Chapter 9

Algorithms

You should be familiar with

- The inner product
- The Euclidean vector norm
- The Frobenius matrix norm
- Matrix multiplication
- Truncation error
- Upper- and lower-triangular matrices
- Tridiagonal matrices

We have introduced some methods for solving problems in linear algebra; for instance, Gaussian elimination and the computation of eigenvalues using the characteristic equation. Each computation consisted of a series of steps leading to a solution of the problem. Such a series of steps is termed an *algorithm*. We will present algorithms of varying complexity throughout the remainder of this book.

Definition 9.1. Starting with input, if any, an *algorithm* is a set of instructions that describe a computation. When executed, the instructions eventually halt and may produce output.

To this point, our presentation of algorithms was done informally and supported by examples. Now we are beginning a rigorous presentation of algorithms in numerical linear algebra, and we need a more precise mechanism for describing how they work. A formal presentation will aid in understanding an algorithm and in implementing it in a programming language. In this book, you will use the MATLAB programming language or something similar, such as Octave, and perhaps you have also used C/C++, Java, or any of many other programming languages. Presenting an algorithm in MATLAB requires that we adhere to the strict syntax of the MATLAB programming language. We should be able to describe the instructions in a simple, less formal way, so they can be converted to statements in the MATLAB or any other programming language. We use *pseudocode* for this purpose. Pseudocode is a language for describing algorithms. It allows the algorithm designer to focus on the logic of the algorithm without being distracted by details of programming language syntax, such as variable declarations, the correct placement of semicolons and braces, and so forth. We provide pseudocode for all major algorithms and, in each case, there is a MATLAB implementation in the book software.

9.1 PSEUDOCODE EXAMPLES

Our pseudocode will use statements very similar to those of MATLAB such as

```
for i = 1:n do
    <statements>
end for
```

and

```
if abserr < tol
    <statements>
end if
```

Other pseudocode constructs include assignment statements, the while loop, a function, and so forth.

Numerical Linear Algebra with Applications. http://dx.doi.org/10.1016/B978-0-12-394435-1.00009-0

9.1.1 Inner Product of Two Vectors

As our first example, we present an algorithm for the computation of the inner product of $n \times 1$ vectors. A for loop forms the sum of the product of corresponding entries

Algorithm 9.1 Inner Product of Two Vectors

```
function INNERPROD(u,v)
   % Input: column vectors u and v
   % Output: ⟨u, v⟩
   inprod = 0.0
   for i = 1:n do
      inprod = inprod + u(i) v(i)
   end for
   return inprod
end function
```

9.1.2 Computing the Frobenius Norm

The algorithm for the computation of the inner product involves a single loop. The Frobenius norm requires that we cycle through all matrix entries, add their squares, and then take the square root. This involves an outer loop to traverse the rows and an inner loop that forms the sum of the squares of the entries of a row.

Algorithm 9.2 Frobenius Norm

```
function FROBENIUS(A)
   % Input: m × n matrix A.
   % Output: the Frobenius norm √(Σ_{i=1}^{m} Σ_{k=1}^{n} a_{ik}^2).
   fro = 0.0

   for i = 1:m do
      for j = 1:n do
         fro = fro + a_{ij}^2
      end for
   end for
   return fro
end function
```

9.1.3 Matrix Multiplication

Matrix multiplication presents a more significant challenge. If A is an $m \times p$ matrix and B is a $p \times n$ matrix, the product is an $m \times n$ matrix whose elements are

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Start with $i = 1$ and apply the formula for $j = 1, 2, \ldots n$. This gives the first row of the product. Follow this by letting $i = 2$ and applying the formula for $j = 1, 2, \ldots n$ to obtain the second row of the product. Continue in this fashion until computing the last row of AB. This requires three nested loops. The outer loop traverses the m rows of A. For each row i, another loop must cycle through the n columns of B. For each column, form the sum of the products of corresponding elements from row i of A and column j of B. (Figure 9.1).

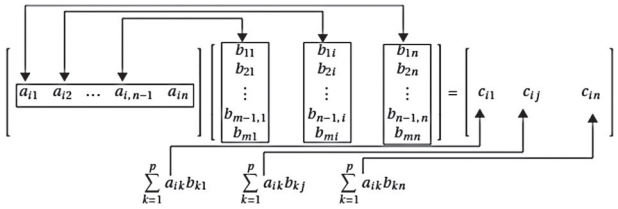

FIGURE 9.1 Matrix multiplication.

Algorithm 9.3 Product of Two Matrices

```
function MATMUL(A,B)
   % Input: m × p matrix A and p × m matrix B
   % Output A × B
   % for each row of A
   for i = 1:m do
      % for each column of B
      for j = 1:n do
         c(i, j) = 0
         % form the sum of the product of corresponding elements from row
         % i of A and column j of B
         for k = 1:p do
            c(i, j) = c(i, j) + a_ik b_kj
         end for
      end for
   end for

   return C
end function
```

9.1.4 Block Matrices

A block matrix is formed from sets of submatrices, and we briefly introduce the concept. In general, these matrices are useful for proving theorems and speeding up algorithms. We will use the idea only a few times in this book and refer the reader to Refs. [1, 2, 23] for an in-depth discussion.

We will confine the discussion to block matrices of order 2×2. Let

$$A = \begin{matrix} \\ m_1 \\ m_2 \\ m = m_1 + m_2 \end{matrix} \begin{matrix} \begin{matrix} p_1 & p_2 \end{matrix} \\ \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \\ p = p_1 + p_2 \end{matrix}$$

The submatrix block A_{11} has dimension $m_1 \times p_1$. In general A_{ij} has dimension $m_i \times p_j$.

