PART 1
Synchronous Algorithms

2

Algorithms for Systems of Linear Equations and Matrix Inversion

Let A be an $n \times n$ real matrix, let b be a vector in $\Re^n$, and consider the system of linear equations

$$Ax = b,$$

where x is an unknown vector to be determined. There is a variety of methods for solving this system, usually classified as *direct* and *iterative*. Direct methods find the exact solution with a finite number of operations, typically of the order of n^3. Iterative methods do not obtain an exact solution of $Ax = b$ in finite time, but they converge to a solution asymptotically. Nevertheless, iterative methods often yield a solution, within acceptable precision, after a relatively small number of iterations, in which case they are preferred to direct methods. This is usually the case when n is very large, for example, when the system $Ax = b$ arises from discretization of a linear partial differential equation, and in many other applications. Iterative methods may also have smaller storage requirements than direct methods, when the matrix A is sparse. A more general problem is the computation of the inverse A^{-1} of A, which can also be solved by either direct or iterative methods.

In this chapter, we study parallel algorithms for solving the system $Ax = b$ and for matrix inversion. Some of these algorithms are just parallel implementations of traditional serial algorithms, whereas others are more recent, developed with the purpose of better

exploiting parallelism. Furthermore, some of these algorithms are of purely theoretical interest, whereas others are widely used in practice. We discuss concepts of complexity and efficiency in the context of direct methods. These concepts are not quite applicable to iterative methods, which do not terminate in finite time. A more appropriate measure for such algorithms is their speed of convergence to the solution. Typical iterative methods converge *geometrically*, or *at the rate of a geometric progression.* This means that the sequence of vectors $\{x(t)\}$ generated by an iterative algorithm has the property $\|x(t) - x^*\| \leq c\rho^t$, where x^* is the solution of the system $Ax = b$, c is a positive constant, ρ is a positive constant smaller than 1, and $\|\cdot\|$ is a vector norm. The smaller the value of ρ, the faster is the convergence of the algorithm.

Throughout this chapter, a synchronous model of computation is assumed. Furthermore, we often assume an idealized model in which any two processors may communicate instantaneously via an interconnection network or a shared memory. In effect, communication costs are excluded from such an analysis. However, at several points, we pause to indicate how some of the more practical algorithms may be implemented on specific architectures with small or negligible communication penalty.

In Section 2.1, we present parallel direct algorithms for the case where the matrix A has a special structure; in particular, A is assumed to be triangular or tridiagonal. In Section 2.2, we present some classical direct methods for solving the system $Ax = b$ for the case of a general matrix A, and describe their parallel implementation. In Section 2.3, we present a very fast direct parallel algorithm for inverting a square matrix. This algorithm is of theoretical interest but is impractical due to excessive processor requirements and poor numerical stability. In Section 2.4, we present a few classical iterative methods for solving the system $Ax = b$, and in Section 2.5, we comment on their parallel implementation, including an example arising in the numerical solution of partial differential equations and a brief discussion of multigrid algorithms. In Section 2.6, we develop the machinery for the convergence analysis of iterative methods. While these results are classical and fairly old, some of the tools introduced here will be used in the much harder convergence proofs of asynchronous iterative algorithms (Chapters 6 and 7). In Section 2.7, we present the conjugate gradient method and comment on its parallelization. In Section 2.8, we study iterative algorithms for the computation of the invariant distribution of a finite state Markov chain. Finally, in Section 2.9, we present a very fast, Newton–like iterative algorithm for matrix inversion.

2.1 PARALLEL ALGORITHMS FOR LINEAR SYSTEMS WITH SPECIAL STRUCTURE

2.1.1 Triangular Matrices and Back Substitution

Let A be a lower triangular square matrix of dimensions $n \times n$, that is, $a_{ij} = 0$ for $i < j$. Our objective is to compute A^{-1}, assuming that A is invertible; equivalently, we assume that $a_{ii} \neq 0$ for all i. We first consider the case where $a_{ii} = 1$ for all i, and we subsequently generalize. We write $A = I - L$, where L is strictly lower triangular, that

is, its elements ℓ_{ij} satisfy $\ell_{ij} = 0$ for $i \leq j$. It is straightforward to verify that the ijth element of L^k is zero if $i - j < k$, so $L^n = 0$.

Lemma 1.1. If $A = I - L$, where L is strictly lower triangular, then

$$A^{-1} = (I + L + L^2 + \cdots + L^{n-1}). \tag{1.1}$$

Proof. Let B be the right–hand side of Eq. (1.1). An easy calculation yields $B(I - L) = I - L^n = I$, because $L^n = 0$. This implies that $B = A^{-1}$. **Q.E.D.**

Equation (1.1) leads to a straighforward algorithm for computing A^{-1}: compute, in parallel, $L^2, \ldots, L^{n-1}$, and then add the results. According to the discussion in Subsection 1.2.3, all of these operations can be performed in time $O(\log^2 n)$ using n^4 processors, excluding communication costs. This algorithm, although very simple, uses an excessive number of processors. A more efficient algorithm is obtained using the following lemma.

Lemma 1.2. If $A = I - L$, where L is strictly lower triangular, then

$$A^{-1} = (I + L^{2^{\lceil \log n \rceil - 1}})(I + L^{2^{\lceil \log n \rceil - 2}}) \cdots (I + L^4)(I + L^2)(I + L). \tag{1.2}$$

Proof. Expand the product in the right–hand side of Eq. (1.2). Since $L^n = 0$, we are left with $I + L + L^2 + \cdots + L^{n-1}$, which is equal to A^{-1}, by Lemma 1.1. **Q.E.D.**

Lemma 1.2 leads to the following algorithm. We compute, by successive squaring, L^2, $L^4, \ldots, L^{2^{\lceil \log n \rceil - 1}}$, we add the identity to each one of these matrices, and, finally, we carry out the multiplications in the right–hand side of Eq. (1.2). The addition of the identity can be carried out in one time unit. The other steps consist of $O(\log n)$ successive matrix multiplications and can be carried out, with n^3 processors, in time $O(\log^2 n)$, excluding communication costs (Subsection 1.2.3).

Suppose now that the assumption $a_{ii} = 1$ fails. We define a diagonal matrix D, such that $d_{ii} = a_{ii}$ for each i, and notice that $D^{-1}A$ is triangular and has unit diagonal elements. Thus, we may first transform A to $D^{-1}A$ (this takes a single time unit using n^2 processors), invert $D^{-1}A$ to obtain $A^{-1}D$ (using the preceding algorithm), and finally right–multiply the result by D^{-1} (this takes a single time unit using n^2 processors) to recover A^{-1}. Therefore, the time required by the algorithm remains $O(\log^2 n)$ using n^3 processors.

We now present a different method, of the "divide–and–conquer" type, whose performance is comparable to that of the preceding algorithm (the assumption $a_{ii} = 1$ is no longer needed). We partition the A matrix into blocks:

$$A = \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix},$$

where A_1 is of size $\lceil n/2 \rceil \times \lceil n/2 \rceil$. Notice that A_1 and A_3 are lower triangular. Moreover, it is easily shown that

$$A^{-1} = \begin{bmatrix} A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{bmatrix},$$

(multiply the above expressions for A and A^{-1} and verify that the product is the identity). Based on the above decomposition, we obtain the following algorithm. Given an $n \times n$ triangular matrix A:

1. If $n = 1$, then obtain A^{-1} in the obvious way.
2. If $n > 1$, then partition A as indicated above and do the following:
   (a) Invert (concurrently) A_1 and A_3. (Since A_1 and A_3 are lower triangular, they can be inverted by using the same algorithm.)
   (b) Multiply A_3^{-1} with A_2 to obtain $A_3^{-1}A_2$.
   (c) Right–multiply the result of (b) by A_1^{-1}.

Notice that steps (b) and (c) take $O(\log n)$ time using n^3 processors. Thus, if $T(n)$ denotes the time required by the algorithm for inverting a matrix of dimensions $n \times n$, we have $T(n) = T(\lceil n/2 \rceil) + O(\log n)$, which yields $T(n) = O(\log^2 n)$ using n^3 processors, excluding communication costs.

The methods presented so far do not simplify when one is faced with the presumably easier task of solving a system $Ax = b$. Furthermore, if communication overhead is properly taken into account, the time requirements can be much larger than $O(\log^2 n)$ for certain processor architectures. For this reason, and in view of their large processor requirements, these algorithms are theoretically interesting but impractical. We now describe a practical method for solving $Ax = b$, called *back substitution*, which is obtained by parallelizing the natural sequential algorithm for this problem.

Under the assumption that A is lower triangular, the ith equation of the system $Ax = b$ is

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{ii}x_i = b_i. \tag{1.3}$$

The following parallel version of the back substitution algorithm employs n processors. Suppose that at the beginning of the ith stage, the values of the variables $x_1, \ldots, x_{i-1}$ and of the expressions $a_{j1}x_1 + \cdots + a_{j,i-1}x_{i-1}$ for each $j \geq i$, are available. Then the ith processor evaluates x_i by solving Eq. (1.3):

$$x_i = \frac{1}{a_{ii}}\left(b_i - a_{i1}x_1 - \cdots - a_{i,i-1}x_{i-1}\right).$$

Finally, each processor j, with $j \geq i + 1$, evaluates the expression $a_{j1}x_1 + \cdots + a_{ji}x_i$ by adding $a_{ji}x_i$ to the previously available expression $a_{j1}x_1 + \cdots + a_{j,i-1}x_{i-1}$. The algorithm terminates at the end of the nth stage, when all variables $x_1, \ldots, x_n$ have been

computed. Clearly, the parallel time required for each stage is constant. Therefore, the total time required by this version of back substitution is $O(n)$ using n processors and excluding communication costs.

We now compare the efficiencies of the algorithms presented so far. We recall that the efficiency of a parallel algorithm is defined (Subsection 1.2.2) to be equal to $T^*(n)/\big(pT_p(n)\big)$, where n is a measure of problem size, p is the number of processors, $T_p(n)$ is the time spent by a parallel algorithm that uses p processors, and $T^*(n)$ is the time required by the fastest sequential algorithm (or by a benchmark sequential algorithm). Notice that any sequential algorithm needs $\Omega(n^2)$ time units to solve the problem $Ax = b$: the reason is that the solution depends on $\Omega(n^2)$ numbers, the entries of A. Furthermore, the back substitution algorithm, if serially implemented, takes $O(n^2)$ time. We therefore have $T^*(n) = \Theta(n^2)$. We then see that the efficiency of the first methods of this subsection is $O\big(1/(n\log^2 n)\big)$, as opposed to the efficiency of back substitution, which is $\Theta(1)$. Back substitution is slower, but the other methods need an excessive number of processors, which is disproportionately large when compared with the additional speedup that these excess processors are providing. In practice, parallel back substitution is universally used, not only for its higher efficiency, but also because of its modest communication requirements, which will be analyzed shortly.

Back substitution can also be used for computing the inverse of a triangular matrix A as follows. Since $AA^{-1} = I$, we see that the ith column x^i of A^{-1} satisfies $Ax^i = e^i$, where e^i is the ith unit vector. Thus, A^{-1} is obtained by solving n systems of equations, and parallel back substitution can be used for each one. These systems can be solved simultaneously [$O(n)$ time using n^2 processors], or one at a time [$O(n^2)$ time using n processors].

We now turn to the implementation of back substitution on special architectures. It is natural to consider a linear array of n processors, whereby the ith processor is given the entries in the ith row of A and the ith component of the vector b. As shown in Fig. 2.1.1, it is possible, by pipelining the communication and interleaving it with the computation, to obtain $O(n)$ execution time. Thus, the communication penalty can only increase the execution time by a constant factor. If this constant factor is large, the impact of the communication penalty can be reduced by using fewer processors and assigning to each one several rows of the matrix A. (See Subsection 1.3.5 for the general effects of reduced numbers of processors on the communication penalty.) Finally, if a hypercube architecture is to be used instead, the communication penalty can only be smaller, since a linear array can be imbedded in a hypercube (Subsection 1.3.4).

There is also an alternative implementation of back substitution in which the ith processor is given the entries in the ith column of A. The issues are somewhat similar as in the previous implementation and the time requirements are again $O(n)$ using n processors (see Fig. 2.1.2).

2.1.2 Tridiagonal Systems and Odd–Even Reduction

We consider a system of equations $Ax = b$, where A is tridiagonal, that is, $a_{ij} = 0$ if $|i - j| > 1$. Such a system is of the form:

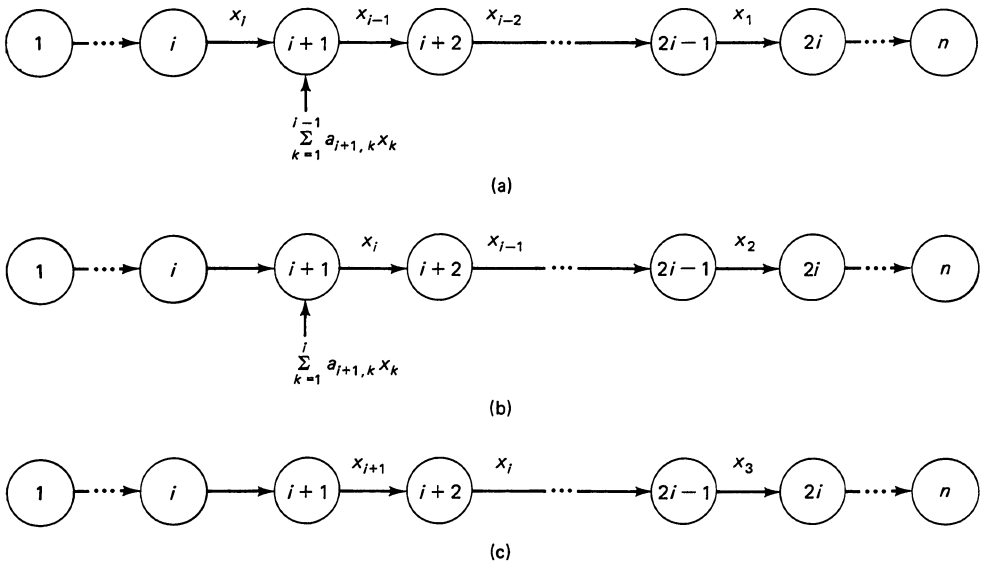


Figure 2.1.1 Implementation of back substitution in a linear array of n processors. The ith processor knows b_i and the entries of the ith row of A. The value of each x_i is transmitted to the right as soon as it is available and communication of such values is pipelined. (a) Snapshot of the algorithm as soon as x_i is computed. At this point, processor $i+1$ has already received $x_1, \ldots, x_{i-1}$ and has evaluated the sum $\sum_{k=1}^{i-1} a_{i+1,k}x_k$. (b) Once x_i is received by processor $i+1$, it is forwarded to the right and the sum $\sum_{k=1}^{i} a_{i+1,k}x_k$ is evaluated. Then, x_{i+1} is computed using Eq. (1.3). (c) The value of x_{i+1} is transmitted to processor $i+2$. The time between the evaluation of two successive components x_i and x_{i+1} is $O(1)$, and the total time of the algorithm is proportional to n.

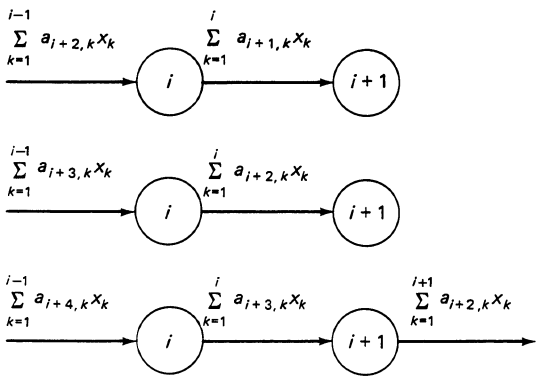


Figure 2.1.2 Alternative implementation of back substitution in a linear array of n processors. The ith processor knows b_i and the entries of the ith column of A. In (a), the value of x_i has just been computed by processor i. As soon as processor $i+1$ receives $\sum_{k=1}^{i} a_{i+1,k}x_k$, it computes x_{i+1}. Then, processor $i+1$ receives the value of $\sum_{k=1}^{i} a_{i+2,k}x_k$, as in (b), adds to it the value of $a_{i+2,i+1}x_{i+1}$, and transmits the sum $\sum_{k=1}^{i+1} a_{i+2,k}x_k$ to processor $i+2$, as in (c). A detailed timing analysis shows that the total execution time is proportional to n.

$$g_1x_1 + h_1x_2 = b_1, \tag{1.4}$$

$$f_ix_{i-1} + g_ix_i + h_ix_{i+1} = b_i, \quad i = 2, 3, \ldots, n-1, \tag{1.5}$$

$$f_nx_{n-1} + g_nx_n = b_n. \tag{1.6}$$

Here, g_i are the diagonal elements of A and f_i (respectively, h_i) are the entries of A below (respectively, above) the diagonal. There are several methods for solving such a system and most of them are easily parallelizable. We describe here a representative one that is called *odd–even reduction.*

The basic idea is that if $g_i \neq 0$, we can solve Eq. (1.5) for x_i in terms of x_{i-1} and x_{i+1}. If we do this for every odd integer i and then substitute the expression for x_i into the remaining equations, we are left with a system of equations involving only the variables x_i, with i even. The resulting system of equations is again tridiagonal and has about half as many variables. The same procedure is then applied recursively.

We now describe the algorithm in more detail. To simplify the equations, we use the convention $x_0 = x_{n+1} = 0$, which makes Eq. (1.5) valid for $i = 1$ and $i = n$. We solve Eq. (1.5) for x_i and obtain

$$x_i = \frac{1}{g_i}\left(b_i - f_i x_{i-1} - h_i x_{i+1}\right). \tag{1.7}$$

We use Eq. (1.7), with i replaced by $i - 1$ and $i + 1$, to eliminate x_{i-1} and x_{i+1} from Eq. (1.5). This yields

$$\frac{f_i}{g_{i-1}}\left(b_{i-1} - f_{i-1}x_{i-2} - h_{i-1}x_i\right) + g_i x_i + \frac{h_i}{g_{i+1}}\left(b_{i+1} - f_{i+1}x_i - h_{i+1}x_{i+2}\right) = b_i,$$

which simplifies to

$$\begin{aligned} -\left(\frac{f_i f_{i-1}}{g_{i-1}}\right)x_{i-2} + \left(g_i - \frac{h_{i-1}f_i}{g_{i-1}} - \frac{h_i f_{i+1}}{g_{i+1}}\right)x_i - \left(\frac{h_i h_{i+1}}{g_{i+1}}\right)x_{i+2} \\ = b_i - \frac{f_i}{g_{i-1}}b_{i-1} - \frac{h_i}{g_{i+1}}b_{i+1}. \end{aligned} \tag{1.8}$$

Consider Eq. (1.8) for each even index i, $1 \leq i \leq n$. It is a system in the variables $x_2, \ldots, x_{2\lfloor n/2 \rfloor}$, and it is clearly a tridiagonal system. We then use the same procedure, recursively, to obtain a smaller system, until we are left with a single equation in a single variable, which we solve directly. We then proceed backwards to obtain the values of the eliminated variables (see Fig. 2.1.3). Exercise 1.2 suggests a modification with which the backward evaluation of eliminated variables is not needed and the time of the algorithm is cut by a factor of two, approximately.

The algorithm breaks down if at some stage a division by zero is attempted [see Eq. (1.8)]. This may occur even if the original matrix A is nonsingular and has nonzero diagonal entries. In practice, this happens somewhat rarely and odd–even reduction is often used, although there are no theoretical guarantees.

For a timing analysis of the algorithm, notice that at each stage, the number of variables is reduced approximately by half. Thus, after $\Theta(\log n)$ stages, all but one of the variables are eliminated. At each stage, we need to compute the coefficients of the

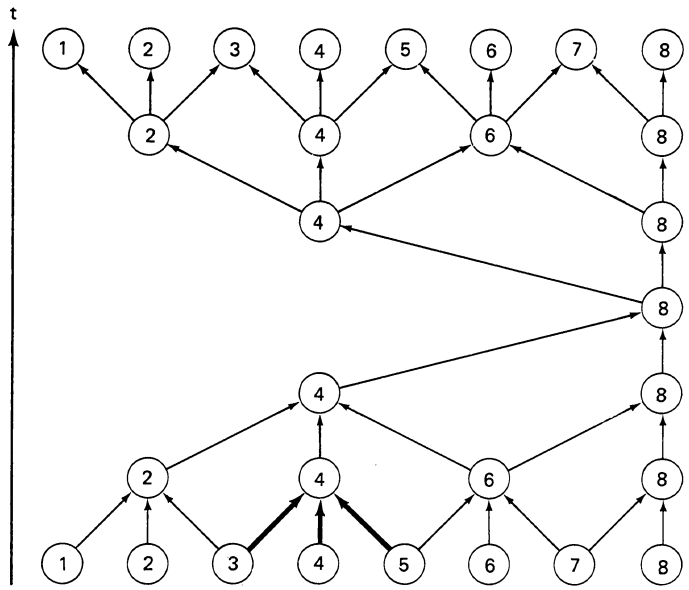


Figure 2.1.3 Illustration of odd–even reduction for $n = 8$. At the first stage, the variables x_1, x_3, x_5, and x_7 are eliminated using Eq. (1.7) to obtain a system of equations of the form of Eq. (1.8) involving x_2, x_4, x_6, and x_8. At the second stage, x_2 and x_6 are eliminated. Then x_4 is eliminated, which yields a single equation involving x_8. Once x_8 is evaluated, the remaining variables can be evaluated by following the reverse steps. For example, once x_2, x_4, x_6, and x_8 are evaluated, the values of x_1, x_3, x_5, and x_7 are readily obtained from Eq. (1.7). The arcs in this diagram indicate data dependencies. For example, the coefficients f_4', g_4', h_4', and b_4' of the equation $f_4'x_2 + g_4'x_4 + h_4'x_6 = b_4'$ obtained after the first reduction depend on f_i, g_i, h_i, and b_i for $i = 3, 4, 5$ [cf. Eq. (1.8)]. This fact is indicated by the thicker lines in the figure.

reduced system. It is immediate from Eq. (1.8) that this may be accomplished in four time units using $O(n)$ processors. A similar comment applies to the back substitution phase, during which the previously eliminated variables are evaluated. It follows that the overall algorithm can be implemented in $O(\log n)$ time using $O(n)$ processors. It is not hard to see that the total number of operations in this algorithm is only $O(n)$, because the computational requirements of each stage are about half of the requirements of the preceding stage. This implies that there exists a sequential algorithm that runs in $O(n)$ time. Furthermore, this is optimal because any sequential algorithm needs $\Omega(n)$ time just to read the input. We thus have $T^*(n) = \Theta(n)$ and we can conclude that the efficiency of the above described parallel implementation of the odd–even reduction algorithm is $\Theta(1/\log n)$. A more efficient implementation is suggested in Exercise 1.3.

We now study the implementation of odd–even reduction on parallel architectures with special interconnection topologies. Consider first a linear array of n processors and assume that each processor knows the entries of the ith row of the matrix A and the ith entry of b, and is responsible for eventually computing the value of x_i. The first stage of the computation [evaluation of the coefficients in Eq. (1.8)] is easily accomplished with $O(1)$ communication, because the ith processor (for i even) only needs to know

the values of the coefficients possessed by neighboring processors. However, after k reductions, we are left with a system in the variables x_i, where i is an integer multiple of 2^k, and this implies that processor $i2^k$ has to communicate with processor $(i+1)2^k$. (This is evident from Fig. 2.1.3.) In particular, at the last reduction step, the communication overhead is $\Omega(n)$. It follows that the parallel execution time is $\Omega(n)$, which is at least as bad as for the serial algorithm. In fact, this conclusion could be reached more easily from the observation that the value of x_1 depends on data possessed by processors at $\Theta(n)$ distance away, and therefore the parallel execution time has to be $\Omega(n)$.

Consider next a linear array of p processors ($p < n$), each one in charge of n/p successive variables (assuming that n/p is an integer). During the first $N = \lfloor \log(n/p) \rfloor$ stages of the algorithm, most computations are local to each processor, the total number of arithmetic operations executed by each processor during these N stages is $O(n/p)$, and only a small amount of communication between neighboring processors is needed. After the Nth stage, we are left with a tridiagonal system in $O(p)$ variables. This system can be solved by transmitting its coefficients to a single processor, which solves it and broadcasts the results back to the p processors. This can be done with $O(p)$ time spent for communication and $O(p)$ time spent for computation. We conclude that the total time is $O(n/p) + O(p)$. By optimizing with respect to p, we obtain $p = \Theta(n^{1/2})$ and an $O(n^{1/2})$ total execution time, which is better than the $O(n)$ serial time but worse than the $O(\log n)$ time obtained assuming all communication is instantaneous.

