Previous Chapter Return to Table of Contents Next Chapter

CHAPTER 31: MATRIX OPERATIONS

Operations on matrices are at the heart of scientific computing. Efficient algorithms for working with matrices are therefore of considerable practical interest. This chapter provides a brief introduction to matrix theory and matrix operations, emphasizing the problems of multiplying matrices and solving sets of simultaneous linear equations.

After Section 31.1 introduces basic matrix concepts and notations, Section 31.2 presents Strassen's surprising algorithm for multiplying two $n \times n$ matrices in $\Theta(n^{\lg 7}) = O(n^{2.81})$ time. Section 31.3 defines quasirings, rings, and fields, clarifying the assumptions required to make Strassen's algorithm work. It also contains an asymptotically fast algorithm for multiplying boolean matrices. Section 31.4 shows how to solve a set of linear equations using LUP decompositions. Then, Section 31.5 explores the close relationship between the problem of multiplying matrices and the problem of inverting a matrix. Finally, Section 31.6 discusses the important class of symmetric positive-definite matrices and shows how they can be used to find a least-squares solution to an overdetermined set of linear equations.

31.1 Properties of matrices

In this section, we review some basic concepts of matrix theory and some fundamental properties of matrices, focusing on those that will be needed in later sections.

Matrices and vectors

A ***matrix*** is a rectangular array of numbers. For example,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad (31.1)$$

(31.1)

is a 2×3 matrix $A = (a_{ij})$, where for $i = 1, 2$ and $j = 1, 2, 3$, the element of the matrix in row i and column j is a_{ij}. We use uppercase letters to denote matrices and corresponding subscripted lowercase letters to denote their elements. The set of all $m \times n$ matrices with real-valued entries is denoted $\mathbf{R}^{m \times n}$. In general, the set of $m \times n$ matrices with entries drawn from a set S is denoted $S^{m \times n}$.

The ***transpose*** of a matrix A is the matrix A^{T} obtained by exchanging the rows and columns of A. For the matrix A of equation (31.1),

$$A^{\mathrm{T}} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} .$$

A ***vector*** is a one-dimensional array of numbers. For example,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \qquad (31.2)$$

(31.2)

is a vector of size 3. We use lowercase letters to denote vectors, and we denote the ith element of a size-n vector x by x_i, for $i = 1, 2, \ldots, n$. We take the standard form of a vector to be as a ***column vector*** equivalent to an $n \times 1$ matrix; the corresponding ***row vector*** is obtained by taking the transpose:

$x^{\mathrm{T}} = (2\ 3\ 5)$.

The ***unit vector*** e_i is the vector whose ith element is 1 and all of whose other elements are 0. Usually, the size of a unit vector is clear from the context.

A ***zero matrix*** is a matrix whose every entry is 0. Such a matrix is often denoted 0, since the ambiguity between the number 0 and a matrix of 0's is usually easily resolved from context. If a matrix of 0's is intended, then the size of the matrix also needs to be derived from the context.

Square $n \times n$ matrices arise frequently. Several special cases of square matrices are of particular interest:

1. A ***diagonal matrix*** has $a_{ij} = 0$ whenever $i \neq j$. Because all of the off-diagonal elements are zero, the matrix can be specified by listing the elements along the diagonal:

$$\mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix} .$$

2 The $n \times n$ ***identity matrix*** I_n is a diagonal matrix with 1's along the diagonal:

I_n = diag(1,1,...,1)

$$= \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix} .$$

When I appears without a subscript, its size can be derived from context. The ith column of an identity matrix is the unit vector e_i.

3. A ***tridiagonal matrix*** T is one for which $t_{ij} = 0$ if $|i - j| > 1$. Nonzero entries appear only on the main diagonal, immediately above the main diagonal ($t_{i,i+1}$ for $i = 1, 2, \ldots, n - 1$), or immediately below the main diagonal ($t_{i+1,i}$ for $i = 1, 2, \ldots, n - 1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}.$$

4. An ***upper-triangular matrix*** U is one for which $u_{ij} = 0$ if $i > j$. All entries below the diagonal are zero:

$$U = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & u_{nn} \end{pmatrix}.$$

An upper-triangular matrix is ***unit upper-triangular*** if it has all 1's along the diagonal.

5. A ***lower-triangular matrix*** L is one for which $l_{ij} = 0$ if $i < j$. All entries above the diagonal are zero:

$$L = \begin{pmatrix} l_{11} & 0 & \ldots & 0 \\ l_{21} & l_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix}.$$

A lower-triangular matrix is ***unit lower-triangular*** if it has all 1's along the diagonal.

6. A ***permutation matrix*** P has exactly one 1 in each row or column, and 0's elsewhere. An example of a permutation matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Such a matrix is called a permutation matrix because multiplying a vector x by a permutation matrix has the effect of permuting (rearranging) the elements of x.

7. A ***symmetric matrix*** A satisfies the condition $A = A^{\mathrm{T}}$. For example,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

is a symmetric matrix.

Operations on matrices

The elements of a matrix or vector are numbers from a number system, such as the real numbers, the complex numbers, or integers modulo a prime. The number system defines how to add and multiply numbers. We can extend these definitions to encompass addition and multiplication of matrices.

We define ***matrix addition*** as follows. If $A = (a_{ij})$ and $B = (b_{ij})$ are $m \times n$ matrices, then their matrix sum $C = (c_{ij}) = A + B$ is the $m \times n$ matrix defined by

$$c_{ij} = a_{ij} + b_{ij}$$

for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. That is, matrix addition is performed componentwise. A zero matrix is the identity for matrix addition:

$$A + 0 = A$$

$$= 0 + A .$$

If Λ is a number and $A = (a_{ij})$ is a matrix, then $\Lambda A = (\Lambda a_{ij})$ is the ***scalar multiple*** of A obtained by multiplying each of its elements by Λ. As a special case, we define the ***negative*** of a matrix $A = (a_{ij})$ to be $-1 \cdot A = -A$, so that the ijth entry of $-A$ is $-a_{ij}$. Thus,

$$A + (-A) = 0$$

$$= (-A) + A .$$

Given this definition, we can define ***matrix subtraction*** as the addition of the negative of a matrix: $A - B = A + (-B)$.

We define ***matrix multiplication*** as follows. We start with two matrices A and B that are ***compatible*** in the sense that the number of columns of A equals the number of rows of B. (In general, an expression containing a matrix product AB is always assumed to imply that matrices A and B are compatible.) If $A = (a_{ij})$ is an $m \times n$ matrix and $B = (b_{jk})$ is an $n \times p$ matrix, then their matrix product $C = AB$ is the $m \times p$ matrix $C = (c_{ik})$, where

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \qquad (31.3)$$

(31.3)

for $i = 1, 2, \ldots, m$ and $k = 1, 2, \ldots, p$. The procedure MATRIX-MULTIPLY in Section 26.1 implements matrix multiplication in the straightforward manner based on equation (31.3), assuming that the matrices are square: $m = n = p$. To multiply $n \times n$ matrices, MATRIX-MULTIPLY performs n^3 multiplications and $n^2(n - 1)$ additions, and its running time is $\Theta(n^3)$.

Matrices have many (but not all) of the algebraic properties typical of numbers. Identity matrices are identities for matrix multiplication:

$$I_m A = A I_n = A$$

for any $m \times n$ matrix A. Multiplying by a zero matrix gives a zero matrix:

$$A\,0 = 0 \ .$$

Matrix multiplication is associative:

$$A(BC) = (AB)C$$

(31.4)

for compatible matrices A, B, and C. Matrix multiplication distributes over addition:

$$A(B + C) = AB + AC \ ,$$

$$(B + C)D = BD + CD \ .$$

(31.5)

Multiplication of $n \times n$ matrices is not commutative, however, unless $n = 1$. For example, if $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ and $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ and $AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$, then

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \ .$$

and

$$x^{\mathrm{T}} y = \sum_{i=1}^{n} x_i y_i$$

Matrix-vector products or vector-vector products are defined as if the vector were the equivalent $n \times 1$ matrix (or a $1 \times n$ matrix, in the case of a row vector). Thus, if A is an $m \times n$ matrix and x is a vector of size n, then Ax is a vector of size m. If x and y are vectors of size n, then

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \\ &= (x^{\mathrm{T}} x)^{1/2} \ . \end{aligned}$$

is a number (actually a 1×1 matrix) called the ***inner product*** of x and y. The matrix xy^{T} is an $n \times n$ matrix Z called the ***outer product*** of x and y, with $z_{ij} = x_i y_j$. The ***(euclidean) norm*** $||x||$ of a vector x of size n is defined by

Thus, the norm of x is its length in n-dimensional euclidean space.

Matrix inverses, ranks, and determinants

We define the ***inverse*** of an $n \times n$ matrix A to be the $n \times n$ matrix, denoted A^{-1} (if it exists), such that $AA^{-1} = I_n = A^{-1}A$. For example,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} .$$

Many nonzero $n \times n$ matrices do not have inverses. A matrix without an inverse is is called ***noninvertible***, or ***singular***. An example of a nonzero singular matrix is

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} .$$

If a matrix has an inverse, it is called ***invertible***, or ***nonsingular***. Matrix inverses, when they exist, are unique. (See Exercise 31.1-4.) If A and B are nonsingular $n \times n$ matrices, then

$$(BA)^{-1} = A^{-1}B^{-1}.$$

(31.6)

The inverse operation commutes with the transpose operation:

$$(A^{-1})^T = (A^T)^{-1} .$$

The vectors $x_1, x_2, \ldots, x_n$ are ***linearly dependent*** if there exist coefficients $c_1, c_2, \ldots, cn$, not all of which are zero, such that $c_1x_1 + c_2x_2 + \ldots + c_nx_n = 0$. For example, the vectors $x_1 = (1\ 2\ 3)^T$, $x_2 = (2\ 6\ 4)^T$, and $x_3 = (4\ 11\ 9)^T$ are linearly dependent, since $2x_1 + 3x_2 - 2x_3 = 0$. If vectors are not linearly dependent, they are ***linearly independent.*** For example, the columns of an identity matrix are linearly independent.

The ***column rank*** of a nonzero $m \times n$ matrix A is the size of the largest set of linearly independent columns of A. Similarly, the ***row rank*** of A is the size of the largest set of linearly independent rows of A. A fundamental property of any matrix A is that its row rank always equals its column rank, so that we can simply refer to the ***rank*** of A. The rank of an $m \times n$ matrix is an integer between 0 and min(m, n), inclusive. (The rank of a zero matrix is 0, and the rank of an $n \times n$ identity matrix is n.) An alternate, but equivalent and often more useful, definition is that the rank of a nonzero $m \times n$ matrix A is the smallest number r such that there exist matrices B and C of respective sizes $m \times r$ and $r \times n$ such that

$$A = BC .$$

A square $n \times n$ matrix has ***full rank*** if its rank is n. A fundamental property of ranks is given by the following theorem.

Theorem 31.1

A square matrix has full rank if and only if it is nonsingular.

An $m \times n$ matrix has ***full column rank*** if its rank is n.

A ***null vector*** for a matrix A is a nonzero vector x such that $Ax = 0$. The following theorem, whose proof is left as Exercise 31.1-8, and its corollary relate the notions of column rank and singularity to null vectors.