Addition and scalar multiplication work as expected. If α is a scalar,

$$\alpha A = \begin{bmatrix} \alpha A_{11} & \alpha A_{12} \\ \alpha A_{21} & \alpha A_{22} \end{bmatrix}$$

and if $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ has the same dimensions as A,

$$A + C = \begin{bmatrix} A_{11} + C_{11} & A_{12} + C_{12} \\ A_{21} + C_{21} & A_{22} + C_{22} \end{bmatrix}.$$

Now we will discuss block matrix multiplication. Form matrix B as follows:

$$\begin{array}{cc} & \begin{array}{cc} n_1 & n_2 \end{array} \\ B = \begin{array}{c} p_1 \\ p_2 \end{array} & \left[\begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array}\right] \\ p = p_1 + p_2 & n = n_1 + n_2 \end{array}$$

To compute an ordinary matrix product AB, the number of columns of A must equal the number of rows in B. For block matrix multiplication, the number of columns of A is p, and the number of rows of B is p. Let's treat the blocks as individual scalar elements in an ordinary 2×2 matrix. Then,

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Although we will not prove it, this is the product AB.

Example 9.1. An example will clarify block matrix multiplication. Let

$$A = \left[\begin{array}{cc|cc} 1 & 4 & 6 & -1 \\ -1 & 3 & 1 & 4 \\ \hline 5 & 2 & -3 & 1 \end{array}\right] \quad B = \left[\begin{array}{cc|c} 3 & 2 & 3 \\ 1 & 6 & 1 \\ \hline 4 & 1 & 3 \\ 2 & 0 & -1 \end{array}\right].$$

For these matrices $m_1 = 2, m_2 = 1, p_1 = 2, p_2 = 2, n_1 = 2$, and $n_2 = 1$, and

$$A_{11} = \begin{bmatrix} 1 & 4 \\ -1 & 3 \end{bmatrix}, \quad A_{12} = \begin{bmatrix} 6 & -1 \\ 1 & 4 \end{bmatrix}, \quad A_{21} = \begin{bmatrix} 5 & 2 \end{bmatrix}, \quad A_{22} = \begin{bmatrix} -3 & 1 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 3 & 2 \\ 1 & 6 \end{bmatrix}, \quad B_{12} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \quad B_{21} = \begin{bmatrix} 4 & 1 \\ 2 & 0 \end{bmatrix}, \quad B_{22} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}.$$

Now,

$$\begin{aligned} AB &= \begin{bmatrix} \begin{bmatrix} 1 & 4 \\ -1 & 3 \end{bmatrix}\begin{bmatrix} 3 & 2 \\ 1 & 6 \end{bmatrix} + \begin{bmatrix} 6 & -1 \\ 1 & 4 \end{bmatrix}\begin{bmatrix} 4 & 1 \\ 2 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 4 \\ -1 & 3 \end{bmatrix}\begin{bmatrix} 3 \\ 1 \end{bmatrix} + \begin{bmatrix} 6 & -1 \\ 1 & 4 \end{bmatrix}\begin{bmatrix} 3 \\ -1 \end{bmatrix} \\ \begin{bmatrix} 5 & 2 \end{bmatrix}\begin{bmatrix} 3 & 2 \\ 1 & 6 \end{bmatrix} + \begin{bmatrix} -3 & 1 \end{bmatrix}\begin{bmatrix} 4 & 1 \\ 2 & 0 \end{bmatrix} & \begin{bmatrix} 5 & 2 \end{bmatrix}\begin{bmatrix} 3 \\ 1 \end{bmatrix} + \begin{bmatrix} -3 & 1 \end{bmatrix}\begin{bmatrix} 3 \\ -1 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} \begin{bmatrix} 29 & 32 \\ 12 & 17 \end{bmatrix} & \begin{bmatrix} 26 \\ -1 \end{bmatrix} \\ \begin{bmatrix} 7 & 19 \end{bmatrix} & 7 \end{bmatrix} \end{aligned}$$

If we ignore the blocks and consider A and B to have dimensions 3×4 and 4×3, respectively, the product is the 3×3 matrix

$$\begin{bmatrix} 29 & 32 & 26 \\ 12 & 17 & -1 \\ 7 & 19 & 7 \end{bmatrix}.$$

■

9.2 ALGORITHM EFFICIENCY

We all know that some algorithms take longer than others. Clearly, multiplying two $n \times n$ matrices takes longer than multiplying an $n \times 1$ column vector by a constant. If you have two algorithms that solve the same problem, one algorithm might be better than another under your current circumstances. For instance, consider the problem of computing eigenvalues. There are a number of algorithms; for instance, finding the roots of the characteristic polynomial, and using the power method (to be discussed later). What we need is a means of measuring the computational effort an algorithm requires. There are many factors that come into play, the number of floating-point arithmetic operations required, the amount of memory

needed, the overhead of array subscripting, the maintenance of control variables in for loops, and so forth. Floating-point operations are slow compared to many other operations, so counting them exactly or approximately helps us compare algorithms.

Definition 9.2. A *flop* is a floating-point operation $\oplus$, $\ominus$, $\otimes$, and $\oslash$. The number of flops required to execute an algorithm is termed the *flop count* of the algorithm.

It is convenient to have a notation that gives us an idea of how much work the algorithm must perform, and we will use "Oh", or "Big-O" notation. For a floating point computation, we want to measure the number of flops required. Given an expression for the number of flops, we say the algorithm is $O\left(n^k\right)$ if the dominant term in the flop count is Cn^k, where C is a constant. We are saying that for large n, the other terms are negligible in comparison to n^k.

