28 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 28.1 introduces basic matrix concepts and notations, Section 28.2 presents Strassen's surprising algorithm for multiplying two $n \times n$ matrices in $\Theta(n^{\lg 7}) = O(n^{2.81})$ time. Section 28.3 shows how to solve a set of linear equations using LUP decompositions. Then, Section 28.4 explores the close relationship between the problem of multiplying matrices and the problem of inverting a matrix. Finally, Section 28.5 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.

One important issue that arises in practice is ***numerical stability***. Due to the limited precision of floating-point representations in actual computers, round-off errors in numerical computations may become amplified over the course of a computation, leading to incorrect results; such computations are numerically unstable. Although we shall briefly consider numerical stability on occasion, we do not focus on it in this chapter. We refer the reader to the excellent book by Golub and Van Loan [125] for a thorough discussion of stability issues.

28.1 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,

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \end{aligned} \tag{28.1}$$

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 (28.1),

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

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

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{28.2}$$

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:

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

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:

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

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

   $$\begin{aligned} I_n &= \operatorname{diag}(1, 1, \ldots, 1) \\ &= \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix} . \end{aligned}$$

   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:

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

   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:

$$\begin{aligned} A + 0 &= A \\ &= 0 + A . \end{aligned}$$

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,

$$\begin{aligned} A + (-A) &= 0 \\ &= (-A) + A . \end{aligned}$$

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

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \qquad (28.3)$$

for $i = 1, 2, \ldots, m$ and $k = 1, 2, \ldots, p$. The procedure MATRIX-MULTIPLY in Section 25.1 implements matrix multiplication in the straightforward manner based on equation (28.3), 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 so 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:

$$A(BC) = (AB)C \qquad (28.4)$$

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

$$\begin{aligned} A(B + C) &= AB + AC , \\ (B + C)D &= BD + CD . \end{aligned} \qquad (28.5)$$

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

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

and

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

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

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

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

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

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 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 28.1-3.) If A and B are nonsingular $n \times n$ matrices, then

$$(BA)^{-1} = A^{-1}B^{-1} . \tag{28.6}$$

The inverse operation commutes with the transpose operation:

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

The vectors $x_1, x_2, \ldots, x_n$ are ***linearly dependent*** if there exist coefficients $c_1, c_2, \ldots, c_n$, not all of which are zero, such that $c_1x_1 + c_2x_2 + \cdots + c_nx_n = 0$. For example, the row vectors $x_1 = (\,1 \quad 2 \quad 3\,)$, $x_2 = (\,2 \quad 6 \quad 4\,)$, and $x_3 = (\,4 \quad 11 \quad 9\,)$ 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. An $m \times n$ matrix has ***full column rank*** if its rank is n. A fundamental property of ranks is given by the following theorem.

Theorem 28.1
A square matrix has full rank if and only if it is nonsingular. ■

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 28.1-9, and its corollary relate the notions of column rank and singularity to null vectors.

Theorem 28.2
A matrix A has full column rank if and only if it does not have a null vector. ■

Corollary 28.3
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

$$\det(A) = \begin{cases} a_{11} & \text{if } n = 1 \,, \\ \displaystyle\sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(A_{[1j]}) & \text{if } n > 1 \,. \end{cases} \tag{28.7}$$

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 28.4 (Determinant properties)

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 (or any two columns) are exchanged.

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

Theorem 28.5

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 $x^{\mathrm{T}} A x > 0$ for all size-n vectors $x \neq 0$. For example, the identity matrix is positive-definite, since for any nonzero vector $x = (\, x_1 \quad x_2 \quad \cdots \quad x_n \,)^{\mathrm{T}}$,

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

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

Theorem 28.6
For any matrix A with full column rank, the matrix $A^{\mathrm{T}}A$ is positive-definite.

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

$$\begin{aligned} x^{\mathrm{T}}(A^{\mathrm{T}}A)x &= (Ax)^{\mathrm{T}}(Ax) \quad \text{(by Exercise 28.1-2)} \\ &= \|Ax\|^2 \ . \end{aligned}$$

Note that $\|Ax\|^2$ is just the sum of the squares of the elements of the vector Ax. Therefore, $\|Ax\|^2 \geq 0$. 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 28.2. Hence, $A^{\mathrm{T}}A$ is positive-definite. ■

Other properties of positive-definite matrices will be explored in Section 28.5.