Theorem 31.2

A matrix A has full column rank if and only if it does not have a null vector.

Corollary 31.3

A square matrix A is singular if and only if it has a null vector.

The ijth ***minor*** of an $n \times n$ matrix A, for $n > 1$, is the $(n - 1) \times (n - 1)$ matrix $A_{[ij]}$ obtained by deleting the ith row and jth column of A. The ***determinant*** of an $n \times n$ matrix A can be defined recursively in terms of its minors by

$$\det(A) = \begin{cases} a_{11} & \text{if } n = 1 , \\ a_{11}\det(A_{[11]}) - a_{12}\det(A_{[12]}) & \\ \quad + \cdots + (-1)^{n+1} a_{1n}\det(A_{[1n]}) & \text{if } n > 1 . \end{cases} \qquad (31.7)$$

(31.7)

The term $(-1)^{i+j}\det(A_{[ij]})$ is known as the ***cofactor*** of the element a_{ij}.

The following theorems, whose proofs are omitted here, express fundamental properties of the determinant.

Theorem 31.4

The determinant of a square matrix A has the following properties:

- If any row or any column of A is zero, then $\det(A) = 0$.
- The determinant of A is multiplied by λ if the entries of any one row (or any one column) of A are all multiplied by λ.
- The determinant of A is unchanged if the entries in one row (respectively, column) are added to those in another row (respectively, column).
- The determinant of A equals the determinant of A^T.
- The determinant of A is multiplied by - 1 if any two rows (respectively, columns) are exchanged.

Also, for any square matrices A and B, we have $\det(AB) = \det(A)\det(B)$.

Theorem 31.5

An $n \times n$ matrix A is singular if and only if $\det(A) = 0$.

Positive-definite matrices

Positive-definite matrices play an important role in many applications. An $n \times n$ matrix A is ***positive-definite*** if $x^T Ax > 0$ for all size-n vectors $x \neq 0$. For example, the identity matrix is positive-definite, since for any nonzero vector $x = (x_1\ x_2\ \ldots\ x_n)^T$,

$$\begin{aligned} x^T I_n x &= x^T x \\ &= \|x\|^2 \\ &= \sum_{i=1}^{n} x_i^2 \\ &> 0 . \end{aligned}$$

As we shall see, matrices that arise in applications are often positive-definite due to the following theorem.

Theorem 31.6

For any matrix A with full column rank, the matrix $A^T A$ is positive-definite.

Proof We must show that $x^T (A^T A)x > 0$ for any nonzero vector x. For any vector x,

$x^T(A^T\ A)x = (Ax)^T(Ax)$ (by Exercise 31.1-3)

$= ||Ax||^2$

≥ 0 .

(31.8)

Note that $||Ax||^2$ is just the sum of the squares of the elements of the vector Ax. Therefore, if $||Ax||^2 = 0$, every element of Ax is 0, which is to say $Ax = 0$. Since A has full column rank, $Ax = 0$ implies $x = 0$, by Theorem 31.2. Hence, $A^T A$ is positive-definite.

Other properties of positive-definite matrices will be explored in Section 31.6.

Exercises

31.1-1

Prove that the product of two lower-triangular matrices is lower-triangular. Prove that the determinant of a (lower- or upper-) triangular matrix is equal to the product of its diagonal elements. Prove that the inverse of a lower-triangular matrix, if it exists, is lower-triangular.

31.1-2

Prove that if P is an $n \times n$ permutation matrix and A is an $n \times$ n matrix, then PA can be obtained from A by permuting its rows, and AP can be obtained from A by permuting its

columns. Prove that the product of two permutation matrices is a permutation matrix. Prove that if P is a permutation matrix, then P is invertible, its inverse is P^T, and P^T is a permutation matrix.

31.1-3

Prove that $(AB)^T = B^T A^T$ and that $A^T A$ is always a symmetric matrix.

31.1-4

Prove that if B and C are inverses of A, then $B = C$.

31.1-5

Let A and B be $n \times n$ matrices such that $AB = I$. Prove that if A' is obtained from A by adding row j into row i, then the inverse B' of A' can be obtained by subtracting column i from column j of B.

31.1-6

Let A be a nonsingular $n \times n$ matrix with complex entries. Show that every entry of A^{-1} is real if and only if every entry of A is real.

31.1-7

Show that if A is a nonsingular symmetric matrix, then A^{-1} is symmetric. Show that if B is an arbitrary (compatible) matrix, then $B\,AB^T$ is symmetric.

31.1-8

Show that a matrix A has full column rank if and only if $Ax = 0$ implies $x = 0$. (*Hint*: Express the linear dependence of one column on the others as a matrix-vector equation.)

31.1-9

Prove that for any two compatible matrices A and B,

$$\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B)),$$

where equality holds if either A or B is a nonsingular square matrix. (*Hint*: Use the alternate definition of the rank of a matrix.)

31.1-10

Given numbers $x_0, x_1, \ldots, x_n$-1, prove that the determinant of the ***Vandermonde matrix***

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

is

$$\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \le j < k \le n-1} (x_k - x_j) .$$

(*Hint:* Multiply column i by $-x_0$ and add it to column $i + 1$ for $i = n - 1, n - 2, \ldots, 1$, and then use induction.)

31.2 Strassen's algorithm for matrix multiplication

This section presents Strassen's remarkable recursive algorithm for multiplying $n \times n$ matrices that runs in $\Theta(n^{\lg 7}) = O(n^{2.81})$ time. For sufficiently large n, therefore, it outperforms the naive $\Theta(n^3)$ matrix-multiplication algorithm MATRIX-MULTIPLY from Section 26.1.

An overview of the algorithm

Strassen's algorithm can be viewed as an application of a familiar design technique: divide and conquer. Suppose we wish to compute the product $C = AB$, where each of A, B, and C are $n \times n$ matrices. Assuming that n is an exact power of 2, we divide each of A, B, and C into four $n/2 \times n/2$ matrices, rewriting the equation $C = AB$ as follows:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} . \qquad (31.9)$$

(31.9)

(Exercise 31.2-2 deals with the situation in which n is not an exact power of 2.) For convenience, the submatrices of A are labeled alphabetically from left to right, whereas those of B are labeled from top to bottom, in agreement with the way matrix multiplication is performed. Equation (31.9) corresponds to the four equations

$$r = ae + bf ,$$

(31.10)

$$s = ag + bh ,$$

(31.11)

$$t = ce + df ,$$

(31.12)

$$u = cg + dh \ .$$

(31.13)

Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. Using these equations to define a straightforward divide-and-conquer strategy, we derive the following recurrence for the time $T(n)$ to multiply two $n \times n$ matrices:

$$T(n) = 8T(n/2) + \Theta(n^2) \ .$$

(31.14)

Unfortunately, recurrence (31.14) has the solution $T(n) = \Theta(n^3)$, and thus this method is no faster than the ordinary one.

Strassen discovered a different recursive approach that requires only 7 recursive multiplications of $n/2 \times n/2$ matrices and $\Theta(n^2)$ scalar additions and subtractions, yielding the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$= \Theta(n^{\lg 7})$$

$$= O(n^{2.81}) \ .$$

(31.15)

Strassen's method has four steps:

1. Divide the input matrices A and B into $n/2 \times n/2$ submatrices, as in equation (31.9).

2. Using $\Theta(n^2)$ scalar additions and subtractions, compute 14 $n/2 \times n/2$ matrices $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$.

3. Recursively compute the seven matrix products $P_i = A_i B_i$ for $i = 1, 2, \ldots, 7$.

4. Compute the desired submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the P_i matrices, using only $\Theta(n^2)$ scalar additions and subtractions.

Such a procedure satisfies the recurrence (31.15). All that we have to do now is fill in the missing details.

Determining the submatrix products

It is not clear exactly how Strassen discovered the submatrix products that are the key to making his algorithm work. Here, we reconstruct one plausible discovery method.

Let us guess that each matrix product P_i can be written in the form

$$P_i = A_i B_i$$

$$= (\alpha_i 1a + \alpha_i 2b + \alpha_i 3c + \alpha_i 4d) \cdot (\beta_i 1e + \beta_i 2f + \beta_i 3g + \beta_i 4h) ,$$

(31.16)

where the coefficients α_{ij}, β_{ij} are all drawn from the set {-1, 0, 1}. That is, we guess that each product is computed by adding or subtracting some of the submatrices of *A,* adding or subtracting some of the submatrices of *B*, and then multiplying the two results together. While more general strategies are possible, this simple one turns out to work.

If we form all of our products in this manner, then we can use this method recursively without assuming commutativity of multiplication, since each product has all of the *A* submatrices on the left and all of the *B* submatrices on the right. This property is essential for the recursive application of this method, since matrix multiplication is not commutative.

For convenience, we shall use 4×4 matrices to represent linear combinations of products of submatrices, where each product combines one submatrix of *A* with one submatrix of *B* as in equation (31.16). For example, we can rewrite equation (31.10) as

$$\begin{aligned} r &= ae + bf \\ &= \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \\ &= \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array} . \end{aligned}$$

The last expression uses an abbreviated notation in which "+" represents +1, "·" represents 0, and "-" represents -1. (From here on, we omit the row and column labels.) Using this notation, we have the following equations for the other submatrices of the result matrix *C*:

$$
\begin{aligned}
s &= ag + bh \\
&= \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},
\end{aligned}
$$

$$
\begin{aligned}
t &= ce + df \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix},
\end{aligned}
$$

$$
\begin{aligned}
u &= cg + dh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.
\end{aligned}
$$

We begin our search for a faster matrix-multiplication algorithm by observing that the submatrix s can be computed as $s = P_1 + P_2$, where P_1 and P_2 are computed using one matrix multiplication each:

$$
\begin{aligned}
P_1 &= A_1 B_1 \\
&= a \cdot (g - h) \\
&= ag - ah \\
&= \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},
\end{aligned}
$$

$$
\begin{aligned}
P_2 &= A_2 B_2 \\
&= (a + b) \cdot h \\
&= ah + bh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
\end{aligned}
$$

The matrix t can be computed in a similar manner as $t = P_3 + P_4$, where

$$
\begin{aligned}
P_3 &= A_3B_3 \\
&= (c+d)\cdot e \\
&= ce+de \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}
\end{aligned}
$$

and

$$
\begin{aligned}
P_4 &= A_4B_4 \\
&= d\cdot(f-e) \\
&= df-de \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}.
\end{aligned}
$$

Let us define an ***essential term*** to be one of the eight terms appearing on the right-hand side of one of the equations (31.10)--(31.13). We have now used 4 products to compute the two submatrices *s* and *t* whose essential terms are *ag*, *bh*, *ce*, and *df* . Note that P_1 computes the essential term *ag*, P_2 computes the essential term *bh*, P_3 computes the essential term *ce*, and P_4 computes the essential term *df*. Thus, it remains for us to compute the remaining two submatrices *r* and *u*, whose essential terms are the diagonal terms *ae*, *bf*, *cg*, and *dh*, without using more than 3 additional products. We now try the innovation P_5 in order to compute two essential terms at once:

$$
\begin{aligned}
P_5 &= A_5B_5 \\
&= (a+d)\cdot(e+h) \\
&= ae+ah+de+dh \\
&= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}.
\end{aligned}
$$

In addition to computing both of the essential terms *ae* and *dh*, P_5 computes the inessential terms *ah* and *de*, which need to be cancelled somehow. We can use P_4 and P_2 to cancel them, but two other inessential terms then appear:

$$
\begin{aligned}
P_5+P_4-P_2 &= ae+dh+df-bh \\
&= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \end{pmatrix}.
\end{aligned}
$$

By adding an additional product

$$
\begin{aligned}
P_6 &= A_6 B_6 \\
&= (b-d)\cdot(f+h) \\
&= bf + bh - df - dh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix},
\end{aligned}
$$

however, we obtain

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= ae + bf \\
&= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
\end{aligned}
$$

We can obtain u in a similar manner from P_5 by using P_1 and P_3 to move the inessential terms of P_5 in a different direction:

$$
\begin{aligned}
P_5 + P_1 - P_3 &= ae + ag - ce + dh \\
&= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.
\end{aligned}
$$

By subtracting an additional product

$$
\begin{aligned}
P_7 &= A_7 B_7 \\
&= (a-c)\cdot(e+g) \\
&= ae + ag - ce - cg \\
&= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},
\end{aligned}
$$

we now obtain

$$
\begin{aligned}
u &= P_5 + P_1 - P_3 - P_7 \\
&= cg + dh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.
\end{aligned}
$$

The 7 submatrix products $P_1, P_2, \ldots, P_7$ can thus be used to compute the product $C = AB$, which completes the description of Strassen's method.