Example 9.2. If an algorithm requires $\frac{4}{3}n^3 + 9n^2 + 8n + 6$ flops, it is an $O\left(n^3\right)$ algorithm. When $n = 100$, the actual value of the expression is

$$V = \frac{4}{3}(100)^3 + 9(100)^2 + 8(100) + 6 = 1.424139 \times 10^6.$$

Discarding the lower order terms,

$$T = \frac{4}{3}(100)^3 = 1.333333 \times 10^6,$$

and $T/V = 0.9362$, so $\frac{4}{3}n^3$ contributes 93% of the expression's value. ■

Suppose we want to form the sum of the components in a 3×1 vector. Use a variable sum, initialize sum to have value zero, and use a for loop as follows:

```
sum = 0
for i = 1:3 do
  sum = sum + x(i)
end for
```

The algorithm begins by adding $x(1)$ to $sum = 0$, so it executes three additions. However, looking at the sum abstractly as

$$x(1) + x(2) + x(3),$$

there are two additions. When counting flops, we will ignore the extra addition.

Example 9.3. Given two $n \times 1$ vectors u and v, the inner product $\langle u, v\rangle = u_1v_1 + u_2v_2 + \cdots + u_nv_n$ requires n multiplications and $n-1$ additions, a total of $n + (n-1) = 2n - 1$ flops. Computing the inner product is an $O(n)$ or a *linear algorithm*. ■

Example 9.4. Computing the Frobenius norm requires one flop for each square (a_{ij}^2), and there are mn squares to compute, for a total of mn flops. The addition of the squares requires $mn - 1$ flops, so the flop count for the algorithm is $2mn - 1$, and computing the Frobenius norm is an $O(mn)$ algorithm. If the matrix A is square, the flop count is $O\left(n^2\right)$. We call this a quadratic algorithm. ■

Example 9.5. You should consult Algorithm 9.3 as you read this flop count analysis. Consider the multiplication of an $m \times p$ matrix A by a $p \times n$ matrix B. The inner loop performs one multiplication and an addition, for a total of $2p$ flops per execution of the loop. This inner loop executes mn times, so the flop count is

$$2mnp. \tag{9.1}$$

This flop count is very useful. It tells you that matrix multiplication is an expensive operation; for instance, if $m = 10$, $n = 8$, $p = 12$, the multiplication requires $10(16)12 = 1920$ flops. Most of the matrices engineers and scientists deal with are $n \times n$, and matrix multiplication costs $2n^3$ flops. We say that square matrix multiplication is a *cubic algorithm*, or is $O\left(n^3\right)$. ■

Throughout this book, we will do a detailed flop count analysis if it is instructional. In other cases, the flop count will be stated without proof.

9.2.1 Smaller Flop Count Is Not Always Better

In Chapter 14, we will introduce the QR decomposition of a matrix, which states that $A = QR$, where R is an upper-triangular matrix and Q has orthonormal columns. The decomposition is obtained using what is termed the Gram-Schmidt process. Chapter 17 presents two additional algorithms for finding the QR decomposition, using Givens rotations or Householder reflections. Although Gram-Schmidt has a lower flop count, both are preferable to Gram-Schmidt for a number of reasons that will be explained later. The flop count using Householder reflections for computing the QR decomposition of an $m \times n$ matrix, $m \geq n$, is

$$4\left(m^2 n - mn^2 + \frac{n^3}{3}\right) + 2n^2\left(m - \frac{n}{3}\right),$$

and the flop count using Givens rotations is

$$\frac{1}{2}(5 + 6n + 6n)\left(2mn - n^2 - n\right).$$

On the average, the Householder reflection method is superior in terms of flop count, but the Givens rotation method lends itself very well to parallelization. If you are using a machine with many cores, for instance, the Givens rotation method will likely be superior.

Another aspect that must be taken into account is memory requirements. An algorithm may have a smaller flop count but require much more memory. Depending on how much memory is on the system, an algorithm with a larger flop count but less memory use may run faster. There are many other things that influence speed. For instance, a better written implementation of algorithm A may run faster than algorithm B even if algorithm A has a larger flop count. For example, algorithm B may not reuse variables already allocated in memory, slowing it down.

Remark 9.1. We will use flop count as the primary means for comparing algorithms; in other words, we will ignore the evaluation of square roots, sine, cosine, etc. Different systems may implement these functions in different ways, some perhaps more efficient than others. However, the flop count will remain the same.

9.2.2 Measuring Truncation Error

When approximating a value using a finite sum of terms from a series, truncation error occurs.

Example 9.6. The McLaurin series for e^x is $e^x = \sum_{n=0}^{\infty} x^n/n!$. If x is small and we approximate e^x by

$$1 + x + \frac{x^2}{2},$$

we are leaving off $\frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$, of which the largest term is $\frac{x^3}{3!}$, and we say the truncation error is $O\left(x^3\right)$. ■

If you have studied numerical integration, you are familiar with fourth-order Runge-Kutta methods for estimating $\int_a^b f(x)\,\mathrm{d}x$. If we divide an interval $a \leq x \leq b$ into n subintervals of length $h = (b-a)/n$ and apply a fourth-order Runge-Kutta method, the error is $O\left(h^4\right)$.

9.3 THE SOLUTION TO UPPER AND LOWER TRIANGULAR SYSTEMS

This section presents algorithms for solving upper- and lower-triangular systems of equations. In addition to providing additional algorithms for study, we will need to use both these algorithms throughout this book.

An *upper-triangular matrix* is an $n \times n$ matrix whose only nonzero entries are below the main diagonal; in other words

$$a_{ij} = 0, \quad j < i, \quad 1 \leq i, j \leq n.$$

If U is an $n \times n$ upper-triangular matrix, we know how to solve the linear system $Ux = b$ using back substitution. In fact, this is the final step in the Gaussian elimination algorithm that we discussed in Chapter 2. Compute the value of $x_n = b_n/u_{nn}$, and then insert this value into equation $(n-1)$ to solve for x_{n-1}. Continue until you have found x_1. Algorithm 9.4 presents back substitution in pseudocode.