Exercises

28.1-1
Show that if A and B are symmetric $n \times n$ matrices, then so are $A + B$ and $A - B$.

28.1-2
Prove that $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$ and that $A^{\mathrm{T}}A$ is always a symmetric matrix.

28.1-3
Prove that matrix inverses are unique, that is, if B and C are inverses of A, then $B = C$.

28.1-4
Prove that the product of two lower-triangular matrices is lower-triangular. Prove that the determinant of a lower-triangular 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.

28.1-5
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.

28.1-6

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.

28.1-7

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.

28.1-8

Show that if A is a nonsingular, symmetric, $n \times n$ matrix, then A^{-1} is symmetric. Show that if B is an arbitrary $m \times n$ matrix, then the $m \times m$ matrix given by the product BAB^{T} is symmetric.

28.1-9

Prove Theorem 28.2. That is, 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.)

28.1-10

Prove that for any two compatible matrices A and B,

$$\operatorname{rank}(AB) \le \min(\operatorname{rank}(A), \operatorname{rank}(B)) ,$$

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

28.1-11

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.)

28.2 Strassen's algorithm for matrix multiplication

This section presents Strassen's remarkable recursive algorithm for multiplying $n \times n$ matrices, which runs in $\Theta(n^{\lg 7}) = O(n^{2.81})$ time. For sufficiently large values of n, therefore, it outperforms the naive $\Theta(n^3)$ matrix-multiplication algorithm MATRIX-MULTIPLY from Section 25.1.

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:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} . \tag{28.8}$$

(Exercise 28.2-2 deals with the situation in which n is not an exact power of 2.) Equation (28.8) corresponds to the four equations

$$r = ae + bg , \tag{28.9}$$

$$s = af + bh , \tag{28.10}$$

$$t = ce + dg , \tag{28.11}$$

$$u = cf + dh . \tag{28.12}$$

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:

$$T(n) = 8T(n/2) + \Theta(n^2) . \tag{28.13}$$

Unfortunately, recurrence (28.13) 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

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \qquad (28.14) \\ &= \Theta(n^{\lg 7}) \\ &= O(n^{2.81}) . \end{aligned}$$

Strassen's method has four steps:

1. Divide the input matrices A and B into $n/2 \times n/2$ submatrices, as in equation (28.8).
2. Using $\Theta(n^2)$ scalar additions and subtractions, compute 14 matrices A_1, B_1, A_2, $B_2, \ldots, A_7$, B_7, each of which is $n/2 \times n/2$.
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 (28.14). 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

$$\begin{aligned} P_i &= A_i B_i \\ &= (\alpha_{i1} a + \alpha_{i2} b + \alpha_{i3} c + \alpha_{i4} d) \cdot (\beta_{i1} e + \beta_{i2} f + \beta_{i3} g + \beta_{i4} h) \ , \end{aligned} \tag{28.15}$$

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 (28.15). For example, we can rewrite equation (28.9) as

$$\begin{aligned} r &= ae + bg \\ &= \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \end{aligned}$$

$$= \begin{array}{ccccc} & e & f & g & h \\ a & + & \cdot & \cdot & \cdot \\ b & \cdot & \cdot & + & \cdot \\ c & \cdot & \cdot & \cdot & \cdot \\ d & \cdot & \cdot & \cdot & \cdot \end{array} .$$

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:

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

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

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

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:

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

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

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

$$\begin{aligned}
P_3 &= A_3 B_3 \\
&= (c+d)\cdot e \\
&= ce + de \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}
\end{aligned}$$

and

$$\begin{aligned}
P_4 &= A_4 B_4 \\
&= d\cdot (g-e) \\
&= dg - de \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix} .
\end{aligned}$$

Let us define an ***essential term*** to be one of the eight terms appearing on the right-hand side of one of the equations (28.9)–(28.12). We have now used 4 products to compute the two submatrices s and t whose essential terms are af, bh, ce, and dg. Note that P_1 computes the essential term af, P_2 computes the essential term bh, P_3 computes the essential term ce, and P_4 computes the essential term dg. Thus, it remains for us to compute the remaining two submatrices r and u, whose essential terms are ae, bg, cf, 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:

$$\begin{aligned}
P_5 &= A_5 B_5 \\
&= (a+d)\cdot (e+h) \\
&= ae + ah + de + dh
\end{aligned}$$

$$= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} .$$

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 canceled somehow. We can use P_4 and P_2 to cancel them, but two other inessential terms then appear:

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

By adding an additional product

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

however, we obtain

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

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:

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

By subtracting an additional product

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

we now obtain

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

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

From a practical point of view, Strassen's algorithm is often not the method of choice for matrix multiplication, for four reasons:

1. The constant factor hidden in the running time of Strassen's algorithm is larger than the constant factor in the naive $\Theta(n^3)$ method.
2. When the matrices are sparse, methods tailored for sparse matrices are faster.
3. Strassen's algorithm is not quite as numerically stable as the naive method.
4. The submatrices formed at the levels of recursion consume space.