Discussion

The large constant hidden in the running time of Strassen's algorithm makes it

impractical unless the matrices are large (n at least 45 or so) and dense (few zero entries). For small matrices, the straightforward algorithm is preferable, and for large, sparse matrices, there are special sparsematrix algorithms that beat Strassen's in practice. Thus, Strassen's method is largely of theoretical interest.

By using advanced techniques beyond the scope of this text, one can in fact multiply $n \times n$ matrices in better than $\Theta(n^{\lg 7})$ time. The current best upper bound is approximately $O(n^{2.376})$. The best lower bound known is just the obvious $\Omega(n^2)$ bound (obvious because we have to fill in n^2 elements of the product matrix). Thus, we currently do not know how hard matrix multiplication really is.

Strassen's algorithm does not require that the matrix entries be real numbers. All that matters is that the number system form an algebraic ring. If the matrix entries do not form a ring, however, sometimes other techniques can be brought to bear to allow his method to apply. These issues are discussed more fully in the next section.

Exercises

31.2-1

Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}.$$

Show your work.

31.2-2

How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.

31.2-3

What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

31.2-4

V. Pan has discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? Compare it with the running time for Strassen's algorithm.

131.2-5

How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's

algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.

31.2-6

Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three real multiplications. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

* 31.3 Algebraic number systems and boolean matrix multiplication

The properties of matrix addition and multiplication depend on the properties of the underlying number system. In this section, we define three different kinds of underlying number systems: quasirings, rings, and fields. We can define matrix multiplication over quasirings, and Strassen's matrix-multiplication algorithm works over rings. We then present a simple trick for reducing boolean matrix multiplication, which is defined over a quasiring that is not a ring, to multiplication over a ring. Finally, we discuss why the properties of a field cannot naturally be exploited to provide better algorithms for matrix multiplication.

Quasirings

Let $(S, \oplus, \odot, \overline{0}, \overline{1})$ denote a number system, where S is a set of elements, $\oplus$ and $\odot$ are binary operations on S (the addition and multiplication operations, respectively), and $\overline{0}$ and $\overline{1}$ are distinct distinguished elements of S. This system is a ***quasiring*** if it satisfies the following properties:

1. $(S, \oplus, \overline{0})$ is a ***monoid***:

- S is ***closed*** under $\oplus$; that is, $a \oplus b \in S$ for all $a, b \in S$.
- $\oplus$ is ***associative***; that is, $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ for all $a, b, c \in S$.
- $\overline{0}$ is an ***identity*** for $\oplus$; that is, $a \oplus \overline{0} = \overline{0} \oplus a = a$ for all $a \in S$.

Likewise, $(S, \odot, \overline{1})$ is a monoid.

2. $\overline{0}$ is an ***annihilator***, that is, $a \odot \overline{0} = \overline{0} \odot a = \overline{0}$ for all $a \in S$.

3. The operator $\oplus$ is ***commutative***; that is, $a \oplus b = b \oplus a$ for all $a, b \in S$.

4. The operator $\odot$ ***distributes*** over $\oplus$; that is, $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ and

$(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$ for all $a,b,c \in S$.

Examples of quasirings include the ***boolean quasiring*** ({0,1}, $\vee$, $\wedge$, 0, 1), where $\vee$ denotes logical OR and $\wedge$ denotes logical AND, and the natural number system (**N**, +, $\cdot$, 0, 1), where + and $\cdot$ denote ordinary addition and multiplication. Any closed semiring (see Section 26.4) is also a quasiring; closed semirings obey additional idempotence and infinite-sum properties.

We can extend $\oplus$ and $\odot$ to matrices as we did for + and $\cdot$ in Section 31.1. Denoting the $n \times n$ identity matrix composed of $\overline{0}$ and $\overline{1}$ by $\overline{I}_n$, we find that matrix multiplication is well defined and the matrix system is itself a quasiring, as the following theorem states.

Theorem 31.7

If $(S, \oplus, \odot, \overline{0}, \overline{1})$ is a quasiring and $n \geq 1$, then $(S^{n \times n}, \oplus, \odot, \overline{0}, \overline{I}_n)$ is a quasiring.

Proof The proof is left as Exercise 31.3-3.

Rings

Subtraction is not defined for quasirings, but it is for a ***ring***, which is a quasiring $(S, \oplus, \odot, \overline{0}, \overline{1})$ that satisfies the following additional property:

5. Every element in S has an ***additive inverse***; that is, for all $a \in S$, there exists an element $b \in S$ such that $a \oplus b = b \oplus a = \overline{0}$. Such a b is also called the ***negative*** of a and is denoted $(-a)$.

Given that the negative of any element is defined, we can define subtraction by $a - b = a + (-b)$.

There are many examples of rings. The integers (**Z**, +, $\cdot$, 0, 1) under the usual operations of addition and multiplication form a ring. The integers modulo n for any integer $n > 1$--that is, ($\mathbf{Z}_n$, +, $\cdot$, 0, 1), where + is addition modulo n and $\cdot$ is multiplication modulo n--form a ring. Another example is the set $\mathbf{R}[x]$ of finite-degree polynomials in x with real coefficients under the usual operations--that is, ($\mathbf{R}[x]$, +, $\cdot$, 0, 1), where + is polynomial addition and $\cdot$ is polynomial multiplication.

The following corollary shows that Theorem 31.7 generalizes naturally to rings.

Corollary 31.8

If $(S, \oplus, \odot, \overline{0}, \overline{1})$ is a ring and $n \geq 1$, then $(S^{n \times n}, \oplus, \odot, \overline{0}, \overline{I}_n)$ is a ring.

Proof The proof is left as Exercise 31.3-3.

Using this corollary, we can prove the following theorem.

Theorem 31.9

Strassen's matrix-multiplication algorithm works properly over any ring of matrix elements.

Proof Strassen's algorithm depends on the correctness of the algorithm for 2×2 matrices, which requires only that the matrix elements belong to a ring. Since the matrix elements do belong to a ring, Corollary 31.8 implies the matrices themselves form a ring. Thus, by induction, Strassen's algorithm works correctly at each level of recursion.

Strassen's algorithm for matrix multiplication, in fact, depends critically on the existence of additive inverses. Out of the seven products $P_1, P_2, \ldots, P_7$, four involve differences of submatrices. Thus, Strassen's algorithm does not work in general for quasirings.

Boolean matrix multiplication

Strassen's algorithm cannot be used directly to multiply boolean matrices, since the boolean quasiring ({0,1}, $\vee$, $\wedge$, 0, 1) is not a ring. There are instances in which a quasiring is contained in a larger system that is a ring. For example, the natural numbers (a quasiring) are a subset of the integers (a ring), and Strassen's algorithm can therefore be used to multiply matrices of natural numbers if we consider the underlying number system to be the integers. Unfortunately, the boolean quasiring cannot be extended in a similar way to a ring. (See Exercise 31.3-4.)

The following theorem presents a simple trick for reducing boolean matrix multiplication to multiplication over a ring. Problem 31-1 presents another efficient approach.

Theorem 31.10

If $M(n)$ denotes the number of arithmetic operations required to multiply two $n \times n$ matrices over the integers, then two $n \times n$ boolean matrices can be multiplied using $O(M(n))$ arithmetic operations.

Proof Let the two matrices be A and B, and let $C = AB$ in the boolean quasiring, that is,

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj} .$$

Instead of computing over the boolean quasiring, we compute the product C' over the ring of integers with the given matrix-multiplication algorithm that uses $M(n)$ arithmetic operations. We thus have

$$c'_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Each term $a_{ik}b_{kj}$ of this sum is 0 if and only if $a_{ik} \wedge b_{kj} = 0$, and 1 if and only if $a_{ik} \wedge b_{kj} =$

1. Thus, the integer sum c'_{ij} is 0 if and only if every term is 0 or, equivalently, if and only if the boolean OR of the terms, which is c_{ij}, is 0. Therefore, the boolean matrix C can be reconstructed with $\Theta(n^2)$ arithmetic operations from the integer matrix C' by simply comparing each c'_{ij} with 0. The number of arithmetic operations for the entire procedure is then $O(M(n)) + \Theta(n^2) = O(M(n))$, since $M(n) = \Omega(n^2)$.

Thus, using Strassen's algorithm, we can perform boolean matrix multiplication in $O(n^{\lg 7})$ time.

The normal method of multiplying boolean matrices uses only boolean variables. If we use this adaptation of Strassen's algorithm, however, the final product matrix can have entries as large as n, thus requiring a computer word to store them rather than a single bit. More worrisome is that the intermediate results, which are integers, may grow even larger. One method for keeping intermediate results from growing too large is to perform all computations modulo $n + 1$. Exercise 31.3-5 asks you to show that working modulo $n + 1$ does not affect the correctness of the algorithm.

Fields

A ring $(S, \oplus, \odot, \overline{0}, \overline{1})$ is a ***field*** if it satisfies the following two additional properties:

6. The operator $\odot$ is ***commutative***; that is, $a \odot b = b \odot a$ for all $a, b \in S$.

7. Every nonzero element in S has a ***multiplicative inverse***; that is, for all $a \in S - \{\overline{0}\}$, there exists an element $b \in S$ such that $a \odot b = b \odot a = \overline{1}$.

Such an element b is often called the i***nverse*** of a and is denoted a^{-1}.

Examples of fields include the real numbers (**R**, +, •, 0, 1), the complex numbers (**C**, +, •, 0, 1), and the integers modulo a prime p: ($\mathbf{Z}_p$, +, •, 0, 1).

Because fields offer multiplicative inverses of elements, division is possible. They also offer commutativity. By generalizing from quasirings to rings, Strassen was able to improve the running time of matrix multiplication. Since the underlying elements of matrices are often from a field--the real numbers, for instance--one might hope that by using fields instead of rings in a Strassen-like recursive algorithm, the running time might be further improved.