Algorithm 9.4 Solving an Upper Triangular System

```
function BACKSOLVE(U,b)
   % Find the solution to Ux = b, where U is an n × n upper-triangular matrix.
   x_n = b_n/u_nn
   for i = n-1:-1:1 do
      sum = 0.0
      for j = i+1:n do
         sum = sum + u_ij x_j
      end for
      x(i) = (b(i) − sum)/u_ii
   end for
   return x
end function
```

NLALIB: The function `backsolve` implements Algorithm 9.4.

A lower-triangular matrix is a matrix all of whose elements above the main diagonal are 0; in other words

$$a_{ij} = 0, \quad j > i, \quad 1 \le i, j \le n.$$

A *lower-triangular system* is one with a lower-triangular coefficient matrix.

$$\begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \dots & 0 \\ \dots & \dots & \dots & \ddots & 0 \\ a_{n1} & a_{n2} & \dots & a_{n,n-1} & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

The solution to a lower-triangular system is just the reverse of the algorithm for solving an upper-triangular system—use *forward substitution*. Solve the first equation for $x_1 = \frac{b_1}{a_{11}}$, and insert this value into the second equation to find x_2, and so forth.

Example 9.7. Solve

$$\begin{bmatrix} 2 & 0 & 0 \\ 3 & 1 & 0 \\ 1 & 4 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 8 \end{bmatrix}$$

$x_1 = 2/2 = 1$
$3\,(1) + x_2 = -1, x_2 = -4$
$1\,(1) + 4\,(-4) + 5x_3 = 8, x_3 = 23/5$

SOLUTION: $x = \begin{bmatrix} 1 & -4 & 23/5 \end{bmatrix}^{\mathrm{T}}$. ■

Algorithm 9.5 Solving a Lower Triangular System

```
function FORSOLVE(L,b)
   % Find the solution to the system Lx = b, where L is an n × n lower-triangular matrix.
   x_1 = b_1/l_11
   for i = 2:n do
      sum = 0.0
      for j = 1:i-1 do
         sum = sum + l_ij x_j
      end for
      x(i) = (b(i) − sum)/l_ii
   end for
   return x
end function
```

NLALIB: The function `forsolve` implements Algorithm 9.5.

Example 9.8. Solve the systems

$$\begin{bmatrix} 1 & -1 & 3 \\ 0 & 2 & 9 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 9 \\ -2 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 2 & 0 \\ 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 9 \\ -2 \end{bmatrix}.$$

```
>> U = [1 -1 3;0 2 9;0 0 1];
>> L = [1 0 0;-1 2 0;3 4 5];
>> b = [1 9 -2]';
>> x = backsolve(U,b)

x =
   20.5000
   13.5000
   -2.0000
>> U\b

ans =
   20.5000
   13.5000
   -2.0000
>> y = forsolve(L,b)

y =
     1
     5
    -5
>> L\b
ans =
     1
     5
    -5
```

■

9.3.1 Efficiency Analysis

Algorithm 9.4 executes 1 division and then begins an outer loop having $n-1$ iterations. The inner loop executes $n-(i+1)+1 = n-i$ times, and each loop iteration performs 1 addition and 1 multiplication, for a total of $2(n-i)$ flops. After the inner loop finishes, 1 subtraction and 1 division execute. The total number of flops required is

$$\begin{aligned} 1 + \sum_{i=1}^{n-1} [2(n-i)+2] &= 1 + 2(n-1) + 2\sum_{i=1}^{n-1} (n-i) \\ &= 1 + 2(n-1) + 2[(n-1) + (n-2) + \cdots + 1] \\ &= 1 + 2(n-1) + 2\left(\frac{n(n-1)}{2}\right) \\ &= n^2 + n - 1 \end{aligned}$$

Thus, back substitution is an $O(n^2)$ (quadratic) algorithm. It is left as an exercise to show that Algorithm 9.5 has exactly the same flop count.

9.4 THE THOMAS ALGORITHM

A tridiagonal matrix is square, and the only nonzero elements are those on the main diagonal, the first subdiagonal, and the first superdiagonal, as shown:

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 & \dots & 0 \\ a_1 & b_2 & c_2 & & & & 0 \\ 0 & a_2 & b_3 & c_3 & & \dots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & & & \ddots & \ddots & c_{n-2} & 0 \\ \vdots & & & & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & 0 & \dots & 0 & a_{n-1} & b_n \end{bmatrix}.$$

A tridiagonal matrix is said to be *sparse*, since $n^2 - [n + 2\,(n-1)] = n^2 - 3n + 2$ entries are zero. Tridiagonal matrices are extremely important in applications; for instance, they occur in finite difference solutions to differential equations and in the computation of cubic splines.

