times n$ nonsingular matrix with $\mathbf{QR}$ factorization $\mathbf{M} = \mathbf{QR}$.

(a) Is this factorization unique?

(b) If $\mathbf{M}$ has another $\mathbf{QR}$ factorization such that $\mathbf{M} = \overline{\mathbf{Q}}\overline{\mathbf{R}}$. What is the connection between $\mathbf{Q}$ and $\overline{\mathbf{Q}}$? $\mathbf{R}$ and $\overline{\mathbf{R}}$?

10.5. Derive the "outer product form" of the Cholesky factorization algorithm based on the proof of Theorem 10.1. [*Hint:* Express

$$\mathbf{M} = \mathbf{L}_1\mathbf{M}_1\mathbf{L}_1^T, \quad \mathbf{M}_1 = \mathbf{L}_2\mathbf{M}_2\mathbf{L}_2^T, \quad \ldots, \quad \mathbf{M}_{n-1} = \mathbf{L}_n\mathbf{I}_n\mathbf{L}_n^T$$

where $\mathbf{L}_i$ for $i = 1, 2, \ldots, n$ are lower triangular matrices and $\mathbf{I}_n$ is the $n \times n$ identity matrix. Then show that $\mathbf{L}$, the Cholesky factor of $\mathbf{M}$, is $\mathbf{L}_1 + \mathbf{L}_2 + \cdots + \mathbf{L}_n - (n-1)\mathbf{I}_n$.]

10.6. Let $\mathbf{A}$ be an $(m \times n)$-dimensional matrix $(m < n)$ with rank $r \le m$. Assume that $\mathbf{A} = \mathbf{U}\mathbf{V}^T$. Show that $\mathbf{U}$ and $\mathbf{V}$ are of rank r.

10.7. Let $\mathbf{A}$ be an $(m \times n)$-dimensional matrix $(m < n)$ with full row-rank. Prove that $\mathbf{A}\mathbf{A}^T$ is symmetric and positive definite. If

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & -2 \\ -1 & 2 & 5 \end{bmatrix}$$

find the Cholesky factor of $\mathbf{A}\mathbf{A}^T$.

10.8. Let $\mathbf{A}$ be an $(m \times n)$-dimensional matrix (say $m < n$). Also let $\mathbf{B}$ be an $n \times m$ matrix such that $\mathbf{B}\mathbf{A} = \mathbf{I}$. Show that $\mathbf{B}$ exists if and only if the columns of $\mathbf{A}$ are linearly independent. Furthermore, show that $\mathbf{B}$ is unique if and only if the rows of $\mathbf{A}$ are linearly independent. Construct such a $\mathbf{B}$ for a given $\mathbf{A}$. [*Note:* $\mathbf{B}$ is called the "left inverse" of $\mathbf{A}$.]

10.9. Consider the matrix $\mathbf{M}$ in Example 10.1. Compute its Cholesky factor $\mathbf{L}$ and then apply the minimum degree reordering to recompute the Cholesky factor. Remember to compare these two results.

10.10. Verify that Algorithm F-2 gives the correct answer, and access $\mathbf{L}$ column by column.

10.11. Prove that $\mathbf{d}^k$, for $k = 1, 2, \ldots$, obtained by (10.23) and (10.24), indeed are mutually conjugate with respect to matrix M.

10.12. For the matrix $\mathbf{A}$ of Exercise 10.7, perform LQ factorization to get the corresponding matrix $\mathbf{Q}$.

10.13. Provide a simple example, say $1 \le m < n \le 10$, such that $\mathbf{A}$ is relatively sparse but $\mathbf{Q}$ is very dense.

10.14. Consider the fixed-point iteration scheme in connection with the LQ factorization technique, where

$$\mathbf{w}^{(j)} = \theta\mathbf{w}^{(j-1)} + (1-\theta)\left[\mathbf{B}\mathbf{w}^{(j-1)} + \mathbf{v}\right]$$

Let $\kappa_{\max}$ and $\kappa_{\min}$ be the largest and smallest eigenvalues of $\mathbf{B}$. Show that

$$\rho(\theta\mathbf{I} + (1-\theta)\mathbf{B}) < \rho(\mathbf{B}) \quad \text{if and only if} \quad 0 > \theta > \frac{-\kappa_{\min}}{1-\kappa_{\min}}$$

10.15. An advantage of the C programming language is due to the dynamic memory allocation which substantially reduces memory requirements for the implementation of the interior-point algorithms. If you know what "dynamic memory allocation" is, discuss the way of implementing the minimum degree reordering, the Cholesky factorization, and the forward/backward solves with the dynamic memory allocation scheme.