This approach seems unlikely to be fruitful. For a recursive divide-and-conquer algorithm based on fields to work, the matrices at each step of the recursion must form a field. Unfortunately, the natural extension of Theorem 31.7 and Corollary 31.8 to fields fails badly. For $n > 1$, the set of $n \times n$ matrices *never* forms a field, even if the underlying number system is a field. Multiplication of $n \times n$ matrices is not commutative, and many $n \times n$ matrices do not have inverses. Better algorithms for matrix multiplication are therefore more likely to be based on ring theory than on field theory.

Exercises

31.3-1

Does Strassen's algorithm work over the number system $(\mathbf{Z}[x], +, \cdot, 0, 1)$, where $\mathbf{Z}[x]$ is the set of all polynomials with integer coefficients in the variable x and $+$ and $\cdot$ are ordinary polynomial addition and multiplication?

31.3-2

Explain why Strassen's algorithm doesn't work over closed semirings (see Section 26.4) or over the boolean quasiring $(\{0, 1\}, \vee, \wedge, 0, 1)$.

31.3-3

Prove Theorem 31.7 and Corollary 31.8.

31.3-4

Show that the boolean quasiring $(\{0, 1\}, \vee, \wedge, 0, 1)$ cannot be embedded in a ring. That is, show that it is impossible to add a "-1" to the quasiring so that the resulting algebraic structure is a ring.

31.3-5

Argue that if all computations in the algorithm of Theorem 31.10 are performed modulo $n + 1$, the algorithm still works correctly.

31.3-6

Show how to determine efficiently if a given undirected input graph contains a triangle (a set of three mutually adjacent vertices).

31.3-7

Show that computing the product of two $n \times n$ boolean matrices over the boolean quasiring is reducible to computing the transitive closure of a given directed $3n$-vertex input graph.

31.3-8

Show how to compute the transitive closure of a given directed n-vertex input graph in time $O(n^{\lg 7} \lg n)$. Compare this result with the performance of the TRANSITIVE-CLOSURE procedure in Section 26.2.

31.4 Solving systems of linear equations

Solving a set of simultaneous linear equations is a fundamental problem that occurs in diverse applications. A linear system can be expressed as a matrix equation in which

each matrix or vector element belongs to a field, typically the real numbers **R**. This section discusses how to solve a system of linear equations using a method called LUP decomposition.

We start with a set of linear equations in n unknowns $x_1, x_2, \ldots, x_n$:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1,$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2,$$

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n . \end{aligned} \qquad (31.17)$$

(31.17)

$$a_n1x_1 + a_n2x_2 + \cdots + a_{nn}x_n = b_n.$$

A set of values for $x_1, x_2, \ldots, x_n$ that satisfy all of the equations (31.17) simultaneously is said to be a ***solution*** to these equations. In this section, we only treat the case in which there are exactly n equations in n unknowns.

We can conveniently rewrite equations (31.17) as the matrix-vector equation

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

or, equivalently, letting $A = (a_{ij})$, $x = (x_j)$, and $b = (b_i)$, as

$$Ax = b \ .$$

(31.18)

If A is nonsingular, it possesses an inverse A^{-1}, and

$$x = A^{-1} b$$

(31.19)

is the solution vector. We can prove that x is the unique solution to equation (31.18) as follows. If there are two solutions, x and x', then $Ax = Ax' = b$ and

$$x = (A^{-1} A)x$$

$$= A^{-1}(Ax)$$

$$= A^{-1}(Ax')$$

$= (A^{-1} A) x'$

$= x'$.

In this section, we shall be concerned predominantly with the case in which A is nonsingular or, equivalently (by Theorem 31.1), the rank of A is equal to the number n of unknowns. There are other possibilities, however, which merit a brief discussion. If the number of equations is less than the number n of unknowns--or, more generally, if the rank of A is less than n--then the system is ***underdetermined***. An underdetermined system typically has infinitely many solutions (see Exercise 31.4-9), although it may have no solutions at all if the equations are inconsistent. If the number of equations exceeds the number n of unknowns, the system is ***overdetermined***, and there may not exist any solutions. Finding good approximate solutions to overdetermined systems of linear equations is an important problem that is addressed in Section 31.6.

Let us return to our problem of solving the system $Ax = b$ of n equations in n unknowns. One approach is to compute A^{-1} and then multiply both sides by A^{-1}, yielding $A^{-1}Ax = A^{-1}b$, or $x = A^{-1}b$. This approach suffers in practice from ***numerical instability***: round-off errors tend to accumulate unduly when floating-point number representations are used instead of ideal real numbers. There is, fortunately, another approach--LUP decomposition--that is numerically stable and has the further advantage of being about a factor of 3 faster.

Overview of LUP decomposition

The idea behind LUP decomposition is to find three $n \times n$ matrices L, U, and P such that

$PA = LU$,

(31.20)

where

- L is a unit lower-triangular matrix,
- U is an upper-triangular matrix, and
- P is a permutation matrix.

We call matrices L, U, and P satisfying equation (31.20) an ***LUP decomposition*** of the matrix A. We shall show that every nonsingular matrix A possesses such a decomposition.

The advantage of computing an LUP decomposition for the matrix A is that linear systems can be solved more readily when they are triangular, as is the case for both matrices L and U. Having found an LUP decomposition for A, we can solve the equation (31.18) $Ax = b$ by solving only triangular linear systems, as follows. Multiplying both sides of $Ax = b$ by P yields the equivalent equation $PAx = Pb$, which by Exercise 31.1-2 amounts to permuting the equations (31.17). Using our decomposition (31.20), we obtain

$$LUx = Pb .$$

We can now solve this equation by solving two triangular linear systems. Let us define $y = Ux$, where x is the desired solution vector. First, we solve the lower-triangular system

$$Ly = Pb$$

(31.21)

for the unknown vector y by a method called "forward substitution." Having solved for y, we then solve the upper-triangular system

$$Ux = y$$

(31.22)

for the unknown x by a method called "back substitution." The vector x is our solution to $Ax = b$, since the permutation matrix P is invertible (Exercise 31.1 -2):

$$\begin{aligned} Ax &= P^{-1} LUx \\ &= P^{-1} Ly \\ &= P^{-1} Pb \\ &= b . \end{aligned}$$

Our next step is to show how forward and back substitution work and then attack the problem of computing the LUP decomposition itself.

Forward and back substitution

Forward substitution can solve the lower-triangular system (31.21) in $\Theta(n^2)$ time, given L, P, and b. For convenience, we represent the permutation P compactly by an array $\pi[1 . . n]$. For $i = 1, 2, \ldots, n$, the entry $\pi[i]$ indicates that $P_{i,\pi[i]} = 1$ and $P_{ij} = 0$ for $j \neq \pi[i]$. Thus, PA has $a_{\pi[i],j}$ in row i and column j, and Pb has $b_{\pi[i]}$ as its ith element. Since L is unit lower-triangular, equation (31.21) can be rewritten as

$$\begin{aligned} y_1 &= b_{\pi[1]}, \\ l_{21}y_1 + y_2 &= b_{\pi[2]}, \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]}, \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]} . \end{aligned}$$

Quite apparently, we can solve for y_1 directly, since the first equation tells us that $y_1 = b_{\pi[1]}$. Having solved for y_1, we can substitute it into the second equation, yielding

$$y_2 = b_{\pi[2]} - l_{21}y_1 .$$

Now, we can substitute both y_1 and y_2 into the third equation, obtaining

$$y_3 = b\Pi_{[3]} - (l_{31}y_1 + l_{32}y_2).$$

In general, we substitute $y_1, y_2, \ldots, y_i$-1 "forward" into the ith equation to solve for y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j .$$

Back substitution is similar to forward substitution. Given U and y, we solve the nth equation first and work backward to to the first equation. Like forward substitution, this process runs in $\Theta(n^2)$ time. Since U is upper-triangular, we can rewrite the system (31.22) as

$$u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_n\text{-}2 + u_{1,n-1}x_n\text{-}1 + u_{1n}x_n = y_1,$$

$$u_{22}x_2 + \cdots + u_{2,n-2}x_n\text{-}2 + u_{2,n-1}x_n\text{-}1 + u_{2n}x_n = y_2,$$

$$\vdots$$

$$u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n = y_{n-2},$$

$$u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n = y_{n-1},$$

$$u_{n,n}x_n = y_n .$$

Thus, we can solve for $x_n, x_n\text{-}1, \ldots, x_1$ successively as follows:

$$x_n = y_n/u_{nn} ,$$

$$x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} ,$$

$$x_{n-2} = (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} ,$$

$$\vdots$$

or, in general,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j\right)/u_{ii} .$$

Given P, L, U, and b, the procedure LUP-SOLVE solves for x by combining forward and back substitution. The pseudocode assumes that the dimension n appears in the attribute $rows[L]$ and that the permutation matrix P is represented by the array π.

```
LUP-SOLVE(L, U, π, b)
1  n ← rows[L]
2  for i ← 1 to n
3       do y_i ← b_π[i] − Σ_{j=1}^{i−1} l_ij y_j
4  for i ← n downto 1
5       do x_i ← (y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
6  return x
```

Procedure LUP-SOLVE solves for y using forward substitution in lines 2-3, and then it solves for x using backward substitution in lines 4-5. Since there is an implicit loop in the summations within each of the **for** loops, the running time is $\Theta(n^2)$.

As an example of these methods, consider the system of linear equations defined by

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix} ,$$

where

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix} ,$$

$$b = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix} ,$$

and we wish to solve for the unknown x. The LUP decomposition is

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix} ,$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix} ,$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} .$$

(The reader can verify that $PA = LU$.) Using forward substitution, we solve $Ly = Pb$ for y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 12.5 \\ 0.1 \end{pmatrix} ,$$

obtaining

$$y = \begin{pmatrix} 10.3 \\ 6.32 \\ -5.569 \end{pmatrix}$$

by computing first y_1, then y_2, and finally y_3. Using back substitution, we solve $Ux = y$ for x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 6.32 \\ -5.569 \end{pmatrix},$$

thereby obtaining the desired answer

$$x = \begin{pmatrix} 0.5 \\ -0.2 \\ 3.0 \end{pmatrix}$$

by computing first x_3, then x_2, and finally x_1.

Computing an LU decomposition

We have now shown that if an LUP decomposition can be computed for a nonsingular matrix A, forward and back substitution can be used to solve the system $Ax = b$ of linear equations. It remains to show how an LUP decomposition for A can be found efficiently. We start with the case in which A is an $n \times n$ nonsingular matrix and P is absent (or, equivalently, $P = I_n$). In this case, we must find a factorization $A = LU$. We call the two matrices L and U an ***LU decomposition*** of A.

The process by which we perform LU decomposition is called ***Gaussian elimination***. We start by subtracting multiples of the first equation from the other equations so that the first variable is removed from those equations. Then, we subtract multiples of the second equation from the third and subsequent equations so that now the first and second variables are removed from them. We continue this process until the system that is left has an upper-triangular form--in fact, it is the matrix U. The matrix L is made up of the row multipliers that cause variables to be eliminated.