A tridiagonal system $Ax = \text{rhs}$ can be solved by storing and using only the entries on the three diagonals

$$\begin{aligned} a &= \begin{bmatrix} a_1 & a_2 & \dots & a_{n-1} \end{bmatrix}^{\mathrm{T}}, \\ b &= \begin{bmatrix} b_1 & b_2 & \dots & b_{n-1} & b_n \end{bmatrix}^{\mathrm{T}}, \\ c &= \begin{bmatrix} c_1 & c_2 & \dots & c_{n-1} \end{bmatrix}^{\mathrm{T}}. \end{aligned}$$

The algorithm is similar to Gaussian elimination, in which the matrix is converted to upper-triangular form and then solved using back substitution, but the algorithm is much more efficient. For the purpose of explanation, we will display matrices. The first action is to divide the row 1 by b_1 to make the pivot in the first row and first column pivot 1. This gives

$$\left[\begin{array}{ccccccc|c} 1 & c_1' & 0 & 0 & \dots & \dots & 0 & \text{rhs}_1' \\ a_1 & b_2 & c_2 & & & & & \text{rhs}_2 \\ 0 & a_2 & b_3 & c_3 & & & & \text{rhs}_3 \\ \vdots & & a_3 & \ddots & \ddots & & & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots \\ 0 & & & & a_{n-2} & b_{n-1} & c_{n-1} & \text{rhs}_{n-1} \\ 0 & 0 & 0 & \dots & 0 & a_{n-1} & b_n & \text{rhs}_n \end{array}\right],$$

where $c_1' = c_1/b_1$ and $\text{rhs}_1' = \text{rhs}_1/b_1$.

Multiply row 1 by a_1 and subtract from row 2:

$$\left[\begin{array}{ccccccc|c} 1 & c_1' & 0 & 0 & \dots & \dots & 0 & \text{rhs}_1' \\ 0 & b_2' & c_2 & 0 & \dots & \dots & & \text{rhs}_2' \\ 0 & a_2 & b_3 & c_3 & & & & \text{rhs}_3 \\ \vdots & & a_3 & \ddots & \ddots & & & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots \\ 0 & & & & a_{n-2} & b_{n-1} & c_{n-1} & \text{rhs}_{n-1} \\ 0 & 0 & 0 & \dots & 0 & a_{n-1} & b_n & \text{rhs}_n \end{array}\right],$$

where $b_2' = b_2 - a_1 c_1'$ and $\text{rhs}_2' = \text{rhs}_2 - a_1 \text{rhs}_1'$.

The steps we have performed define a process that will end in an upper-triangular matrix. To see this, continue by dividing row 2 by b'_2, multiplying row 2 by a_2 and subtracting from row 3 to eliminate a_2 in row 3:

$$\left[\begin{array}{ccccccc|c} 1 & c'_1 & 0 & 0 & \dots & \dots & 0 & \text{rhs}'_1 \\ 0 & 1 & c'_2 & 0 & \dots & \dots & & \text{rhs}''_2 \\ 0 & 0 & b'_3 & c_3 & & & & \text{rhs}'_3 \\ \vdots & & a_3 & \ddots & \ddots & & & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots \\ 0 & & & & a_{n-2} & b_{n-1} & c_{n-1} & \text{rhs}_{n-1} \\ 0 & 0 & 0 & \dots & 0 & a_{n-1} & b_n & \text{rhs}_n \end{array}\right],$$

where

$$\begin{aligned} c'_2 &= c_2/b'_2 = c_2/\left(b_2 - a_1 c'_1\right), \\ \text{rhs}''_2 &= \text{rhs}'_2/b'_2 = \left(\text{rhs}_2 - a_1 \text{rhs}'_1\right) / \left(b_2 - a_1 c'_1\right), \end{aligned}$$

and

$$\begin{aligned} b'_3 &= b_3 - a_2 c'_2, \\ \text{rhs}'_3 &= \text{rhs}_3 - a_2 \text{rhs}''_2 \end{aligned}$$

Note that b'_3 and rhs'_3 will be used to compute c'_3 and rhs''_3 in the next elimination step. Continue the process row by row until the matrix is in upper-triangular form with ones on its diagonal:

$$\left[\begin{array}{ccccccc|c} 1 & c'_1 & 0 & 0 & \dots & \dots & 0 & \text{rhs}'_1 \\ 0 & 1 & c'_2 & 0 & \dots & \dots & & \text{rhs}''_2 \\ 0 & 0 & 1 & c'_3 & & & & \text{rhs}''_3 \\ \vdots & & 0 & \ddots & \ddots & & & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots \\ 0 & & & & 0 & 1 & c'_{n-1} & \text{rhs}''_{n-1} \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 & \text{rhs}''_n \end{array}\right]$$

The resulting matrix is an *upper bidiagonal matrix*. It has nonzero entries only on the main diagonal and the one above. Back substitution is very fast for such a matrix, since the determination of x_i requires using only the computed value of $x_{i+1}$and rhs_i. Back substitution gives

$$\begin{aligned} x_n &= \text{rhs}''_n, \\ x_{n-1} &= \text{rhs}''_{n-1} - c'_{n-1} x_n, \\ &\vdots \\ x_1 &= \text{rhs}'_1 - c'_1 x_2. \end{aligned}$$

Algorithm 9.5 formalizes the process.

The algorithm does involve division, so the Thomas algorithm can fail as a result of division by zero. A condition called *diagonal dominance* will guarantee that the algorithm will never encounter a zero divisor.

Algorithm 9.6 The Thomas Algorithm

```
1:  function THOMAS(a,b,c,rhs)
2:    % The function solves a tridiagonal system of linear equations Ax = rhs
3:    % using the linear Thomas algorithm. a is the lower diagonal, b the
4:    % diagonal, and c the upper diagonal.
5:
6:    % Begin elimination steps, resulting in a bidiagonal matrix
7:    % with 1s on its diagonal.
8:     c_1 = c_1/b_1
9:     rhs_1 = rhs_1/b_1
10:    for i = 2:n-1 do
11:       c_i = c_i/(b_i − a_{i−1} c_{i−1})
12:       rhs_i = (rhs_i − a_{i−1} rhs_{i−1}) / (b_i − a_{i−1} c_{i−1})
13:    end for
14:    rhs_n = (rhs_n − a_{n−1} rhs_{n−1}) / (b_n − a_{n−1} c_{n−1})
15:    % Now perform back substitution
16:    x_n = rhs_n
17:    for i = n-1:-1:1 do
18:       x_i = rhs_i − c_i x_{i+1}
19:    end for
20:    return x
21:  end function
```

NLALIB: The function `thomas` implements Algorithm 9.6.