The latter two reasons were mitigated around 1990. Higham [145] demonstrated that the difference in numerical stability had been overemphasized; although Strassen's algorithm is too numerically unstable for some applications, it is within acceptable limits for others. Bailey et al. [30] discuss techniques for reducing the memory requirements for Strassen's algorithm.

In practice, fast matrix-multiplication implementations for dense matrices use Strassen's algorithm for matrix sizes above a "crossover point," and they switch to the naive method once the subproblem size reduces to below the crossover point. The exact value of the crossover point is highly system dependent. Analyses that count operations but ignore effects from caches and pipelining have produced crossover points as low as $n = 8$ (by Higham [145]) or $n = 12$ (by Huss-Lederman et al. [163]). Empirical measurements typically yield higher crossover points, with some as low as $n = 20$ or so. For any given system, it is usually straightforward to determine the crossover point by experimentation.

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 exactly how hard matrix multiplication really is.

Exercises

28.2-1

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.

28.2-2

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

28.2-3

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?

28.2-4

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? How does it compare to Strassen's algorithm?

28.2-5

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.

28.2-6

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.

28.3 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 $x_1, x_2, \ldots, x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n . \end{aligned} \tag{28.16}$$

A set of values for $x_1, x_2, \ldots, x_n$ that satisfy all of the equations (28.16) simultaneously is said to be a ***solution*** to these equations. In this section, we treat only the case in which there are exactly n equations in n unknowns.

We can conveniently rewrite equations (28.16) 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 $A = (a_{ij})$, $x = (x_i)$, and $b = (b_i)$, as

$$Ax = b . \tag{28.17}$$

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

$$x = A^{-1}b \tag{28.18}$$

is the solution vector. We can prove that x is the unique solution to equation (28.17) as follows. If there are two solutions, x and x', then $Ax = Ax' = b$ and

$$\begin{aligned} x &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x' . \end{aligned}$$

In this section, we shall be concerned predominantly with the case in which A is nonsingular or, equivalently (by Theorem 28.1), 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, 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 28.5.

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. There is, fortunately, another approach—LUP decomposition—that is numerically stable and has the further advantage of being faster in practice.

Overview of LUP decomposition

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

$$PA = LU \ , \qquad (28.19)$$

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 (28.19) 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 (28.17) $Ax = b$ 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 28.1-5 amounts to permuting the equations (28.16). Using our decomposition (28.19), 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

$$Ly = Pb \tag{28.20}$$

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

$$Ux = y \tag{28.21}$$

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 28.1-5):

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

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 (28.20) 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 (28.20) can be rewritten as

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

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 the first equation. Like forward substitution, this process runs in $\Theta(n^2)$ time. Since U is upper-triangular, we can rewrite the system (28.21) as

$$\begin{array}{rcl}
u_{11}x_1 + u_{12}x_2 + \cdots + \; u_{1,n-2}x_{n-2} + \; u_{1,n-1}x_{n-1} + \; u_{1n}x_n & = & y_1 \; , \\
u_{22}x_2 + \cdots + \; u_{2,n-2}x_{n-2} + \; u_{2,n-1}x_{n-1} + \; u_{2n}x_n & = & y_2 \; , \\
& \vdots & \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n & = & y_{n-2} \; , \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n & = & y_{n-1} \; , \\
u_{n,n}x_n & = & y_n \; .
\end{array}$$

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

$$\begin{array}{rcl}
x_n & = & y_n / u_{n,n} \; , \\
x_{n-1} & = & (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \; , \\
x_{n-2} & = & (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} \; , \\
& \vdots &
\end{array}$$

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($L, U, \pi, b$)
$n \gets rows[L]$
for $i \gets 1$ to $n$
    do $y_i \gets b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$
for $i \gets n$ downto 1
    do $x_i \gets \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j\right) / u_{ii}$
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