Our algorithm to implement this strategy is recursive. We wish to construct an LU decomposition for an $n \times n$ nonsingular matrix A. If $n = 1$, then we're done, since we can choose $L = I_1$ and $U = A$. For $n > 1$, we break A into four parts:

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

where v is a size-$(n - 1)$ column vector, w^{T} is a size-$(n - 1)$ row vector, and A' is an $(n - 1) \times (n - 1)$ matrix. Then, using matrix algebra (verify the equations by simply multiplying through), we can factor A as

$$
\begin{aligned}
A &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} .
\end{aligned}
$$

The 0's in the first and second matrices of the factorization are row and column vectors, respectively, of size n - 1. The term vw^{T}/a_{11}, formed by taking the outer product of v and w and dividing each element of the result by a_{11}, is an $(n - 1) \times (n - 1)$ matrix, which conforms in size to the matrix A' from which it is subtracted. The resulting $(n - 1) \times (n - 1)$ matrix

$$A' - vw^{\mathrm{T}}/a_{11}$$

(31.23)

is called the ***Schur complement*** of A with respect to a_{11}.

We now recursively find an LU decomposition of the Schur complement. Let us say that

$$A' - vw^{\mathrm{T}}/a_{11} = L'U' ,$$

where L' is unit lower-triangular and U' is upper-triangular. Then, using matrix algebra, we have

$$
\begin{aligned}
A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\
&= LU ,
\end{aligned}
$$

thereby providing our LU decomposition. (Note that because L' is unit lower-triangular, so is L, and because $U'$is upper-triangular, so is U.)

Of course, if $a_{11} = 0$, this method doesn't work, because it divides by 0. It also doesn't work if the upper leftmost entry of the Schur complement $A' - vw^{\mathrm{T}}/a_{11}$ is 0, since we divide by it in the next step of the recursion. The elements by which we divide during LU decomposition are called ***pivots***, and they occupy the diagonal elements of the matrix U. The reason we include a permutation matrix P during LUP decomposition is that it allows us to avoid dividing by zero elements. Using permutations to avoid division by 0 (or by small numbers) is called ***pivoting***.

An important class of matrices for which LU decomposition always works correctly is the class of symmetric positive-definite matrices. Such matrices require no pivoting, and thus the recursive strategy outlined above can be employed without fear of dividing by 0. We shall prove this result, as well as several others, in Section 31.6.

Our code for LU decomposition of a matrix A follows the recursive strategy, except that

an iteration loop replaces the recursion. (This transformation is a standard optimization for a "tail-recursive" procedure--one whose last operation is a recursive call to itself.) It assumes that the dimension of A is kept in the attribute $rows[A]$. Since we know that the output matrix U has 0's below the diagonal, and since LU-SOLVE does not look at these entries, the code does not bother to fill them in. Likewise, because the output matrix L has 1's on its diagonal and 0's above the diagonal, these entries are not filled in either. Thus, the code computes only the "significant" entries of L and U.

```
LU-DECOMPOSITION(A)
 1  n ← rows[A]
 2  for k ← 1 to n
 3       do u_kk ← a_kk
 4          for i ← k + 1 to n
 5               do l_ik ← a_ik/u_kk        ▷ l_ik holds v_i
 6                  u_ki ← a_ki             ▷ u_ki holds w_i^T
 7          for i ← k + 1 to n
 8               do for j ← k + 1 to n
 9                       do a_ij ← a_ij − l_ik u_kj
10  return L and U
```

The outer **for** loop beginning in line 2 iterates once for each recursive step. Within this loop, the pivot is determined to be $u_{kk} = a_{kk}$ in line 3. Within the **for** loop in lines 4-6 (which does not execute when $k = n$), the v and w^T vectors are used to update L and U. The elements of the v vector are determined in line 5, where v_i is stored in l_{ik}, and the elements of the w^T vector are determined in line 6, where w_i^T is stored in u_{ki}. Finally, the elements of the Schur complement are computed in lines 7-9 and stored back in the matrix A. Because line 9 is triply nested, LU-DECOMPOSITION runs in time $\Theta(n^3)$.

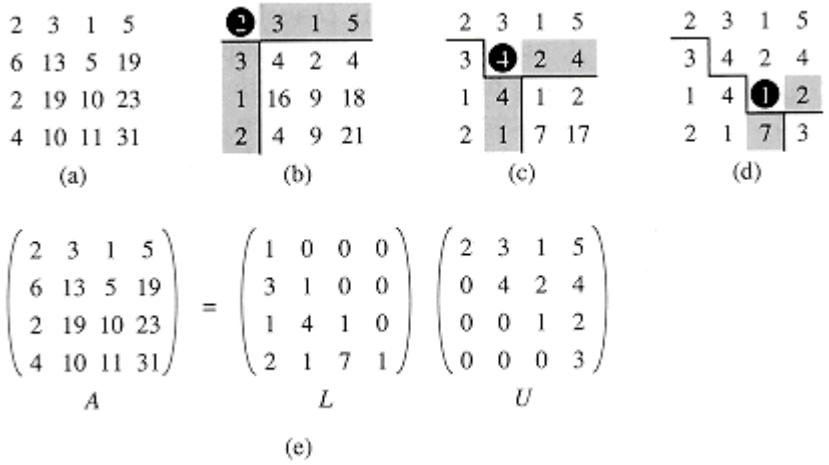

Figure 31.1 The operation of LU-DECOMPOSITION. (a) The matrix A. (b) The element $a_{11} = 2$ in black is the pivot, the shaded column is v/a_{11}, and the shaded row is w^T. The elements of U computed thus far are above the horizontal line, and the elements of L are to the left of the vertical line. The Schur complement matrix $A' - vw^T/a_{11}$ occupies the lower right. (c) We now operate on the Schur complement matrix produced from part (b). The element $a_{22} = 4$ in black is the pivot, and the shaded column and row are v/a_{22} and w^T (in the partitioning of the Schur

complement), respectively. Lines divide the matrix into the elements of U computed so far (above), the elements of L computed so far (left), and the new Schur complement (lower right). (d) The next step completes the factorization. (The element 3 in the new Schur complement becomes part of U when the recursion terminates.) (e) The factorization A = LU.

Figure 31.1 illustrates the operation of LU-DECOMPOSITION. It shows a standard optimization of the procedure in which the significant elements of L and U are stored "in place" in the matrix A. That is, we can set up a correspondence between each element a_{ij} and either l_{ij} (if $i > j$) or u_{ij} (if $i \leq j$) and update the matrix A so that it holds both L and U when the procedure terminates. The pseudocode for this optimization is obtained from the above pseudocode merely by replacing each reference to l or u by a; it is not difficult to verify that this transformation preserves correctness.

Computing an LUP decomposition

Generally, in solving a system of linear equations $Ax = b$, we must pivot on off-diagonal elements of A to avoid dividing by 0. Not only is division by 0 undesirable, so is division by any small value, even if A is nonsingular, because numerical instabilities can result in the computation. We therefore try to pivot on a large value.

The mathematics behind LUP decomposition is similar to that of LU decomposition. Recall that we are given an $n \times n$ nonsingular matrix A and wish to find a permutation matrix P, a unit lower-triangular matrix L, and an upper-triangular matrix U such that $PA = LU$. Before we partition the matrix A, as we did for LU decomposition, we move a nonzero element, say a_k1, from the first column to the (1,1) position of the matrix. (If the first column contains only 0's, then A is singular, because its determinant is 0, by Theorems 31.4 and 31.5.) In order to preserve the set of equations, we exchange row 1 with row k, which is equivalent to multiplying A by a permutation matrix Q on the left (Exercise 31.1-2). Thus, we can write QA as

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

where $v = (a_{21}, a_{31}, \ldots, a_n1)^{\mathrm{T}}$, except that a_{11} replaces a_k1; $w^{\mathrm{T}} = (a_k2, a_k3, \ldots, a_{kn})$; and A' is an $(n - 1) \times (n - 1)$ matrix. Since $a_k1 \neq 0$, we can now perform much the same linear algebra as for LU decomposition, but now guaranteeing that we do not divide by 0:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix}. \end{aligned}$$

The Schur complement $A' - vw^{\mathrm{T}}/a_k1$ is nonsingular, because otherwise the second matrix in the last equation has determinant 0, and thus the determinant of matrix A is 0; but this means that A is singular, which contradicts our assumption that A is nonsingular.

Consequently, we can inductively find an LUP decomposition for the Schur complement, with unit lower-triangular matrix L', upper-triangular matrix U', and permutation matrix P', such that

$$P'(A' - vw^T/a_{k1}) = L'U' .$$

Define

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q ,$$

which is a permutation matrix, since it is the product of two permutation matrices (Exercise 31.1-2). We now have

$$\begin{aligned} PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\ &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & P'(A' - vw^T/a_{k1}) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & U' \end{pmatrix} \\ &= LU , \end{aligned}$$

yielding the LUP decomposition. Because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.

Notice that in this derivation, unlike the one for LU decomposition, both the column vector v/a_{k1} and the Schur complement $A' - vw^T/a_{k1}$ must be multiplied by the permutation matrix P'.

Like LU-DECOMPOSITION, our pseudocode for LUP decomposition replaces the recursion with an iteration loop. As an improvement over a direct implementation of the recursion, we dynamically maintain the permutation matrix P as an array π, where $\pi[i] = j$ means that the ith row of P contains a 1 in column j. We also implement the code to compute L and U "in place" in the matrix A. Thus, when the procedure terminates,

$$a_{ij} = \begin{cases} l_{ij} & \text{if } i > j , \\ u_{ij} & \text{if } i \le j . \end{cases}$$

```
LUP-DECOMPOSITION(A)

1  n ← rows[A]

2  for i ← 1 to n
```

```
3       do Π[i] ← i
4  for k ← 1 to n - 1
5       do p ← 0
6          for i ← k to n
7             do if |a_ik| > p
8                   then p ← |a_ik|
9                        k' ← i
10         if p = 0
11            then error "singular matrix"
12         exchange Π[k] ↔ Π[k']
13         for i ← 1 to n
14              do exchange a_ki ↔ a_k'i
15         for i ← k + 1 to n
16              do a_ik ← a_ik/a_kk
17                 for j ← k + 1 to n
18                      do a_ij ← a_ij - a_ik a_kj
```

Figure 31.2 illustrates how LUP-DECOMPOSITION factors a matrix. The array Π is initialized by lines 2-3 to represent the identity permutation. The outer **for** loop beginning in line 4 implements the recursion. Each time through the outer loop, lines 5-9 determine the element $a_{k'k}$ with largest absolute value of those in the current first column (column k) of the $(n - k + 1) \times (n - k + 1)$ matrix whose LU decomposition must be found.