Definition 9.3. A square matrix is diagonally dominant if the absolute value of each diagonal element is greater than the sum of the absolute values of the other elements in its row, or

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| .$$

For instance, the tridiagonal matrix

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 1 & 3 & -1 & 0 \\ 0 & 5 & -7 & 1 \\ 0 & 0 & 3 & 8 \end{bmatrix}$$

is diagonally dominant. This condition is easy to check and often occurs in problems.

Theorem 9.1. *If A is a diagonally dominant tridiagonal matrix with diagonals a, b, and c, the Thomas algorithm never encounters a division by zero.*

Remark 9.2. If a matrix is not diagonally dominant, the Thomas algorithm may work. Diagonal dominance is only a sufficient condition.

9.4.1 Efficiency Analysis

To determine the flop count for the Thomas algorithm, note that lines 8 and 9 account for two divisions. The for loop beginning at line 10 executes $n-2$ times, and each execution involves 2 divisions, 3 subtractions, and 3 multiplications, for a total of 8 $(n-2)$ flops. The statement at line 14 involves 2 subtractions, 1 division, and 2 multiplications, a total of 5 flops. The for loop beginning at line 17 executes $n-1$ times, and each execution performs 1 subtraction and 1 multiplication, for a total of 2 $(n-1)$ flops. The total flop count is then

$$2 + 8\,(n-2) + 5 + 2\,(n-1) = 10n - 11.$$

The Thomas algorithm is linear ($O(n)$). As we will see in Chapter 11, the Gaussian elimination algorithm for a general $n \times n$ matrix requires approximately $\frac{2}{3}n^3$ flops. It is not uncommon when using finite difference methods for the solution of partial differential equations that tridiagonal systems of order 500×500 or higher must be solved. Standard Gaussian elimination will not take advantage of the sparsity of the tridiagonal system and will require approximately $\frac{2}{3}(500)^3 = 83333333$ flops. Using the Thomas algorithm requires $10(500) - 11 = 4989$ flops, quite a savings!

Example 9.9. Diagonals $a^{4999\times1}$, $b^{5000\times1}$, $c^{4999\times1}$, and right-hand side $\text{rhs}^{5000\times1}$ are generated randomly, and the example times the execution of function `thomas` when solving the 5000×5000 tridiagonal system formed from these vectors. The function `trid` in this book software distribution builds an $n \times n$ tridiagonal matrix from diagonals a, b, and c. The example computes the time required to solve the system using the MATLAB '\' operator. Again, we see the advantages of designing an algorithm that takes advantage of matrix structure.

```
>> a = randn(4999,1);
>> b =randn(5000,1);
>> c = randn(4999,1);
>> rhs = randn(5000,1);
>> tic;x1 = thomas(a,b,c,rhs);toc;
Elapsed time is 0.032754 seconds.
>> T = trid(a,b,c);
>> tic;x2 = T\rhs;toc;
Elapsed time is 0.386797 seconds.
```

■

9.5 CHAPTER SUMMARY

Stating an Algorithm Using Pseudocode

Starting with input, if any, an algorithm is a set of instructions that describe a computation. When executed, the instructions eventually halt and may produce output. This chapter begins a rigorous presentation of algorithms in numerical linear algebra using pseudocode. Pseudocode is a language for describing algorithms that allows the algorithm designer to focus on the logic of the algorithm without being distracted by details of programming language syntax. We provide pseudocode for all major algorithms and, in each case, there is a MATLAB implementation in the book software.

Algorithms are presented for computing the inner product, the Frobenius norm, and matrix multiplication. We also discuss block matrix formulation and operations with block matrices including multiplication. Using block matrices often simplifies the discussion of an algorithm.

Algorithm Efficiency

We measure the efficiency of an algorithm by explicitly counting or estimating the number of flops (floating point operations) it requires. Suppose an algorithm requires $n^3 + n^2 + 6n + 8$ flops. Using Big-O notation, we say it is an $O(n^3)$ algorithm, meaning that the dominant term is n^3. As n increases, the n^3 term accounts for almost all the value; for instance, if $n = 250$, $n^2+6n+8 = 64,008$, and $n^3 = 1.5625 \times 10^7$. There is a mathematical description of this and similar notation for expression algorithm efficiency (see Ref. [24, pp. 52-61]). As examples, the inner product is an $O(n)$ algorithm, and computing the Frobenius norm is $O(n^2)$. Matrix multiplication is an interesting example. Multiplying an $m \times p$ by a $p \times n$ matrix requires $2mnp$ flops. If the matrix is $n \times n$, then the product requires $2n^3$ flops, and is an $O(n^3)$ algorithm. If two matrices are 500×500, their product requires 2.5×10^8 flops. Fortunately, when matrices in applications become that large, they are usually sparse, meaning there is a low percentage of nonzero entries. There are algorithms for rapid multiplication of sparse matrices, and we will deal with sparse matrices in Chapters 21 and 22.

It is possible that a lower flop count may not be better. This can occur when an algorithm with a higher flop count can be parallelized, but one with a lower flop count cannot. An algorithm with a lower flop count may require excessive amounts of memory and, as a result, perform more slowly.

We will have occasion to approximate a function by terms of a series. For instance, the McLaurin series for $\sin x$ is

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

For small x, if we use $x - x^3/3!$ as an approximation, then the truncation error is $O(x^5)$. In order to approximate the solution to differential equations, we will use finite difference equations. If h is small, then

$$\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

has a truncation error $O\left(h^2\right)$.