We finally consider a hypercube architecture with n processors. Let a linear array of processors be imbedded in the hypercube, according to a reflected Gray code (Subsection 1.3.4). We use the terminology of Subsection 1.3.4: the *logical distance* of two processors is their distance as elements of the linear array, and their *physical distance* is their distance when all of the hypercube arcs are available. We found that a linear array is inappropriate because processors $i2^k$ and $(i+1)2^k$ need to communicate at the kth stage. Let us now recall from Subsection 1.3.4 that, with a reflected Gray code imbedding, processors at logical distance 2^k, $k > 0$, have a physical distance equal to 2 and, furthermore, all pairs of processors of the form $\left(i2^k, (i+1)2^k\right)$, $i = 1, 2, \ldots$, can communicate to each other, simultaneously, in two time units. It follows that each stage of the odd–even reduction algorithm can be accomplished with only $O(1)$ time spent for communication, thereby maintaining the total execution time at $O(\log n)$ using n processors. In practice, the communication requirements per stage may be dominant when compared to the computation requirements per stage, even though both are $O(1)$. It may then be appropriate to use a number of processors smaller than n, which reduces the associated communication penalty, as discussed in Subsection 1.3.5.

We close by pointing out that the odd–even reduction algorithm can also be applied to the solution of *block–tridiagonal* systems of equations. Such systems have the structure of Eqs. (1.4) to (1.6), except that each x_i is now a vector of dimension k and each f_i, g_i, and h_i is a matrix of dimensions $k \times k$. A key difference is that a term such as $1/g_{i-1}$ [see Eq. (1.8)] has to be interpreted as a matrix inverse. For this reason, each processor in a parallel implementation has to invert a $k \times k$ matrix at each stage. The computational requirements for each processor are now substantially larger [$O(k^3)$ per stage if the matrices g_i do not have any special structure], but the increase of the communication

requirements is smaller [only $O(k^2)$ numbers need to be transmitted by each processor at each stage]. We conclude that the communication penalty is reduced as k increases.

EXERCISES

1.1. Consider the execution of parallel back substitution in a linear array of p processors, where $p < n$, and n/p is an integer. Suppose that each processor is given n/p consecutive rows of the matrix A, as well as the corresponding entries of the vector b. Assume that each message consists of a single real number and that all messages incur the same delay. Let $T(n, p)$ be the time spent by the algorithm, under the assumption that computation is instantaneous. Design the details of the algorithm so that $T(n, p)$ is made as small as possible and find an expression for $T(n, p)$.

1.2. **[HoJ81]** Modify the odd–even reduction algorithm so that at the end of the last stage of variable eliminations, the value of x_i is obtained for all i, thus avoiding the need for a backward substitution phase. Furthermore, this should be done with only n processors. *Hint:* With the algorithm of Fig. 2.1.3, the value of x_8 is obtained after the last elimination stage. Construct, for each i, a similar algorithm that produces the value of x_i, and run all those algorithms simultaneously.

1.3. Provide a parallel implementation of odd–even reduction that uses $O(n/\log n)$ processors and takes $O(\log n)$ time, neglecting communications costs.

2.2 PARALLEL DIRECT METHODS FOR GENERAL LINEAR EQUATIONS

Let A be a square matrix of size $n \times n$. Most direct methods for solving a system of linear equations of the form $Ax = b$ proceed by applying a set of simple transformations on both sides of the equation until the matrix A becomes triangular, and then solve the resulting system of equations by back substitution. These transformations consist of successively left–multiplying the matrix A by a sequence of matrices $M^{(1)}, \ldots, M^{(K)}$. The matrices $M^{(i)}$ are chosen with two objectives in mind: (a) multiplication of an arbitrary matrix by $M^{(i)}$ should have low computational requirements, and (b) the product $M^{(K)}M^{(K-1)} \cdots M^{(1)}A$ should be triangular. There are several ways of accomplishing this; two such methods, together with their parallel implementations, are discussed in this section.

Notice that the actual solution of the system $Ax = b$ is only a small step further from what was described above: in particular, while computing $M^{(K)} \cdots M^{(1)}A$, we may also compute the product $M^{(K)} \cdots M^{(1)}b$. Then, as long as each $M^{(i)}$ is invertible, the original system $Ax = b$ is equivalent to

$$(M^{(K)} \cdots M^{(1)}A)x = (M^{(K)} \cdots M^{(1)}b). \tag{2.1}$$

By construction, $M^{(K)} \cdots M^{(1)} A$ is triangular and the system (2.1) can be solved using back substitution in $O(n)$ time with n processors (Section 2.1).

If the inverse of A is desired, a similar procedure will do. While computing $M^{(K)} \cdots M^{(1)} A$, we also compute the product $M^{(K)} \cdots M^{(1)}$. Then, as long as each $M^{(i)}$ is invertible, A^{-1} can be computed using the equation

$$A^{-1} = \left(M^{(K)} \cdots M^{(1)} A\right)^{-1} \left(M^{(K)} \cdots M^{(1)}\right).$$

Since $M^{(K)} \cdots M^{(1)} A$ is triangular, it can be inverted using back substitution in $O(n)$ time using n^2 processors, or in $O(n^2)$ time using n processors. For these reasons, we will limit our further discussion to the task of triangularizing the matrix A.

2.2.1 Gaussian Elimination

Gaussian elimination is the classical procedure for solving linear equations whereby each variable, say the ith variable x_i, is expressed as a function of the variables $x_{i+1}, \ldots, x_n$ and is eliminated from the system. After $n-1$ such steps, we are left with an equation in the single variable x_n, which is easily solved.

We now give a recursive definition of the algorithm. Let $C^{(0)} = A$ and $C^{(i)} = M^{(i)} \cdots M^{(1)} A$. Suppose that $C^{(i-1)}$ has been already computed for some i, $1 \leq i \leq n-1$, and has the property that all subdiagonal entries of the jth column are zero for every j smaller than i. Equation (2.2) shows the structure of $C^{(i-1)}$, where an asterisk represents a generically nonzero entry:

$$C^{(i-1)} = \begin{bmatrix} * & \cdots & \cdots & \cdots & \cdots & * \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & \ddots & & & \\ & & 0 & * & \cdots & * \\ \vdots & & \vdots & \vdots & & \\ 0 & \cdots & 0 & * & \cdots & * \end{bmatrix} \begin{matrix} \\ \\ \\ i \\ \\ \\ \end{matrix} \qquad (2.2)$$

(Notice that $C^{(0)}$ satisfies the above requirement, vacuously.) We now show how to determine a matrix $M^{(i)}$ so that $C^{(i)} = M^{(i)} C^{(i-1)}$ also has the desired property.

Let us assume for now that $C_{ii}^{(i-1)} \neq 0$. We let $M^{(i)} = I - N^{(i)}$, where $N^{(i)}$ is a matrix with the following structure:

$$N^{(i)} = \begin{bmatrix} 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & & & & & \vdots \\ \vdots & \ddots & \ddots & & & & \\ & & 0 & 0 & \cdots & \cdots & 0 \\ & & \vdots & * & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & * & 0 & \cdots & 0 \end{bmatrix} \begin{matrix} \\ \\ \\ i \\ \\ \\ \\ \end{matrix}$$
$$\qquad\qquad\qquad\qquad\qquad\qquad\quad i$$

In particular, the nonzero entries of $N^{(i)}$ are given by $N_{ji}^{(i)} = C_{ji}^{(i-1)}/C_{ii}^{(i-1)}$, $j > i$. We verify that $C^{(i)} = \big(I - N^{(i)}\big)C^{(i-1)}$ has the desired property. Since the first $i-1$ columns of $N^{(i)}$ are zero, it is seen that the first $i-1$ columns of $C^{(i)}$ are the same as the corresponding columns of $C^{(i-1)}$. Consider now a subdiagonal entry of $C^{(i)}$ in the ith column, that is, an entry of the form $C_{ji}^{(i)}$, with $j > i$. Then,

$$C_{ji}^{(i)} = C_{ji}^{(i-1)} - \frac{C_{ji}^{(i-1)}}{C_{ii}^{(i-1)}} C_{ii}^{(i-1)} = 0.$$

We have thus completed an inductive proof that, with $M^{(i)}$ chosen as above, the matrices $C^{(i)}$ will be of the form of Eq. (2.2). In particular, $C^{(n-1)}$ is upper triangular, which was our stated goal.

Let us also note for future reference that the first i rows of $C^{(i)}$ and $C^{(i-1)}$ are equal and that for $j > i$, $k > i$, we have

$$C_{jk}^{(i)} = C_{jk}^{(i-1)} - \sum_{\ell=1}^{n} N_{j\ell}^{(i)} C_{\ell k}^{(i-1)} = C_{jk}^{(i-1)} - N_{ji}^{(i)} C_{ik}^{(i-1)} = C_{jk}^{(i-1)} - \frac{C_{ji}^{(i-1)}}{C_{ii}^{(i-1)}} C_{ik}^{(i-1)}. \quad (2.3)$$

The above method, called *Gaussian elimination without pivoting*, fails when $C_{ii}^{(i-1)}$ is zero. Even if $C_{ii}^{(i-1)}$ is nonzero but has a small magnitude, $M^{(i)}$ will have some very large entries and numerical problems are expected to arise. For this reason, Gaussian elimination without pivoting is used primarily when A is a symmetric matrix. In that case, it is known that $C_{ii}^{(i-1)}$ is never zero, nor do numerical problems arise, provided that A is invertible [GoV83].

For nonsymmetric problems, one may have to apply some kind of *pivoting*, that is, interchange two rows or two columns of $C^{(i-1)}$ so that the ith diagonal entry is nonzero and has, preferably, a large magnitude. The most common method, called *row pivoting*, works as follows: having computed $C^{(i-1)}$, find some $j^* \geq i$ such that

$$|C_{j^*i}^{(i-1)}| = \max_{j \geq i} |C_{ji}^{(i-1)}|. \quad (2.4)$$

Then, interchange rows i and j^* and proceed to compute $C^{(i)}$ as before.

The interchange of the ith and jth row of a matrix is equivalent to multiplication of that matrix, from the left, by the permutation matrix P^{ij}. (The entries of P^{ij} that are not shown are all zero.)

$$P^{ij} = \begin{bmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & - & 1 & - & & \\ & & & | & 1 & | & & & \\ & & & | & \ddots & | & & & \\ & & & | & 1 & | & & & \\ & & & 1 & - & 0 & - & & \\ & & & | & & | & 1 & & \\ & & & | & & | & & \ddots & \\ & & & | & & | & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ i \\ \\ \\ \\ j \\ \\ \\ \\ \end{matrix}$$
$$\qquad\qquad i \qquad\qquad j$$

In the context of the solution of linear systems, the right–hand side of the equation should be also multiplied by P^{ij}; this corresponds to interchanging the ith and jth entries of the vector in the right–hand side of the equation.

Gaussian elimination with row pivoting may suffer from poor numerical stability, but has been found to work well in practice [GoV83]. In particular, the maximum in Eq. (2.4) is always positive, except if the matrix A is singular, and therefore the method never breaks down unnecessarily (Exercise 2.1).

Let us point out here that the odd–even reduction algorithm for tridiagonal systems, presented in Subsection 2.1.2, can be viewed as a special case of Gaussian elimination, because odd–even reduction successively eliminates certain variables by expressing them as functions of the remaining variables. For general systems of equations, the variables can be eliminated only one at a time. However, the special structure of tridiagonal systems allows us to eliminate about half of the variables in a single step. Notice that in odd–even reduction, the variables are not eliminated in the usual order. For this reason, odd–even reduction is actually equivalent to Gaussian elimination performed on a matrix PAP', where P is an appropriate permutation matrix. In particular, P is such that it takes the odd–numbered rows and puts them on top; accordingly, the odd–numbered variables are the first to be eliminated. Notice also that odd–even reduction eliminates the variables according to a predetermined order. For this reason, it is equivalent to Gaussian elimination without pivoting. The case where odd–even reduction fails, due to an attempt to divide by zero, corresponds to the situation where pivoting is necessary. However, pivoting is undesirable in the solution of tridiagonal systems because it destroys the special structure of the system.

Parallel Implementations

We turn to the parallel implementation of the above methods. We first consider the case of no pivoting and do a preliminary analysis that ignores the communication penalty. By using the special structure of the matrix $M^{(i)}$, it follows that each entry of $C^{(i)}$ can

be computed with just one multiplication, one division, and one subtraction [see Eq. (2.3)]. By using n^2 processors, this can be done for all entries of $C^{(i)}$ simultaneously, in three time units. Thus, the total parallel time to compute $C^{(n-1)}$ is approximately $3n$, using n^2 processors. With fewer processors, say n, the required time is $O(n^2)$. To determine the efficiency of parallel Gaussian elimination, we compare it against the $T^*(n) = \Theta(n^3)$ benchmark; although there exist sequential algorithms that need less than $O(n^3)$ time units, every practical algorithm, including the serial implementation of Gaussian elimination, needs $\Theta(n^3)$ time. We then see that the efficiency of parallel Gaussian elimination is $\Theta(1)$, for the cases where n^2 or n processors are used.

We now consider particular interconnection topologies and verify that the associated communication penalty is not excessive. In particular, Gaussian elimination without pivoting can be implemented on a square mesh of n^2 processors, with the total execution time remaining of the order of n. We associate each processor with a particular entry of the matrices being manipulated and the resulting movement of data is illustrated in Fig. 2.2.1. In order to keep the time requirements as small as $O(n)$, it is essential that messages communicated are appropriately pipelined, and that computations and communications are interleaved, meaning that the computations of stage $i + 1$ start before all messages sent during the ith stage are received; the details are left as an exercise (Exercise 2.2). Without such interleaving, the time requirements of the algorithm are $\Omega(n^2)$ [Saa86]. With the above mentioned interleaving, the time spent for communications is of the same order of magnitude as the time spent in arithmetic computations; in particular, if the time required for a single communication is substantially larger than the time required for a single computation, then the communication penalty can be significant and should be alleviated by using a smaller number of processors.

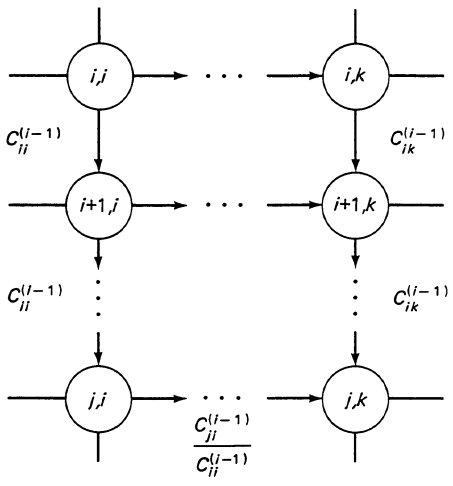


Figure 2.2.1 The movement of data in the execution of Gaussian elimination without pivoting in a mesh-connected architecture with n^2 processors. Suppose that the entries of $C^{(i-1)}$ have been computed. The value of $C_{ii}^{(i-1)}$ is propagated downward. Upon reception of this value, each processor (j, i), $j > i$, computes the ratio $C_{ji}^{(i-1)}/C_{ii}^{(i-1)}$ and transmits it to the processors to its right. In the meantime, the value of $C_{ik}^{(i-1)}$, for $k > i$, is propagated downward. Each processor (j, k), with $j > i$ and $k > i$, eventually receives $C_{ji}^{(i-1)}/C_{ii}^{(i-1)}$ and $C_{ik}^{(i-1)}$ and computes $C_{jk}^{(i)}$ according to Eq. (2.3). It is seen that a typical phase of the algorithm needs $\Theta(n)$ time for communication. Thus, if a new phase starts only after the previous one is completed, the algorithm needs $\Theta(n^2)$ time. On the other hand, by appropriately interleaving different phases, the execution time is reduced to $O(n)$.

On a hypercube architecture with $\Theta(n^2)$ processors, Gaussian elimination can be executed in $O(n)$ time because a mesh can be imbedded in a hypercube. Finally, a linear array of n processors can simulate an $n \times n$ mesh with a $O(n)$ reduction in speed; it follows that parallel Gaussian elimination can be executed in a linear array of n processors in $O(n^2)$ time.

The computational requirements of Gaussian elimination with pivoting are similar, except that at the beginning of a typical stage, we need to perform a comparison of $O(n)$ numbers and pick the largest [see Eq. (2.4)]. Let us neglect any communication costs for the time being. Under reasonable models of parallel computation in which a processor can compare two (but not more than two) numbers in unit time, we need $\Omega(\log n)$ time to compare n numbers no matter how many processors are available. (See e.g., Prop. 2.1 in Subsection 1.2.2.) For this reason, Gaussian elimination with pivoting takes $\Omega(n \log n)$ time when n^2 processors are available; the corresponding efficiency is $O(1/\log n)$, which decreases to zero as n increases. On the other hand, an $O(n \log n)$ parallelization is possible using only $O(n^2/\log n)$ processors, leading to $\Theta(1)$ efficiency. (See Prop. 2.4 in Subsection 1.2.2 and Exercise 2.4.) If n processors are available, then each stage takes $O(n)$ time, leading to a total execution time of $O(n^2)$. The efficiency is again $\Theta(1)$, similarly with Gaussian elimination without pivoting.

Pivoting is particularly undesirable when implementation in particular architectures is considered. For example, with an n^2–processor mesh–connected architecture, there is no implementation with $O(n \log n)$ running time (Exercise 2.3). On the other hand, $O(n \log n)$ time implementations are possible with a hypercube with $O(n^2)$ or even $O(n^2/\log n)$ processors (Exercise 2.4). Finally, if a linear array of n processors is used, with each processor associated with a particular column of the matrices being manipulated, then a row interchange does not lead to any data movement across processors, and the execution time remains $O(n^2)$, much as in Gaussian elimination without pivoting [ISS86].

Recall that the solution of a linear system of equations $Ax = b$ has a last phase during which a triangular system is solved, involving the upper triangular matrix produced at the last stage of the Gaussian elimination algorithm. We assume that back substitution will be used here. In a serial environment, back substitution needs $O(n^2)$ time which is negligible compared to the $O(n^3)$ time needed by Gaussian elimination. However, in a parallel environment, and if n^2 processors are used, the time devoted to back substitution is $O(n)$ (Subsection 2.1.1) and is comparable to the time needed for Gaussian elimination; thus, efficiency of implementation of back substitution becomes important. A related point concerns the case where a linear array of n processors is used and row pivoting is employed. If we employ an implementation of Gaussian elimination where the ith processor is associated to the ith column (as opposed to the ith row), we should also employ an implementation of back substitution where each processor knows a column of the triangular matrix produced by the Gaussian elimination algorithm. This is to avoid excessive movement of data between the Gaussian elimination phase and the back substitution phase.

The discussion in this subsection is summarized in Table 2.1.

TABLE 2.1 Bounds on the timing of Gaussian elimination for several architectures. The upper bounds are the same as those obtained for the same number of processors, if communication is assumed instantaneous.

Number of Processors	Architecture	Without Pivoting	With Pivoting
n^2	Hypercube	$O(n)$	$O(n \log n)$
n^2	Mesh	$O(n)$	$\Omega(n^{4/3})$
$n^2/\log n$	Hypercube	$O(n \log n)$	$O(n \log n)$
n	Hypercube	$O(n^2)$	$O(n^2)$
n	Linear Array	$O(n^2)$	$O(n^2)$

2.2.2 Triangularization Using Givens Rotations

We describe here an alternative method for triangularizing a square matrix. An important difference from Gaussian elimination is that the matrices $M^{(i)}$ are chosen to be orthogonal, that is, they have the property

$$\|M^{(i)}x\|_2 = \|x\|_2, \qquad \forall x \in \Re^n, \tag{2.5}$$

where $\|\cdot\|_2$ is the Euclidean norm. An equivalent definition of orthogonality is to require that $(M^{(i)})'M^{(i)} = I$. Orthogonal transformations are desirable in numerical analysis [GoV83] because they do not amplify the magnitude of past numerical errors.

Let C be a given matrix and suppose that its ith and jth rows have the property that there exists some $k \geq 1$ such that

$$C_{i\ell} = C_{j\ell} = 0, \qquad \forall \ell < k, \tag{2.6}$$

$$C_{jk} \neq 0, \tag{2.7}$$

that is, C has the structure

$$C = \begin{bmatrix} & * & & & * & \\ 0 & \cdots & 0 & * & \cdots & * \\ & & & & & \\ 0 & \cdots & 0 & * & \cdots & * \\ & * & & \vdots & * & \\ & & & k & & \end{bmatrix} \begin{matrix} \\ i \\ \\ j \\ \\ \end{matrix}$$

where the entries that are not shown are generically nonzero. Consider now a matrix M of the form

$$M = \begin{bmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & c & & & s & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & s & & & -c & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ i \\ \\ \\ \\ j \\ \\ \\ \end{matrix} \tag{2.8}$$

$$\begin{matrix} & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & i & & & j & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ \end{matrix}$$

where all entries not shown are zero, and

$$c = \frac{C_{ik}}{\left(C_{ik}^2 + C_{jk}^2\right)^{1/2}}, \tag{2.9}$$

$$s = \frac{C_{jk}}{\left(C_{ik}^2 + C_{jk}^2\right)^{1/2}}. \tag{2.10}$$

Such a matrix M is called a Givens rotation. It is orthogonal (Exercise 2.5), and has the following properties:

(a) All rows of MC, other than the ith and the jth row, are the same as the corresponding rows of C.
(b) $[MC]_{i\ell} = [MC]_{j\ell} = 0$ for all ℓ such that $\ell < k$.
(c) $[MC]_{jk} = 0$.

Properties (a) and (b) are straightforward consequences of the structure of C and M. For property (c), notice that

$$[MC]_{jk} = \sum_{\ell=1}^{n} M_{j\ell} C_{\ell k} = \sum_{\ell=i,j} M_{j\ell} C_{\ell k} = M_{ji} C_{ik} + M_{jj} C_{jk} = s C_{ik} - c C_{jk} = 0.$$

We can use such Givens rotations to set the subdiagonal entries of a given matrix A to zero, one at a time, as follows. Suppose that, after a number of stages, we have obtained a matrix C. We then find some i, j, and k such that $k < j$ (that is, the jkth entry is below the diagonal) and such that the properties of Eqs. (2.6) and (2.7) are satisfied. We then choose M as described above. It follows that the jkth entry of MC is equal to zero; the rows, other than i and j, are not disturbed and the zeros in the first $k-1$ entries of rows i and j are not destroyed. Since each such stage annihilates one more entry, this process must eventually stop. This happens if there exist no i, j, and k such

that $k < j$ and such that Eqs. (2.6) and (2.7) hold. It is easily seen that this implies that we have obtained an upper triangular matrix.

We now discuss the parallel implementation of such an algorithm. Notice that a Givens rotation M affects only two rows. If two successive Givens rotations were to affect disjoint sets of rows, then these rotations could be applied simultaneously without changing the result. In fact, when n rows are available, up to $\lfloor n/2 \rfloor$ rotations can be applied simultaneously, as long as they operate on disjoint pairs of rows, and the result is the same as if they were applied sequentially. Although it is not possible to apply exactly $\lfloor n/2 \rfloor$ rotations at each time stage, one can come fairly close. We need here a systematic way of deciding which entries to annihilate at each stage. This is accomplished by means of a *schedule* that consists of two functions, $T(j,k)$ and $S(j,k)$, defined for $k < j$. The interpretation of these functions is the following: the jkth entry is annihilated at stage $T(j,k)$ by a Givens rotation operating on rows j and $S(j,k)$.

Any schedule has to satisfy two requirements:

(a) Concurrent rotations operate on different rows; mathematically, if $T(j,k) = T(j',k')$ and $(j,k) \neq (j',k')$, then the sets $\{j, S(j,k)\}$ and $\{j', S(j',k')\}$ are disjoint.

(b) If $T(j,k) = t$ and $S(j,k) = i$, then $T(j,\ell) < t$ and $T(i,\ell) < t$ for all $\ell < k$; this is to guarantee that Eq. (2.6) is satisfied at the time that the jkth entry is annihilated.

One particular schedule, illustrated in Fig. 2.2.2, is given by

$$T(j,k) = n - j + 2k - 1, \tag{2.11}$$

$$S(j,k) = j - 1. \tag{2.12}$$

We verify that this schedule has the desired properties. Suppose that $T(j,k) = T(j',k')$. Equation (2.11) yields $-j + 2k = -j' + 2k'$. If $j = j'$, then $k = k'$. If $j \neq j'$, then $j - j'$ is even and, in particular, $|j - j'| \geq 2$. Therefore, the sets $\{j, j-1\}$ and $\{j', j'-1\}$ are disjoint, which implies that the sets $\{j, S(j,k)\}$ and $\{j', S(j',k')\}$ are also disjoint, and property (a) is satisfied. For the second property, Eq. (2.11) shows that $T\big(S(j,k),\ell\big) = T(j-1,\ell) = T(j,\ell) + 1 = n - j + 2\ell < n - j + 2k - 1 = T(j,k)$ for $\ell < k$.