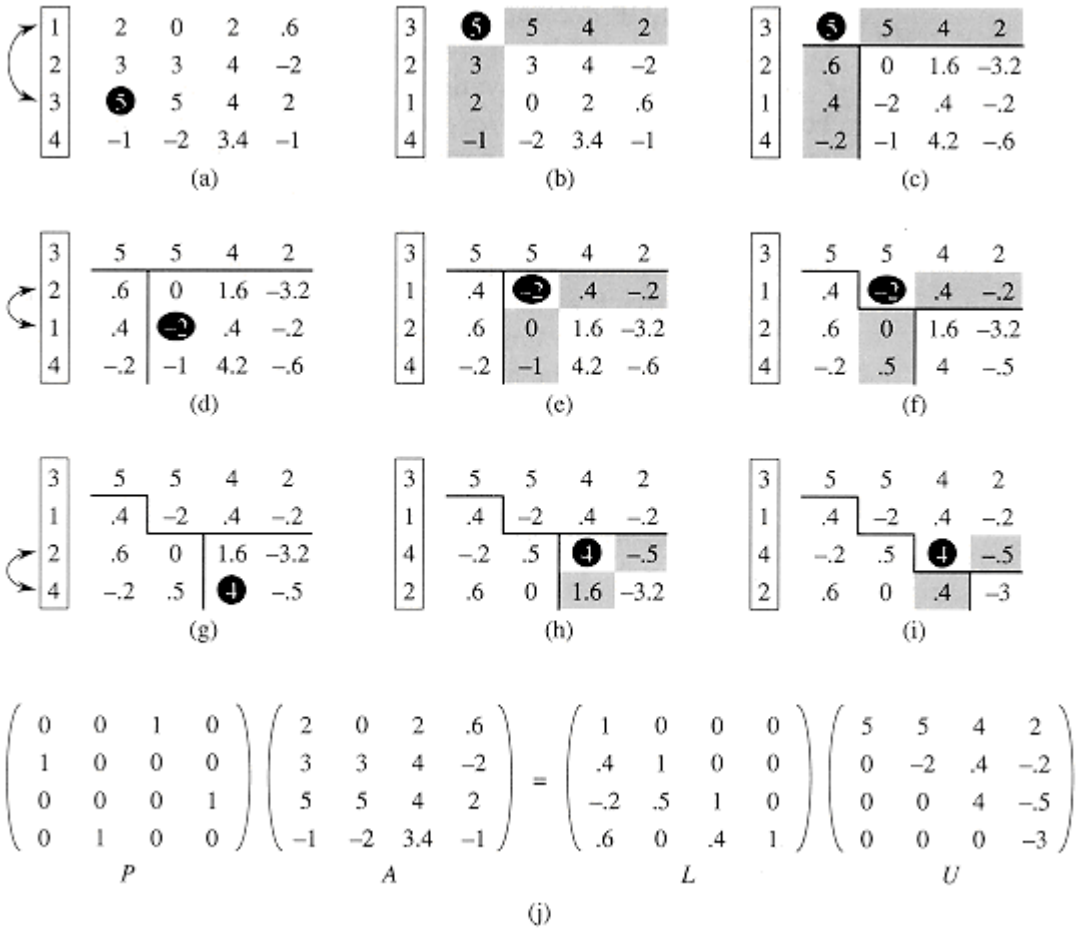

Figure 31.2 The operation of LUP-DECOMPOSITION. (a) The input matrix A with the identity permutation of the rows on the left. The first step of the algorithm determines that the element 5 in black in the third row is the pivot for the first column. (b) Rows 1 and 3 are swapped and the permutation is updated. The shaded column and row represent v and w^T. (c) The vector v is replaced by v/5, and the the lower right of the matrix is updated with the Schur complement. Lines divide the matrix into three regions: elements of U (above), elements of L (left), and elements of the Schur complement (lower right).(d)-(f) The second step.(g)-(i) The third step finishes the algorithm. (j) The LUP decomposition PA = LU.

If all elements in the current first column are zero, lines 10-11 report that the matrix is singular. To pivot, we exchange $\pi[k']$ with $\pi[k]$ in line 12 and exchange the kth and k'th rows of A in lines 13-14, thereby making the pivot element a_{kk}. (The entire rows are swapped because in the derivation of the method above, not only is $A' - vw^T/a_k 1$ multiplied by P', but so is $v/a_k 1$.) Finally, the Schur complement is computed by lines 15-18 in much the same way as it is computed by lines 4-9 of LU-DECOMPOSITION, except that here the operation is written to work "in place."

Because of its triply nested loop structure, the running time of LUP-DECOMPOSITION is $\Theta(n^3)$, the same as that of LU-DECOMPOSITION. Thus, pivoting costs us at most a constant factor in time.

Exercises

31.4-1

Solve the equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

by using forward substitution.

31.4-2

Find an LU decomposition of the matrix

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

31.4-3

Why does the **for** loop in line 4 of LUP-DECOMPOSITION run only up to $n - 1$, whereas the corresponding **for** loop in line 2 of LU-DECOMPOSITION runs all the way to n?

31.4-4

Solve the equation

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

by using an LUP decomposition.

31.4-5

Describe the LUP decomposition of a diagonal matrix.

31.4-6

Describe the LUP decomposition of a permutation matrix A, and prove that it is unique.

31.4-7

Show that for all $n \geq 1$, there exist singular $n \times n$ matrices that have LU decompositions.

31.4-8

Show how we can efficiently solve a set of equations of the form $Ax = b$ over the boolean quasiring $(\{0, 1\}, \vee, \wedge, 0, 1)$.

31.4-9

Suppose that A is an $m \times n$ real matrix of rank m, where $m < n$. Show how to find a size-n vector x_0 and an $m \times (n - m)$ matrix B of rank $n - m$ such that every vector of the form $x_0 + By$, for $y \in \mathbf{R}^{n-m}$, is a solution to the underdetermined equation $Ax = b$.

31.5 Inverting matrices

Although in practice we do not generally use matrix inverses to solve systems of linear equations, preferring instead to use more numerically stable techniques such as LUP decomposition, it is sometimes necessary to compute a matrix inverse. In this section, we show how LUP decomposition can be used to compute a matrix inverse. We also discuss the theoretically interesting question of whether the computation of a matrix inverse can be sped up using techniques such as Strassen's algorithm for matrix multiplication. Indeed, Strassen's original paper was motivated by the problem of showing that a set of a linear equations could be solved more quickly than by the usual method.

Computing a matrix inverse from an LUP decomposition

Suppose that we have an LUP decomposition of a matrix A in the form of three matrices L, U, and P such that $PA = LU$. Using LU-SOLVE, we can solve an equation of the form $Ax = b$ in time $\Theta(n^2)$. Since the LUP decomposition depends on A but not b, we can solve a second set of equations of the form $Ax = b'$ in additional time $\Theta(n^2)$. In general, once we have the LUP decomposition of A, we can solve, in time $\Theta(kn^2)$, k versions of the equation $Ax = b$ that differ only in b.

The equation

$$AX = I_n$$

(31.24)

can be viewed as a set of n distinct equations of the form $Ax = b$. These equations define the matrix X as the inverse of A. To be precise, let X_i denote the ith column of X, and recall that the unit vector e_i is the ith column of I_n. Equation (31.24) can then be solved for X by using the LUP decomposition for A to solve each equation

$$AX_i = e_i$$

separately for X_i. Each of the n X_i can be found in time $\Theta(n^2)$, and so the computation of X from the LUP decomposition of A takes time $\Theta(n^3)$. Since the LUP decomposition of A can be computed in time $\Theta(n^3)$, the inverse A^{-1} of a matrix A can be determined in time $\Theta(n^3)$.

Matrix multiplication and matrix inversion

We now show that the theoretical speedups obtained for matrix multiplication translate to speedups for matrix inversion. In fact, we prove something stronger: matrix inversion is equivalent to matrix multiplication, in the following sense. If $M(n)$ denotes the time to multiply two $n \times n$ matrices and $I(n)$ denotes the time to invert a nonsingular $n \times n$ matrix, then $I(n) = \Theta(M(n))$. We prove this result in two parts. First, we show that $M(n) =$

$O(I(n))$, which is relatively easy, and then we prove that $I(n) = O(M(n))$.

Theorem 31.11

If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Proof Let A and B be $n \times n$ matrices whose matrix product C we wish to compute. We define the $3n \times 3n$ matrix D by

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} .$$

The inverse of D is

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix} ,$$

and thus we can compute the product AB by taking the upper right $n \times n$ submatrix of D^{-1}.

We can construct matrix D in $\Theta(n^2) = O(I(n))$ time, and we can invert D in $O(I(3n)) = O(I(n))$ time, by the regularity condition on $I(n)$. We thus have

$M(n) = O(I(n))$.

Note that $I(n)$ satisfies the regularity condition whenever $I(n)$ does not have large jumps in value. For example, if $I(n) = \Theta(n^c \lg^d n)$ for any constants $c > 0$, $d \geq 0$, then $I(n)$ satisfies the regularity condition.

Reducing matrix inversion to matrix multiplication

The proof that matrix inversion is no harder than matrix multiplication relies on some properties of symmetric positive-definite matrices that will be proved in Section 31.6.

Theorem 31.12

If we can multiply two $n \times n$ real matrices in time $M(n)$, where $M(n) = \Omega(n^2)$ and $M(n) = O(M(n + k))$ for $0 \leq k \leq n$, then we can compute the inverse of any real nonsingular $n \times n$ matrix in time $O(M(n))$.

Proof We can assume that n is an exact power of 2, since we have

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

for any $k > 0$. Thus, by choosing k such that $n + k$ is a power of 2, we enlarge the matrix to

a size that is the next power of 2 and obtain the desired answer A^{-1} from the answer to the enlarged problem. The regularity condition on $M(n)$ ensures that this enlargement does not cause the running time to increase by more than a constant factor.

For the moment, let us assume that the $n \times n$ matrix A is symmetric and positive-definite. We partition A into four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} . \qquad (31.25)$$

(31.25)

Then, if we let

$$S = D - CB^{-1}C^{\mathrm{T}}$$

(31.26)

be the Schur complement of A with respect to B, we have

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} , \qquad (31.27)$$

(31.27)

since $AA^{-1} = I_n$, as can be verified by performing the matrix multiplication. The matrices B^{-1} and S^{-1} exist if A is symmetric and positive-definite, by Lemmas 31.13, 31.14, and 31.15 in Section 31.6, because both B and S are symmetric and positive-definite. By Exercise 31.1-3, $B^{-1}C^{\mathrm{T}} = (CB^{-1})^{\mathrm{T}}$ and $B^{-1}C^{\mathrm{T}}S^{-1} = (S^{-1}CB^{-1})^{\mathrm{T}}$. Equations (31.26) and (31.27) can therefore be used to specify a recursive algorithm involving 4 multiplications of $n/2 \times n/2$ matrices:

$C \leftarrow B^{-1}$,

$(CB^{-1}) \leftarrow C^{\mathrm{T}}$,

$S^{-1} \leftarrow (CB^{-1})$,

$(CB^{-1})^{\mathrm{T}} \leftarrow (S^{-1}CB^{-1})$.

Since we can multiply $n/2 \times n/2$ matrices using an algorithm for $n \times n$ matrices, matrix inversion of symmetric positive-definite matrices can be performed in time

$$I(n) \le 2I(n/2) + 4M(n) + O(n^2)$$

$$= 2I(n/2) + O(M(n))$$

$$= O(M(n)) .$$

It remains to prove that the asymptotic running time of matrix multiplication can be obtained for matrix inversion when A is invertible but not symmetric and positive-definite. The basic idea is that for any nonsingular matrix A, the matrix $A^{\mathrm{T}}A$ is symmetric

(by Exercise 31.1-3) and positive-definite (by Theorem 31.6). The trick, then, is to reduce the problem of inverting A to the problem of inverting $A^{\mathrm{T}}A$.

