APPENDIX A

Solution of Linear Systems by Matrix Methods

A.1 OVERVIEW OF MATRICES

A *matrix* is a rectangular array of *elements* arranged in rows and columns. If the matrix contains m rows and n columns, the matrix is said to be of size $m \times n$. The element in the ith row and jth column of the matrix A is denoted by a_{ij}. For example,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}$$

is a 4×3 matrix.

Types of Matrices

Matrices may belong to one of the many types discussed here.

1. Square Matrix A matrix in which rows m is equal to columns n.

2. Identity Matrix A square matrix in which all the diagonal elements are one and other elements are zero. That is

$$\begin{aligned} a_{ij} &= 1 \quad \text{for} \quad i = j \\ a_{ij} &= 0 \quad \text{for} \quad i \neq j \end{aligned}$$

Identity matrices are denoted by I. For example, a 3 × 3 identity is written as

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

3. Row Vector A matrix with one row and n columns.

4. Column Vector A matrix with m rows and one column.

5. Transpose Matrix The matrix A^T is called the transpose of A if the element a_{ij} in A is equal to element a_{ji} in A^T for all i and j. For example, if

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

then,

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

It is clear that A^T is obtained by interchanging the rows and the columns of A.

6. Zero Matrix A matrix in which every element is zero.

7. Equal Matrix A matrix is said to be equal to another matrix if and only if they have the same order and the corresponding elements are equal. That is,

$$\text{matrix } A = \text{matrix } B$$

if $a_{ij} = b_{ij}$ for all i and j.

Matrix Algebra

We can perform only three operations, namely, addition, subtraction and multiplication on the matrices. The division, although not defined, is replaced by the concept of inversion discussed later.

Two matrices A and B can be added together (or subtracted from each other) if they are of the same order. Then

$$\text{sum } C = A + B$$

can be obtained by adding the corresponding elements. That is,

$$c_{ij} = a_{ij} + b_{ij} \qquad \text{for all } i \text{ and } j$$

Similarly,

$$\text{difference } E = A - B$$

can be obtained by subtracting the corresponding elements. That is,

$$e_{ij} = a_{ij} - b_{ij} \qquad \text{for all } i \text{ and } j$$

Two matrices A and B can be multiplied in the order AB if and only if the number of columns of A is equal to the number of rows of B. That is, if A is of order $m \times r$, then B should be of order $r \times n$, where m and n are arbitrary values. In such cases, we may obtain

$$P = AB$$

which is of order $m \times n$. Elements of the matrix P is given by

$$p_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj} \qquad \text{for all } i \text{ and } j$$

The following general properties apply to matrix algebra.

1. $A \pm B = B \pm A$
2. $A \pm (B \pm C) = (A \pm B) \pm C$
3. $(A \pm B)^T = A^T \pm B^T$
4. $IA = AI$, where I is identity matrix
5. $(AB)C = A(BC)$
6. $C(A + B) = CA + CB$
7. $(A + B)C = AC + BC$
8. $\alpha(AB) = (\alpha A)B = A(\alpha B)$, where α is a scalar

Traces and Determinants

The *trace* of a matrix is the number obtained by adding its diagonal elements (from upper left corner to the lower right corner). For example, the trace of the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \text{ is } a_{11} + a_{22} + a_{33}$$

The *determinant* of a 2×2 matrix, say A, is written in the form

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

and the determinant is computed as

$$a_{11} a_{22} - a_{12} a_{21}$$

Similarly, for a 3×3 matrix the determinant is given by

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$= a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$
\begin{aligned}
&= a_{11}\,(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{31}a_{23}) \\
&\qquad + a_{13}(a_{21}a_{32} - a_{22}a_{31}) \\
&= a_{11}a_{22}a_{33} + a_{12}\text{[illegible]}\ a_{13}a_{21}a_{32} - a_{11}a_{32}a_{23} \\
&\qquad - a_{21}a_{12}a_{33} - a_{31}a_{22}a_{13}
\end{aligned}
$$

Note that there are 6 product terms added together. For larger matrices, the determinant is much more difficult to define and compute manually. In general, for an $n \times n$ matrix, the determinant will contain a sum of $n!$ signed product terms, each having n elements.