We now estimate the number of stages required by the schedule of Eqs. (2.11) and (2.12). This is done by maximizing $T(j,k)$ over all j and k corresponding to subdiagonal entries. A little thought shows that the maximum is attained for $j = n$ and $k = n-1$ and is equal to $2n - 3$. Thus, a total of $2n - 3$ parallel stages are sufficient. Concerning the computational requirements of each stage, ignoring communication costs, each Givens rotation can be performed using n processors in $O(1)$ time. This is because only $O(n)$ entries (two rows) are affected by each rotation, and the new value of each entry can be computed with a constant number of operations. Since we have up to $n/2$ simultaneous rotations at each stage, it follows that the computations at each stage can be performed in $O(1)$ time using n^2 processors. The total count for the entire algorithm is $O(n)$ time using n^2 processors. With n processors, the time requirement becomes $O(n^2)$. For either

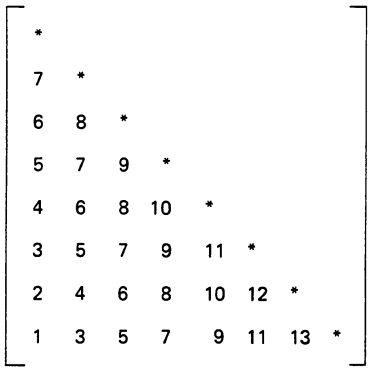


Figure 2.2.2 The schedule of Eq. (2.11) for the case $n = 8$. The numbers indicate the stage at which the corresponding entry is annihilated.

case the efficiency is $\Theta(1)$ when compared with the benchmark $T^*(n) = \Theta(n^3)$ for serial algorithms.

We notice that the time requirements of the Givens rotation method are the same as for Gaussian elimination without pivoting when n^2 processors are used and communication costs are ignored; they are also the same as for Gaussian elimination with row pivoting when n processors are used. A more detailed analysis shows that Gaussian elimination is faster, by a small constant factor, when the same number of processors is used. This may be compensated by the better numerical properties of the Givens rotation method [GoV83].

As far as parallel implementation in special architectures is concerned, the algorithm can be implemented in a mesh of n^2 processors in time $O(n)$ [BBK84] and this implies that it can also be efficiently implemented in a hypercube of $O(n^2)$ processors [in $O(n)$ time] or a linear array of n processors [in $O(n^2)$ time]. The implementation in a mesh involves pipelining, and interleaving of computation and communication, and results in a very regular pattern for the movement of data. However, the details are somewhat tedious and the reader is referred to [BBK84].

EXERCISES

2.1. Show that if Gaussian elimination with row pivoting is used but the maximum in Eq. (2.4) is zero at some intermediate stage of the algorithm, then the matrix A is singular.

2.2. Show that Gaussian elimination without pivoting can be implemented in a mesh–connected architecture of n^2 processors in $O(n)$ time. *Hint:* See Fig. 2.2.1.

2.3. Consider a two–dimensional mesh–connected architecture. Assume that a processor cannot compare more than two numbers in unit time and that communication between neighboring processors takes unit time.

(a) Show that the problem of computing the maximum of n numbers requires $\Omega(n^{1/3})$ time units regardless of the number of processors and even if each processor knows the values of all numbers to be compared.

(b) Show that Gaussian elimination with row pivoting needs $\Omega(n^{4/3})$ time.

2.4. **[Cap87]** Show that Gaussian elimination with pivoting can be executed in a hypercube with $O(n^2/\log n)$ processors in time $O(n \log n)$. *Hint:* Arrange the processors as a $n \times (n/\log n)$ mesh. Assign to each processor $\Theta(\log n)$ entries of the same column. Show that the entries in each column can be compared in $O(\log n)$ time. Also, show that a row interchange, as well as any other communication required in a typical stage of the algorithm, takes $O(\log n)$ time.

2.5. Verify that the matrix M defined by Eqs. (2.8) to (2.10) is orthogonal.

2.6. Find a schedule for the algorithm based on Givens rotations for the case of an 8×8 matrix that needs fewer than 13 parallel stages. (Compare with Fig. 2.2.2.)

2.3 A FAST DIRECT MATRIX INVERSION ALGORITHM

All of the methods of Section 2.2 require time at least proportional to n for the solution of a system of n linear equations, and this raises the question whether a substantially faster algorithm is possible, assuming that an unlimited number of processors is available. While it was believed for some time that this was not possible, the algorithm in this section, due to Csanky [Csa76], produces the inverse of a square matrix in $O(\log^2 n)$ time. This algorithm is only of theoretical interest because it is prone to numerical problems and because it uses an excessive number (n^4) of processors. It is an open question whether there exist parallel matrix inversion algorithms whose time requirements are smaller than $O(\log^2 n)$. In fact, this problem is open even for the case of triangular matrices.

Given an $n \times n$ nonsingular matrix A, we consider the characteristic polynomial ϕ of A, defined by

$$\phi(\lambda) = \det(\lambda I - A) = \prod_{i=1}^{n} (\lambda - \lambda_i), \tag{3.1}$$

where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of A. Let $c_1, \ldots, c_n$ be the coefficients of the characteristic polynomial, that is,

$$\phi(\lambda) = \lambda^n + c_1 \lambda^{n-1} + \cdots + c_{n-1}\lambda + c_n. \tag{3.2}$$

By comparing Eqs. (3.1) and (3.2), we see that $c_n = (-1)^n \prod_{i=1}^{n} \lambda_i$; in particular, A is invertible if and only if $c_n \neq 0$, which we assume. The computation of A^{-1} uses the Cayley–Hamilton theorem (Prop. A.18 in Appendix A), which states that $\phi(A) = A^n + c_1 A^{n-1} + \cdots + c_{n-1}A + c_n I = 0$. Therefore, A^{-1} is given by

$$A^{-1} = -\frac{1}{c_n}\left(A^{n-1} + c_1 A^{n-2} + \cdots + c_{n-1} I\right). \tag{3.3}$$

By using the discussion in Subsection 1.2.3, the matrices $A^2, \ldots, A^{n-1}$ can be all generated in time $O(\log^2 n)$ using n^4 processors. Therefore, it only remains to find a "fast" method for computing the coefficients of the characteristic polynomial.

We define the trace $\mathrm{tr}(A)$ of a matrix A as the sum of its diagonal entries. The trace of a matrix is also equal to the sum of the eigenvalues (Prop. A.22 in Appendix A). We define $s_k = \sum_{i=1}^{n} \lambda_i^k$, where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of A. Since $\lambda_1^k, \ldots, \lambda_n^k$ are the eigenvalues of A^k [Prop. A.17(d) in Appendix A], it follows that $s_k = \mathrm{tr}(A^k)$. In particular, the coefficients $s_1, \ldots, s_n$ can be computed by summing the diagonal entries of the matrices A^k. We now use a classical method, known as Leverrier's method [LeV40], that allows us to compute the coefficients of the characteristic polynomial in terms of $s_1, \ldots, s_n$.

Proposition 3.1. The coefficients $c_1, \ldots, c_n$ and the coefficients $s_1, \ldots, s_n$ satisfy the following system of equations:

$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ s_1 & 2 & 0 & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ s_{k-1} & \cdots & s_1 & k & 0 & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 \\ s_{n-1} & \cdots & s_{k-1} & \cdots & s_1 & n \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_n \end{bmatrix} = - \begin{bmatrix} s_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ s_n \end{bmatrix}. \tag{3.4}$$

Proof. The derivative of the characteristic polynomial is given by

$$\frac{d\phi}{d\lambda}(\lambda) = n\lambda^{n-1} + c_1(n-1)\lambda^{n-2} + \cdots + c_{n-1}. \tag{3.5}$$

An alternative expression for $d\phi/d\lambda$ is obtained by differentiating both sides of Eq. (3.1). This yields

$$\frac{d\phi}{d\lambda}(\lambda) = \frac{d}{d\lambda}\left[\prod_{i=1}^{n}(\lambda - \lambda_i)\right] = \sum_{i=1}^{n}\prod_{j\neq i}(\lambda - \lambda_i) = \sum_{i=1}^{n}\frac{\phi(\lambda)}{\lambda - \lambda_i}. \tag{3.6}$$

We now use the series expansion

$$\frac{1}{\lambda - \lambda_i} = \frac{1}{\lambda(1 - \lambda_i/\lambda)} = \frac{1}{\lambda}\left(1 + \frac{\lambda_i}{\lambda} + \frac{\lambda_i^2}{\lambda^2} + \cdots\right), \tag{3.7}$$

which is valid for $|\lambda| > |\lambda_i|$. We use Eq. (3.7) in Eq. (3.6) to obtain

$$\begin{aligned} \frac{d\phi}{d\lambda}(\lambda) &= \frac{\phi(\lambda)}{\lambda}\sum_{i=1}^{n}\left(1 + \frac{\lambda_i}{\lambda} + \frac{\lambda_i^2}{\lambda^2} + \cdots\right) \\ &= \left(\lambda^n + c_1\lambda^{n-1} + \cdots + c_n\right)\left(\frac{n}{\lambda} + \frac{s_1}{\lambda^2} + \frac{s_2}{\lambda^3} + \cdots\right), \end{aligned} \tag{3.8}$$

where the last equality followed by changing the order of the summation and using the definition of s_k. The right–hand side of Eq. (3.8) is equal to the right–hand side of Eq. (3.5) for all values of λ satisfying $|\lambda| > |\lambda_i|$ and for all i. It follows that the coefficients of each power of λ must be the same in both expressions. By comparing the coefficients of λ^{n-k-1} for $k = 1, \ldots, n$, we obtain Eq. (3.4). **Q.E.D.**

Notice that Eq. (3.4) is a lower triangular system of equations. Using the results of Section 2.1, we can solve for $c_1, \ldots, c_n$ in time $O(\log^2 n)$ using n^3 processors. We can now summarize the algorithm:

1. Compute A^k for $k = 2, \ldots, n$.
2. Compute s_k for $k = 1, \ldots, n$.
3. Solve the system (3.4) for $c_1, \ldots, c_n$.
4. Evaluate A^{-1} using Eq. (3.3).

Each one of these four steps can be executed in $O(\log^2 n)$ time using n^4 processors, and, therefore, this estimate is valid for the overall algorithm as well.

EXERCISES

3.1. (Cholesky factorization [Luo87]) Any symmetric positive definite matrix A can be expressed in the form $A = L'DL$, where L is a lower triangular matrix, and D is a diagonal matrix with positive diagonal entries. Devise a parallel algorithm that computes L and D in time $O(\log^3 n)$, where n is the size of the matrix A. *Hint:* Compute a square matrix X, of approximately half the size of A, such that

$$\begin{bmatrix} I & 0 \\ X & I \end{bmatrix} A \begin{bmatrix} I & X' \\ 0 & I \end{bmatrix} = \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix},$$

where B_1 and B_2 are some matrices, and proceed recursively.

2.4 CLASSICAL ITERATIVE METHODS FOR SYSTEMS OF LINEAR EQUATIONS

We present here a few classical iterative methods for solving linear equations. Such methods are widely used, especially for the solution of large problems such as those arising from the discretization of linear partial differential equations. For this reason, there is an extensive theory dealing with such methods, some of which is developed in Section 2.6.

Let A be an $n \times n$ matrix, let b be a vector in $\Re^n$, and consider the system of linear equations

$$Ax = b, \tag{4.1}$$

where x is an unknown vector to be determined. We assume that A is invertible, so that $Ax = b$ has a unique solution. We write the ith equation of the system $Ax = b$ as

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \tag{4.2}$$

where a_{ij} are the entries of A; also, x_j and b_i are the components of x and b, respectively. We assume that $a_{ii} \neq 0$ and solve for x_i to obtain

$$x_i = -\frac{1}{a_{ii}}\left[\sum_{j\neq i} a_{ij}x_j - b_i\right]. \tag{4.3}$$

If all components x_j, $j \neq i$, of the solution of $Ax = b$ are known, the remaining component x_i can be determined from Eq. (4.3). If instead some approximate estimates for the components x_j, $j \neq i$, are available, then we can use Eq. (4.3) to obtain an estimate of x_i. This can be done for each component of x simultaneously, leading to the following algorithm:

Jacobi algorithm. Starting with some initial vector $x(0) \in \Re^n$, evaluate $x(t)$, $t = 1, 2, \ldots$, using the iteration

$$x_i(t+1) = -\frac{1}{a_{ii}}\left[\sum_{j\neq i} a_{ij}x_j(t) - b_i\right]. \tag{4.4}$$

The Jacobi algorithm produces an infinite sequence $\{x(t)\}$ of elements of $\Re^n$. If this sequence converges to a limit x, then by taking the limit of both sides of Eq. (4.4) as t tends to infinity, we see that x satisfies Eq. (4.3) for each i, which is equivalent to x being a solution of $Ax = b$. Of course, it is possible that the algorithm diverges [$x(t)$ does not converge]; see Fig. 2.4.1. Conditions for convergence will be explored in Section 2.6.

In the above algorithm, each component of $x(t+1)$ was evaluated based on Eq. (4.3) and the estimate $x(t)$ of the solution. If this algorithm is executed on a serial computer, by the time that $x_i(t+1)$ is evaluated, we already have available some new estimates $x_j(t+1)$ for the components of x with index j smaller than i. It may be preferable to employ these new estimates of x_j, $j < i$, when updating x_i. This leads to the following algorithm:

Gauss–Seidel algorithm. Starting with some initial vector $x(0) \in \Re^n$, evaluate $x(t)$, $t = 1, 2, \ldots$, using the iteration

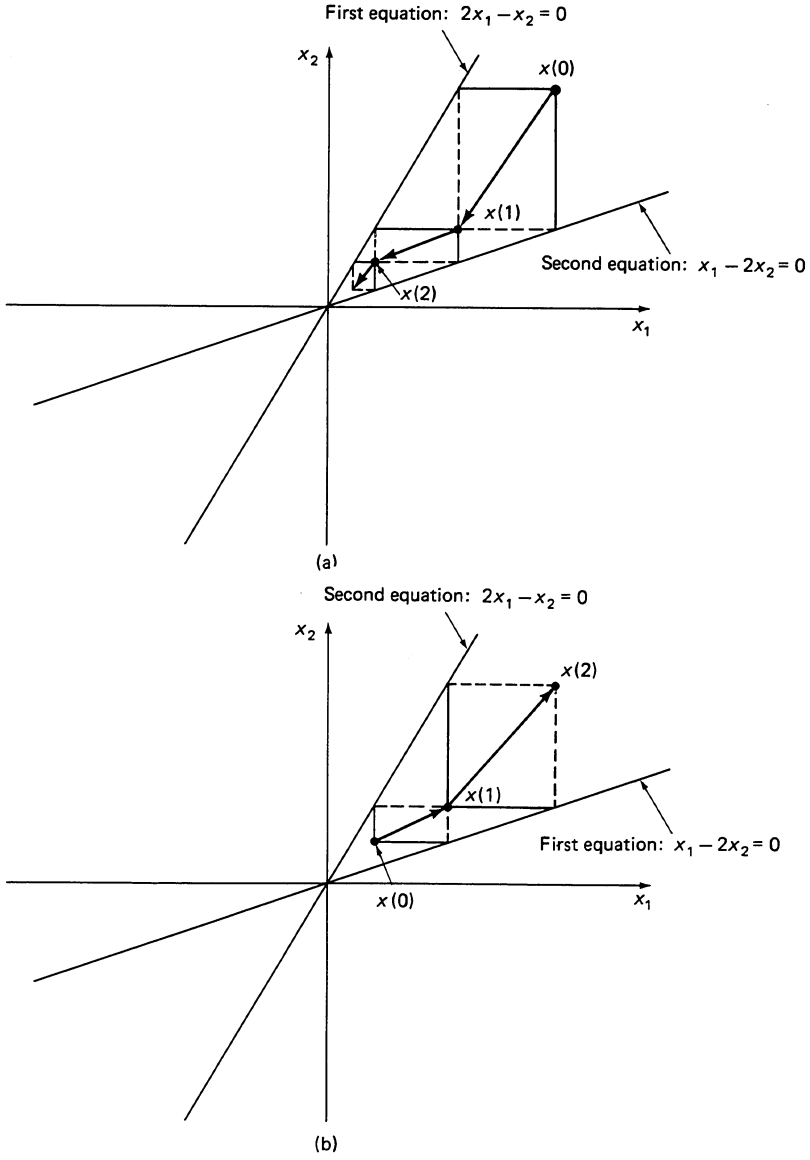


Figure 2.4.1 Illustration of the Jacobi algorithm for solving a system of linear equations. At each iteration, the ith equation is solved for the ith component with all other components fixed at their values at the start of the iteration. In (a), the Jacobi algorithm is applied to the system

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

and the iteration converges; in (b), the algorithm is applied to the equivalent system

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

in which the assignment of equations to components has been reversed, and the iteration diverges. In Section 2.6, it will be seen that convergence is enhanced if equations are assigned to components so that the diagonal elements a_{ii} are large (in absolute value) relative to the other coefficients a_{ij}, $i \neq j$.

$$x_i(t+1) = -\frac{1}{a_{ii}}\left[\sum_{j<i} a_{ij}x_j(t+1) + \sum_{j>i} a_{ij}x_j(t) - b_i\right]. \tag{4.5}$$

Figure 2.4.2 illustrates convergence and divergence of the algorithm. As discussed in Subsection 1.2.4, there are many different Gauss–Seidel algorithms for the same system of equations, depending on the particular order with which the variables are updated. In Eq. (4.5), we first update x_1, then x_2, etc. It is equally meaningful to start by updating x_n, then x_{n-1}, and proceed backwards, with x_1 being updated last. Any other order of updating is possible. Different orders of updating may produce substantially different results for the same system of equations.

A variation of the Jacobi and Gauss–Seidel methods is obtained if we use a nonzero scalar γ (called the *relaxation parameter*), and rewrite Eq. (4.3) in the equivalent form

$$x_i = (1-\gamma)x_i - \frac{\gamma}{a_{ii}}\left[\sum_{j\neq i} a_{ij}x_j - b_i\right], \tag{4.6}$$

thereby leading to the following algorithms:

Jacobi overrelaxation (JOR). Jacobi overrelaxation is similar to the Jacobi algorithm except that Eq. (4.4) is replaced by

$$x_i(t+1) = (1-\gamma)x_i(t) - \frac{\gamma}{a_{ii}}\left[\sum_{j\neq i}^{n} a_{ij}x_j(t) - b_i\right]. \tag{4.7}$$

In particular, if $0 < \gamma < 1$, the new value of x_i obtained from Eq. (4.7) is a convex combination of the old value of x_i and the new value of x_i that would have been obtained if the Jacobi iteration (4.4) was used. The next algorithm is a similar modification of the Gauss–Seidel algorithm.

Successive overrelaxation (SOR). Successive overrelaxation is the same as the Gauss–Seidel algorithm except that Eq. (4.5) is replaced by

$$x_i(t+1) = (1-\gamma)x_i(t) - \frac{\gamma}{a_{ii}}\left[\sum_{j<i} a_{ij}x_j(t+1) + \sum_{j>i} a_{ij}x_j(t) - b_i\right]. \tag{4.8}$$

Notice that the Jacobi and Gauss–Seidel algorithms are equivalent to the JOR and SOR algorithms, respectively, when $\gamma = 1$. The JOR and SOR algorithms are widely used because they often converge faster if γ is suitably chosen.

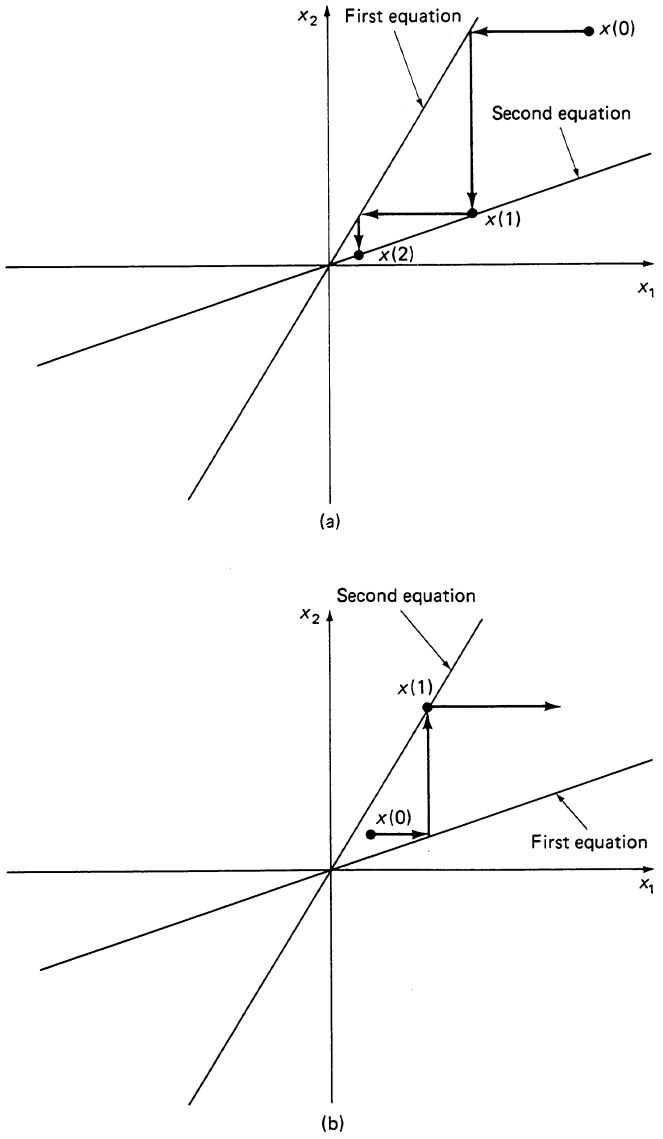


Figure 2.4.2 Illustration of convergence and divergence in the Gauss–Seidel algorithm for the same systems of equations as in Fig. 2.4.1.

Richardson's method. Our next iterative method, which is sometimes called *Richardson's method* [HaY81], is obtained by rewriting the equation $Ax = b$ in the form $x = x - \gamma(Ax - b)$. The method is described by

$$x(t+1) = x(t) - \gamma\Big[Ax(t) - b\Big], \tag{4.9}$$

where γ is a scalar relaxation parameter. A variant of Richardson's method is obtained if the iteration $x := x - \gamma(Ax - b)$ is executed in Gauss–Seidel fashion. This algorithm will be referred to as the RGS method and is described by the update equation

$$x_i(t+1) = x_i(t) - \gamma \left[\sum_{j<i} a_{ij} x_j(t+1) + \sum_{j \geq i} a_{ij} x_j(t) - b_i \right]. \tag{4.10}$$

A more general class of algorithms is obtained by using an invertible matrix B to transform the equation $Ax = b$ to the equivalent equation

$$x = x - B(Ax - b) \tag{4.11}$$

and then applying the iteration

$$x(t+1) = x(t) - B\Big[Ax(t) - b\Big]. \tag{4.12}$$

A Gauss–Seidel variant of this iteration is also possible.

The discussion following the presentation of the Jacobi algorithm applies to all of the other methods, and shows that the following is true.

Proposition 4.1. If the sequence $\{x(t)\}$, generated by any of the above presented algorithms converges, then it converges to a solution of $Ax = b$.

2.5 PARALLEL IMPLEMENTATION OF CLASSICAL ITERATIVE METHODS

In this section, we comment on the parallelization of the iterative methods introduced in Section 2.4. Most of the computation in such methods consists of matrix–vector multiplications and the relevant facts have already been covered in Subsection 1.3.6. We concentrate on the case of message–passing architectures because the issues tend to be somewhat simpler for the case of shared memory systems. We discuss the case where the matrices involved are dense, as well as the sparse case, which is typical in problems originating from partial differential equations (Subsection 2.5.1). Finally, in Subsection 2.5.2, we describe multigrid methods and comment on their parallelizability.