The reduction is based on the observation that when A is an $n \times n$ nonsingular matrix, we have

$$A^{-1} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}},$$

since $((A^{\mathrm{T}}A)^{-1} A^{\mathrm{T}})A = (A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A) = I_n$ and a matrix inverse is unique. Therefore, we can compute A^{-1} by first multiplying A^{T} by A to obtain $A^{\mathrm{T}}A$, then inverting the symmetric positive-definite matrix $A^{\mathrm{T}}A$ using the above divide-and-conquer algorithm, and finally multiplying the result by A^{T}. Each of these three steps takes $O(M(n))$ time, and thus any nonsingular matrix with real entries can be inverted in $O(M(n))$ time.

The proof of Theorem 31.12 suggests a means of solving the equation $Ax = b$ without pivoting, so long as A is nonsingular. We multiply both sides of the equation by A^{T}, yielding $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b$. This transformation doesn't affect the solution x, since A^{T} is invertible, so we can factor the symmetric positive-definite matrix $A^{\mathrm{T}}A$ by computing an LU decomposition. We then use forward and back substitution to solve for x with the right-hand side $A^{\mathrm{T}}b$. Although this method is theoretically correct, in practice the procedure LUP-DECOMPOSITION works much better. LUP decomposition requires fewer arithmetic operations by a constant factor, and it has somewhat better numerical properties.

Exercises

31.5-1

Let $M(n)$ be the time to multiply $n \times n$ matrices, and let $S(n)$ denote the time required to square an $n \times n$ matrix. Show that multiplying and squaring matrices have essentially the same difficulty: $S(n) = \Theta(M(n))$.

31.5-2

Let $M(n)$ be the time to multiply $n \times n$ matrices, and let $L(n)$ be the time to compute the LUP decomposition of an $n \times n$ matrix. Show that multiplying matrices and computing LUP decompositions of matrices have essentially the same difficulty: $L(n) = \Theta(M(n))$.

31.5-3

Let $M(n)$ be the time to multiply $n \times n$ matrices, and let $D(n)$ denote the time required to find the determinant of an $n \times n$ matrix. Show that finding the determinant is no harder than multiplying matrices: $D(n) = O(M(n))$.

31.5-4

Let $M(n)$ be the time to multiply $n \times n$ boolean matrices, and let $T(n)$ be the time to find the transitive closure of $n \times n$ boolean matrices. Show that $M(n) = O(T(n))$ and $T(n) =$

$O(M(n)\lg n)$.

31.5-5

Does the matrix-inversion algorithm based on Theorem 31.12 work when matrix elements are drawn from the field of integers modulo 2? Explain.

31.5-6

Generalize the matrix-inversion algorithm of Theorem 31.12 to handle matrices of complex numbers, and prove that your generalization works correctly. (*Hint*: Instead of the transpose of A, use the ***conjugate transpose*** A^*, which is obtained from the transpose of A by replacing every entry with its complex conjugate. Instead of symmetric matrices, consider ***Hermitian*** matrices, which are matrices A such that $A = A^*$.)

31.6 Symmetric positive-definite matrices and least-squares approximation

Symmetric positive-definite matrices have many interesting and desirable properties. For example, they are nonsingular, and LU decomposition can be performed on them without our having to worry about dividing by 0. In this section, we shall prove several other important properties of symmetric positive-definite matrices and show an interesting application to curve fitting by a least-squares approximation.

The first property we prove is perhaps the most basic.

Lemma 31.13

Any symmetric positive-definite matrix is nonsingular.

Proof Suppose that a matrix A is singular. Then by Corollary 31.3, there exists a nonzero vector x such that $Ax = 0$. Hence, $x^T Ax = 0$, and A cannot be positive-definite.

The proof that we can perform LU decomposition on a symmetric positive-definite matrix A without dividing by 0 is more involved. We begin by proving properties about certain submatrices of A. Define the kth ***leading submatrix*** of A to be the matrix A_k consisting of the intersection of the first k rows and first k columns of A.

Lemma 31.14

If A is a symmetric positive-definite matrix, then every leading submatrix of A is symmetric and positive-definite.

Proof That each leading submatrix A_k is symmetric is obvious. To prove that A_k is positive-definite, let x be a nonzero column vector of size k, and let us partition A as follows:

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} .$$

Then, we have

$$\begin{aligned} x^{\mathrm{T}} A_k x &= (x^{\mathrm{T}} \quad 0) \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} x \\ 0 \end{pmatrix} \\ &= (x^{\mathrm{T}} \quad 0) A \begin{pmatrix} x \\ 0 \end{pmatrix} \\ &> 0 , \end{aligned}$$

since A is positive-definite, and hence A_k is also positive-definite.

We now turn to some essential properties of the Schur complement. Let A be a symmetric positive-definite matrix, and let A_k be a leading $k \times k$ submatrix of A. Partition A as

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} . \qquad (31.28)$$

(31.28)

Then, the ***Schur complement*** of A with respect to A_k is defined to be

$$S = C - B A_k^{-1} B^{\mathrm{T}} . \qquad (31.29)$$

(31.29)

(By Lemma 31.14, A_k is symmetric and positive-definite; therefore, A_k^{-1} exists by Lemma 31.13, and S is well defined.) Note that our earlier definition (31.23) of the Schur complement is consistent with definition (31.29), by letting $k = 1$.

The next lemma shows that the Schur-complement matrices of symmetric positive-definite matrices are themselves symmetric and positive-definite. This result was used in Theorem 31.12, and its corollary is needed to prove the correctness of LU decomposition for symmetric positive-definite matrices.

Lemma 31.15

If A is a symmetric positive-definite matrix and A_k is a leading data $k \times k$ submatrix of A, then the Schur complement of A with respect to A_k is symmetric and positive-definite.

Proof That S is symmetric follows from Exercise 31.1-7. It remains to show that S is positive-definite. Consider the partition of A given in equation (31.28).

For any nonzero vector x, we have $x^{\mathrm{T}}Ax > 0$ by assumption. Let us break x into two subvectors y and z compatible with A_k and C, respectively. Because A_k^{-1} exists, we have

$$\begin{aligned} x^{\mathrm{T}}Ax &= (y^{\mathrm{T}}z^{\mathrm{T}})\begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} \\ &= y^{\mathrm{T}}A_k y + y^{\mathrm{T}}B^{\mathrm{T}}z + z^{\mathrm{T}}By + z^{\mathrm{T}}Cz \\ &= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}A_k(y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z\,, \quad (31.30) \end{aligned}$$

(31.30)

by matrix magic. (Verify by multiplying through.) This last equation amounts to "completing the square" of the quadractic form. (See Exercise 31.6-2.)

Since $x^{\mathrm{T}}Ax > 0$ holds for any nonzero x, let us pick any nonzero z and then choose $y = -A_k^{-1}B^{\mathrm{T}}z$, which causes the first term in equation (31.30) to vanish, leaving

$$z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z = z^{\mathrm{T}}Sz$$

as the value of the expression. For any $z \neq 0$, we therefore have $z^{\mathrm{T}}Sz = x^{\mathrm{T}}Ax > 0$, and thus S is positive-definite.

Corollary 31.16

LU decomposition of a symmetric positive-definite matrix never causes a division by 0.

Proof Let A be a symmetric positive-definite matrix. We shall prove something stronger than the statement of the corollary: every pivot is strictly positive. The first pivot is a_{11}. Let e_1 be the first unit vector, from which we obtain $a_{11} = e_1^{\mathrm{T}}Ae_1 > 0$. Since the first step of LU decomposition produces the Schur complement of A with respect to $A_1 = (a_{11})$, Lemma 31.15 implies that all pivots are positive by induction.

Least-squares approximation

Fitting curves to given sets of data points is an important application of symmetric positive-definite matrices. Suppose that we are given a set of m data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m),$$

where the y_i are known to be subject to measurement errors. We would like to determine a function $F(x)$ such that

$$y_i = F(x_i) + \eta_i,$$

(31.31)

for $i = 1, 2, \ldots, m$, where the approximation errors ηi are small. The form of the function F depends on the problem at hand. Here, we assume that it has the form of a linearly weighted sum,

$$F(x) = \sum_{j=1}^{n} c_j f_j(x) ,$$

where the number of summands n and the specific ***basis functions*** f_j are chosen based on knowledge of the problem at hand. A common choice is $f_j(x) = x^j$-1, which means that

$F(x) = c_1 + c_2x + c_3x^2 + \cdots + c_nx^n$-1

is a polynomial of degree n - 1 in x.

By choosing $n = m$, we calculate each y_i *exactly* in equation (31.31). Such a high-degree F "fits the noise" as well as the data, however, and generally gives poor results when used to predict y for previously unseen values of x. It is usually better to choose n significantly smaller than m and hope that by choosing the coefficients c_j well, we can obtain a function F that finds the significantl patterns in the data points without paying undue attention to the noise. Some theoretical principles exist for choosing n, but they are beyond the scope of this text. In any case, once n is chosen, we end up with an overdetermined set of equations that we wish to solve as well as we can. We now show how this can be done.

Let

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix}$$

denote the matrix of values of the basis functions at the given points; that is, $a_{ij} = f_j(x_i)$. Let $c = (c_k)$ denote the desired size-n vector of coefficients. Then,

$$\begin{aligned} Ac &= \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \\ &= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix} \end{aligned}$$

is the size-m vector of "predicted values" for y. Thus,

$\eta = Ac - y$

is the size-m vector of ***approximation errors***.

To minimize approximation errors, we choose to minimize the norm of the error vector η, which gives us a ***least-squares solution***, since

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2\right)^{1/2} .$$

Since

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right)^2 ,$$

we can minimize $||\eta||$ by differentiating $||\eta||^2$ with respect to each c_k and then setting the result to 0:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right) a_{ik} = 0 . \qquad (31.32)$$

(31.32)

The n equations (31.32) for $k = 1, 2, \ldots, n$ are equivalent to the single matrix equation

$(Ac - y)^T A = 0$

or, equivalently (using Exercise 31.1-3), to

$A^T(Ac - y) = 0$,

which implies

$A^T Ac = A^T y$.

(31.33)

In statistics, this is called the ***normal equation***. The matrix $A^T A$ is symmetric by Exercise 31.1-3, and if A has full column rank, then $A^T A$ is positive-definite as well. Hence, $(A^T A)^{-1}$ exists, and the solution to equation (31.33) is

$c = ((A^T A)^{-1} A^T) y$

$= A^+ y$,

(31.34)

where the matrix $A^+ = ((A^T A)^{-1} A^T)$ is called the ***pseudoinverse*** of the matrix A. The pseudoinverese is a natural generalization of the notion of a matrix inverse to the case in which A is nonsquare. (Compare equation (31.34) as the approximate solution to $Ac = y$ with the solution $A^{-1}b$ as the exact solution to $Ax = b$.)

As an example of producing a least-squares fit, suppose that we have 5 data points

$(-1,2),(1,1),(2,1),(3,0),(5,3),$

shown as black dots in Figure 31.3. We wish to fit these points with a quadratic polynomial

$F(x) = c_1 + c_2x + c_3x^2$.