Some of the important properties of determinants that would be helpful in computing their values are:

1. Interchanging two rows (or two columns) of a matrix changes the sign of the determinant but not the value.
2. If two rows (or columns) of a matrix are identical then its determinant is zero.
3. If a matrix contains a row (or column) of all zeros, then its determinant is zero.
4. Value of the determinant of a matrix does not change when a scalar multiple of one row (or column) is added to another row (or column).
5. If every element of a row (or column) is multiplied by a scalar α, the value of the determinant is multiplied by α.
6. If A and B are square matrices of same size, then $|AB| = |A|\,|B|$.
7. For a triangular matrix (in which all the elements below (or above) the diagonal are zero), the determinant is the product of the diagonal elements.

Minors and Cofactors

The *minor* M_{ij} of the element a_{ij} of the determinant $|A|$ is obtained by striking out the ith row and jth column. That is,

$$
M_{11} = \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix}, \qquad M_{22} = \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix}
$$

and so on. That is, the minor of a particular element is the determinant that remains after the row and the column that contain the element have been deleted.

The *cofactor* of an element is its minor with a sign attached. The cofactor d_{ij} of an element a_{ij} is given by

$$
d_{ij} = (-1)^{i+j} M_{ij}
$$

The value of the determinant of a matrix can be obtained by expanding the determinant by cofactors. This is done by choosing any column or row and determining the sum of the product of each element in the chosen row or column and its cofactor.

Adjoint Matrix

If d_{ij} is the cofactor of the element a_{ij} of the square matrix A, then, by definition, the *adjoint matrix* of A is given by

$$\text{adj}(A) = D^T$$

where

$$D = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ d_{21} & d_{22} & \dots & d_{2n} \\ \vdots & \vdots & & \\ d_{n1} & d_{n2} & \dots & d_{nn} \end{bmatrix}$$

Inverse of a Matrix

If B and C are two $n \times n$ square matrices such that

$$BC = CB = I \text{ (Identity matrix)}$$

then, B is called the *inverse* of C and C the inverse of B. The common notation for inverses is B^{-1} and C^{-1}. That is,

$$B^{-1} B = I$$
$$C^{-1} C = I$$

The concept of matrix inversion is useful in solving linear systems of equations.

A.2 SOLUTION OF LINEAR SYSTEMS BY DETERMINANTS

We can solve a system of linear equations by determinants using a method called *Cramer's Rule*. For the sake of simplicity, we consider a 2×2 system such as

$$ax_1 + bx_2 = c$$
$$dx_1 + ex_2 = f$$

We can solve for the variable x_1 by eliminating the variable x_2. Thus,

$$x_1 = \frac{ce - bf}{ae - bd}$$

Similarly,

$$x_2 = \frac{af - cd}{ae - bd}$$

Alternatively, we can express x_1 and x_2 using determinants. That is,

$$x_i = \frac{\begin{vmatrix} c & b \\ f & e \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}} \quad \text{and} \quad x_2 = \frac{\begin{vmatrix} a & c \\ d & f \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}}$$

This is known as *Cramer's rule*.

Example A.1

Solve the following system of equations using Cramer's rule:

$$2x_1 + 3x_2 = 12$$
$$4x_1 - x_2 = 10$$

$$x_1 = \frac{\begin{vmatrix} 12 & 3 \\ 10 & -1 \end{vmatrix}}{\begin{vmatrix} 2 & 3 \\ 4 & -1 \end{vmatrix}} = \frac{-12-30}{-2-12} = \frac{42}{14} = 3$$

$$x_2 = \frac{\begin{vmatrix} 2 & 12 \\ 4 & 10 \end{vmatrix}}{\begin{vmatrix} 2 & 3 \\ 4 & -1 \end{vmatrix}} = \frac{20-48}{-14} = \frac{28}{14} = 2$$