The Jacobi, JOR, and Richardson's algorithms for the solution of the system $Ax = b$ are straightforward to implement in parallel since each iteration involves a matrix–vector multiplication. Suppose that there are n available processors, and that the ith processor is responsible for computing $x_i(t)$ at each iteration t. (The case where the number of processors is smaller than the number of variables is qualitatively similar and will be discussed shortly.) Suppose that the ith processor knows the entries of the ith row of A. (This is the row storage method of Subsection 1.3.6.) To compute $x_i(t+1)$, processor

i has to know the values of $x_j(t)$ computed at the previous iteration by processors j for which $a_{ij} \neq 0$. If most of the entries of A are nonzero, it is easier to transmit x_j to all processors i, even if a_{ij} equals zero, because selective transmissions may introduce some unwarranted overhead. We are thus dealing with a multinode broadcast. As discussed in Subsection 1.3.4, the time to perform a multinode broadcast is $O(n)$ in a linear array or a mesh, and $O(n/\log n)$ in a hypercube. Given that each processor has to perform $\Theta(n)$ arithmetic operations at each stage (assuming that the matrix A has no special sparsity structure), the percentage of time spent on communications diminishes with n for a hypercube and remains constant for a linear array. In practice, this constant factor could be substantial and communication could dominate the execution time; in that case, there should be fewer processors, with more components assigned to each one, and the communication penalty can be made insignificant (Subsection 1.3.5).

In an alternative implementation, there are again n processors, with the ith processor in charge of the ith component x_i. However, we assume that the ith processor has access to the ith column of A, as opposed to the ith row of A. (This is the column storage method of Subsection 1.3.6.) Assuming that A is a fully dense matrix, the computation proceeds as follows. Each processor i evaluates $a_{ji}x_i$ for $j = 1, \ldots, n$. Then for each j, the quantities $a_{ji}x_i$ for $i = 1, \ldots, n$ are propagated to processor j, with partial sums formed along the way, which is a multinode accumulation. As discussed in Subsection 1.3.4, the time required for a multinode accumulation is equal to the time for a multinode broadcast, and we conclude that the communication requirements of the row and column storage methods are the same, for the dense case. Which of the two implementations is preferable may depend on fine details of a particular parallel computer.

We now consider the case where the matrix A is sparse. The sparsity structure of A determines a directed graph $G = (N, \mathcal{A})$, where $N = \{1, \ldots, n\}$ and the set of arcs $\mathcal{A}$ is the set $\{(i, j) \mid i \neq j \text{ and } a_{ji} \neq 0\}$ of all processor pairs (i, j) such that i needs to communicate to j. We recognize this as the dependency graph introduced in Subsection 1.2.4. Given a special architecture, efficient parallel implementations are obtained if the above dependency graph can be imbedded into the graph describing the interconnection topology. In that case, all communication takes place between neighboring processors and the communication penalty is minimal. An example will be shown in the next subsection. When such an imbedding cannot be found, the communication requirements of the row and column storage methods can be substantially different (see the discussion in Subsection 1.3.6), and this is an important difference from the dense case. There are also some alternative storage methods which are discussed in [McV87] and [FJL88].

We now consider the Gauss–Seidel, SOR, and RGS algorithms. These are not well suited for parallel implementation in general. To see this, consider the Gauss–Seidel algorithm and suppose that the matrix A is fully dense, that is, $a_{ij} \neq 0$ for all i and j. Then, for processor i to compute $x_i(t+1)$, the value of $x_j(t+1)$ is needed for every $j < i$. Hence, the algorithm is inherently sequential, because no two components of x can be updated simultaneously. This is quite unfortunate because SOR algorithms, with a suitable choice of γ, are often much faster [HaY81]. However, as discussed in Subsection 1.2.4, this difficulty may be often circumvented if the matrix A is sparse by employing a coloring scheme. This is always the case when the matrix A is obtained

from discretization of a partial differential equation, and an example will be discussed in Subsection 2.5.1.

In practice, the number p of processors is often substantially smaller than the number n of variables. In this case, several variables can be assigned to each processor. All of the preceding discussion applies with minor modifications, and the communication penalty will typically be reduced under these circumstances, as discussed in Subsection 1.3.5.

A final point of interest concerns termination of iterative algorithms. Typical termination criteria used in practice evaluate an expression such as $\|Ax(t) - b\|$, where $\|\cdot\|$ is some norm, and terminate the algorithm if its value is small enough. Such testing for termination does not introduce any significant overhead in the case where A is dense. For example, suppose that $\|\cdot\|$ is the maximum norm $\|\cdot\|_\infty$. At each iteration, every processor computes the value of $\max_i |[Ax]_i - b_i|$, where i ranges over the indices of the variables assigned to that processor. These values can then be compared using a spanning tree, with each processor propagating toward the root of the tree the largest of its own value and the values it has received. Thus, termination detection requires a single node accumulation and adds little to the communication penalty when compared to the multinode broadcast or multinode accumulation involved in each iteration.

The considerations concerning termination detection are different when A is sparse and the variables have been assigned to the processors so that each iteration only requires nearest neighbor communication. Here the time spent on communications needed for executing one iteration is proportional to the number of variables assigned to each processor, whereas the communication time needed for termination testing is proportional to the diameter of the interconnection network. Consequently, unless the diameter is comparable or smaller than the number of variables assigned to each processor, it is meaningful to test for termination only once in a while. Alternatively, testing for termination could take place while the main algorithm continues with subsequent iterations. In the latter case, the value of $Ax - b$ used for the termination decision will be outdated by the time that this decision is made, but this typically does not have any particularly adverse consequences.

2.5.1 An Example: Poisson's Equation

As a prototype of a linear partial differential equation, we consider Poisson's equation on the unit square:

$$\frac{\partial^2 f}{\partial x^2}(x,y) + \frac{\partial^2 f}{\partial y^2}(x,y) = g(x,y), \qquad (x,y) \in [0,1]^2, \tag{5.1}$$

where $g : [0,1]^2 \mapsto \Re$ is a known function. The objective is to find a function $f : [0,1]^2 \mapsto \Re$ that satisfies Poisson's equation and has prescribed values on the boundary of the unit square. In order to solve this equation numerically, we consider the values of f only on a finite grid of points in the unit square. Let N be an integer larger than 2 and let

$$f_{i,j} = f\left(\frac{i}{N}, \frac{j}{N}\right), \qquad 0 \le i, j \le N,$$

$$g_{i,j} = g\left(\frac{i}{N}, \frac{j}{N}\right), \qquad 0 < i, j < N.$$

Assuming that f is sufficiently smooth and that Δ is a small scalar, we can use a central difference approximation for the second derivative of f (see Prop. A.33 in Appendix A) to obtain

$$\frac{\partial^2 f}{\partial x^2}(x, y) \approx \frac{1}{\Delta^2}\Big[f(x + \Delta, y) - 2f(x, y) + f(x - \Delta, y)\Big].$$

We use a similar approximation for $\partial^2 f/\partial y^2$, we let (x, y) be one of the grid points in the interior of the unit square, and we let $\Delta = 1/N$ to obtain

$$f_{i,j} = \frac{1}{4}\Big(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1}\Big) - \frac{1}{4N^2} g_{i,j}, \qquad 0 < i, j < N. \tag{5.2}$$

This is a system of $(N-1)^2$ linear equations in $(N-1)^2$ unknowns, the unknowns being the values of f at the interior grid points. (Recall that the values of f on the boundary are given.) This system can be represented in the form $Ax = b$, where A is an $(N-1)^2 \times (N-1)^2$ matrix, x is a vector with the unknown values of $f_{i,j}$ at the interior grid points, and where b is a vector depending on $g_{i,j}$ and the known boundary values of f. Then, all of the methods of Section 2.4 become applicable. However, we do not need to write an explicit matrix representation in order to apply these methods; Eq. (5.2) already contains all the information needed. We notice that Eq. (5.2) expresses one variable in terms of the others, and we have one such equation for each variable. Thus, Eq. (5.2) has the same structure as Eq. (4.3) on which the iterative methods of the preceding section were based. Therefore, the updating equation of the JOR algorithm is

$$f_{i,j}(t+1) = (1-\gamma) f_{i,j}(t) + \frac{\gamma}{4}\Big[f_{i+1,j}(t) + f_{i-1,j}(t) + f_{i,j+1}(t) + f_{i,j-1}(t)\Big] - \frac{\gamma}{4N^2} g_{i,j}, \qquad 0 < i, j < N. \tag{5.3}$$

If the right-hand side of Eq. (5.3) involves a boundary point, then the given boundary value is used. That is, we are setting $f_{i,j}(t) = f_{i,j}$ whenever i or j is equal to 0 or N.

The parallel implementation of the JOR algorithm is straightforward. We assign a different processor to each interior grid point and notice that the processor responsible for updating $f_{i,j}$ can execute the iteration (5.3), provided that it knows the values of f at neighboring grid points, as computed at the previous iteration. Thus, the most natural parallel architecture is a mesh of $(N-1)^2$ processors such that neighboring processors correspond to neighboring grid points (see Fig. 2.5.1). At each stage of the algorithm, each processor transmits its most recently computed value of $f_{i,j}$ to its neighbors and then each processor uses the values received to update its own value according to Eq.

(5.3). In practice, the number of grid points is likely to exceed the number of processors, in which case it is meaningful to assign a block of adjacent grid points to each processor. Concurrency is reduced but the communication penalty is also reduced due to the area–perimeter effect discussed in Subsection 1.3.5.

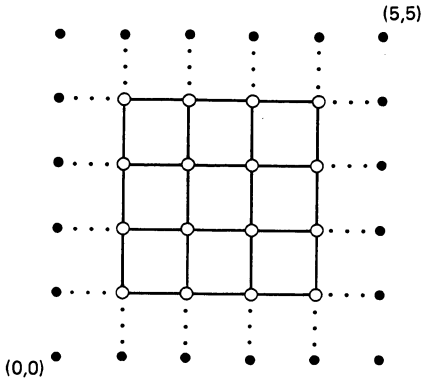


Figure 2.5.1 A 6×6 grid for the solution of Poisson's equation. A processor is assigned to each interior grid point and processors are connected as indicated. Furthermore, processors near the boundary need to know the boundary conditions for f at the points indicated by dashed lines. If the arcs are viewed as bidirectional, the graph of processors is also the dependency graph corresponding to the Jacobi and JOR algorithms.

We now turn to the parallel implementation of SOR. In the language of Subsection 1.2.4, this is the Gauss–Seidel algorithm based on the JOR iteration (5.3). The dependency graph of iteration (5.3) is shown in Fig. 2.5.1; it is identical to the dependency graph considered in Example 2.1 of Subsection 1.2.4, and the discussion in that subsection applies. In particular, this graph can be colored using two colors only. Furthermore, if each processor is assigned to two adjacent grid points, then a sweep (that is, an update of all components) takes the same time for either of the JOR and SOR algorithms. (In practice, each processor is often assigned several grid points and the time requirements of each iteration are again the same for JOR and SOR.) Generally, the SOR algorithm is preferred because it has a better rate of convergence [Var62].

Parallel implementations of SOR are also possible for more complicated discretizations of partial differential equations. For example, a so–called nine–point discretization gives rise to the dependency graph that was shown in Figure 1.2.12 of Subsection 1.2.4. According to Exercise 2.7 in that subsection, this dependency graph can be colored using four colors. It follows that if the number of grid points assigned to each processor is equal to 4 (or an integer multiple of 4), then the time for the execution of one iteration of SOR is the same as the time for one iteration of JOR.

We now recall from Subsection 1.3.4 that a mesh can be imbedded in a hypercube. It follows that the JOR and SOR algorithms for Poisson's equation can be implemented in a hypercube with communication taking place only between nearest neighbors. Thus, hypercubes are well suited for the numerical solution of partial differential equations.

2.5.2 Multigrid Methods

Multigrid methods are a special class of iterative algorithms for the numerical solution of partial differential equations. In these methods, a partial differential equation like Poisson's equation, is discretized for several choices of the grid spacings, and iterations

take place on several such grids. The rationale behind such methods is that a fine grid is required to obtain an accurate solution, but iterations on coarser grids typically converge with fewer iterations. An appropriate combination of iterations on fine and coarse grids leads to the fastest known algorithms for the solution of certain types of partial differential equations. We will not be concerned here with the analytical aspects of such methods; the reader is referred to [Hac85]. We shall concentrate instead on their structure and on the data dependencies involved, and we shall explore the potential for their parallelization. For simplicity, we restrict our discussion to two–dimensional grids although the discussion generalizes to higher dimensions.

Let $G_0 = \{(i,j) \mid 1 \leq i,j \leq N\}$, where N is assumed to be a power of 2. We view G_0 as the finest grid. We also define coarser grids, $G_1, \ldots, G_D$, where D is an integer smaller than $\log N$, by letting

$$G_d = \left\{(i,j) \in G_0 \mid i \text{ and } j \text{ are integer multiples of } 2^d\right\}, \qquad d = 1,2,\ldots,D,$$

(see Fig. 2.5.2). With each grid G_d and each $(i,j) \in G_d$, we associate a variable $x_{ij,d}$. A multigrid algorithm involves the following three types of computations:

(a) Relaxations on a given grid G_d. The next value of $x_{ij,d}$ is determined as a function of the values of its neighbors on the grid G_d. We assume that such relaxations take place only on a single grid at a time.

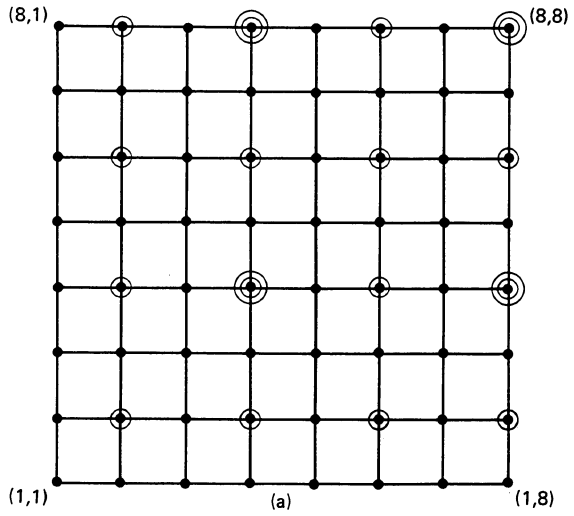
(b) Transfer to a finer grid (*interpolation*). The values of the finer grid variables $x_{ij,d-1}$ are computed in terms of the values of the coarser grid variables $x_{ij,d}$, according to some local rule. For example, $x_{ij,d-1}$ could be set equal to the average of the values of $x_{k\ell,d}$, where the average is taken over all $(k,\ell) \in G_d$ which are at a minimal distance from (i,j) (see Fig. 2.5.2).

(c) Transfer to a coarser grid (*projection*). The values of the coarser grid variables $x_{ij,d+1}$ are computed in terms of the values of the variables $x_{ij,d}$, according to some local rule. The simplest rule is to let $x_{ij,d+1} := x_{ij,d}$.

Suppose that we have N^2 processors available, and that the ijth processor is assigned the responsibility of computing the values of the variables $x_{ij,d}$ for every d such that $(i,j) \in G_d$. In order to execute the relaxation iterations on the finest grid efficiently, it is reasonable to use a two–dimensional mesh–connected array of processors. However, we notice that when relaxation iterations are executed on some coarser grid G_d, $d \neq 0$, then neighboring grid points on G_d correspond to distant processors on our mesh of processors and for this reason the mesh topology is unsuitable for multigrid methods.

Let us now consider a hypercube and suppose that a mesh of N^2 processors has been imbedded in a hypercube with N^2 nodes, using a reflected Gray code, as in Subsection 1.3.4. We recall certain key properties of this imbedding (see Fig. 2.5.3).

(a) For $j = 1,\ldots,N$, the jth "column" of the grid, which is the linear array $C_j = \{(i,j) \in G_0 \mid 1 \leq i \leq N\}$, is mapped to a smaller hypercube, with N nodes, contained in the original hypercube. The same is true for each "row" $R_i = \{(i,j) \in$



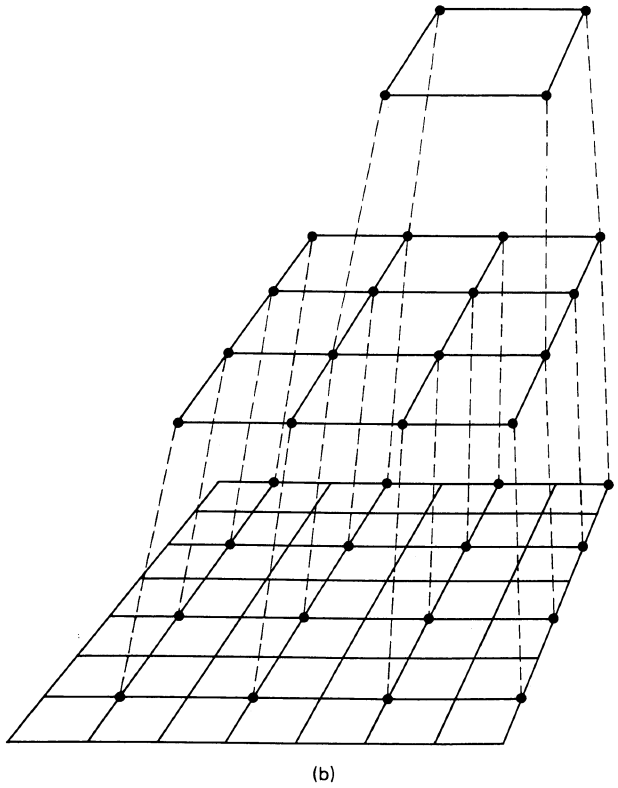


Figure 2.5.2 Two views of the different grids for the case $N = 8$. (a) Grid G_0 consists of all points in the diagram. Grid G_1 consists of the points marked with a circle and grid G_2 consists of the points marked with a double circle. In an interpolation step from G_1 to G_0, the value of $x_{33,0}$ could be set equal to the average of $x_{22,1}$, $x_{24,1}$, $x_{44,1}$, and $x_{42,1}$ associated with the closest grid points belonging to G_1. Similarly, the value of $x_{43,0}$ could be set to the average of $x_{44,1}$ and $x_{42,1}$. (b) Grids G_0, G_1, and G_2 are drawn separately. The dashed lines identify corresponding points in different grids.

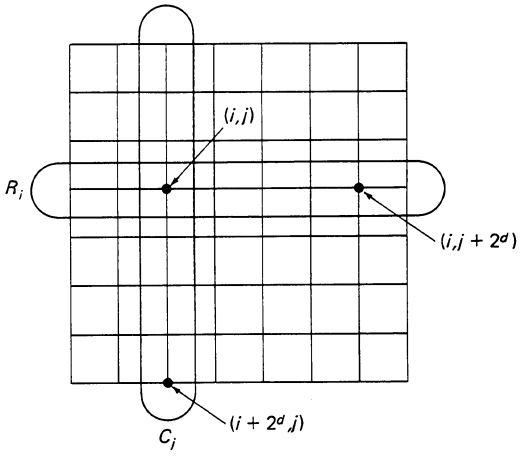


Figure 2.5.3 A mesh is imbedded into a hypercube so that each column C_j and each row R_i is imbedded in a smaller hypercube. The distance between points (i,j) and $(i+2^d,j)$ [respectively, $(i,j+2^d)$], when the arcs of the hypercube C_j (respectively, R_i) are used, is at most 2. Thus, any two processors that need to communicate during the relaxation iterations on grid G_d can do so within two time units. The hypercubes corresponding to each R_i and C_j do not share any arcs; also, concurrent communication within each R_i and each C_j is possible. It follows that all required communication can take place simultaneously for all $(i,j) \in G_d$.

$G_0 \mid 1 \leq j \leq n\}$. Furthermore, the mapping of each such column or row corresponds to a reflected Gray code. Finally, the subhypercubes to which each C_j and R_i is mapped do not share any arcs.

(b) Suppose that a linear array is mapped into a hypercube according to a reflected Gray code. Let the nodes be numbered according to their position in the linear array. Then, the physical distance of nodes i and $i+2^d$ in the hypercube is equal to 1 if $d = 0$, and is equal to 2 if d is a positive integer. Furthermore, communication between nodes i and $i+2^d$ of the linear array can take place simultaneously for all i, in two time units, using the "subrings of level d" (see Fig. 1.3.21 in Subsection 1.3.4).

It follows that the communication time per relaxation iteration is equal to at most 2, for every grid G_d. Similar reasoning shows that grid transfers can also be implemented in a hypercube using local communication only. We conclude that multigrid algorithms can be implemented on hypercubes so that the time devoted to communication is proportional to the computation time. In practice, several points of the finest grid could be assigned to each processor, and this reduces the communication penalty even further. As a most extreme case, if the number of grid points in the coarsest grid is equal to the number of processors, then most of the data dependencies involve variables residing inside the same processor and the communication penalty is rather small. In fact, in this case, it can be seen that there is no need for communication between distant processors even when a mesh–connected architecture is used.

We have assumed so far that relaxations take place in only one grid at a time. In an alternative method, called *concurrent multigrid*, relaxations are performed on several grids simultaneously. Hypercubes are again suitable for such methods, but the mapping problem is somewhat more involved [ChS86].

2.6 CONVERGENCE ANALYSIS OF CLASSICAL ITERATIVE METHODS

We have so far defined and discussed the parallel implementation of iterative methods for solving the system $Ax = b$, where A is an $n \times n$ invertible matrix. All of these methods have the property that if they converge, then they do so to the desired solution (Prop. 4.1). In this section, we derive conditions that guarantee convergence. The mathematical tools that we introduce here will also be of use in the analysis of asynchronous iterative algorithms in Chapters 6 and 7. Furthermore, the ideas of this section have natural generalizations to the contexts of nonlinear optimization and solution of systems of nonlinear equations (Chapter 3). We first develop a uniform representation of the different algorithms.

Let D be a diagonal matrix whose diagonal entries are equal to the corresponding diagonal entries of A, and let $B = A - D$, so that B is zero along the diagonal. Assuming that the diagonal entries of A are nonzero, the Jacobi algorithm can be written in matrix form as [cf. Eq. (4.4)]

$$x(t+1) = -D^{-1}Bx(t) + D^{-1}b. \tag{6.1}$$

Similarly, one iteration of the JOR algorithm is described by

$$x(t+1) = \Big[(1-\gamma)I - \gamma D^{-1}B\Big]x(t) + \gamma D^{-1}b, \tag{6.2}$$

where I is the $n \times n$ identity matrix. To derive a similar matrix representation of the Gauss–Seidel and SOR algorithms, we decompose A as $A = L + D + U$, where L is strictly lower triangular, D is diagonal, and U is strictly upper triangular. Then, the SOR iteration (4.8) can be rewritten as

$$x(t+1) = (1-\gamma)x(t) - \gamma D^{-1}\Big[Lx(t+1) + Ux(t) - b\Big],$$

which is equivalent to

$$x(t+1) = \big(I + \gamma D^{-1}L\big)^{-1}\Big[(1-\gamma)I - \gamma D^{-1}U\Big]x(t) + \gamma\Big(I + \gamma D^{-1}L\Big)^{-1}D^{-1}b. \tag{6.3}$$

Finally, Richardson's method is described by

$$x(t+1) = (I - \gamma A)x(t) + \gamma b \tag{6.4}$$

and a representation resembling Eq. (6.3) is possible for the RGS algorithm of Eq. (4.10). Equations (6.1) to (6.4) can all be written in the form

$$x(t+1) = Mx(t) + Gb, \tag{6.5}$$

where M and G are suitable matrices determined by A and the particular algorithm being used. (We call M the *iteration matrix* of an iterative algorithm.) Therefore, we only need to study the convergence of iteration (6.5). Let us assume that $I - M$ is invertible[†], so that there exists a unique x^* satisfying $x^* = Mx^* + Gb$, and let $y(t) = x(t) - x^*$. Then, $y(t+1) = My(t)$, which implies that $y(t) = M^t y(0)$ for every t. It is seen that the sequence $\{x(t)\}$ converges to x^* for all choices of $x(0)$ if and only if $y(t)$ converges to zero for all choices of $y(0)$. This happens if and only if M^t converges to zero, which is the case if and only if all of the eigenvalues of M have a magnitude smaller than 1, that is, if the *spectral radius* $\rho(M)$ is smaller than 1 (Def. A.9 and Prop. A.21 in Appendix A). We have thus proved the following fundamental result:

Proposition 6.1. Assume that $I - M$ is invertible, let x^* satisfy $x^* = Mx^* + Gb$ and let $\{x(t)\}$ be the sequence generated by the iteration $x(t+1) = Mx(t) + Gb$. Then, $\lim_{t\to\infty} x(t) = x^*$ for all choices of $x(0)$ if and only if $\rho(M) < 1$.

Notice that the above condition for convergence refers only to the iteration matrix M and not to the vector b. We conclude that the iteration $x := Mx + Gb$ converges for all choices of b if and only if it converges for a particular choice of b, say $b = 0$.

Let us recall that the rate at which M^t converges to zero is basically governed by $\rho(M)$ (Prop. A.20 in Appendix A). For this reason, we will be comparing the rate of convergence of alternative iterative methods on the basis of the corresponding spectral radii.