We start with the matrix of basis-function values

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix},$$

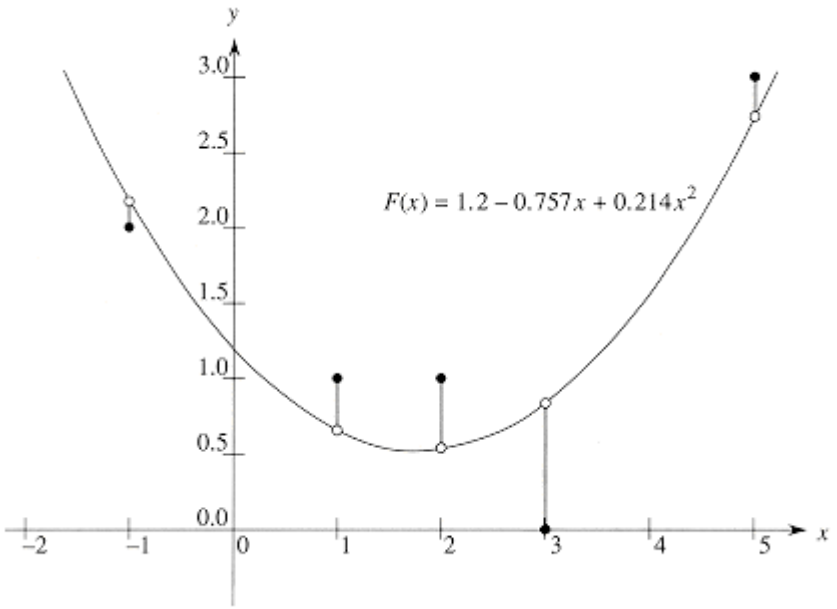

Figure 31.3 The least-squares fit of a quadratic polynomial to the set of data points {(-1, 2), (1, 1), (2, 1),(3, 0), (5, 3)}. The black dots are the data points, and the white dots are their estimated values predicted by the polynomial F(x) = 1.2 - 0.757x + 0.214x^2, the quadratic polynomial that minimizes the sum of the squared errors. The error for each data point is shown as a shaded line.

whose pseudoinverse is

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix}.$$

Multiplying y by A^+, we obtain the coefficient vector

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix},$$

which corresponds to the quadratic polynomial

$F(x) = 1.200 - 0.757x + 0.214x^2$

as the closest-fitting quadratic to the given data, in a least-squares sense.

As a practical matter, we solve the normal equation (31.33) by multiplying y by A^{T} and then finding an LU decomposition of $A^{\mathrm{T}} A$. If A has full rank, the matrix $A^{\mathrm{T}} A$ is guaranteed to be nonsingular, because it is symmetric and positive-definite. (See Exercise 31.1-3 and Theorem 31.6.)

Exercises

31.6-1

Prove that every diagonal element of a symmetric positive-definite matrix is positive.

31.6-2

Let $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ be a 2×2 symmetric positive-definite matrix. Prove that its determinant $ac - b^2$ is positive by "completing the square" in a manner similar to that used in the proof of Lemma 31.15.

31.6-3

Prove that the maximum element in a symmetric positive-definite matrix lies on the diagonal.

31.6-4

Prove that the determinant of each leading submatrix of a symmetric positive-definite matrix is positive.

31.6-5

Let A_k denote the kth leading submatrix of a symmetric positive-definite matrix A. Prove that $\det(A_k)/\det(A_k\text{-}1)$ is the kth pivot during LU decomposition, where by convention $\det(A_0) = 1$.

31.6-6

Find the function of the form

$F(x) = c_1 + c_2 x \lg x + c_3 e^x$

that is the best least-squares fit to the data points

$(1,1),(2,1),(3,3),(4,8)$.

31.6-7

Show that the pseudoinverse A^+ satisfies the following four equations:

$AA^+A = A$,

$A^+AA^+ = A^+$,

$(AA^+)^T = AA^+$,

$(A^+A)^T = A^+A$.

Problems

31-1 Shamir's boolean matrix multiplication algorithm

In Section 31.3, we observed that Strassen's matrix-multiplication algorithm cannot be applied directly to boolean matrix multiplication because the boolean quasiring Q = ({0, 1}, $\vee$, $\wedge$, 0, 1) is not a ring. Theorem 31.10 showed that if we used arithmetic operations on words of $O(\lg n)$ bits, we could nevertheless apply Strassen's method to multiply $n \times n$ boolean matrices in $O(n^{\lg 7})$ time. In this problem, we investigate a probabilistic method that uses only bit operations to achieve nearly as good a bound but with some small chance of error.

a. Show that R = ({0, 1}, $\oplus$, $\wedge$, 0, 1), where $\oplus$ is the XOR (exclusive-or) function, is a ring.

Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ boolean matrices, and let $C = (c_{ij}) = AB$ in the quasiring Q. Generate $A' = (a'_{ij})$ from A using the following randomized procedure:

- If $a_{ij} = 0$, then let $a'_{ij} = 0$.
- If $A_{ij} = 1$, then let $a'_{ij} = 1$ with probability 1/2 and let $a'_{ij} = 0$ with probability 1/2. The random choices for each entry are independent.

b. Let $C' = (c'_{ij}) = A'B$ in the ring R. Show that $c_{ij} = 0$ implies $c'_{ij} = 0$. Show that $c_{ij} = 1$ implies $c'_{ij} = 1$ with probability 1/2.

c. Show that for any $\epsilon > 0$, the probability is at most ϵ/n^2 that a given c'_{ij} never takes on the value c_{ij} for $\lg(n^2/\epsilon)$ independent choices of the matrix A'. Show that the probability is at least $1 - \epsilon$ that all c'_{ij} take on their correct values at least once.

d. Give an $O(n^{\lg 7} \lg n)$-time randomized algorithm that computes the product in the boolean quasiring Q of two $n \times n$ matrices with probability at least $1 - 1/n^k$ for any constant $k > 0$. The only operations permitted on matrix elements are $\wedge$, $\vee$, and $\oplus$.

31-2 Tridiagonal systems of linear equations

Consider the tridiagonal matrix

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} .$$

a. Find an LU decomposition of A.

b. Solve the equation $Ax = (1\ 1\ 1\ 1\ 1)^T$ by using forward and back substitution.

c. Find the inverse of A.

d. Show that for any $n \times n$ symmetric, positive-definite, tridiagonal matrix A and any n-vector b, the equation $Ax = b$ can be solved in $O(n)$ time by performing an LU decomposition. Argue that any method based on forming A^{-1} is asymptotically more expensive in the worst case.

e. Show that for any $n \times n$ nonsingular, tridiagonal matrix A and any n-vector b, the equation $Ax = b$ can be solved in $O(n)$ time by performing an LUP decomposition.

31-3 Splines

A practical method for interpolating a set of points with a curve is to use ***cubic splines.*** We are given a set $\{(x_i, y_i): i = 0,1, \ldots, n\}$ of $n + 1$ point-value pairs, where $x_0 < x_1 < \cdots < x_n$. We wish to fit a piecewise-cubic curve (spline) $f(x)$ to the points. That is, the curve $f(x)$ is made up of n cubic polynomials $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ for $i = 0,1, \ldots, n - 1$, where if x falls in the range $x_i \le x \le x_i + 1$, then the value of the curve is given by $f(x) = f_i(x - x_i)$. The points x_i at which the cubic polynomials are "pasted" together are called ***knots***. For simplicity, we shall assume that $x_i = i$ for $i = 0,1, \ldots, n$.

To ensure continuity of $f(x)$, we require that

$$f(x_i) = f_i(0) = y_i ,$$

$$f(x_i+1) = f_i(1) = y_i+1$$

for $i = 0, 1, \ldots, n - 1$. To ensure that $f(x)$ is sufficiently smooth, we also insist that there be continuity of the first derivative at each knot:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

for $i = 0, 1, \ldots, n - 1$.

a. Suppose that for $i = 0, 1, \ldots, n$, we are given not only the point-value pairs $\{(x_i, y_i)\}$ but also the first derivatives $D_i = f'(x_i)$ at each knot. Express each coefficient a_i, b_i, c_i, and d_i in terms of the values y_i, y_i+1, D_i, and D_i+1. (Remember that $x_i = i$.) How quickly can the $4n$ coefficients be computed from the point-value pairs and first derivatives?

The question remains of how to choose the first derivatives of $f(x)$ at the knots. One

method is to require the second derivatives to be continuous at the knots:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

for $i = 0, 1, \ldots, n - 1$. At the first and last knots, we assume that $f''(x_0) = f_0''(0) = 0$ and $f''(x_n) = f_n''(1) = 0$; these assumptions make $f(x)$ a ***natural*** cubic spline.

b. Use the continuity constraints on the second derivative to show that for $i = 1, 2, \ldots, n - 1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) .$$

(31.35)

c. Show that

$$2D_0 + D_1 = 3(y_1 - y_0) ,$$

(31.36)

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) .$$

(31.37)

d. Rewrite equations (31.35)--(31.37) as a matrix equation involving the vector $D = \langle D_0, D_1, \ldots, D_n \rangle$ of unknowns. What attributes does the matrix in your equation have?

e. Argue that a set of $n + 1$ point-value pairs can be interpolated with a natural cubic spline in $O(n)$ time (see Problem 31-2).

f. Show how to determine a natural cubic spline that interpolates a set of $n + 1$ points (x_i, y_i) satisfying $x_0 < x_1 < \cdots < x_n$, even when x_i is not necessarily equal to i. What matrix equation must be solved, and how quickly does your algorithm run?

Chapter notes

There are many excellent texts available that describe numerical and scientific computation in much greater detail than we have room for here. The following are especially readable: George and Liu [81], Golub and Van Loan [89], Press, Flannery, Teukolsky, and Vetterling[161, 162], and Strang[181, 182].

The publication of Strassen's algorithm in 1969 [183] caused much excitement. Before then, it was hard to imagine that the naive algorithm could be improved upon. The asymptotic upper bound on the difficulty of matrix multiplication has since been considerably improved. The most asymptotically efficient algorithm for multiplying $n \times n$

matrices to date, due to Coppersmith and Winograd [52], has a running time of $O(n^{2.376})$. The graphical presentation of Strassen's algorithm is due to Paterson [155]. Fischer and Meyer [67] adapted Strassen's algorithm to boolean matrices (Theorem 31.10) .

Gaussian elimination, upon which the LU and LUP decompositions are based, was the first systematic method for solving linear systems of equations. It was also one of the earliest numerical algorithms. Although it was known earlier, its discovery is commonly attributed to C. F. Gauss (1777-1855). In his famous paper [183], Strassen also showed that an $n \times n$ matrix can be inverted in $O(n^{\lg 7})$ time. Winograd [203] originally proved that matrix multiplication is no harder than matrix inversion, and the converse is due to Aho, Hopcroft, and Ullman [4].

Strang [182] has an excellent presentation of symmetric positive-definite matrices and on linear algebra in general. He makes the following remark on page 334: "My class often asks about *unsymmetric* positive definite matrices. I never use that term."

Go to Chapter 32 Back to Table of Contents