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

where

$$\begin{aligned} A &= \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} , \\ b &= \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} , \end{aligned}$$

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

$$\begin{aligned} L &= \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} , \\ U &= \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} , \\ P &= \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} . \end{aligned}$$

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

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix} ,$$

obtaining

$$y = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

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

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix} ,$$

thereby obtaining the desired answer

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

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:

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

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

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} . \end{aligned} \tag{28.22}$$

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

$$A' - vw^{\mathrm{T}}/a_{11} \tag{28.23}$$

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

We claim that if A is nonsingular, then the Schur complement is nonsingular, too. Why? Suppose that the Schur complement, which is $(n-1) \times (n-1)$, is singular. Then by Theorem 28.1, it has row rank strictly less than $n - 1$. Because the bottom $n - 1$ entries in the first column of the matrix

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

are all 0, the bottom $n - 1$ rows of this matrix must have row rank strictly less than $n - 1$. The row rank of the entire matrix, therefore, is strictly less than n. Applying Exercise 28.1-10 to equation (28.22), A has rank strictly less than n, and from Theorem 28.1 we derive the contradiction that A is singular.

Because the Schur complement is nonsingular, we can now recursively find an LU decomposition of it. 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.)

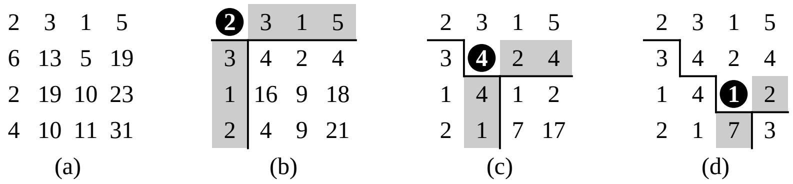
Of course, if $a_{11} = 0$, this method doesn't work, because it divides by 0. It also doesn't work if the upper leftmost entry of the Schur complement $A' - vw^{\mathrm{T}}/a_{11}$ is 0, since we divide by it in the next step of the recursion. The elements by which we divide during LU decomposition 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 28.5.

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 LUP-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  $n \leftarrow rows[A]$
 2  for $k \leftarrow 1$ to $n$
 3       do $u_{kk} \leftarrow a_{kk}$
 4          for $i \leftarrow k+1$ to $n$
 5              do $l_{ik} \leftarrow a_{ik}/u_{kk}$     ▷ $l_{ik}$ holds $v_i$
 6                 $u_{ki} \leftarrow a_{ki}$          ▷ $u_{ki}$ holds $w_i^{\mathrm{T}}$
 7          for $i \leftarrow k+1$ to $n$
 8              do for $j \leftarrow k+1$ to $n$
 9                     do $a_{ij} \leftarrow a_{ij} - l_{ik}u_{kj}$
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. (We don't need to divide by a_{kk} in line 9



$$\underset{A}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}$$

(e)

Figure 28.1 The operation of LU-DECOMPOSITION. **(a)** The matrix A. **(b)** The element $a_{11} = 2$ in the black circle 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 $A' - vw^{\mathrm{T}}/a_{11}$ occupies the lower right. **(c)** We now operate on the Schur complement matrix produced from part (b). The element $a_{22} = 4$ in the black circle 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 $A = LU$.

because we already did so when we computed l_{ik} in line 5.) Because line 9 is triply nested, LU-DECOMPOSITION runs in time $\Theta(n^3)$.

Figure 28.1 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 somewhere in 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 28.4 and 28.5.) 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 28.1-5). 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:

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

As we saw for LU decomposition, if A is nonsingular, then the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ is nonsingular, too. Therefore, we can inductively find an LUP decomposition for it, with unit lower-triangular matrix L', upper-triangular matrix U', and permutation matrix P', such that