Solving Upper- and Lower-Triangular Systems

We have studied back substitution that solves a matrix equation of the form

$$\begin{bmatrix} a_{11} & a_{12} & \dots & \dots & a_{1n} \\ 0 & a_{22} & \dots & \dots & a_{2n} \\ 0 & 0 & \ddots & & \\ \vdots & \vdots & \ddots & \ddots & \\ 0 & 0 & \dots & 0 & a_{nn} \end{bmatrix} x = b.$$

This is an $O\left(n^2\right)$ algorithm. If the matrix is in lower-triangular form, we use forward substitution, and it requires the same number of flops.

The Thomas Algorithm for Solving a Tridiagonal Linear System

Tridiagonal systems of equations occur often. When we approximate the solution to the one-dimensional heat equation in Chapter 12 and develop cubic splines in the same chapter, the solution involves solving a tridiagonal system. The Thomas algorithm is based on a clever use of Gaussian elimination and yields a solution in $O(n)$ flops. Solving a general linear system using Gaussian elimination requires approximately $\frac{2}{3}n^3$ flops, so the savings in using the Thomas algorithm is huge!

9.6 PROBLEMS

Whenever a problem asks for the development of an algorithm, do a flop count.

9.1 What is the flop count for each code segment?

a.
```
sum = 0.0;
for i = 1:n
     sum = sum + n^2;
end
```

b.
```
x = 0.0:.01:2*pi;
n = length(x);
sum = 0.0;
for i = 1:length
     sum = sum + 1/(x(i)^2 + x(i) + 1);
end
```

c.
```
A = rand(5,8);
B = rand(8,6);
C = rand(6,12);
D = A*B*C;
```

d.
```
x = rand(10,1);
y = rand((10,1);
z = x*y';
```

9.2 Give the flop count for each matrix operation.

a. Multiplication of $m \times n$ matrix A by an $n \times 1$ vector x.

b. The product xy^{T} if x is an $m \times 1$ vector and y is a $p \times 1$ vector.

c. If u and v are $n \times 1$ vectors, the computation of $\left(\langle v, u\rangle / \|u\|^2\right) u$.

d. $\|A\|_\infty$ for $m \times n$ matrix A.

e. $\|A\|_1$ for $m \times n$ matrix A.
f. trace (A), where A is an $n \times n$ matrix.

9.3 What is the action of the following algorithm?

```
function PROBLEM(u,v)
   sum1 = 0.0
   sum2 = 0.0
   for i = 1:n do
      sum1 = sum1 + u_i v_i
      sum2 = sum2 + u_i^2
   end for
   k = sum1/sum2
   for i = 1:n do
      u_i = k u_i
   end for
   return u
end function
```

9.4 Determine the action of the following algorithm, and find the number of comparisons if the algorithm returns *true*. There is a name attached to this type of matrix. Determine what it is.

```
function ASYM(A)

   for i = 1:n do
      for j = i+1:n do
         if a_ij ≠ −a_ji then
            return false
         end if
      end for
   end for
   return true
end function
```

9.5 Let u be an $m \times 1$ column vector $\begin{bmatrix} u_1 & u_2 & \dots & u_m \end{bmatrix}^{\mathrm{T}}$ and v be a $1 \times n$ row vector $\begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}$. The *tensor product* of u and v, written $u \otimes v$, is the $m \times n$ matrix

$$\begin{bmatrix} u_1v_1 & u_1v_2 & \dots & u_1v_n \\ u_2v_1 & u_2v_2 & \dots & u_2v_n \\ \dots & \dots & \dots & \dots \\ u_mv_1 & u_mv_2 & \dots & u_mv_n \end{bmatrix}$$

Write an algorithm, tensorprod, for the computation of $u \otimes v$.

9.6 Show that matrix multiplication can be implemented using tensor products as defined in Problem 9.5, and write an algorithm that does it.

9.7 What is the action of the following algorithm? Determine its flop count.

```
function PROBLEM(A,B)
   for i = 1:n do
      for j = i:n do
         sum = 0.0;
         for k = i:j do
            sum = sum + a(i, k) b(k, j)
         end for
         P(i, j) = sum
      end for
   end for
   return P
end function
```

Hint: To help determine its action, trace the algorithm using 3×3 matrices. These formulas are useful when determining the flop count:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

9.8 Assume that the operation $A. * B$, where A and B are $n \times n$ matrices multiplies corresponding entries to form a new matrix. For instance,

$$\begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} . * \begin{bmatrix} 4 & 3 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 6 \\ -3 & 10 \end{bmatrix}.$$

Write an algorithm to form $A. * B$.

9.9 Definition 6.4 within the problems of Chapter 6 defines the cross product of two vectors. Develop pseudocode for a function crossprod that computes the cross product of vectors u and v.

9.10 Write an efficient algorithm, addsym, that forms the sum of two $n \times n$ symmetric matrices.

9.11 Section 9.4 defined a tridiagonal matrix. Develop an algorithm trimul that forms the product of two $n \times n$ tridiagonal matrices.

9.12 An upper bidiagonal matrix is a matrix with a main diagonal and one upper diagonal:

$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ & a_{22} & a_{23} & & \\ & & \ddots & \ddots & \\ & & & a_{n-1,n-1} & a_{n-1,n} \\ 0 & & & & a_{nn} \end{bmatrix}$$

Develop an algorithm, bisolve, to solve a system of equations $Ax = b$ that uses only the nonzero elements.