A.3 SOLUTION OF LINEAR SYSTEMS BY MATRIX INVERSION

A linear system of n equations in n unknowns can be represented in matrix form as

$$AX = B$$

where A, X and B are matrices and are given by

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

A is called *coefficient matrix* and B is known as *constant vector*. X is the required solution and, therefore, it is called the *solution vector*. If we multiply the matrix equation

$$AX = B$$

By A^{-1} on both sides, we get

$$A^{-1}AX = A^{-1}B$$

A^{-1} is the *inverse matrix* of A. We know that $A^{-1}A = I$ is the *identity matrix* and is given by

$$I = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Therefore,

$$A^{-1}AX = IX = X$$

is the solution of the system of equations and is obtained from

$$X = A^{-1}B = CB$$

If we know the inverse of the matrix A, we can obtain the solution vector X by post-multiplying it by B.

The inverse A^{-1} of a square matrix A exists, if and only if, A is *nonsingular* (i.e. $\det A \neq 0$). A^{-1} is the matrix obtained from A by replacing each element a_{ij} by its *cofactor* d_{ij} and then transposing the resulting matrix and dividing it by the *determinant* of A.

$$C = A^{-1} = \frac{\text{adj}(A)}{\det A}$$

where adj(A) is the *adjoint* of matrix A and is given by the transpose of the *cofactor* matrix of A.
Then

$$\text{adj}(A) = D^T$$

where

$$D = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ d_{21} & d_{22} & \dots & d_{2n} \\ \vdots & \vdots & & \\ d_{n1} & d_{n2} & \dots & d_{nn} \end{bmatrix}$$

d_{ij} is the cofactor of the element a_{ij} and is given by

$$d_{ij} = (-1)^{i+j} M_{ij}$$

M_{ij} is called the *minor* of a_{ij} and is taken as the determinant of matrix A after deleting ith row and jth column.

[illegible]

Solve the following system using matrix inversion method

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 7 \\ x_1 - x_2 + x_3 &= 0 \\ 4x_1 + 2x_2 - 3x_3 &= 4 \end{aligned}$$

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

$$M_{11} = \begin{vmatrix} -1 & 1 \\ 2 & -3 \end{vmatrix} = 1 \qquad d_{11} = 1$$

$$M_{12} = \begin{vmatrix} 1 & 1 \\ 4 & -3 \end{vmatrix} = -7 \qquad d_{12} = 7$$

$$M_{13} = \begin{vmatrix} 1 & -1 \\ 4 & -2 \end{vmatrix} = 6 \qquad d_{13} = 6$$

$$M_{21} = \begin{vmatrix} 1 & 1 \\ 2 & -3 \end{vmatrix} = -5 \qquad d_{21} = 5$$

$$M_{22} = \begin{vmatrix} 2 & 1 \\ 4 & -3 \end{vmatrix} = -10 \qquad d_{22} = -10$$

$$M_{23} = \begin{vmatrix} 2 & 1 \\ 4 & 2 \end{vmatrix} = 0 \qquad d_{23} = 0$$

$$M_{31} = \begin{vmatrix} 1 & 1 \\ -1 & 1 \end{vmatrix} = 2 \qquad d_{31} = 2$$

$$M_{32} = \begin{vmatrix} 2 & 1 \\ 1 & 1 \end{vmatrix} = 1 \qquad d_{32} = -1$$

$$M_{33} = \begin{vmatrix} 2 & 1 \\ 1 & -1 \end{vmatrix} = -3 \qquad d_{33} = -3$$

$$D = \begin{bmatrix} 1 & 7 & 6 \\ 5 & -10 & 0 \\ 2 & -1 & -3 \end{bmatrix}$$

$$\text{Adj}A = D^T = \begin{bmatrix} 1 & 5 & 2 \\ 7 & -10 & -1 \\ 6 & 0 & -3 \end{bmatrix}$$

$$\det A = a_{11}d_{11} + a_{12}d_{12} + a_{13}d_{13} = 15$$

$$A^{-1} = C = \begin{bmatrix} \frac{1}{15} & \frac{5}{15} & \frac{2}{15} \\ \frac{7}{15} & \frac{-10}{15} & \frac{-1}{15} \\ \frac{6}{15} & 0 & \frac{-3}{15} \end{bmatrix}$$