Proposition 6.1 is the most general possible convergence result, but it is of limited use because the eigenvalues of M are rarely known exactly. Thus, more refined tools are called for. A very useful method for proving convergence of iterative methods consists of introducing a suitable distance function, or norm, and showing that each iteration reduces the distance of the current iterate from the desired point of convergence. For linear equations, the most useful norms are quadratic norms, like the Euclidean norm, and (weighted) maximum norms.

2.6.1 Background on Maximum Norms and Nonnegative Matrices

Let us recall some notation and a few facts from Appendix A. If w is a vector in $\Re^n$, the notations $w > 0$ and $w \geq 0$ indicate that all components of w are positive or nonnegative, respectively. Similarly, if A is a matrix, the notations $A > 0$ and $A \geq 0$ indicate that all entries of A are positive or nonnegative, respectively. For any two vectors w and v, the notation $w > v$ stands for $w - v > 0$. The notations $w < v$, $w \geq v$, $A > B$, etc. are to be interpreted accordingly. Given a vector w, we denote by $|w|$ the vector whose ith component equals $|w_i|$. Similarly, for any matrix A, we denote by $|A|$ the matrix whose entries are the absolute values of the entries of A. Given a vector $w > 0$, we define the weighted maximum norm $\|\cdot\|_\infty^w$ by $\|x\|_\infty^w = \max_i |x_i/w_i|$. In the special case

[†] In fact, for the algorithms introduced in Section 2.4, $I - M$ is guaranteed to be invertible, as long as A is invertible (Exercise 6.1).

where $w_i = 1$ for each i, we suppress the superscript w. Thus, $\|x\|_\infty = \max_i |x_i|$. The unit ball with respect to the norm $\|\cdot\|_\infty^w$, that is, the set of all x such that $\|x\|_\infty^w \leq 1$ is actually a box rather than a sphere, as illustrated in Fig. 2.6.1.

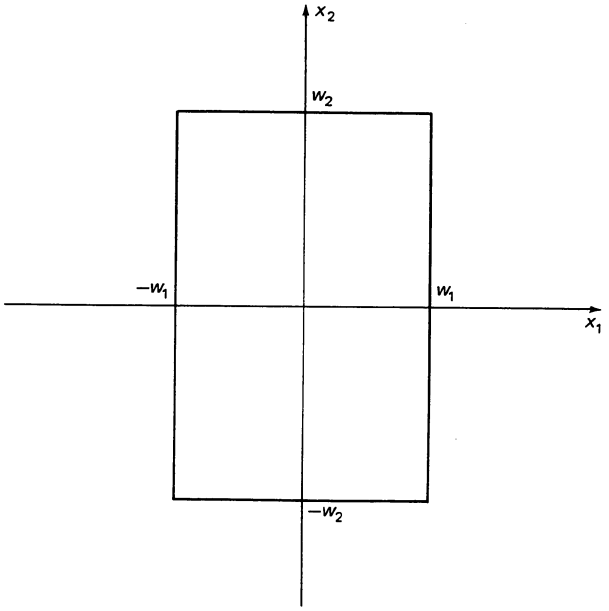


Figure 2.6.1 The unit ball $\{x \mid \|x\|_\infty^w \leq 1\} = \{x \mid |x| \leq w\}$ in the two–dimensional case.

The vector norm $\|\cdot\|_\infty^w$ induces a matrix norm, also denoted by $\|\cdot\|_\infty^w$, defined by

$$\|A\|_\infty^w = \max_{x \neq 0} \frac{\|Ax\|_\infty^w}{\|x\|_\infty^w},$$

where A is an $n \times n$ matrix. An alternative but equivalent definition of this norm is

$$\|A\|_\infty^w = \max_i \frac{1}{w_i} \sum_{j=1}^n |a_{ij}| w_j, \tag{6.6}$$

where a_{ij} are the entries of A [Prop. A.13(a) in Appendix A].

The following proposition lists a few useful facts. See Fig. 2.6.2 for an illustration of parts (b) and (d).

Proposition 6.2. Let M and N be $n \times n$ matrices and let $w \in \Re^n$ be a positive vector.

(a) The matrix M is nonnegative ($M \geq 0$) if and only if it maps nonnegative vectors into nonnegative vectors.

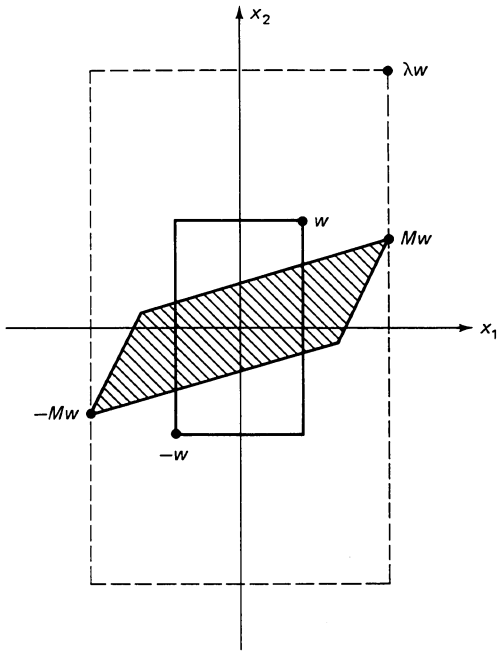


Figure 2.6.2 Illustration of the norm of a matrix $M \geq 0$. The shaded region is the image, under the mapping M, of the unit ball with respect to $\|\cdot\|_\infty^w$, that is, the set $\{Mx \mid \|x\|_\infty^w \leq 1\}$. The scalar λ is the smallest number such that $Mw \leq \lambda w$. In view of Prop. 6.2(d), we have $\lambda = \|M\|_\infty^w$.

(b) If $M \geq 0$, then $\|M\|_\infty^w = \|Mw\|_\infty^w$.

(c) We have $\| \, |M| \, \|_\infty^w = \|M\|_\infty^w$.

(d) Let $M \geq 0$. Then, for any scalar $\lambda > 0$, we have $\|M\|_\infty^w \leq \lambda$ if and only if $Mw \leq \lambda w$.

(e) We have $\rho(M) \leq \|M\|_\infty^w$.

(f) If $M \geq N \geq 0$, then $\|M\|_\infty^w \geq \|N\|_\infty^w$.

Proof.

(a) If $M \geq 0$ and $x \geq 0$, then it is obvious that $Mx \geq 0$. For the converse, suppose that the ijth entry of M is negative. Let x be the jth unit vector. Then the ith entry of Mx is negative.

(b) Using Eq. (6.6), we have

$$\|M\|_\infty^w = \max_i \frac{1}{w_i} \sum_{j=1}^n m_{ij} w_j = \max_i \frac{1}{w_i} [Mw]_i = \|Mw\|_\infty^w.$$

(c) This is an immediate consequence of Eq. (6.6).

(d) Using part (b), we have $\|M\|_\infty^w \leq \lambda$ if and only if $\|Mw\|_\infty^w \leq \lambda$, which is equivalent to $Mw \leq \lambda w$.

(e) This follows from the inequality $\rho(M) \leq \|M\|$, which holds for every induced norm $\|\cdot\|$ (Prop. A.20 in Appendix A).

(f) This is an immediate consequence of Eq. (6.6). **Q.E.D.**

Showing that $\|M\|_\infty^w < 1$ is a simple method for proving that $\rho(M) < 1$, as long as the weighting vector w can be suitably chosen. However, this method is not universally applicable. In particular, there exist matrices M such that $\rho(M) < 1$, but for which there exists no $w > 0$ such that $\|M\|_\infty^w < 1$ (Exercise 6.2). Still, for nonnegative matrices, the property $\rho(M) < 1$ is equivalent to the existence of a positive vector w for which $\|M\|_\infty^w < 1$. This is a consequence of the Perron–Frobenius theorem, the most important result in the theory of nonnegative matrices, which will be presented and proved after a few definitions.

Given an $n \times n$ matrix M, with $n \geq 2$, we form a directed graph $G = (N, A)$, with nodes $N = \{1, \ldots, n\}$ and arcs $A = \{(i, j) \mid i \neq j \text{ and } m_{ij} \neq 0\}$, where m_{ij} is the ijth entry of M.

Definition 6.1. An $n \times n$ matrix M, with $n \geq 2$, is called *irreducible* if for every $i, j \in N$ there exists a positive path in the above constructed graph G (i.e., a path with all arcs oriented as in the graph G) from i to j. For the case $n = 1$, M is called irreducible if its only element is nonzero.

An alternative and often useful characterization of irreducibility is the following:

Proposition 6.3. An $n \times n$ matrix $M \geq 0$, with $n \geq 2$, is irreducible if and only if $(I + M)^{n-1} > 0$.

Proof. By using the definition of matrix multiplication, it is easy to check that for $i \neq j$, the ijth entry of $(I + M)^{n-1}$ is equal to the sum of positive multiples of all products of the form $\prod_{k=1}^{K-1} m_{i_k i_{k+1}}$, where $i_1 = i$, $i_K = j$, and $K \leq n$. Since $M \geq 0$, this sum is positive if and only if one of the summands is positive, which is the case if and only if there exists a positive path from i to j in the graph G associated with M. When $i = j$, the corresponding entry of $(I + M)^{n-1}$ is at least unity. **Q.E.D.**

Proposition 6.4. If an $n \times n$ matrix $M \geq 0$ is irreducible and if some nonnegative vector $x \geq 0$ satisfies $Mx = 0$, then $x = 0$.

Proof. The case $n = 1$ is trivial and we assume that $n \geq 2$. If all the entries of the jth column of M are equal to zero, then there is no path from any $i \neq j$ to j in the graph associated with M and therefore M is not irreducible. We conclude that every column of M has a nonzero entry and there exists a positive number a such that $\sum_{i=1}^n m_{ij} \geq a$ for each j. Suppose that $x \geq 0$ and $Mx = 0$. Then,

$$0 = \sum_{i=1}^{n} [Mx]_i = \sum_{i=1}^{n} \sum_{j=1}^{n} m_{ij} x_j = \sum_{j=1}^{n} x_j \sum_{i=1}^{n} m_{ij} \geq a \sum_{j=1}^{n} x_j.$$

Since $a > 0$, we conclude that $\sum_{j=1}^{n} x_j \leq 0$, and since $x \geq 0$, we obtain $x = 0$. **Q.E.D.**

We now state and prove a few variants of the Perron–Frobenius theorem. The proof presented here uses the Brouwer Fixed Point Theorem [GuP74], whose proof is beyond the scope of this book. A longer proof that proceeds from first principles is outlined in Exercise 6.3.

Consider the unit simplex

$$S = \left\{ x \in \Re^n \;\middle|\; x \geq 0 \text{ and } \sum_{i=1}^{n} x_i = 1 \right\}.$$

Proposition 6.5. (*Brouwer Fixed Point Theorem*) If $f : S \mapsto S$ is a continuous function, then there exists some $w \in S$ such that $f(w) = w$.

Proposition 6.6. (*Perron–Frobenius Theorem*) Let M be an $n \times n$ nonnegative matrix.

(a) If M is irreducible, then $\rho(M)$ is an eigenvalue of M and there exists some $w > 0$ such that $Mw = \rho(M)w$. Furthermore, such a w is unique within a scalar multiple; that is, if some v also satisfies $Mv = \rho(M)v$, then $v = \alpha w$ for some scalar α. Finally, $\|M\|_\infty^w = \rho(M)$.

(b) $\rho(M)$ is an eigenvalue of M and there exists some $w \geq 0$, $w \neq 0$, such that $Mw = \rho(M)w$.

(c) For every $\epsilon > 0$, there exists some $w > 0$ such that $\rho(M) \leq \|M\|_\infty^w \leq \rho(M) + \epsilon$.

Proof. The case $n = 1$ is trivial and it will be assumed that $n \geq 2$.

(a) Let $M \geq 0$ be irreducible. We define a function $f : S \mapsto S$ with components

$$f_i(x) = \frac{[Mx]_i}{\sum_{j=1}^{n} [Mx]_j}, \qquad i = 1, \ldots, n.$$

Suppose that the denominator in the definition of f is equal to zero for some $x \in S$. Since $M \geq 0$ and $x \geq 0$, we conclude that $Mx = 0$. Then, Prop. 6.4 implies that $x = 0$, which contradicts the assumption $x \in S$. We conclude that the denominator is always nonzero and, therefore, f is well-defined and continuous on S. By Prop. 6.5, there exists some $w \in S$ such that $f(w) = w$. Let $\lambda = \sum_{i=1}^{n} [Mw]_i$. We have $\lambda > 0$ because otherwise $Mw = 0$, which would imply that $w = 0$. The equality $f(w) = w$ can be rewritten as $Mw = \lambda w$. In particular, λ is an eigenvalue of

M. We now have $(I + M)w = (1 + \lambda)w$ and $(I + M)^{n-1}w = (1 + \lambda)^{n-1}w$. By Prop. 6.3, $(I + M)^{n-1} > 0$, and since $w \geq 0$, $w \neq 0$, it is easily seen that $(I + M)^{n-1}w > 0$. This implies that $w > 0$.

We now show that $\lambda = \rho(M)$. Since λ is an eigenvalue of M, we obtain $\lambda \leq \rho(M)$. On the other hand, using the obvious fact $\|w\|_\infty^w = 1$ and Prop. 6.2(b), we obtain

$$\lambda = \|\lambda w\|_\infty^w = \|Mw\|_\infty^w = \|M\|_\infty^w \geq \rho(M),$$

where the last inequality is Prop. 6.2(e). The equality $\lambda = \rho(M)$ follows.

We now prove uniqueness of w. Suppose that a vector v satisfies $Mv = \rho(M)v$. If $v = 0$, then $v = \alpha w$, with $\alpha = 0$, and we are done. We now assume that $v \neq 0$ and we define $z = v/\|v\|_\infty^w$. Notice that $w \geq |z|$ and that $w_i = |z_i|$ for some i. By possibly replacing z by $-z$, we may and will assume that $w_i = z_i$ for some i. Either $z = w$, in which case v is a scalar multiple of w, or $z \neq w$, in which case we will obtain a contradiction. We have $M(w - z) = \rho(M)(w - z)$. Since M is irreducible, $(I + M)^{n-1} > 0$, and since $w - z \geq w - |z| \geq 0$ and $w \neq z$, we obtain $(I + M)^{n-1}(w - z) > 0$. On the other hand, $(I + M)^{n-1}(w - z) = \big(1 + \rho(M)\big)^{n-1}(w - z)$, whose ith component is equal to zero, by construction. This is a contradiction and proves the desired result.

(b) Let N_δ be an $n \times n$ matrix with all entries equal to δ, where δ is a positive scalar. Since $M + N_\delta$ is positive, it is irreducible and part (a) of the proposition applies. Thus, for each $\delta > 0$, there exists some $w_\delta > 0$ such that

$$(M + N_\delta)w_\delta = \rho(M + N_\delta)w_\delta. \tag{6.7}$$

Without any loss of generality, let us assume that each w_δ has been scaled so that its largest component is equal to 1. Then the set $\{w_\delta \mid \delta > 0\}$ is bounded. Since every bounded sequence in $\Re^n$ has a convergent subsequence (Prop. A.5 in Appendix A), it follows that there exists some vector w_0 and a sequence $\{\delta_k\}$ such that $\lim_{k\to\infty} \delta_k = 0$ and $\lim_{k\to\infty} w_{\delta_k} = w_0$. Since w_0 is the limit of positive vectors, it has to be nonnegative. By construction, $\|w_{\delta_k}\|_\infty = 1$ for each k, and since w_{δ_k} converges to w_0, it follows that $\|w_0\|_\infty = 1$; in particular, $w_0 \neq 0$.

We now take the limit of both sides of Eq. (6.7), as δ tends to zero along the sequence $\{\delta_k\}$. We see that Mw_{δ_k} converges to Mw_0; also $\|N_{\delta_k}w_{\delta_k}\|_\infty \leq n\delta_k\|w_{\delta_k}\|_\infty = n\delta_k$, which converges to zero. Thus, the left–hand side of Eq. (6.7) converges to Mw_0. We now use the fact that the spectral radius is a continuous function (Prop. A.19 in Appendix A) to conclude that $\rho(M + N_{\delta_k})$ converges to $\rho(M)$. Thus, the limit of the right–hand side of Eq. (6.7) is equal to $\rho(M)w_0$. This implies that $Mw_0 = \rho(M)w_0$ and proves part (b).

(c) The first inequality holds for every $w > 0$, by Prop. 6.2(e). Let $\delta > 0$ be small enough so that $\rho(M + N_\delta) \leq \rho(M) + \epsilon$, which is possible because of the continuity of the spectral radius. Let $w_\delta > 0$ be as in part (b). Then, using parts (f) and (b) of Prop. 6.2,

$$\|M\|_\infty^{w_\delta} \leq \|M + N_\delta\|_\infty^{w_\delta} = \|(M + N_\delta)w_\delta\|_\infty^{w_\delta}$$
$$= \rho(M + N_\delta)\|w_\delta\|_\infty^{w_\delta} = \rho(M + N_\delta) \leq \rho(M) + \epsilon.$$

Q.E.D.

Proposition 6.6(c) is illustrated by the following example. Let

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

The graph associated with this matrix is shown in Fig. 2.6.3 and we see that M is not irreducible. The characteristic polynomial of M is $\lambda(\lambda - 1)$ from which it follows that $\rho(M) = 1$. On the other hand, for any vector $w > 0$, we have $\|M\|_\infty^w = (w_1 + w_2)/w_2 > 1$. Thus, there does not exist any $w > 0$ such that $\|M\|_\infty^w = \rho(M)$. However, by taking w_1 arbitrarily small, $\|M\|_\infty^w$ comes arbitrarily close to $\rho(M)$, as predicted by Prop. 6.6(c).

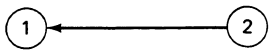


Figure 2.6.3 The graph associated with the matrix

$$M = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

which is not irreducible.

We now present a few useful consequences of the Perron-Frobenius theorem.

Corollary 6.1. If M is a square nonnegative matrix then the following are equivalent:

(i) $\rho(M) < 1$.
(ii) There exists some $w > 0$ such that $\|M\|_\infty^w < 1$.
(iii) There exist some $\lambda < 1$ and $w > 0$ such that $Mw \leq \lambda w$.

Proof. Assume that (i) holds. Let $\epsilon > 0$ be small enough so that $\rho(M) + \epsilon < 1$. By using Prop. 6.6(c), there exists some $w > 0$ such that $\|M\|_\infty^w \leq \rho(M) + \epsilon < 1$, which proves (ii). If (ii) holds, then, using the definition of $\|\cdot\|_\infty^w$, we obtain $Mw < w$, which proves (iii). Finally, if (iii) holds, then $\|M\|_\infty^w \leq \lambda < 1$ and using Prop. 6.2(e), we obtain $\rho(M) < 1$, which proves (i) and concludes the proof. **Q.E.D.**

Corollary 6.2. Given any square matrix M, there exists some $w > 0$ such that $\|M\|_\infty^w < 1$ if and only if $\rho(|M|) < 1$.

Proof. By Prop. 6.2(c), we have $\|M\|_\infty^w = \big\| |M| \big\|_\infty^w$. The result follows from the equivalence of parts (i) and (ii) of Cor. 6.1 applied to the matrix $|M|$. **Q.E.D.**

Corollary 6.3. For any square matrix M, $\rho(M) \leq \rho(|M|)$.

Proof. By Prop. 6.6(c), for any $\epsilon > 0$, there exists some $w > 0$ such that

$$\rho(M) \leq \|M\|_\infty^w = \big\| \, |M| \, \big\|_\infty^w \leq \rho(|M|) + \epsilon.$$

Since ϵ was arbitary, we can take the limit as ϵ decreases to zero to obtain the desired result. **Q.E.D.**

2.6.2 Convergence Analysis Using Maximum Norms

We now use the Perron–Frobenius theorem and its corollaries to obtain sufficient conditions for convergence of iterative algorithms, and also to make a comparison between the Jacobi and the Gauss–Seidel methods.

Definition 6.2. A square matrix A with entries a_{ij} is *(row) diagonally dominant* if

$$\sum_{j \neq i} |a_{ij}| < |a_{ii}|, \qquad \forall i.$$

Let A be a square row diagonally dominant matrix, and consider the iteration matrix M associated with the Jacobi algorithm for solving the equation $Ax = b$. Then, $|m_{ij}| = |a_{ij}|/|a_{ii}|$ for $j \neq i$, and $|m_{ii}| = 0$ [cf. Eq. (6.1)]. The assumption of diagonal dominance translates to the condition $\sum_{j=1}^n |m_{ij}| < 1$ for all i. Equivalently, $\|M\|_\infty < 1$ from which we conclude that $\rho(M) < 1$. We have thus proved the following.

Proposition 6.7. If A is row diagonally dominant, then the Jacobi method for solving $Ax = b$ converges.

We continue with a general comparison of methods of the Jacobi and of the Gauss–Seidel type. Let us fix an $n \times n$ matrix M associated to an iteration $x := Mx + b$, where b is some fixed vector. Let $\hat{M}$ be the corresponding Gauss–Seidel iteration matrix, that is, the iteration matrix obtained if the components in the original iteration are updated one at a time. In particular, the matrix $\hat{M}$ satisfies

$$[\hat{M}x]_i = \sum_{j<i} m_{ij}[\hat{M}x]_j + \sum_{j \geq i} m_{ij}x_j, \tag{6.8}$$

for every $x \in \Re^n$.

Proposition 6.8. Suppose that $\rho(|M|) < 1$. Then, $\rho(\hat{M}) \leq \rho(|M|)$. (In particular, if $M \geq 0$ and the Jacobi–type iteration $x := Mx + b$ converges, then the corresponding Gauss–Seidel iteration also converges.)

Proof. Let us assume that $\rho(|M|) < 1$ and let us fix some $\epsilon > 0$ such that $\lambda = \rho(|M|) + \epsilon < 1$. Proposition 6.6(c) shows that there exists some $w > 0$ such that

$$\| \, |M| w \, \|_\infty^w = \| \, |M| \, \|_\infty^w \leq \rho(|M|) + \epsilon = \lambda.$$

Therefore, $|M|w \leq \lambda w$. Equivalently, for all i,

$$\sum_{j=1}^{n} |m_{ij}| w_j \leq \lambda w_i.$$

Consider now some x such that $\|x\|_\infty^w = 1$ and let $y = \hat{M}x$. We will prove, by induction on i, that $|y_i| \leq \lambda w_i$. Indeed, assuming that $|y_j| \leq \lambda w_j$ for $j < i$, we obtain, from Eq. (6.8),

$$|y_i| \leq \sum_{j<i} |m_{ij}| \cdot |y_j| + \sum_{j \geq i} |m_{ij}| w_j \leq \sum_{j<i} |m_{ij}| w_j + \sum_{j \geq i} |m_{ij}| w_j \leq \lambda w_i.$$

We conclude that $|\hat{M}x| \leq \lambda w$ for every x satisfying $\|x\|_\infty^w = 1$. This implies that $\|\hat{M}\|_\infty^w \leq \lambda$. Therefore, $\rho(\hat{M}) \leq \lambda = \rho(|M|) + \epsilon$. Since this is true for every $\epsilon > 0$, the result follows. **Q.E.D.**

We now focus on the solution of a system of equations $Ax = b$, where the entries of the A matrix satisfy $a_{ij} \leq 0$ for all $i \neq j$, and $a_{ii} > 0$ for all i. The iteration matrix M_J associated with the Jacobi algorithm for solving such a system [cf. Eq. (4.4)] is given by $[M_J]_{ii} = 0$ and $[M_J]_{ij} = -a_{ij}/a_{ii}$ for $j \neq i$, and it follows that $M_J \geq 0$. Let M_{GS} be the iteration matrix for the Gauss–Seidel algorithm of Eq. (4.5). It is easily seen that $M_{GS} \geq 0$.

Proposition 6.9. (*Stein–Rosenberg Theorem*) Under the above assumption on A, the following are true:

(a) If $\rho(M_J) < 1$, then $\rho(M_{GS}) \leq \rho(M_J)$.

(b) If $\rho(M_J) \geq 1$, then $\rho(M_{GS}) \geq \rho(M_J)$.