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

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 28.1-5). We now have

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

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

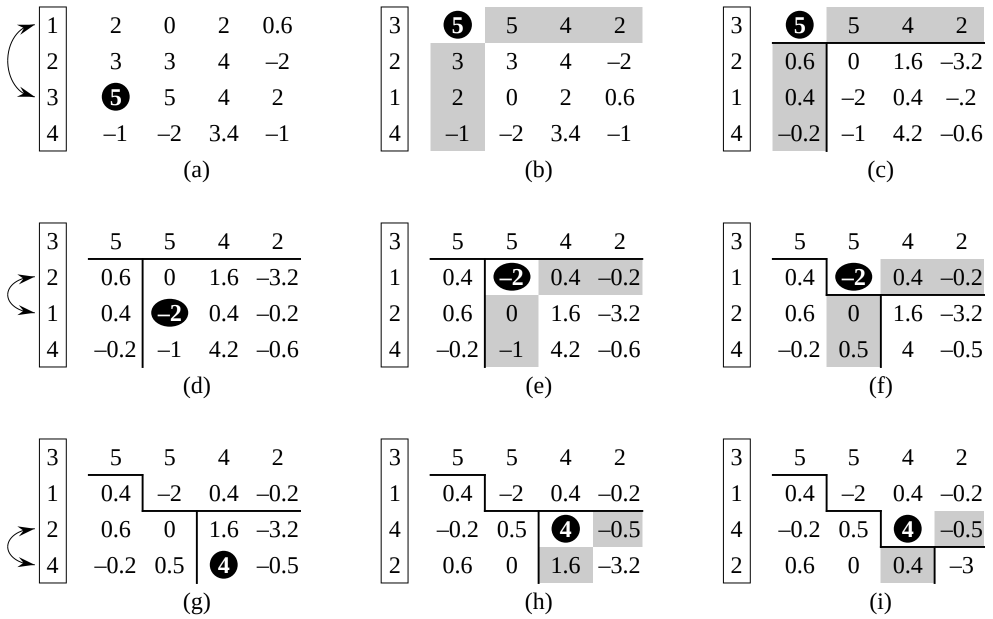
yielding the LUP decomposition. Because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.