$$B = \begin{bmatrix} 7 \\ 0 \\ 4 \end{bmatrix}$$

We know that $X = CB$ and therefore

$$x_1 = \frac{7}{15} + 0 + \frac{8}{15} = 1$$

$$x_2 = \frac{49}{15} - 0 - \frac{4}{15} = 3$$

$$x_3 = \frac{42}{15} + 0 - \frac{12}{15} = 2$$

A.4 GAUSS-JORDAN MATRIX INVERSION

The Gauss-Jordan elimination technique (discussed in Chapter 7) can be used to invert a matrix effectively. The square matrix A which is to be inverted is augmented by a unit matrix of the same order as follows.

$$\left[\begin{array}{cccc|cccc} a_{11} & a_{12} & \dots & a_{1n} & 1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2n} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & & & \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} & 0 & 0 & \dots & 1 \end{array}\right]$$

If we carry out Gauss-Jordan elimination using the first row as a pivot row we get

$$\left[\begin{array}{cccc|cccc} 1 & a_{12} & \cdots & a_{1n} & a_{1,n+1} & 0 & \cdots & 0 \\ 0 & a_{21} & \cdots & a_{2n} & a_{2,n+1} & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & & \\ 0 & a_{n2} & \cdots & a_{nn} & a_{n,n+1} & 0 & \cdots & 1 \end{array}\right]$$

When we repeat the process using the second row as the pivot row, the result is

$$\left[\begin{array}{ccccc|ccccc} 1 & 0 & a_{13} & \cdots & a_{1n} & a_{1,n+1} & a_{1,n+2} & 0 & \cdots & 0 \\ 0 & 1 & a_{23} & \cdots & a_{2n} & a_{2,n+1} & a_{2,n+2} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & a_{3n} & a_{3,n+1} & a_{3,n+2} & 1 & \cdots & 0 \\ 0 & 0 & a_{n3} & \cdots & a_{nn} & a_{n,n+1} & a_{n,n+2} & 0 & \cdots & 1 \end{array}\right]$$

This elimination process, if continued for all the n rows, yields the final result as follows:

$$\left[\begin{array}{ccccc|cccc} 1 & 0 & 0 & \cdots & 0 & a_{1,n+1} & a_{1,n+2} & \cdots & a_{1,n+n} \\ 0 & 1 & 0 & \cdots & 0 & a_{2,n+1} & a_{2,n+2} & \cdots & a_{2,n+n} \\ 0 & 0 & 1 & \cdots & 0 & \vdots & & & \\ \vdots & & & & & & & & \\ 0 & 0 & 0 & \cdots & 1 & a_{n,n+1} & a_{n,n+1} & \cdots & a_{n,n+n} \end{array}\right]$$

The matrix a_{ij} for $i = 1$ to n and $j = n + 1$ to $2n$ is the inverse matrix of A.

$$C = A^{-1} = [a_{ij}] \begin{cases} i = 1, \ldots, n \\ j = n + 1, \ldots, 2n \end{cases}$$

where the element

$$c_{ij} = a_{i,n+j} \text{ for } i = 1, \ldots, n \qquad \text{and} \qquad j = 1, \ldots, n.$$

Example A.3

Find the inverse of the coefficient matrix of the system given in Example A2 using Gauss-Jordan elimination technique.

$$\text{Augmented } A = \left[\begin{array}{ccc|ccc} 2 & 1 & 1 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 4 & 2 & -3 & 0 & 0 & 1 \end{array}\right]$$

Pivot row-1

$$\left[\begin{array}{ccc|ccc} 1 & 1/2 & 1/2 & 1/2 & 0 & 0 \\ 0 & -3/2 & 1/2 & -1/2 & 1 & 0 \\ 0 & 0 & -5 & -2 & 0 & 1 \end{array}\right]$$

Pivot row-2

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 2/3 & 1/3 & 1/3 & 0 \\ 0 & 1 & -1/3 & 1/3 & -2/3 & 0 \\ 0 & 0 & -5 & -2 & 0 & 1 \end{array}\right]$$

Pivot row-3

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1/15 & 1/3 & 2/15 \\ 0 & 1 & 0 & 7/15 & -2/3 & -1/15 \\ 0 & 0 & 1 & 2/5 & 0 & -1/5 \end{array}\right]$$

The last three columns represent the inverse of the matrix

$$\begin{array}{ccc} 2 & 1 & 1 \\ 1 & -1 & 1 \\ 4 & 2 & -3 \end{array}$$

Compare this with the result obtained in the previous section.