Proof. Part (a) is simply a restatement of Prop. 6.8, because we have $|M_J| = M_J$. For part (b), start with some $w \neq 0$ such that $w \geq 0$ and $M_J w = \rho(M_J)w$, which exists because of Prop. 6.6(b). Let $y = M_{GS}w$. Proceeding as in the proof of Prop. 6.8, with the inequalities reversed, we obtain $y_i \geq \rho(M_J)w_i$ for each i. Therefore, $M_{GS}w = y \geq \rho(M_J)w$. Let $N = M_{GS}/\rho(M_J)$. Then, $Nw \geq w$ and $N^t w \geq w$, for all t. Since $w \neq 0$, it follows that N^t does not converge to zero, as t tends to infinity, and using Prop. A.21 in Appendix A, we have $\rho(N) \geq 1$. Thus, $\rho(M_{GS}) = \rho(M_J)\rho(N) \geq \rho(M_J)$, which concludes the proof. **Q.E.D.**

Propositions 6.8 and 6.9 are useful because the iteration matrix M_J corresponding to the Jacobi method may be simpler to analyze when compared to the iteration

matrix M_{GS} of the Gauss–Seidel method. Proposition 6.8 shows that, for the special case of nonnegative iteration matrices, if a Jacobi–type algorithm converges, then the corresponding Gauss–Seidel algorithm also converges, and its convergence rate is no worse than that of the Jacobi algorithm. This provides a justification for algorithms of the Gauss–Seidel type, that is, algorithms in which more recent data are used whenever available. Furthermore, notice that the proofs of Props. 6.8 and 6.9 remain valid when different updating orders of the components are considered. The implication is that iterative algorithms involving positive iteration matrices possess some intrinsic robustness with respect to the order of updates. This turns out to be a key element in the context of asynchronous algorithms, as will be seen in Chapter 6.

2.6.3 Convergence Analysis Using a Quadratic Cost Function

Quadratic cost functions are particularly useful in dealing with systems of equations $Ax = b$, where A is a symmetric positive definite matrix, which we are going to assume in this subsection (see Appendix A for a definition and properties of positive definite and symmetric matrices). Let us also recall that positive definite matrices are always invertible, so that the equation $Ax = b$ is guaranteed to have a unique solution. We will measure the progress of an algorithm by introducing the cost function $F(x) = \frac{1}{2}x'Ax - x'b$. Since A is positive definite, F is a strictly convex function [Prop. A.40(d) in Appendix A], and a vector x^* minimizes F if and only if $\nabla F(x^*) = 0$. We notice that $\nabla F(x) = Ax - b$, which shows that x^* is the solution of $Ax = b$ if and only if x^* minimizes F. (In particular, F has a unique minimum.)

We now interpret the Gauss–Seidel method in terms of the cost function F. Suppose that at some stage of the algorithm, we have a current vector x, and it is the turn of the ith component of x to be updated. Let $\bar{x}$ be the vector obtained after the update. Using the definition of the Gauss–Seidel method, we see that $\bar{x}$ is chosen so that the ith equation in the system $Ax = b$ is satisfied. Since the equations $Ax = b$ and $\nabla F(x) = 0$ are equivalent, it follows that $\bar{x}$ is chosen so that $\nabla_i F(\bar{x}) = 0$. Thus, the vector $\bar{x}$ is defined by the properties that all components except for the ith one are fixed and the ith component is chosen so as to satisfy $\nabla_i F(\bar{x}) = 0$. But this is equivalent to choosing $\bar{x}$ so as to minimize $F(y)$ over all y differing from x only along the ith coordinate. There are two distinct cases to be considered: either at the current point x we have $\nabla_i F(x) \neq 0$, in which case the update has nonzero size and $F(\bar{x}) < F(x)$; or $\nabla_i F(x) = 0$, x is not changed, and the value of F remains the same. In the latter case, one of two things will happen. Either a component j will be eventually found such that $\nabla_j F(x) \neq 0$ and a cost decrease will result; or $\nabla_j F(x) = 0$ for all j, in which case we are at a point at which $\nabla F(x) = 0$, that is, at a solution of $Ax = b$. To summarize, the Gauss–Seidel method proceeds by minimizing $F(x)$ successively along each coordinate, as illustrated in Fig. 2.6.4. By using this fact, it is not hard to show that $F(x)$ converges to the minimum of F (see Fig. 2.6.5). Accordingly, the vector x converges to the solution of $Ax = b$.

Let us now turn to the SOR method. As long as γ is positive, the direction of each update is the same as for the update that would be made by the Gauss–Seidel algorithm (starting from the same point), and the magnitude of the update is scaled by a factor of

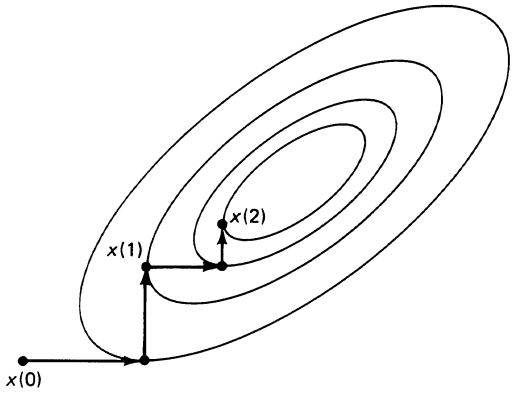


Figure 2.6.4 For a positive definite matrix A, the Gauss–Seidel method can be viewed as a coordinate descent method for minimizing the function $F(x) = \frac{1}{2}x'Ax - x'b$. The curves in this figure correspond to the level sets of F, that is, sets of points on which F is constant.

γ. It is then natural to look for the range of values of γ for which the cost does not increase. Since $F(x)$ is a quadratic function of x_i, the other coordinates being fixed, and since quadratic functions are symmetric around their minimizing point, it follows that (whenever the vector x is changed) we have a cost decrease if and only if $0 < \gamma < 2$. This discussion motivates the following result.

Proposition 6.10. Let A be symmetric and positive definite, and let x^* be the solution of $Ax = b$.

(a) If $\gamma \in (0, 2)$, then the sequence $\{x(t)\}$ generated by the SOR algorithm converges to x^*.

(b) If $\gamma \notin (0, 2)$, then for every choice of $x(0)$ different than x^*, the sequence generated by the SOR algorithm does not converge to x^*.

The proof of Prop. 6.10 consists of filling the gaps in the argument we provided above and is left as an exercise. Actually, part (a) of the proposition is a special case of Prop. 2.2 of Section 3.2, which covers the case of a general (not necessarily quadratic) cost function F.

Recall that the JOR method is the same as SOR except that all components are updated simultaneously. Since each component is updated by SOR in a direction that does not increase the cost, the same should be true for JOR, as long as $\gamma > 0$ is sufficiently small. Finally, Richardson's method is the same as JOR except that each component's update is not scaled by the factor $1/a_{ii}$. Thus, again each component is updated in a direction that does not increase the cost. This discussion motivates the following result. Its proof is omitted because it is a special case of the more general Prop. 2.1 of Section 3.2. However, the interested reader should have no difficulty in providing a proof.

Proposition 6.11. If A is symmetric and positive definite and if $\gamma > 0$ is sufficiently small, then the JOR and Richardson's algorithms converge to the solution of $Ax = b$.

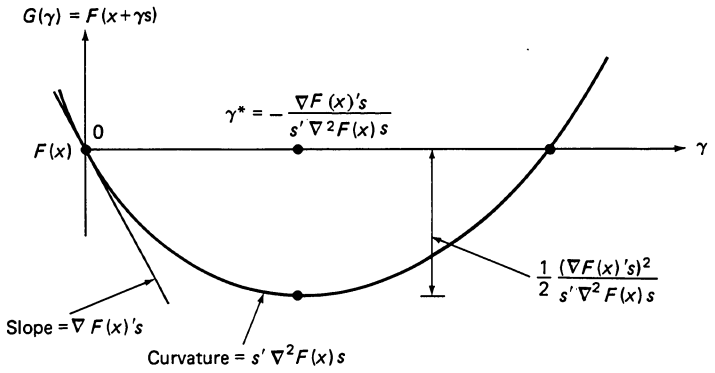


Figure 2.6.5 Illustration of cost reduction along a descent direction. Let A be a symmetric positive definite square matrix and consider a function F, given by $F(x) = \frac{1}{2}x'Ax$, along a direction s starting from the vector x. The function $G(\gamma) = F(x + \gamma s) = \frac{1}{2}(x + \gamma s)'A(x + \gamma s)$ of the stepsize is quadratic and is minimized at

$$\gamma^* = -\frac{\nabla F(x)'s}{s'\nabla^2 F(x)s} = -\frac{x'As}{s'As}.$$

The corresponding reduction is

$$F(x) - \min_{\gamma} F(x + \gamma s) = \frac{1}{2}\frac{\big(\nabla F(x)'s\big)^2}{s'\nabla^2 F(x)s}.$$

By using this figure, it can be shown that if $s^1, \ldots, s^n$ is a set of linearly independent directions and F is minimized at each iteration along one of these directions, and each of these directions is used infinitely often, then convergence to the unique minimizing vector of F is guaranteed. (The sequence $\{x(t)\}$ belongs to the bounded set $\{x | F(x) \leq F\big(x(0)\big)\}$ and therefore has at least one limit point. The sequence of cost reductions must tend to zero, implying that the sequence of stepsizes tends to zero. Thus, every limit point x^* of the sequence $\{x(t)\}$ satisfies $\nabla F(x^*)'s^i = 0$ for $i = 1, \ldots, n$. Since $s^1, \ldots, s^n$ are linearly independent, we have $\nabla F(x^*) = 0$ and, therefore, $x^* = 0$.) If the direction vectors $s^1, \ldots, s^n$, are the unit vectors in $\Re^n$, we recover the Gauss–Seidel algorithm.

2.6.4 The Poisson Equation Revisited

We consider the system of equations

$$f_{i,j} = \frac{1}{4}\Big(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1}\Big) - \frac{1}{4N^2}g_{i,j}, \qquad 0 < i, j < N. \tag{6.9}$$

obtained from the discretization of the Poisson equation on a square [cf. Eq. (5.2)]. We have one such equation for each variable $f_{i,j}$ corresponding to an interior grid point. These equations can be represented in the matrix form $Ax = b$ as follows. With every integer k such that $1 \leq k \leq (N-1)^2$, we associate a different interior grid point (i_k, j_k).

We arrange the unknowns $f_{i,j}$ in order of increasing k to form a vector f of unknowns. Then, the kth equation of the system $Ax = b$ is given by Eq. (6.9), with (i, j) replaced by (i_k, j_k). It follows that the matrix A has the following structure: (i) $a_{kk} = 1$ for each k, and (ii) if $k \neq \ell$, then $a_{k\ell}$ is equal to either $-\frac{1}{4}$ or it is equal to zero; in particular, it is equal to $-\frac{1}{4}$ if and only if $|i_k - i_\ell| + |j_k - j_\ell| = 1$, that is, if and only if (i_k, j_k) and (i_ℓ, j_ℓ) are neighboring interior grid points. It is clear that $a_{k\ell} = a_{\ell k}$ for every k, ℓ, and the matrix A is symmetric.

The eigenvalues of the iteration matrix corresponding to the Jacobi algorithm can be calculated explicitly; however, we prefer to show convergence using a quadratic cost function.

Proposition 6.12. The matrix A constructed above is symmetric positive definite.

Proof. Symmetry has been already established. Using the discussion preceding the statement of the proposition, $f'Af$ is given by the following expression:

$$\sum_{i,j}\Big[f_{i,j}\Big(f_{i,j} - \frac{1}{4}\sum_{k,\ell} f_{k,\ell}\Big)\Big], \tag{6.10}$$

where the first summation is over all interior grid points and the second summation is over all (k, ℓ) that correspond to interior grid points neighboring with (i, j). By rearranging the terms in the expression (6.10), we can rewrite it as

$$\frac{1}{4}\sum_{(i,j),(k,\ell)}\Big(f_{i,j}^2 + f_{k,\ell}^2 - 2f_{i,j}f_{k,\ell}\Big) + \frac{1}{4}\sum_{(i,j)} f_{i,j}^2\chi_{i,j}, \tag{6.11}$$

where the summation is over all pairs $\big((i, j), (k, \ell)\big)$ of neighboring interior grid points, and $\chi_{i,j}$ is the number of boundary points neighboring with (i, j). Using the fact $a^2 + b^2 - 2ab = (a - b)^2 \geq 0$, it follows that $f'Af \geq 0$ and therefore A is nonnegative definite. To prove positive definiteness, we assume that $f'Af = 0$, or, equivalently, that the expression (6.11) is equal to zero, and we will show that $f = 0$. If the expression (6.11) is equal to zero, it follows that $f_{i,j} = 0$ for all points (i, j) neighboring with the boundary, and $f_{i,j} = f_{k,\ell}$ for all pairs $\big((i, j), (k, \ell)\big)$ of neighboring points. This implies that $f_{i,j} = 0$ for every (i, j). **Q.E.D.**

Since A is positive definite, the Gauss–Seidel algorithm converges by virtue of Prop. 6.10. We now notice that the diagonal elements of A are positive and the off–diagonal elements are less than or equal to zero. Consequently, Prop. 6.9 applies and shows that the Jacobi algorithm also converges. We have thus proved the following.

Proposition 6.13. The Jacobi and Gauss–Seidel algorithms for solving the system (6.9) converge and the Gauss–Seidel algorithm converges faster, in the sense of Prop. 6.9.

Proposition 6.13 is one of the reasons for our interest in the parallel implementation of the Gauss–Seidel algorithm. In fact, in the present context, the equally efficiently parallelizable SOR algorithm is known to converge substantially faster than the Gauss–Seidel algorithm, when the relaxation parameter is suitably chosen ([Var62] and [HaY81]). The above discussion also extends to the numerical solution of more general classes of elliptic partial differential equations.

EXERCISES

6.1. Assuming that a square matrix A is invertible and has a nonzero diagonal, show that $I - M$ is invertible, where M is the iteration matrix defined by Eq. (6.5), corresponding to any one of the algorithms introduced in Section 2.4.

6.2. Find an example of a symmetric matrix M with no zero entries and such that $\rho(M) < 1$, but $\|M\|_\infty^w \geq 1$ for every positive vector w. *Hint:* The smallest possible example is of size 3×3.

6.3. This exercise leads to a proof of the Perron–Frobenius theorem (Prop. 6.6) that does not involve the Brouwer Fixed Point Theorem. Let M be an $n \times n$ irreducible nonnegative matrix. Our objective is to show that M has a positive eigenvector w.

Let $X = \{x \in \Re^n \mid x \geq 0,\ x \neq 0\}$. For every $x \in X$, let $r(x) = \sup\{\rho \mid Mx \geq \rho x\}$. Let $S = \{x \in X \mid \sum_{i=1}^n x_i = 1\}$. Let $\lambda = \sup\{r(x) \mid x \in X\}$.

(a) Show that $\lambda > 0$. *Hint:* Let e be the vector in $\Re^n$ with all coordinates equal to 1; use the irreducibility of M to show that $Me > 0$ and conclude that $r(e) > 0$.

(b) Show that $\lambda = \sup\{r(x) \mid x \in S\}$. *Hint:* Any vector in X can be scaled so that it belongs to S.

(c) Let $Q = \{(I+M)^{n-1}x \mid x \in S\}$. Show that all elements of Q are positive vectors.

(d) Show that $\lambda = \sup\{r(x) \mid x \in Q\}$. *Hint:* By definition, $\sup\{r(x) \mid x \in Q\} \leq \lambda$. For the reverse inequality, show that $r\big((I+M)^{n-1}x\big) \geq r(x)$ for $x \in S$, and use the result of (b).

(e) Show that $r(\cdot)$ is a continuous function on Q. *Hint:* Show that

$$r(x) = \min_{1 \leq i \leq n} \left\{ \frac{1}{x_i} \sum_{j=1}^n m_{ij} x_j \right\}, \qquad x \in Q.$$

(f) Show that there exists some $w \in Q$ such that $r(w) = \lambda$.

(g) Let $z = Mw - \lambda w$. Show that $z = 0$, which proves that M has a positive eigenvector w. *Hint:* Use the definition of w to show that $z \geq 0$. If $z \neq 0$, then $(I+M)^{n-1}z > 0$, which shows that $M(I+M)^{n-1}w > \lambda(I+M)^{n-1}w$. Show that this contradicts the definition of λ.

6.4. Prove Prop. 6.10. *Hint:* For part (b) show that $F\big(x(t)\big) \geq F\big(x(0)\big)$ for all t.

6.5. Prove Prop. 6.11.

2.7 THE CONJUGATE GRADIENT METHOD

We consider a system $Ax = b$ of linear equations. It will be assumed throughout this section that A is an $n \times n$ symmetric and positive definite matrix. (If A is invertible but not symmetric, one may apply the methodology of this section to the system $A'Ax = A'b$.) Conjugate direction methods are motivated by a desire to accelerate the speed of convergence of the classical iterative methods for this particular class of problems. While they are guaranteed to find the solution after at most n iterations, they are best viewed as iterative methods, since usually fewer than n iterations are executed, particularly for large problems. These methods are in fact applicable to nonquadratic optimization problems as well. For such problems, they do not, in general, terminate after a finite number of iterations but still, when properly implemented, have attractive convergence and rate of convergence behavior.

For convenience, it will be assumed that $b = 0$. The modifications for the general case will be indicated later. The method is motivated in terms of the cost function

$$F(x) = \frac{1}{2}x'Ax.$$

This function is strictly convex, because of the positive definiteness of A [Prop. A.40(d) in Appendix A], and is minimized at $x = 0$, which is also the unique solution of the system $Ax = 0$.

An iteration of the method has the general form

$$x(t+1) = x(t) + \gamma(t)s(t), \qquad t = 0, 1, \ldots, \tag{7.1}$$

where $s(t) \in \Re^n$ is a direction of update, and $\gamma(t)$ is a scalar stepsize defined by the line minimization

$$F\big(x(t) + \gamma(t)s(t)\big) = \min_{\gamma \in \Re} F\big(x(t) + \gamma s(t)\big). \tag{7.2}$$

The distinguishing feature of this method is the choice of the direction vectors $s(t)$; they are chosen so that they are mutually *A–conjugate*, that is, they have the property

$$s(t)'As(r) = 0, \qquad \text{if } t \neq r.$$

In what follows, we shall employ the notation

$$g(t) = Ax(t) = \nabla F\big(x(t)\big).$$

Some important consequences of conjugacy are the following.

Proposition 7.1. Suppose that $s(0), s(1), \ldots, s(t)$ are nonzero and mutually A–conjugate. Then:

(a) The vectors $s(0), s(1), \ldots, s(t)$ are linearly independent.

(b) We have

$$g(k+1)'s(i) = 0, \qquad \text{if } 0 \leq i \leq k \leq t. \tag{7.3}$$

(c) For $k = 0, 1, \ldots, t+1$, the vector $x(k)$ minimizes F over the linear manifold

$$M_k = \Big\{ x(0) + \alpha_0 s(0) + \cdots + \alpha_{k-1} s(k-1) \;\Big|\; \alpha_0, \ldots, \alpha_{k-1} \in \Re \Big\}.$$

(d) The vectors $x(k)$ satisfy $F\big(x(k+1)\big) \leq F\big(x(k)\big)$ for all $k \leq t$.

Proof.

(a) Suppose the contrary. Then, there exists some $r \leq t$ and some scalars $\alpha_0, \ldots, \alpha_{r-1}$ such that

$$s(r) = \sum_{i=0}^{r-1} \alpha_i s(i).$$

This implies that

$$s(r)'As(r) = \sum_{i=0}^{r-1} \alpha_i s(r)'As(i) = 0$$

which is impossible because $s(r) \neq 0$ and A is positive definite.

(b) By Eq. (7.2) and the chain rule, we have

$$0 = \frac{\partial}{\partial \gamma} F\big(x(k) + \gamma s(k)\big)\Big|_{\gamma=\gamma(k)} = g(k+1)'s(k).$$

Thus, it only remains to prove Eq. (7.3) for $i < k$. We have for $i = 0, 1, \ldots, k-1$,

$$g(k+1)'s(i) = x(k+1)'As(i) = \Big(x(i+1) + \sum_{j=i+1}^{k} \gamma(j)s(j)\Big)'As(i)$$

$$= x(i+1)'As(i) = g(i+1)'s(i) = 0.$$

(c) It suffices to show that the partial derivatives

$$\frac{\partial F}{\partial \alpha_i}\big(x(0) + \alpha_0 s(0) + \cdots + \alpha_{k-1} s(k-1)\big),$$

evaluated at $\alpha_0 = \gamma(0), \ldots, \alpha_{k-1} = \gamma(k-1)$, are equal to zero for each $i \leq k-1$. This is equivalent to $g(k)'s(i) = 0$ for $i = 0, \ldots, k-1$, which is true by part (b).

(d) Since $M_k \subset M_{k+1}$, the result follows from part (c). **Q.E.D.**

2.7.1 Description of the Algorithm

We now describe the most important way of generating the conjugate directions $s(t)$. We start at some $x(0)$ and select $s(0) = -g(0) = -Ax(0)$. More generally, given the current vector $x(t)$, we evaluate the gradient $g(t) = Ax(t)$. If $g(t) = 0$, then $x(t) = 0$ and the algorithm terminates. Otherwise, a reasonable direction of update could be $s(t) = -g(t)$, which is a steepest descent direction [compare with Eq. (4.9) in Section 2.4; see also Section 3.2]. However, in general, this choice does not guarantee conjugacy. This suggests that we generate $s(t)$ according to the formula

$$s(t) = -g(t) + \sum_{i=0}^{t-1} c_i s(i), \tag{7.4}$$

where the coefficients c_i are chosen so that $s(t)$ is conjugate to $s(0), \ldots, s(t-1)$. Assuming that $s(0), \ldots, s(t-1)$ are already mutually conjugate, we need

$$0 = s(t)'As(j) = -g(t)'As(j) + \sum_{i=0}^{t-1} c_i s(i)'As(j) = -g(t)'As(j) + c_j s(j)'As(j),$$

$$j = 0, \ldots, t-1,$$

which gives

$$c_j = \frac{g(t)'As(j)}{s(j)'As(j)}. \tag{7.5}$$

Notice that

$$g(j+1) - g(j) = A\big(x(j+1) - x(j)\big) = \gamma(j)As(j),$$

and Eq. (7.5) becomes

$$c_j = \frac{g(t)'\big(g(j+1) - g(j)\big)}{s(j)'\big(g(j+1) - g(j)\big)}. \tag{7.6}$$

(This step is legitimate provided that $\gamma(j) \neq 0$, and will be justified later.) Equation (7.4) shows that $g(j)$ is a linear combination of $s(0), \ldots, s(j)$, and, using Prop. 7.1(b), we obtain $g(t)'g(j) = 0$ for $j = 0, \ldots, t-1$. We conclude that $c_j = 0$ if $j < t-1$, and, from Eq. (7.4), the conjugate direction $s(t)$ is given as a linear combination of the gradient $g(t)$ and the previous conjugate direction $s(t-1)$, that is,

$$s(t) = -g(t) + \beta(t)s(t-1), \tag{7.7}$$

where

$$\beta(t) = \frac{g(t)'g(t)}{s(t-1)'\big(g(t)-g(t-1)\big)}. \tag{7.8}$$

An alternative expression for $\beta(t)$ is obtained by rewriting the denominator in Eq. (7.8) as

$$s(t-1)'\big(g(t)-g(t-1)\big) = \big(-g(t-1)+\beta(t-1)s(t-2)\big)'\big(g(t)-g(t-1)\big) = g(t-1)'g(t-1).$$

[In the last step, we have used Prop. 7.1(b), together with the fact that $g(t-1)$ is a linear combination of $s(0), \ldots, s(t-1)$.] This leads to

$$\beta(t) = \frac{g(t)'g(t)}{g(t-1)'g(t-1)}. \tag{7.9}$$

The next vector $x(t+1)$ is determined by $x(t+1) = x(t) + \gamma(t)s(t)$, where $\gamma(t)$ is defined by line minimization as in Eq. (7.2). A formula for $\gamma(t)$ is found by setting to zero the derivative (with respect to γ) of the function minimized in Eq. (7.2). Using the chain rule, we obtain

$$s(t)'A\big(x(t)+\gamma(t)s(t)\big) = 0,$$

which leads to

$$\gamma(t) = -\frac{s(t)'g(t)}{s(t)'As(t)}. \tag{7.10}$$

The conjugate direction method based on Eqs. (7.7), (7.9), and (7.10) is called the *conjugate gradient algorithm.* We will assume that the algorithm is terminated at the first time t such that $g(t) = 0$. We now show that the algorithm is well–defined, and, for this, we need to show that, until termination occurs, no division by zero is attempted in Eqs. (7.9) and (7.10). This is clearly true for Eq. (7.9). Concerning Eq. (7.10), since the matrix A is positive definite, the denominator will be equal to zero only if $s(t) = 0$. We therefore need to show that before termination [that is, if $g(0), \ldots, g(t)$ are nonzero], we have $s(t) \neq 0$. We proceed by induction. If $g(0) \neq 0$, then $s(0) \neq 0$. If $g(0), \ldots, g(t)$ are nonzero, we use the induction hypothesis that $s(t-1) \neq 0$ and we show that $s(t) \neq 0$. Indeed, if $s(t)$ were equal to zero, then $g(t)$ would be collinear with $s(t-1)$, because of Eq. (7.7). On the other hand, $g(t)$ is orthogonal to $s(t-1)$ [Prop. 7.1(b)] and, since $s(t-1) \neq 0$, we must have $g(t) = 0$, which is a contradiction.