Notice that in this derivation, unlike the one for LU decomposition, both the column vector v/a_{k1} and the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ 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($A$)
n ← rows[A]
for i ← 1 to n
     do $\pi[i] \leftarrow i$
for k ← 1 to n
     do p ← 0
        for i ← k to n
            do if $|a_{ik}| > p$
                  then $p \leftarrow |a_{ik}|$
                       $k' \leftarrow i$
        if p = 0
          then error "singular matrix"
        exchange $\pi[k] \leftrightarrow \pi[k']$
        for i ← 1 to n
            do exchange $a_{ki} \leftrightarrow a_{k'i}$
        for i ← k + 1 to n
            do $a_{ik} \leftarrow a_{ik}/a_{kk}$
               for j ← k + 1 to n
                   do $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$
```



$$\underset{P}{\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}} \underset{A}{\begin{pmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.4 & 1 & 0 & 0 \\ -0.2 & 0.5 & 1 & 0 \\ 0.6 & 0 & 0.4 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0.4 & -0.2 \\ 0 & 0 & 4 & -0.5 \\ 0 & 0 & 0 & -3 \end{pmatrix}}$$

(j)

Figure 28.2 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 the black circle 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 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. No further changes occur on the fourth, and final, step. **(j)** The LUP decomposition $PA = LU$.

Figure 28.2 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. 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 $A' - vw^{\mathrm{T}}/a_{k1}$ multiplied by P', but so is v/a_{k1}.) 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, LUP-DECOMPOSITION has a running time of $\Theta(n^3)$, which is the same as that of LU-DECOMPOSITION. Thus, pivoting costs us at most a constant factor in time.

Exercises

28.3-1

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.

28.3-2

Find an LU decomposition of the matrix

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

28.3-3

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.

28.3-4

Describe the LUP decomposition of a diagonal matrix.

28.3-5

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

28.3-6

Show that for all $n \geq 1$, there exists a singular $n \times n$ matrix that has an LU decomposition.

28.3-7

In LU-DECOMPOSITION, is it necessary to perform the outermost **for** loop iteration when $k = n$? How about in LUP-DECOMPOSITION?

28.4 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 prove that matrix multiplication and computing the inverse of a matrix are equivalently hard problems, in that (subject to technical conditions) we can use an algorithm for one to solve the other in the same asymptotic running time. Thus, we can use Strassen's algorithm for matrix multiplication to invert a matrix. 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 LUP-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 run LUP-SOLVE on 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

$$AX = I_n \tag{28.24}$$

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 (28.24) 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 columns 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, then there is a way to invert an $n \times n$ matrix in time $O(M(n))$. Moreover, if $I(n)$ denotes the time to invert a nonsingular $n \times n$ matrix, then there is a way to multiply two $n \times n$ matrices in time $O(I(n))$. We prove these results as two separate theorems.

Theorem 28.7 (Multiplication is no harder than inversion)
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 $I(n) = \Theta(n^c \lg^d n)$ for any constants $c > 0$ and $d \geq 0$.

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 28.5.

Theorem 28.8 (Inversion is no harder than multiplication)
Suppose we can multiply two $n \times n$ real matrices in time $M(n)$, where $M(n) = \Omega(n^2)$ and $M(n)$ satisfies the two regularity conditions $M(n+k) = O(M(n))$ for any k in the range $0 \le k \le n$ and $M(n/2) \le cM(n)$ for some constant $c < 1/2$. 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 first 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:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} . \tag{28.25}$$

Then, if we let

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.26}$$

be the Schur complement of A with respect to B (we shall see more about this form of Schur complement in Section 28.5), we have

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} , \tag{28.27}$$

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 28.9, 28.10, and 28.11 in Section 28.5, because both B and S are symmetric and positive-definite. By Exercise 28.1-2, $B^{-1}C^{\mathrm{T}} = (CB^{-1})^{\mathrm{T}}$ and $B^{-1}C^{\mathrm{T}}S^{-1} = (S^{-1}CB^{-1})^{\mathrm{T}}$. Equations (28.26) and (28.27) can therefore be used to specify a recursive algorithm involving four multiplications of $n/2 \times n/2$ matrices:

$$\begin{aligned} & C \cdot B^{-1} , \\ & (CB^{-1}) \cdot C^{\mathrm{T}} , \\ & S^{-1} \cdot (CB^{-1}) , \\ & (CB^{-1})^{\mathrm{T}} \cdot (S^{-1}CB^{-1}) . \end{aligned}$$

Thus, we can invert an $n \times n$ symmetric positive-definite matrix by inverting two $n/2 \times n/2$ matrices (B and S), performing these four multiplications of $n/2 \times n/2$ matrices (which we can do with an algorithm for $n \times n$ matrices), plus an additional cost of $O(n^2)$ for extracting submatrices from A and performing a constant number of additions and subtractions on these $n/2 \times n/2$ matrices. We get the recurrence