9.13 Develop an algorithm lowtrimul that forms the product of two lower triangular matrices.

9.14 Show that the flop count for Algorithm 9.5 is $n^2 + n - 1$.

9.15 Compute the product of the two block matrices.

$$A = \begin{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 & 1 \\ 5 & -1 & 3 \\ 2 & 7 & 5 \end{bmatrix} \\ \begin{bmatrix} 3 \\ -1 \end{bmatrix} & \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \end{bmatrix} \end{bmatrix}, \quad B = \begin{bmatrix} \begin{bmatrix} 1 & 5 \end{bmatrix} & \begin{bmatrix} -1 & 5 & 7 \end{bmatrix} \\ \begin{bmatrix} 1 & -1 \\ 0 & 2 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 & 2 \\ 1 & 1 & 1 \\ 4 & 0 & -1 \end{bmatrix} \end{bmatrix}.$$

9.6.1 MATLAB Problems

9.16

a. Implement the function matmul specified by Algorithm 9.3.

b. Run the following code, and explain the results by listing all the factors you can think of that determine the speed of matrix multiplication.

```
>> A = rand(1000,1000);
>> B = rand(1000,1000);
>> tic;C = A*B;toc;
>> tic;C = matmul(A,B);toc;
```

When a problem requires you to write a MATLAB function, always thoroughly test it with a variety of data. Some problems prescribe test data.

9.17 Implement Problem 9.4 in the MATLAB function `asym`, test it on the following matrices, and create one test case of your own.

$$A = \begin{bmatrix} 1 & -2 & 5 & 8 & 9 \\ 2 & -3 & 12 & 4 & 16 \\ -5 & -12 & 5 & -18 & 2 \\ -8 & -4 & 18 & 7 & 1 \\ -9 & -16 & -2 & -1 & 2 \end{bmatrix},$$

$$B = \begin{bmatrix} 1 & 2 & 3 & -9 \\ -2 & 3 & 1 & 6 \\ -3 & -1 & -2 & 4 \\ 9 & 6 & -4 & 5 \end{bmatrix}.$$

9.18 Implement Problem 9.9 with a MATLAB function `crossprod`. After implementation, create vectors u and v, and compute $u \times v$, $v \times u$, $\langle u \times v, u \rangle$, and $\langle v \times u, v \rangle$.

9.19 This problem deals with both upper- and lower-triangular matrix multiplication.

a. Write the algorithm in Problem 9.7 as a MATLAB function, `updtrimul` and test it with random matrices of size 3×3, 8×8, and 25×25. For your test, compute $\|\text{updtrimul}\,(A, B) - A * B\|_2$. NOTE: The following statements generate a random, integer, upper-triangular matrix whose entries are in the range $-100 \leq a_{ij} \leq 100$:

```
>> n = value;
>> A = randi([-100 100],n,n);
>> A = triu(A);
```

b. Implement the function lowtrimul from Problem 9.13 and test it as in part (a).

9.20 Write a MATLAB function, `addsym`, that implements the algorithm of Problem 9.10 and test it with matrices

```
A = rand(8,8);
A = A + A';
B = rand(8,8);
B = B + B';
```

9.21 Write a MATLAB function `bisolve` that implements the algorithm of Problem 9.12. The following statements generate a random 5×5 bidiagonal matrix and a random right-hand side. Use these statements as your first test of `bisolve`. Modify the statements and test your function with a 25×25 bidiagonal matrix.

```
>> d = rand(5,1);
>> ud = rand(4,1);
>> A = diag(d) + diag(ud,1);
>> b = rand(5,1);
```

9.22 Problem 9.5 defines the tensor product of two vectors. Implement a function, `tensorprod`, that computes the tensor product of an $m \times 1$ column vector, u, with a $1 \times n$ row vector, v.

9.23 Write a function, tenmatprd, that implements the multiplication of matrices using the tensor product.

9.24 Write a function, `trimul`, that computes the product of two tridiagonal matrices. Test your function using the following statements:

```
>> n = value;
>> a = rand(n-1,1);
>> b = rand(n,1);
>> c = rand(n-1,1);
>> A = trid(a,b,c)
```

9.25 For n odd, consider the "X-matrix"

$$C = \begin{bmatrix} a_1 & & & & & & b_n \\ & a_2 & & & & \iddots & \\ & & \ddots & & b_{k+1} & & \\ & & & a_k & & & \\ & & \iddots & & \ddots & & \\ & b_2 & & & & a_{n-1} & \\ b_1 & & & & & & a_n \end{bmatrix},$$

where the diagonals contain no zero values.

a. The center element is $a_k = b_k$. Find a formula for k.
b. Develop a function `x = xmatsolve(a,b,rhs)` that solves the system $Cx = b$. The function must only use the nonzero values $\{a_i\}, \{b_i\}$. Output an error message and terminate under the following conditions:

- n is even.
- a and b do not both have n elements.
- The diagonals do not share a common center.

c. Develop a function `X = buildxmat(a,b)` that builds an X-matrix. Let $n = length\,(a)$. Generate an error message if n is even or $length\,(b) \neq n$.
d. Develop a function `testxmatsolve(a,b,rhs)` that times the execution of `x = xmatsolve(a,b,rhs)`, builds the matrix `X = buildxmat(a,b)`, and times the execution of the MATLAB command `X\rhs`.
e. Let `a = randn(5001,1); b = randn(5001,1);` and call `buildxmat`. Comment on the results.

9.26 Develop a MATLAB function

```
C = blockmul(A,B,m1,m2,p1,p2,n1,n2)
```

that computes the product of the block matrices `A` and `B` and returns the result into block matrix `C`. The scalars `m1`, `m2`, `p1`, `p2`, `n1`, and `n2` are as described in Section 9.1.4. Test your function using the block matrices of Example 9.1 and Problem 9.15.