We prove next that $\gamma(t)$ is nonzero unless the algorithm has terminated, which is needed to justify the step that led us from Eq. (7.5) to Eq. (7.6). In particular, we demonstrate that if $x(t) \neq 0$, then $\gamma(t) \neq 0$. For $t = 0$, we have $s(0) = -g(0)$. If

$x(0) \neq 0$ then $g(0) \neq 0$ and Eq. (7.10) shows that $\gamma(0) \neq 0$. Suppose now that $t \geq 1$, $x(t) \neq 0$, and $\gamma(t) = 0$. Equation (7.10) yields $s(t)'g(t) = 0$. We form the inner product of both sides of Eq. (7.7) with $g(t)$ to obtain

$$0 = s(t)'g(t) = -g(t)'g(t) + \beta(t)g(t)'s(t-1) = -g(t)'g(t),$$

where the last equality followed from Prop. 7.1(b). Thus, $g(t) = 0$, which contradicts our assumption that $x(t) \neq 0$.

We close this subsection by pointing out that Eqs. (7.7), (7.9), and (7.10) also define the conjugate gradient method for the case of a linear system $Ax = b$ when $b \neq 0$. The only difference is that $g(t)$ should be defined to be equal to $Ax(t) - b$. This can be demonstrated by repeating the previous arguments in terms of the cost function $F(x) = (1/2)(x - x^*)'A(x - x^*)$, where $x^* = A^{-1}b$.

2.7.2 Speed of Convergence

We now show that the conjugate gradient method terminates after at most n iterations, with $x(n) = 0$. Furthermore, the second part of the result to follow will lead us to interesting bounds on the speed of convergence when the number of iterations is smaller than n. For this purpose, it is convenient to introduce the linear manifold

$$H_k = \Big\{x(0) + \alpha_1 Ax(0) + \cdots + \alpha_k A^k x(0) \mid \alpha_1, \ldots, \alpha_k \in \Re\Big\}.$$

Proposition 7.2. For the conjugate gradient method, the following hold:

(a) The algorithm terminates after at most n steps; that is, there exists some $t \leq n$ such that $g(t) = 0$ and $x(t) = 0$.

(b) If $s(0), \ldots, s(t-1)$ are all nonzero, then $M_t = H_t$, where M_t is as in Prop. 7.1.

Proof.

(a) If the algorithm does not terminate after at most n steps, then $g(0), \ldots, g(n)$ are all nonzero, and, as shown earlier, the vectors $s(0), \ldots, s(n)$ are also nonzero and, therefore, linearly independent [Prop. 7.1(a)]. This is impossible since they belong to the n–dimensional vector space $\Re^n$. At termination we have $g(t) = 0$ and the equality $x(t) = 0$ follows because $g(t) = Ax(t)$ and A is nonsingular.

(b) Let $\bar{M}_t$ (respectively, $\bar{H}_t$) be the subspace of $\Re^n$ spanned by the collection of vectors $\{s(0), \ldots, s(t-1)\}$ [respectively, $\{Ax(0), \ldots, A^t x(0)\}$]. It is sufficient to show that $\bar{H}_t = \bar{M}_t$, and we proceed by induction. The result is true for $t = 1$ because $s(0) = -Ax(0)$. Suppose that $\bar{M}_{t-1} = \bar{H}_{t-1}$. Using Eq. (7.1), we see that $x(t-1) - x(0)$ is a linear combination of $s(0), \ldots, s(t-2)$ and therefore belongs to $\bar{M}_{t-1}$. Using the induction hypothesis, we obtain $x(t-1) - x(0) \in \bar{H}_{t-1}$. We then use the definition of $\bar{H}_t$ to see that $Ax(t-1) - Ax(0) \in \bar{H}_t$. Since $Ax(0) \in \bar{H}_t$, we conclude that $g(t-1) = -Ax(t-1) \in \bar{H}_t$. Furthermore, by the induction

hypothesis, $s(t-2) \in \bar{M}_{t-1} = \bar{H}_{t-1} \subset \bar{H}_t$. Equation (7.7) then implies that $s(t-1) \in \bar{H}_t$. We conclude that $\bar{M}_t \subset \bar{H}_t$. It remains to show that $\bar{M}_t$ cannot be a proper subset of $\bar{H}_t$. This follows because the vectors $s(0), \ldots, s(t-1)$ are linearly independent and the dimension of $\bar{M}_t$ is t, whereas the dimension of H_t cannot be more than t, since it is spanned by t vectors. **Q.E.D.**

An important consequence of Props. 7.2(b) and 7.1(c) is that $x(t)$ minimizes $F(x)$ over the set H_t. By definition, H_t is the set of all vectors of the form $P(A)x(0)$, where P is a t–degree polynomial whose zero-th order term is equal to 1. Let $\mathcal{P}$ be the class of all such polynomials. We have

$$F\big(x(t)\big) \leq \frac{1}{2} x(0)' P(A)' A P(A) x(0), \qquad \forall P \in \mathcal{P}.$$

Let $A^{1/2}$ be a square root of A, as defined in Prop. A.27 in Appendix A. Since $A^{1/2}$ commutes with A, we obtain for every $P \in \mathcal{P}$,

$$\begin{aligned} F\big(x(t)\big) &\leq \frac{1}{2} \Big(P(A)A^{1/2}x(0)\Big)' \Big(P(A)A^{1/2}x(0)\Big) = \frac{1}{2}\big\|P(A)A^{1/2}x(0)\big\|_2^2 \\ &\leq \frac{1}{2}\|P(A)\|_2^2 \cdot \|A^{1/2}x(0)\|_2^2 = \frac{1}{2}\|P(A)\|_2^2 \cdot \big(x(0)'Ax(0)\big) = \|P(A)\|_2^2 F\big(x(0)\big). \end{aligned}$$

Let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of A. Then, the eigenvalues of $P(A)$ are equal to $P(\lambda_1), \ldots, P(\lambda_n)$ [this is proved similarly with Prop. A.17(d) in Appendix A]. Since A is symmetric, $P(A)$ is also symmetric and $\|P(A)\|_2$ is equal to the largest magnitude of the eigenvalues of $P(A)$ [Prop. A.24(a) in Appendix A]. We therefore conclude that

$$F\big(x(t)\big) \leq \max_{1 \leq i \leq n} \big(P(\lambda_i)\big)^2 F\big(x(0)\big), \qquad \forall P \in \mathcal{P}. \tag{7.11}$$

Inequality (7.11) provides an infinite class of bounds for $F\big(x(t)\big)$, parametrized by the polynomial $P(\cdot)$. If some prior knowledge on the location of the eigenvalues of A is available, interesting bounds are obtained by choosing the polynomial $P(\cdot)$ appropriately. As an example, assume that there exist some a and b, with $0 < a < b$, such that A has no eigenvalues smaller than a, and those eigenvalues that are larger than b take k distinct values. It can be shown (Exercise 7.1) that for every $x(0)$, the vector $x(k+1)$ satisfies

$$F\big(x(k+1)\big) \leq \left(\frac{b-a}{b+a}\right)^2 F\big(x(0)\big). \tag{7.12}$$

This shows that the method converges fast if most of the eigenvalues of A are clustered in a small interval and the remaining eigenvalues lie to the right of the interval. Another consequence is that if the eigenvalues of A take at most k distinct values, then the conjugate gradient method will find the minimum of F after at most k iterations. To

see this, take the interval $[a, b]$ to be an arbitrarily small interval around the smallest eigenvalue of A.

2.7.3 Preconditioned Conjugate Gradient Method

This method is really the conjugate gradient method carried out in a new coordinate system. Let T be a symmetric invertible matrix and consider the system of equations $TATy = Tb$. If y solves the latter system, the vector $x = Ty$ vector solves the original system $Ax = b$. We will assume again that $b = 0$ and apply the conjugate gradient method to the system $TATy = 0$. [The same equations are valid even if $b \neq 0$, provided that $g(t)$ is defined to be equal to $Ax(t) - b$.] The method is described by [compare with Eqs. (7.1), (7.7), (7.9) and (7.10)]

$$y(t+1) = y(t) + \gamma(t)\tilde{s}(t), \tag{7.13}$$

where $\tilde{s}(t)$ is generated by

$$\tilde{s}(0) = -TATy(0), \tag{7.14}$$

$$\tilde{s}(t) = -TATy(t) + \beta(t)\tilde{s}(t-1), \qquad t = 1, 2, \ldots, \tag{7.15}$$

and where

$$\beta(t) = \frac{\big(TATy(t)\big)'\big(TATy(t)\big)}{\big(TATy(t-1)\big)'\big(TATy(t-1)\big)}, \tag{7.16}$$

$$\gamma(t) = -\frac{\tilde{s}(t)'TATy(t)}{\tilde{s}(t)'TAT\tilde{s}(t)}. \tag{7.17}$$

Setting $x(t) = Ty(t)$, $g(t) = Ax(t)$, $s(t) = T\tilde{s}(t)$ and, $H = T^2$, we obtain from Eqs. (7.13) to (7.17) the equivalent method

$$x(t+1) = x(t) + \gamma(t)s(t),$$

$$s(0) = -Hg(0), \qquad s(t) = -Hg(t) + \beta(t)s(t-1), \quad t = 1, 2, \ldots,$$

where

$$\beta(t) = \frac{g(t)'Hg(t)}{g(t-1)'Hg(t-1)}, \qquad \gamma(t) = -\frac{s(t)'g(t)}{s(t)'As(t)}.$$

Notice that the algorithm can be carried out without having to compute the matrix product TAT. Our earlier results guarantee that the algorithm converges in n iterations. Concerning the rate of convergence, inequality (7.11) is again applicable, except that the eigenvalues of $TAT = H^{1/2}AH^{1/2}$ are involved, replacing the eigenvalues of A.

For this reason, preconditioning may substantially enhance the speed of convergence, although finding a good choice of the scaling matrix H remains mostly an art.

2.7.4 Parallel Implementation

For simplicity, we only discuss the case of no preconditioning. Assuming that at the beginning of the tth iteration, $x(t)$, $s(t-1)$, and $g(t-1)'g(t-1)$ have already been computed, we need to evaluate the vectors $g(t) = Ax(t)$, the inner product $g(t)'g(t)$, determine $\beta(t)$ and $s(t)$ using Eqs. (7.9) and (7.7), then evaluate $As(t)$, and finally compute the inner products $s(t)'\big(As(t)\big)$, $s(t)'g(t)$, from which $\gamma(t)$ is determined [cf. Eq. (7.10)]. Neglecting vector additions and scalar–vector multiplications, the main computational requirements are two matrix–vector multiplications and three inner product computations. Furthermore, in all matrix–vector multiplications, the same matrix A is involved.

We only discuss the case of message–passing architectures. If n processors are available, it is natural to let the ith processor be in control of the ith component of the vectors of interest, that is, $x(t)$, $s(t)$, and $g(t)$. Inner products of such vectors are computed by letting the ith processor compute the product of the ith components and then accumulating partial sums along a tree of processors. This is a single node accumulation and takes time proportional to the diameter of the interconnection network. Then, the computed values of the inner products are broadcast to all processors. We now assume that each processor is given the entries in a different row of A. Then, a matrix–vector product like $Ax(t)$ may be computed by broadcasting the vector $x(t)$ (this is a multinode broadcast) and having the ith processor compute the inner product of x with the ith row of A. Alternatively, the ith processor could compute $[A]_{ji}x_i$ for each j, and these quantities could be propagated to each processor j, with partial sums formed along the way, as discussed in Subsection 1.3.6.† If fewer than n processors are available, the issues involved are the same except that there are more components, and more rows of the matrix A, assigned to each processor.

In the case where A is a sparse matrix, the multiplication of any vector by A may be performed more efficiently by using a special interconnection topology that exploits the sparsity structure of A. For example, if A arises from the discretization of a partial differential equation, a mesh–connected processor array is suitable, and the required matrix–vector multiplications can be executed in $O(1)$ time. Unfortunately, the speed of inner product computations is limited by the diameter of the interconnection network. For example, in an $n^{1/2} \times n^{1/2}$ mesh–connected array, $\Omega(n^{1/2})$ time units are required to evaluate an inner product $x'y$ when each component of x and y lies at a different processor. This creates a bottleneck that can be alleviated if there are some special hardware facilities (e.g., additional connections) allowing quick evaluation of inner products. Another option for reducing the communication penalty and increasing efficiency is to use fewer processors and to assign enough components to each processor

† Notice that there is no significant difference between row and column storage schemes for matrix–vector multiplication because the matrix A is symmetric.

so that the number of computations of each processor, per stage, is comparable to the diameter of the interconnection network. This constrains the number of processors that can be efficiently employed and limits the attainable speedup (Exercise 7.3).

EXERCISES

7.1. **[Lue84]** Prove inequality (7.12). *Hint:* Use the polynomial

$$P(\lambda) = \frac{2}{(a+b)\lambda_1 \cdots \lambda_k} \left(\frac{a+b}{2} - \lambda\right)(\lambda_1 - \lambda) \cdots (\lambda_k - \lambda),$$

where $\lambda_1, \ldots, \lambda_k$ are the values of the eigenvalues of A that are larger than b.

7.2. **[Ber74]** Let A be of the form

$$A = M + \sum_{i=1}^{k} v_i v_i',$$

where M is positive definite and symmetric and $v_1, \ldots, v_k$ are some vectors in $\Re^n$. Show that the preconditioned conjugate gradient method with $H = M^{-1}$ terminates in at most $k+1$ steps.

7.3. Consider the discretized Poisson equation (Subsection 2.5.1) for a square domain with N grid points. Suppose that a two–dimensional mesh of p processors is used to execute the conjugate gradient algorithm, with each processor assigned N/p grid points in a square subdomain. Find the order of magnitude dependence of p on N that optimizes the execution time of each iteration. Assume that the time for one computation and the communication delay across any link are comparable.

7.4. Suppose that the conjugate gradient has not terminated at the kth iteration, that is, $x(k) \neq 0$. Show that $F\big(x(k+1)\big) < F\big(x(k)\big)$. [This strengthens Prop. 7.1(d).]

2.8 COMPUTATION OF THE INVARIANT DISTRIBUTION OF A MARKOV CHAIN

Markov chains are widely used as probabilistic models of stochastic systems, in queueing theory, for example. In applications, one often deals with Markov chains with very large state spaces, involving tens of thousands of states. It is often desired to compute the steady–state (invariant) probability distribution for such chains, and this is a computational task that calls for parallel computation. We present two variants of an easily parallelizable iterative algorithm.

Let P be the one–step transition probability matrix of a discrete–time homogeneous n–state Markov chain. The reader may wish to consult Appendix D for the relevant definitions. For the purposes of this section, we only need the following two properties of P:

$$P \geq 0, \tag{8.1}$$

$$\sum_{j=1}^{n} p_{ij} = 1. \tag{8.2}$$

Any matrix with these two properties is called a *stochastic* matrix.

A nonnegative row vector π^* whose components sum to 1, and that has the property $\pi^* = \pi^* P$ is called an *invariant distribution* of the Markov chain associated with P; the computation of such a vector π^* is the subject of this section.

Consider the following algorithm. We start with a row vector $\pi(0) \geq 0$ whose components add to 1, and we employ the iteration

$$\pi(t+1) = \pi(t)P. \tag{8.3}$$

Equivalently,

$$\pi(t) = \pi(0)P^t, \qquad t \geq 0.$$

We present below some conditions under which this iteration converges to an invariant distribution π^*. We first recall some definitions from Appendix D.

Given an $n \times n$ stochastic matrix P, we form a directed graph $G = (N, A)$, where N is the set of states, and $A = \{(i,j) \mid p_{ij} > 0,\ i \neq j\}$, the set of arcs, is the set of all transitions that have positive probability. We say that P is *irreducible* if for every $i, j \in N$ there exists a positive path in the above graph from i to j, or if $n = 1$. Notice that this is the same definition as the one given in Section 2.6.

The stochastic matrix P is called *periodic* if there exists some $k > 1$ and some disjoint nonempty subsets $N_0, \ldots, N_{k-1}$ of the state space N such that if $i \in N_\ell$ and $p_{ij} > 0$, then $j \in N_{\ell+1 (\mathrm{mod}\ k)}$. We say that P is *aperiodic* if it is not periodic. Finally, P is called *primitive* if there exists a positive integer t such that $P^t > 0$. Some examples are provided in Fig. 2.8.1.

Proposition 8.1. A stochastic matrix P is primitive if and only if it is irreducible and aperiodic.

Proposition 8.1 is a well–known result in the theory of Markov chains [Var62]. We omit its proof because it will not be used. Instead, it will be assumed that we are dealing with a primitive matrix.

Proposition 8.2. Let P be a stochastic matrix.

(a) The spectral radius $\rho(P)$ of P is equal to 1.

(b) If π is a row vector whose entries sum to 1, then the row vector πP has the same property.

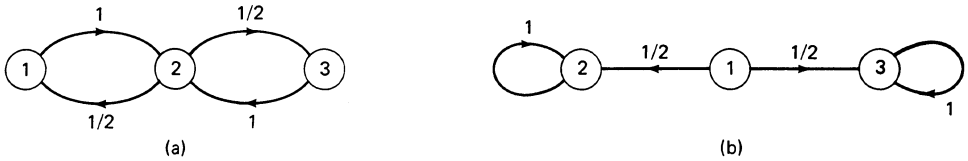


Figure 2.8.1 Some examples of Markov chains. For the Markov chain in (a), we have

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1/2 & 0 & 1/2 \\ 0 & 1 & 0 \end{bmatrix}$$

and it is easily seen that $P^t = P$ if $t > 0$ is odd, and

$$P^t = \begin{bmatrix} 1/2 & 0 & 1/2 \\ 0 & 1 & 0 \\ 1/2 & 0 & 1/2 \end{bmatrix}$$

if $t > 0$ is even. In particular, P^t does not converge. The matrix P is irreducible but neither primitive nor aperiodic. For the Markov chain in (b), we have

$$P^t = P = \begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad \forall t \geq 1.$$

Notice that P is not irreducible but is aperiodic. The sequence $\{P^t\}$ converges, since it is constant, but the limits of different rows are different.

Proof.

(a) Let e be the column vector with all components equal to 1. From Eq. (8.2), we obtain $Pe = e$. Thus, 1 is an eigenvalue of P and $\rho(P) \geq 1$. On the other hand, $\rho(P) \leq \|P\|_\infty = 1$.

(b) We have

$$\sum_{i=1}^{n} [\pi P]_i = \sum_{i=1}^{n} \sum_{k=1}^{n} \pi_k p_{ki} = \sum_{k=1}^{n} \pi_k \sum_{i=1}^{n} p_{ki} = \sum_{k=1}^{n} \pi_k = 1.$$

Q.E.D.

Convergence of the iteration $\pi(t+1) = \pi(t)P$ is obtained from the following result. The examples in Fig. 2.8.1 illustrate the ways in which Prop. 8.3 fails to hold if P is not primitive.

Proposition 8.3. Let P be a primitive stochastic matrix. Then:

(a) There exists a unique row vector π^* such that $\pi^* = \pi^* P$ and $\sum_{i=1}^n \pi_i^* = 1$. Furthermore, $\pi^* > 0$.

(b) The limit of P^t, as t tends to infinity, exists and is the matrix with all rows equal to π^*.

(c) If $\sum_{i=1}^n \pi_i(0) = 1$, then the iteration $\pi(t+1) = \pi(t)P$ converges to π^*.

Proof.

(a) Since P is primitive, it is irreducible. (This follows from the unproved Prop. 8.1, but is also a straightforward consequence of Prop. 6.3.) It then follows that P' is also irreducible. Since the eigenvalues of P' coincide with the eigenvalues of P [Prop. A.17(f) in Appendix A], we use Prop. 8.2(a) to obtain $\rho(P') = 1$. We then apply the Perron–Frobenius theorem (Prop. 6.6) to P' to assert the existence of a positive vector w such that $P'w = w$; equivalently, $w'P = w'$. The existence of π^* follows by normalizing w' so that its components sum to 1. Uniqueness follows from the uniqueness result in Prop. 6.6.

(b) Fix some $x \in \Re^n$. Let $M_t(x) = \max_i [P^t x]_i$ and $m_t(x) = \min_i [P^t x]_i$. The fact $\sum_{j=1}^n p_{ij} = 1$ and an easy induction show that $M_t(x)$ is nonincreasing and $m_t(x)$ is nondecreasing. They must therefore converge to some limits denoted by $M(x)$, $m(x)$.

Let T be such that $P^T > 0$, and let α be the smallest of the entries of P^T. Since the sum of the entries in each row of P^T is equal to 1 (Prop. D.3 in Appendix D), it is easily seen that

$$M_{t+T}(x) \leq (1-\alpha)M_t(x) + \alpha m_t(x),$$

and

$$m_{t+T}(x) \geq (1-\alpha)m_t(x) + \alpha M_t(x).$$

Subtracting these two inequalities, we obtain

$$M_{t+T}(x) - m_{t+T}(x) \leq (1-2\alpha)\big(M_t(x) - m_t(x)\big).$$

Taking the limit, as t tends to infinity, and using the positivity of α, we obtain $M(x) = m(x)$. It follows that $P^t x$ converges to $m(x)e$, where e is the vector with all components equal to 1. Let $e^1, \ldots, e^n$ be the unit vectors in $\Re^n$. Letting $x = e^1, \ldots, e^n$, we conclude that $P^t = P^t I = P^t[e^1\ e^2\ \ldots\ e^n]$ converges to $[m(e^1)e\ \ldots\ m(e^n)e]$, which is a matrix with all rows equal to the row vector $y = [m(e^1)\ldots m(e^n)]$. We now need to show that $y = \pi^*$. Since $Pe = e$, we

have $P^t e = e$ for all t, and $\lim_{t\to\infty} P^t\left(\sum_{i=1}^n e^i\right) = \lim_{t\to\infty} P^t e = e$. Therefore, $\sum_{i=1}^n y_i = 1$. Finally, since $\lim_{t\to\infty} P^t = (\lim_{t\to\infty} P^t)P$, it follows that $y = yP$. Using the uniqueness result of part (a), we obtain $y = \pi^*$.

(c) By the result of part (b), all of the entries in the ith column of the limit of P^t are equal to π_i^*. It follows that the ith entry of $\pi(0)P^t$ converges to $\sum_{j=1}^n \pi_j(0)\pi_i^* = \pi_i^*$. **Q.E.D.**

The iteration $\pi := \pi P$ admits a straightforward parallel implementation in which the ith processor is assigned the task of updating the ith component of the vector π and at each stage communicates its newly computed value to every other processor j such that $p_{ij} \neq 0$. This assumes that each processor j knows the entries in the jth column of P. A different implementation is obtained if each processor j knows the entries of the jth row of P; see the discussion in Subsection 1.3.6. An asynchronous version of this algorithm will be studied in Section 7.3.

We now consider a variant of the iteration $\pi := \pi P$. The new algorithm is almost the same except that one of the components of π, say the first one, is not iterated upon. Thus, the algorithm is described by

$$\pi_1(t+1) = \pi_1(t), \tag{8.4}$$

$$\pi_i(t+1) = \sum_{j=1}^n \pi_j(t)p_{ji}, \qquad i = 2, \ldots, n. \tag{8.5}$$

The initialization of the algorithm is arbitrary, provided that $\pi_1(0) \neq 0$. In order to represent the algorithm in matrix form, we partition the matrix P as shown:

$$P = \begin{bmatrix} p_{11} & a \\ b & \tilde{P} \end{bmatrix}.$$

Here, a (respectively, b) is a row (respectively, column) vector of dimension $n-1$ and $\tilde{P}$ is the matrix of dimensions $(n-1)\times(n-1)$ obtained by deleting the first row and the first column of P. Let $\tilde{\pi}(t)$ be the row vector $\big(\pi_2(t), \ldots, \pi_n(t)\big)$. Then, Eq. (8.5) can be rewritten as

$$\tilde{\pi}(t+1) = \tilde{\pi}(t)\tilde{P} + \pi_1(0)a. \tag{8.6}$$

This iteration converges provided that $\rho(\tilde{P}) < 1$ (Prop. 6.1). The following result provides conditions for this to be the case and characterizes the limit of $\pi(t)$.