$$\begin{aligned} I(n) &\le 2I(n/2) + 4M(n) + O(n^2) \\ &= 2I(n/2) + \Theta(M(n)) \\ &= O(M(n)) \; . \end{aligned}$$

The second line holds because $M(n) = \Omega(n^2)$, and the third line follows because the second regularity condition in the statement of the theorem allows us to apply case 3 of the master theorem (Theorem 4.1).

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 28.1-2) and positive-definite (by Theorem 28.6). 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 28.8 suggests a means of solving the equation $Ax = b$ by using LU decomposition 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, and 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

28.4-1

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: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time squaring algorithm, and an $S(n)$-time squaring algorithm implies an $O(S(n))$-time matrix-multiplication algorithm.

28.4-2

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: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time LUP-decomposition algorithm, and an $L(n)$-time LUP-decomposition algorithm implies an $O(L(n))$-time matrix-multiplication algorithm.

28.4-3

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 multiplying matrices and computing the determinant have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time determinant algorithm, and a $D(n)$-time determinant algorithm implies an $O(D(n))$-time matrix-multiplication algorithm.

28.4-4

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. (See Section 25.2.) Show that an $M(n)$-time boolean matrix-multiplication algorithm implies an $O(M(n) \lg n)$-time transitive-closure algorithm, and a $T(n)$-time transitive-closure algorithm implies an $O(T(n))$-time boolean matrix-multiplication algorithm.

28.4-5

Does the matrix-inversion algorithm based on Theorem 28.8 work when matrix elements are drawn from the field of integers modulo 2? Explain.

28.4-6 ⋆

Generalize the matrix-inversion algorithm of Theorem 28.8 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 $A = A^*$.)

28.5 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 28.9
Any positive-definite matrix is nonsingular.

Proof Suppose that a matrix A is singular. Then by Corollary 28.3, there exists a nonzero vector x such that $Ax = 0$. Hence, $x^{\mathrm{T}}Ax = 0$, 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 28.10
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, we assume that it is not and derive a contradiction. If A_k is not positive-definite, then there exists a size-k vector $x_k \neq 0$ such that $x_k^{\mathrm{T}} A_k x_k \leq 0$. Letting A be $n \times n$, we define the size-n vector $x = (\, x_k^{\mathrm{T}} \quad 0 \,)^{\mathrm{T}}$, where there are $n - k$ 0's following x_k. Then we have

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

which contradicts A being 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

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} . \tag{28.28}$$

We generalize definition (28.23) to define the ***Schur complement*** of A with respect to A_k as

$$S = C - BA_k^{-1}B^{\mathrm{T}} . \tag{28.29}$$

(By Lemma 28.10, A_k is symmetric and positive-definite; therefore, A_k^{-1} exists by Lemma 28.9, and S is well defined.) Note that our earlier definition (28.23) of the Schur complement is consistent with definition (28.29), 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 28.8, and its corollary is needed to prove the correctness of LU decomposition for symmetric positive-definite matrices.

Lemma 28.11 (Schur complement lemma)

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

Proof Because A is symmetric, so is the submatrix C. By Exercise 28.1-8, the product $BA_k^{-1}B^{\mathrm{T}}$ is symmetric, and by Exercise 28.1-1, S is symmetric.

It remains to show that S is positive-definite. Consider the partition of A given in equation (28.28). For any nonzero vector x, we have $x^{\mathrm{T}}Ax > 0$ by the assumption that A is positive-definite. Let us break x into two subvectors y and z compatible with A_k and C, respectively. Because A_k^{-1} exists, we have

$$\begin{aligned} x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\ &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}}z \\ By + Cz \end{pmatrix} \\ &= y^{\mathrm{T}}A_k y + y^{\mathrm{T}}B^{\mathrm{T}}z + z^{\mathrm{T}}By + z^{\mathrm{T}}Cz \\ &= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}A_k(y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z , \end{aligned} \tag{28.30}$$

by matrix magic. (Verify by multiplying through.) This last equation amounts to "completing the square" of the quadratic form. (See Exercise 28.5-2.)

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 (28.30) 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 28.12
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 28.11 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 the approximation errors

$$\eta_i = F(x_i) - y_i , \tag{28.31}$$

are small for $i = 1, 2, \ldots, m$. 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 can calculate each y_i *exactly* in equation (28.31). 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 significant 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 whose solution we wish to approximate. We now show how this can be done.

Let

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

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,