Once the inverse of A is known, the solution vector X can be obtained by simple matrix multiplication. That is

$$X = A^{-1}B = CB$$

$$x_1 = c_{11}b_1 + c_{12}b_2 + \dots + c_{1n}b_n$$

$$x_2 = c_{21}b_1 + c_{22}b_2 + \dots + c_{2n}b_n$$

and so on.

In general,

$$x_i = \sum_{j=1}^{n} c_{ij}b_j \qquad i = 1, \dots, n$$

Note that $c_{ij} = a_{i,\,n+j}$ of augmented A in the final step.

Key Terms

Adjoint matrix
Coefficient matrix
Cofactor
Column vector
Cramer's rule
Determinant
Equal matrix
Identity matrix
Inverse matrix
Minor
Nonsingular matrix
Row vector
Square matrix
Trace of a matrix
Transpose matrix
Zero matrix

APPENDIX B

Solution of Polynomials by Graeffe's Root Squaring Method

Graeffe's root squaring method is a direct method of finding roots of a polynomial with real coefficients. This method deserves attention both for its historical interest and a novel idea involved.

Graeffe's method transforms a polynomial $p_n(x)$ into another polynomial of the same degree but whose roots are the squares of the roots of the original polynomial. Because of the squaring property, the roots of the new polynomial will be spread apart more widely than in the original one, when the roots are greater than 1 in absolute value. Repeating this process until the roots are really far apa , we can compute the roots directly from the coefficients.

We consider here a simple example to illustrate the root squaring technique.

Let

$$p_0(x) = (x - 1)(x - 2)(x - 3) \tag{B.1}$$

Then, we consider another function $p_1(y)$ such that

$$\begin{aligned} p_1(y) &= -p_0(x)\, p_0(-x) \\ &= -(x - 1)(x - 2)(x - 3)(-x - 1)(-x - 2)(-x - 3) \\ &= (x - 1)(x - 2)(x - 3)(x + 1)(x + 2)(x + 3) \\ &= (x^2 - 1)(x^2 - 4)(x^2 - 9) \\ &= (y - 1)(y - 4)(y - 9) \end{aligned} \tag{B.2}$$

where

$$y = x^2$$

We know that the roots of $p_0(x) = 0$ are $x_1 = 1$, $x_2 = 2$, and $x_3 = 3$. And from Eq. (B.2) the roots of $p_1(y) = 0$ are $y_1 = 1$, $y_2 = 4$, and $y_3 = 9$. It shows that roots of $p_1(y) = 0$ are the squares of the roots of $p_0(y) = 0$. This implies that if we compute the roots of $p_1(y)$, then we can obtain the roots of $p_0(x)$ from

$$x_1 = \sqrt{y_1}$$

$$x_2 = \sqrt{y_2}$$

$$x_3 = \sqrt{y_3}$$

Now, let us repeat the procedure for finding the roots of $p_1(y)$. Consider a third polynomial

$$\begin{aligned} p_2(z) &= -p_1(y)\, p_1(-y) \\ &= (z-1)(z-16)(z-81) \qquad \text{(B.3)} \end{aligned}$$

The roots of Eq. (B.3) are

$$\begin{tikzcd}
z_1 = 1\ (= y_1^2) & y_1\ (= x_1^2) \arrow[l] & x_1 \arrow[l] \\
z_2 = 16\ (= y_2^2) & y_2\ (= x_2^2) \arrow[l] & x_2 \arrow[l] \\
z_3 = 81\ (= y_3^2) & y_3\ (= x_3^2) \arrow[l, "\text{Iteration 2}"'] & x_3 \arrow[l, "\text{Iteration 1}"']
\end{tikzcd}$$

That is, after the second iteration, we can estimate the roots of original equation $p_0(x)$ from the relation

$$z_i = (x_i^2)^2 = x_i^4, \qquad i = 1, 2, 3$$

Suppose we have done the squaring process k times and the roots of the final equation are r_i, then

$$\boxed{r_i = x_i^{(2k)} = (x_i)^{2k}} \qquad \text{(B.4)}$$

Remember that we never have $p_0(x)$ in factored form as given in Eq. (B.1), but the result is the same.