Proposition 8.4. Consider the directed graph associated with P, and assume that there exists a positive path from every state to state 1. Let $\{X(t) \mid t = 0, 1, \ldots\}$ be a Markov chain whose one–step transition probabilities are given by P. Let T be a positive integer and let

$$\delta_T = \min_{i=2,\ldots,n} \Pr\Big(\text{there exists some } \tau \leq T \text{ such that } X(\tau) = 1 \;\Big|\; X(0) = i\Big). \tag{8.7}$$

(a) If T is chosen large enough, then $\delta_T > 0$.

(b) If $\delta_T > 0$, then

$$\rho(\tilde{P}) \leq \left(\|\tilde{P}^T\|_\infty\right)^{1/T} \leq (1 - \delta_T)^{1/T} < 1.$$

(c) The sequence $\{\pi(t)\}$ generated by Eqs. (8.4) and (8.5) converges to a vector π^* satisfying $\pi^* = \pi^* P$. If $\pi_1(0)$ is positive and $\pi(0) \geq 0$, then π^* is nonzero and all of its entries are nonnegative.

Proof.

(a) The positivity of δ_T is a straightforward consequence of the fact that for each $i \neq 1$, there exists a sequence of positive probability transitions leading from i to 1. We simply need to take T large enough so that for each $i \neq 1$, there exists at least one such path that uses no more than T arcs.

(b) Let

$$Q = \begin{bmatrix} 1 & 0 \\ b & \tilde{P} \end{bmatrix},$$

which is easily seen to be a stochastic matrix, and let $\{Y(t) \mid t = 0, 1, \ldots\}$ be an associated Markov chain. We notice that $Y(t)$ has the same transition probabilities with $X(t)$ except that state 1 is an *absorbing* state: once $Y(t)$ becomes 1, it never changes. It follows that

$$\delta_T \leq \Pr\Big(\text{there exists some } \tau \leq T \text{ such that } Y(\tau) = 1 \Big| Y(0) = i\Big) = [Q^T]_{i1}, \qquad i = 2, \ldots, n. \tag{8.8}$$

We now notice that Q^T is of the form

$$Q^T = \begin{bmatrix} 1 & 0 \\ c & \tilde{P}^T \end{bmatrix}, \tag{8.9}$$

where c is an $(n-1)$–dimensional column vector with all entries bounded below by δ_T. Since Q is a stochastic matrix, so is Q^T and each row sums to 1. It follows from Eq. (8.9) that the sum of the entries in any row of $\tilde{P}^T$ is bounded above by $1 - \delta_T$. Therefore, $\rho(\tilde{P}^T) \leq \|\tilde{P}^T\|_\infty \leq 1 - \delta_T$. The result follows because $\rho(\tilde{P}) = \left(\rho(\tilde{P}^T)\right)^{1/T}$.

(c) Since $\rho(\tilde{P}) < 1$, the system $\tilde{\pi} = \tilde{\pi}\tilde{P} + \pi_1(0)a$ has a unique solution $\tilde{\pi}$ and the sequence $\{\tilde{\pi}(t)\}$ converges to it. It follows that $\pi(t)$ converges to the vector $\pi^* = \big(\pi_1(0), \tilde{\pi}\big)$ and $\tilde{\pi}$ satisfies $\tilde{\pi} = \tilde{\pi}P + \pi_1(0)a$ [cf. Eq. (8.6)]. This shows that $\pi_i^* = [\pi^* P]_i$ for $i \neq 1$, and it remains to show that this equality also holds for $i = 1$. We have $\pi_j^* = \sum_{i=1}^n \pi_i^* p_{ij}$ for $j \neq 1$. Summing this equality for every $j \neq 1$, we obtain

$$\sum_{j=2}^{n} \pi_j^* = \sum_{j=2}^{n} \sum_{i=1}^{n} \pi_i^* p_{ij} = \sum_{i=1}^{n} \pi_i^* (1 - p_{i1})$$

which after cancellations yields $\pi_1^* = \sum_{i=1}^n \pi_i^* p_{i1} = [\pi^* P]_1$, as desired.

Suppose now that $\pi_1(0) > 0$. Then $\pi_1^* = \pi_1(0) > 0$. Also, if $\tilde{\pi}(0)$ is a nonnegative vector, it follows from Eq. (8.5) that $\tilde{\pi}(t)$ is nonnegative for every t, and the same conclusion obtains for π^*, since it is the limit of nonnegative vectors, which concludes the proof. **Q.E.D.**

Notice that Prop. 8.4(b) provides us with an estimate of the convergence rate of the algorithm of Eqs. (8.4) and (8.5). Furthermore, the conditions for convergence are less stringent than those imposed in Prop. 8.3. For example, the Markov chain in Fig 2.8.1(a) satisfies the conditions of Prop. 8.4 but not those of Prop. 8.3. It should be pointed out that the components of the limit vector π^* do not, in general, add to 1. However, this may be remedied by multiplying the vector π^* obtained at termination of the algorithm, by a suitable scalar. The issues concerning the parallelization of the algorithm of Eqs. (8.4) and (8.5) are the same as for the algorithm of Eq. (8.3). Actually, this algorithm is also guaranteed to converge even if it is executed in a Gauss–Seidel fashion, without any deterioration of the convergence rate and with a potential improvement. This is because the iteration matrix $\tilde{P}$ is nonnegative, its spectral radius is less than 1, and Prop. 6.8 applies. This Gauss–Seidel algorithm is not always parallelizable, but in typical large scale Markov chains, the matrix P is very sparse and a coloring scheme can be applied (cf. Subsection 1.2.4). It will be seen in Section 6.3 that the algorithm of Eqs. (8.4) and (8.5) also converges when executed asynchronously.

EXERCISES

8.1. **(Power Method for the Eigenvalue Problem.)** The iteration $\pi := \pi P$ is a special case of a more general algorithm for finding an eigenvalue with largest magnitude of a given matrix A, together with an associated eigenvector. The algorithm is initialized with some $x(0) \neq 0$ and consists of the iteration

$$x(t+1) = \frac{Ax(t)}{\|Ax(t)\|}, \tag{8.10}$$

where $\|\cdot\|$ is an arbitrary norm. Suppose that A has distinct eigenvalues $\lambda_1, \ldots, \lambda_n$ and corresponding nonzero eigenvectors $x^1, \ldots, x^n$. Suppose that $|\lambda_i| < |\lambda_1|$ for each $i \neq 1$ and that the iteration is initialized with some $x(0)$ that does not belong to the span of $x^2, \ldots, x^n$.

(a) Show that the sequence $\{x(t)\}$ has a limit x satisfying $Ax = \lambda_1 x$. *Hint:* Use the Jordan form of the matrix A.

(b) Show that iteration (8.3), with $\pi(0) \geq 0$ and $\sum_{i=1}^{n} \pi_i(0) = 1$, is of the form of Eq. (8.10), for a suitable choice of the norm $\|\cdot\|$.

8.2. This exercise and the next extend Prop. 8.3 to cover all cases where there is only one ergodic class (Appendix D) associated with the matrix P. Let P be an irreducible, but not necessarily primitive, stochastic matrix. Let α be some constant satisfying $0 < \alpha < 1$ and consider the matrix Q defined by $Q = (1 - \alpha)I + \alpha P$.

(a) Show that Q is a primitive stochastic matrix. *Hint:* Use Prop. 6.3.

(b) Show that there exists a row vector $\pi^* > 0$ satisfying $\pi^* P = \pi^*$, and, furthermore, such a vector is unique up to multiplication by a scalar. Suggest an algorithm for computing π^*.

8.3. Suppose that a stochastic matrix P has the structure

$$P = \begin{bmatrix} A & B \\ 0 & C \end{bmatrix},$$

where A, B, and C, are matrices of dimensions $n_1 \times n_1$, $n_1 \times (n - n_1)$, and $(n - n_1) \times (n - n_1)$, respectively. In particular, C is a stochastic matrix and we assume that it is also irreducible. We also assume that starting at any one of the first n_1 states, there is a positive probability that the state of the associated Markov chain becomes equal to one of the last $n - n_1$ states. Show that there exists a unique row vector $\pi^* \geq 0$ such that $\pi^* P = \pi^*$ and whose components sum to 1. *Hint:* To prove uniqueness, use the techique in the proof of Prop. 8.4 to show that $\rho(A) < 1$, conclude that the first n_1 components of π^* are equal to zero, and finally use the irreducibility of C.

8.4. Let P be an irreducible stochastic matrix and suppose that there exists some i such that $p_{ii} > 0$. Show that P is primitive.

2.9 FAST ITERATIVE MATRIX INVERSION

We consider here a method based on the classical Newton algorithm for iterative improvement of an approximate inverse of a square matrix A, assumed throughout to be invertible. This method is motivated by the desire for very small execution time $[O(\log^2 n)]$ without the drawbacks of the direct inversion algorithm of Section 2.3 (excessive number of processors and lack of numerical robustness). Small execution time rests on a distinctive property of Newton methods, *quadratic convergence*, that is, convergence at the same speed as the sequence $\{\rho^{2^t}\}$, where ρ is a positive constant smaller than 1. This is much faster than the geometric convergence rate $(\{\rho^t\})$ exhibited by the classical iterative methods analyzed in Section 2.6.

Given a square matrix B of the same dimensions as A, we define a residual matrix $R(B) = I - BA$. Thus, $R(B)$ measures, in some sense, how far B is from being the inverse of A. Consider the following algorithm:

1. Start with some B_0 such that $\|I - B_0A\|_2 < 1$.
2. Iteratively improve B_k by letting $B_{k+1} = \big(I + R(B_k)\big)B_k = 2B_k - B_kAB_k$.

This iteration can be interpreted as Newton's method for solving the equation $X^{-1} - A = 0$ (see Exercise 9.1).

Some algebraic manipulation gives

$$\begin{aligned} R(B_{k+1}) &= I - B_{k+1}A = I - \big(I + R(B_k)\big)B_kA \\ &= (I - B_kA) - R(B_k)B_kA = R(B_k)\big(I - B_kA\big) = \big(R(B_k)\big)^2. \end{aligned}$$

We therefore obtain $R(B_k) = \big(R(B_0)\big)^{2^k}$. Using Prop. A.12(c) in Appendix A, we have

$$\big\|R(B_k)\big\|_2 = \Big\|\big(R(B_0)\big)^{2^k}\Big\|_2 \le \|R(B_0)\|_2^{2^k}. \tag{9.1}$$

Since we have assumed that $\|R(B_0)\| < 1$, the previous inequality shows that the norm of $R(B_k)$ converges very rapidly to zero; equivalently, B_k converges very rapidly to A^{-1}. So, the algorithm will be succesful provided that B_0 has been suitably chosen. The following choice of B_0 turns out to be convenient:

$$B_0 = \frac{A'}{\operatorname{tr}(A'A)}, \tag{9.2}$$

where $\operatorname{tr}(A'A)$, the *trace* of $A'A$, is defined as the sum of the diagonal entries of $A'A$. Other choices of B_0, some of them suitable in special cases, are given in Exercises 9.2 to 9.4.

For any nonsingular square matrix A, we let $\kappa(A) = \|A\|_2 \cdot \|A^{-1}\|_2$. This quantity is called the *condition number* of A and plays a prominent role in numerical analysis, as a measure of the difficulty of computing A^{-1} in the face of roundoff errors [GoV83]. Let $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$ be the eigenvalues of $A'A$. These eigenvalues are real and nonnegative because the matrix $A'A$ is symmetric nonnegative definite (Prop. A.26 in Appendix A). Furthermore, they are nonzero because A is assumed nonsingular. Another definition of the condition number is given by $\kappa(A) = (\lambda_n/\lambda_1)^{1/2}$. To see that the two definitions are equivalent, notice that $\lambda_n = \rho(A'A) = \|A'A\|_2 = \|A\|_2^2$ [Props. A.24(a) and A.25(d) in Appendix A], and that $1/\lambda_1 = \|(A'A)^{-1}\|_2 = \|A^{-1}(A^{-1})'\|_2 = \|A^{-1}\|_2^2$ [Props. A.25(f) and A.25(d) in Appendix A].

Proposition 9.1. If $B_0 = A'/\operatorname{tr}(A'A)$, then $\|I - B_0A\|_2 \le 1 - 1/\big(n\kappa^2(A)\big)$.

Proof. We have $\operatorname{tr}(A'A) = \lambda_1 + \cdots + \lambda_n$ [Prop. A.22(a) in Appendix A]. It follows that the ith eigenvalue ρ_i of $I - B_0A = I - A'A/\operatorname{tr}(A'A)$ is equal to $1 - \lambda_i/(\lambda_1 + \cdots + \lambda_n)$. Since $\lambda_i \le \lambda_1 + \cdots + \lambda_n$ for each i, we see that $\rho_i \ge 0$ for each i. Furthermore, since $\lambda_1 + \cdots + \lambda_n \le n\lambda_n$, we obtain

$$\rho_i \leq 1 - \frac{\lambda_i}{n\lambda_n} \leq 1 - \frac{\lambda_1}{n\lambda_n} = 1 - \frac{1}{n\kappa^2(A)},$$

for each i. It follows that $\rho(I - B_0A) = \max_i |\rho_i| \leq 1 - 1/\big(n\kappa^2(A)\big)$. The desired result follows because $I - B_0A$ is symmetric and $\|I - B_0A\|_2 = \rho(I - B_0A)$ [Prop. A.24(a) in Appendix A]. **Q.E.D.**

The above proposition shows that if B_0 is chosen according to Eq. (9.2), then $\|I - B_0A\|_2 < 1$. Furthermore, as long as the matrix A is not extremely ill–conditioned, $\|I - B_0A\|_2$ is sufficiently smaller than 1 to lead to a practical algorithm. In particular, we have the following corollary of Prop. 9.1.

Corollary 9.1. Suppose that $\kappa(A) \leq n^d$ for some constant d, and let c be any positive integer. If $B_0 = A'/\mathrm{tr}(A'A)$, then the matrix B_k produced after $k \geq (c + 2d + 1)\log n$ iterations of the algorithm satisfies

$$\|I - B_kA\|_2 \leq 2^{-n^c}, \tag{9.3}$$

$$\|B_k - A^{-1}\|_2 \leq 2^{-n^c}\,\frac{\kappa(A)}{\|A\|_2}. \tag{9.4}$$

Furthermore, such a B_k can be computed in parallel in time $O\big((c + d)\log^2 n\big)$ using n^3 processors.

Proof. The time and processor bounds are obvious, since each iteration involves two matrix multiplications, which can be performed in time $O(\log n)$, using n^3 processors (see Subsection 1.2.3). [The computation of $\mathrm{tr}(A'A)$ is of no concern: it can be performed in time $O(\log n)$ using n^3 processors.]

Using Prop. 9.1 and Eq. (9.1),

$$\|I - B_kA\|_2 \leq \left(1 - \frac{1}{n\kappa^2(A)}\right)^{n^{c+2d+1}} \leq \left(1 - \frac{1}{n\kappa^2(A)}\right)^{(n\kappa^2(A))n^c} \leq 2^{-n^c},$$

which proves Eq. (9.3). [The last inequality uses the bound $(1 - 1/\alpha)^\alpha \leq 1/2$, for $\alpha \geq 2$.] Finally,

$$\|B_k - A^{-1}\|_2 = \|(B_kA - I)A^{-1}\|_2 \leq \|I - B_kA\|_2 \cdot \|A^{-1}\|_2 = \|I - B_kA\|_2 \frac{\kappa(A)}{\|A\|_2},$$

which completes the proof. **Q.E.D.**

Corollary 9.1 shows that, for all practical purposes, the computation time is a small multiple of $\log^2 n$. This is the same as for the direct algorithm of Section 2.3, except that only n^3 processors are used here as opposed to n^4 processors in Section 2.3. Furthermore, unlike the algorithm of Section 2.3, the present algorithm is robust with respect to numerical errors.

If fewer, say n, processors are available, a very accurate approximation of the inverse is obtained in $O(n^2 \log n)$ time steps. This is because the algorithm needs $O(\log n)$ stages and each stage can be executed in $O(n^2)$ time using n processors. This is somewhat slower than Gaussian elimination or the Givens rotation method (Section 2.2), which require $O(n^2)$ time with n processors.

The algorithm of this section can also be used for solving systems of linear equations: the solution of $Ax = b$ is found by first computing A^{-1} and then letting $x = A^{-1}b$. However, such a method may have certain drawbacks, especially if storage requirements are taken into account. In classical iterative methods (Section 2.4), we need to store the entries of the A matrix, together with the vector x on which we iterate, which is $O(n^2)$ storage. With the algorithm of this section, we need to store the A matrix and, at each stage, the current estimate of the inverse, which is again $O(n^2)$ storage. Suppose now that the A matrix is sparse. In a classical iterative method, only the nonzero entries of A need to be stored and storage requirements are drastically reduced. On the other hand, even if A is sparse, its inverse will not be sparse in general. Therefore, the approximate inverses B_k will not be sparse either. Consequently, with the algorithm of this section, sparsity does not reduce the storage requirements.

EXERCISES

9.1. (Newton Interpretation of the Iteration $B_{k+1} = 2B_k - B_kAB_k$.) Let $f : \Re^n \mapsto \Re^n$ be a continuously differentiable function. For every $x \in \Re^n$ and $d \in \Re^n$, we have

$$f(x + d) = f(x) + \nabla f(x)'d + h(x, d),$$

where h is some function with the property $\lim_{d \to 0} h(x, d)/\|d\| = 0$ for every $x \in \Re^n$, and $\|\cdot\|$ is an arbitrary vector norm (see Appendix A). We wish to solve the equation $f(x) = 0$. Starting from a current value of x, we ignore the term $h(x, d)$, approximate the function $f(x + d)$ by $f(x) + \nabla f(x)'d$, choose some d that sets the latter expression to zero, and let $x := x + d$. The equation $f(x) + \nabla f(x)'d = 0$ yields $d = -\left(\nabla f(x)'\right)^{-1} f(x)$ and the iteration becomes

$$x := x - \left(\nabla f(x)'\right)^{-1} f(x),$$

which is known as Newton's method. Let us now fix an $n \times n$ matrix A. For any invertible $n \times n$ matrix X, let $f(X) = A - X^{-1}$. Inverting A is equivalent to solving the equation $f(X) = 0$.

(a) Show that $f(X + D) = A - X^{-1} + X^{-1}DX^{-1} + h(X, D)$, where the function h has the property $\lim_{D \to 0} h(X, D)/\|D\| = 0$ for every invertible matrix X, and $\|\cdot\|$ is an arbitrary matrix norm.

Hint: Use the formula $(I - C)^{-1} = I + C + C^2 + \cdots$, which is valid for any matrix C of small enough norm.

(b) Choose a value of D that sets $f(X + D)$ to zero, while ignoring the term $h(X, D)$ of part (a).

(c) With D chosen as in (b), show that the iteration $X := X + D$ coincides with the iteration $X := 2X - XAX$.

9.2. [PaR85] Suppose that B_0 is chosen according to the formula

$$B_0 = \frac{A'}{\|A\|_\infty \cdot \|A\|_1},$$

as opposed to Eq. (9.2). Show that Prop. 9.1 remains valid. *Hint:* Use the inequalities $\|A\|_2^2 \leq \|A\|_\infty \cdot \|A\|_1 \leq n\|A\|_2^2$ (see Appendix A), and proceed as in Prop. 9.1.

9.3. [PaR85] Suppose that A is symmetric positive definite, and let

$$B_0 = \frac{I}{\|A\|_\infty}.$$

Show that $\|I - B_0A\|_2 \leq 1 - 1/\left(n^{1/2}\kappa(A)\right)$.

9.4. [PaR85] Suppose that the matrix A has the property

$$\left(1 - \frac{1}{n^c}\right)|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|, \qquad \forall i,$$

and let B_0 be a diagonal matrix whose ith diagonal entry is equal to $1/a_{ii}$. Show that $\|I - B_0A\|_\infty \leq 1 - 1/n^c$.

NOTES AND SOURCES

Serial algorithms for matrix computations are the subject of [FaF63], [Hou64], [FoM67], [Ste73], and [GoV83]. For general surveys and discussion of parallel algorithms, see [Sam77], [Hel78], [Sam81], and [GHN87]. The survey paper [OrV85] focuses on parallel solution of partial differential equations but also provides an extensive discussion and list of references on parallel methods for linear equations.

2.1. The $O(\log^2 n)$ time algorithm for triangular matrix inversion based on Eq. (1.2) is from [Orc74] and [Hel74], and the "divide–and–conquer" algorithm is from [BoM75]. For other fast algorithms of this type, see [Hel78]. For a detailed discussion of the communication overhead of implementations of back substitution in a ring of processors, see [ISS86], which includes a comparison of the row and column storage schemes.

Odd–even reduction is due to [Hoc65] and its parallelization is discussed in [HoJ81]. See also [Hel76] for the block–tridiagonal case. Several parallel algorithms and discussions of communication issues can be found in [Sto75], [Hel78], [SaK77], [SaK78], [HoJ81], [GaV84], [Joh85a], and [Joh87b].

Other special structures that have been studied include banded systems ([Joh85b] and [SaS87]) and Toeplitz systems ([GrS81], [Bin84], and [GKK87]).

2.2. See [GoV83] and [Hou64] for a detailed description and analysis of direct methods for general linear equations. The communication penalty of parallel matrix multiplication and inversion has been first considered in [Gen78], where an $\Omega(n)$ lower bound on the execution time for mesh–connected architectures is established.

A detailed timing analysis of parallel Gaussian elimination with $O(n)$ processors can be found in [LKK83] and processor scheduling for the case of sparse matrices is studied in [WiH80], in the absence of communication delays. The communication penalty is considered in [ISS86] for the case of a ring architecture, and a variety of lower bounds on the communication penalty is provided in [Saa86]. Other pivoting rules, different than the one given by Eq. (2.4), have also been considered ([Sam85a]) and some of them lead to efficient [$\Theta(n)$ time] implementations in mesh–connected architectures (see, e.g., [Sam81] and [GeK81]).

Parallel triangularization using Givens rotations, and the schedule of Eqs. (2.11) and (2.12), have been suggested in [SaK78]; see also [LKK83] for a detailed timing analysis for the case of $O(n)$ processors in the absence of communication delays. The reference [CoR86] contains a result showing that the schedule of Eqs. (2.11) and (2.12) is very close to being optimal, even though it can be somewhat improved (compare with Exercise 2.6). An implementation of the Givens method in a mesh of n^2 processors is given in [BBK84]. Implementations on systolic arrays are discussed in [Kun88].

Givens rotations are also used in a variety of other algorithms in which the objective is to set the entries of a matrix to zero, and most such algorithms admit efficient parallel implementations in mesh–connected, as well as in special purpose VLSI architectures [Kun88]. One important application area is in eigenvalue and singular value problems. Recent work on these problems includes [BrL85], [IpS85], [DoS87], and [LPS87].

2.3. The algorithm of this section is from [Csa76]. Its processor requirements have been reduced in [PrS78]. An $O(\log^2 n)$ algorithm for Cholesky factorization (faster than the one in Exercise 3.1) is given in [Dat85].

2.4. For further reading on iterative methods, see [Var62], [You71], and [HaY81].

2.5. Parallel algorithms for linear PDEs are surveyed in [OrV85]. Algorithms and timing estimates for a particular parallel computer are considered in [Gal85]. See also [RAP87] for the effects of different partitionings of the problem domain and of different discretization methods.

For a precise description and convergence analyses of multigrid algorithms, see [Hac85]. Parallel implementations of multigrid algorithms have been discussed in [Bra81], [ChS85], [ChS86], and [McV87]. The implementation on a hypercube presented here is from [ChS86].

2.6. There are several textbooks with comprehensive analyses of linear iterative methods and their convergence rate, such as [Var62] and [You71]. The Brouwer Fixed Point Theorem and the core of the proof the Perron–Frobenius theorem can be found in [GuP74]. The proof outlined in Exercise 6.3 is taken from [Var62], which provides several references on alternative proofs of this result; see also [Sen81].

2.7. The conjugate gradient method is due to [HeS52]. For its properties when applied to nonlinear problems, see [Pol71]. For further readings for the linear case, see [FaF63] and [Lue84].

2.8. For further readings on Markov chains and their steady state behavior, see [Ash] and [Ros83a]. The algorithm of Eqs. (8.4) and (8.5), and its Gauss–Seidel variant, appear to be new.

2.9. The Newton method for matrix inversion is a classical algorithm, see [IsK66] and [Hou64]. The choice of B_0 in Prop. 9.1 is from [IsK66]. The use of this algorithm for parallel matrix inversion has been suggested in [Boj84a] and [PaR85]. Our complexity analysis is taken from [PaR85] which emphasizes the issue of appropriately choosing the initial matrix B_0.