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

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:

$$\frac{d\,\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right)a_{ik} = 0 \; . \tag{28.32}$$

The n equations (28.32) for $k = 1, 2, \ldots, n$ are equivalent to the single matrix equation

$$(Ac - y)^{\mathrm{T}} A = 0$$

or, equivalently (using Exercise 28.1-2), to

$$A^{\mathrm{T}}(Ac - y) = 0 \; ,$$

which implies

$$A^{\mathrm{T}}Ac = A^{\mathrm{T}}y \; . \tag{28.33}$$

In statistics, this is called the ***normal equation***. The matrix $A^{\mathrm{T}}A$ is symmetric by Exercise 28.1-2, and if A has full column rank, then by Theorem 28.6, $A^{\mathrm{T}}A$ is positive-definite as well. Hence, $(A^{\mathrm{T}}A)^{-1}$ exists, and the solution to equation (28.33) is

$$\begin{aligned} c &= ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})\,y \\ &= A^{+}y \; , \end{aligned} \tag{28.34}$$

where the matrix $A^{+} = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})$ is called the ***pseudoinverse*** of the matrix A. The pseudoinverse is a natural generalization of the notion of a matrix inverse to the case in which A is nonsquare. (Compare equation (28.34) 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 five data points

$$\begin{aligned} (x_1, y_1) &= (-1, 2) \; , \\ (x_2, y_2) &= (1, 1) \; , \\ (x_3, y_3) &= (2, 1) \; , \\ (x_4, y_4) &= (3, 0) \; , \\ (x_5, y_5) &= (5, 3) \; , \end{aligned}$$

shown as black dots in Figure 28.3. 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

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

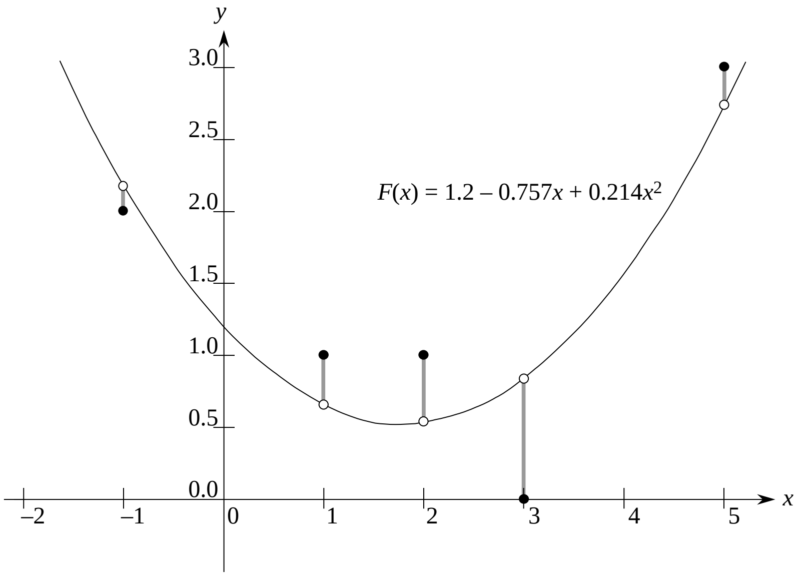


Figure 28.3 The least-squares fit of a quadratic polynomial to the set of five data points $\{(-1,2),(1,1),(2,1),(3,0),(5,3)\}$. The black dots are the data points, and the white dots are their estimated values predicted by the polynomial $F(x) = 1.2 - 0.757x + 0.214x^2$, the quadratic polynomial that 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 (28.33) 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 28.1-2 and Theorem 28.6.)

Exercises

28.5-1

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

28.5-2

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 28.11.

28.5-3

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

28.5-4

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

28.5-5

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

28.5-6

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

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

28.5-7

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

$$\begin{aligned} AA^+A &= A , \\ A^+AA^+ &= A^+ , \\ (AA^+)^{\mathrm{T}} &= AA^+ , \\ (A^+A)^{\mathrm{T}} &= A^+A . \end{aligned}$$

Problems

28-1 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 $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{\mathrm{T}}$ 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.

28-2 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

$$\begin{aligned} f(x_i) &= f_i(0) = y_i , \\ f(x_{i+1}) &= f_i(1) = y_{i+1} \end{aligned}$$

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 $i = 0, 1, \ldots, n-2$.

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 $i = 0, 1, \ldots, n-2$. At the first and last knots, we assume that $f''(x_0) = f_0''(0) = 0$ and $f''(x_n) = f_{n-1}''(1) = 0$; these assumptions make $f(x)$ a ***natural*** cubic spline.

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

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) . \tag{28.35}$$

c. Show that

$$\begin{aligned} 2D_0 + D_1 &= 3(y_1 - y_0) , && (28.36) \\ D_{n-1} + 2D_n &= 3(y_n - y_{n-1}) . && (28.37) \end{aligned}$$

d. Rewrite equations (28.35)–(28.37) 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 28-1).

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 [113], Golub and Van Loan [125], Press, Flannery, Teukolsky, and Vetterling [248, 249], and Strang [285, 286].

Golub and Van Loan [125] discuss numerical stability. They show why $\det(A)$ is not necessarily a good indicator of the stability of a matrix A, proposing instead to use $\|A\|_\infty \|A^{-1}\|_\infty$, where $\|A\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$. They also address the question of how to compute this value without actually computing A^{-1}.

The publication of Strassen's algorithm in 1969 [287] 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 [70], has a running time of $O(n^{2.376})$. The graphical presentation of Strassen's algorithm is due to Paterson [238].

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 [287], Strassen also showed that an $n \times n$ matrix can be inverted in $O(n^{\lg 7})$ time. Winograd [317] originally proved that matrix multiplication is no harder than matrix inversion, and the converse is due to Aho, Hopcroft, and Ullman [5].

Another important matrix decomposition is the ***singular value decomposition***, or ***SVD***. In the SVD, an $m \times n$ matrix A is factored into $A = Q_1 \Sigma Q_2^{\mathrm{T}}$, where Σ is an $m \times n$ matrix with nonzero values only on the diagonal, Q_1 is $m \times m$ with mutually orthonormal columns, and Q_2 is $n \times n$, also with mutually orthonormal columns. Two vectors are ***orthonormal*** if their inner product is 0 and each vector has a norm of 1. The books by Strang [285, 286] and Golub and Van Loan [125] contain good treatments of the SVD.

Strang [286] has an excellent presentation of symmetric positive-definite matrices and of linear algebra in general.