Now, let us consider a third degree polynomial in standard form as

$$p_0(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \qquad \text{(B.5)}$$

Then,

$$\begin{aligned} p_1(y) &= -p_0(-x)\, p(x) \\ &= (a_3x^3 + a_2x^2 + a_1x + a_0)\,(a_3x^3 - a_2x^2 + a_1x - a_0) \\ &= a_3^2x^6 - (a_2^2 - 2a_1a_3)x^4 + (a_1^2 - 2a_0a_2)x^2 - a_0{}^2 \\ &= b_3y^3 + b_2y^2 + b_1y_1 + b_0 \qquad \text{(B.6)} \end{aligned}$$

where

$$y = x^2$$

and

$$\boxed{\begin{aligned} b_3 &= +\,a_3^2 \\ b_2 &= -(a_2^2 - 2a_1a_3) \\ b_1 &= +\,(a_1^2 - 2a_0a_2) \\ b_0 &= -a_0{}^2 \end{aligned}} \tag{B.7}$$

We can thus show that for a general polynomial of degree n,

$$p_0(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 \tag{B.8}$$

after first squaring process,

$$\boxed{\begin{aligned} b_n &= +\,a_n^2 \\ b_{n-1} &= -(a_{n-1}^2 - 2a_n a_{n-2}) \\ b_{n-2} &= +\,(a_{n-2}^2 - 2a_{n-1}\,a_{n-3} + 2a_n a_{n-4}) \\ &\vdots \\ b_0 &= (-1)^n\,a_0^2 \end{aligned}} \tag{B.9}$$

This process can be repeated replacing 'a' values by 'b' values in Eq. (B.9) each time. Let us suppose that our final equation after k iterations (i.e. squaring k times) is

$$B_n y^n + B_{n-1} y^{n-1} + \ldots + B_0 = 0 \tag{B.10}$$

Assuming that the roots of Eq. (B.10) are now more widely separated, we have

$$|y_1| >> |y_2| >> |y_3| \ldots >> y_n$$

Then,

$$y_1 \approx -\frac{B_{n-1}}{B_n}$$

$$y_2 \approx -\frac{B_{n-2}}{B_{n-1}}$$

$$\vdots$$

$$y_n \approx -\frac{B_0}{B_1}$$

That is,

$$y_i \approx -\frac{B_{n-i}}{B_{n-i+1}},\ i = 1, 2, \ldots, n$$

$$= (x_i)^{2^k} \tag{B.11}$$

Thus,

$$x_i = 2^k\text{th root of } y_i = \sqrt[2^k]{\left|\frac{B_{n-i}}{B_{n-i+1}}\right|} \tag{B.12}$$

The main advantage of Graeffe's root squaring method over other methods is that it does not require any initial guessing about roots. The method is also capable of giving all the roots but the limitation is that the polynomial should have only real coefficients.

Example B.1

Apply root squaring technique to estimate the roots of

$$x^3 - 3x^2 - 6x + 8 = 0$$

Table below shows the coefficients of successive polynomials (using Eq. (B.7)) as well as the roots estimated (using Eq. (B.12)).

k	*Coefficients*				*Roots estimated*		
	a_3	a_2	a_1	a_0	x_1	x_2	x_3
0	1	−3	−6	8			
1	1	−21	84	−64	4.5826	2.0	0.8729
2	1	−273	4368	−4096	4.0648	2.0	0.9841
3	1	−65793	16843008	−16777216	4.0020	2.0	0.9995

The exact values are 1, −2, and 4. Signs must be determined by substituting the estimates in the original polynomial. When we substitute an estimated root, if the value of the polynomial is zero, then the root is positive; otherwise negative.

APPENDIX C

Difference Operators and Central Difference Interpolation Formulae

C.1 INTRODUCTION

In Chapter 9 we have already discussed briefly the application of finite differences for interpolating the function values. Here we consider again the finite differences of functions and discuss in detail various operators used on them. We also discuss here some of the central difference formulae used for interpolation.

C.2 FINITE DIFFERENCES

Suppose we have a function, $f(x)$, whose values are known (or tabulated) at a set of points $x_0, x_1, x_2, ..., x_n$. Let us denote the function values $f(x_0)$, $f(x_1), ..., f(x_n)$ by $f_0, f_1, ..., f_n$. The difference between any two consecutive function values is called the *finite difference*. The difference in function values

$$f(x_{i+1}) - f(x_i) = f_{i+1} - f_i \tag{C.1}$$

is known as the *first forward difference* at $x = x_i$. We may denote this first difference at $x = x_i$ as

$$\boxed{\Delta f_i = f_{i+1} - f_i} \tag{C.2}$$

where Δ is an operator called the *forward difference operator*.

By applying the process repeatedly we generate the second forward difference, third forward difference, and so on. Thus the kth forward difference of $f(x)$ at $x = x_i$ is given by

$$\boxed{\Delta^k f_i = \Delta(\Delta^{k-1} f_i) = \Delta^{k-1} f_{i+1} - \Delta^{k-1} f_i} \tag{C.3}$$

Example C.1

Find expression for $\Delta^2 f_i$ and $\Delta^3 f_i$.

$$\begin{aligned}
\Delta^2 f_i &= \Delta(\Delta f_i)\\
&= \Delta(f_{i+1} - f_i)\\
&= \Delta f_{i+1} - \Delta f_i\\
&= f_{i+2} - f_{i+1} - f_{i+1} + f_i\\
&= f_{i+2} - 2f_{i+1} + f_i\\
\Delta^3 f_i &= \Delta(\Delta^2 f_i)\\
&= \Delta(f_{i+2} - 2f_{i+1} + f_i)\\
&= f_{i+3} - 3f_{i+2} + 3f_{i+1} - f_i
\end{aligned}$$

The difference in function values

$$f(x_i) - f(x_{i-1}) = f_i - f_{i-1} \tag{C.4}$$

is known as the *first backward difference* at $x = x_i$. This is denoted by

$$\boxed{\nabla f_i = f_i - f_{i-1}} \tag{C.5}$$

where ∇ is an operator called the *backward difference operator*.

The kth backward difference of $f(x)$ at $x = x_i$ is defined by

$$\boxed{\nabla^k f_i = \nabla(\nabla^{k-1} f_i) = \nabla^{k-1} f_i - \nabla^{k-1} f_{i-1}} \tag{C.6}$$

Forward and backward differences introduce asymmetry. Sometimes we may need formulae which are symmetrical about the points of interest. Such formulae are based on central differences.

The difference in function values

$$f_{i+1/2} - f_{i-1/2} \tag{C.7}$$

is known as the *first central difference* of $f(x)$ at $x = x_i$. This is denoted by

$$\boxed{\delta f_i = f_{i+1/2} - f_{i-1/2}} \tag{C.8}$$

where δ is called the *central difference operator*.

The kth central difference at $x = x_i$ is defined as

$$\boxed{\delta^k f_i = \delta(\delta^{k-1} f_i) = \delta^{k-1} f_{i+1/2} - \delta^{k-1} f_{i-1/2}} \tag{C.9}$$

Note that if the function values are only known at $x_1, x_2, \ldots, x_n$, then $f(x_{i+1/2})$ is indeterminate and therefore $\delta f(x_i)$ is not computable. However, we can evaluate $\delta^2 f(x_i)$ as follows:

$$\delta^2 f_i = \delta(f_{i+1/2} - f_{i-1/2})$$

$$= f_{i+1} - f_i - f_i + f_{i-1}$$

$$= f_{i+1} - 2f_i + f_{i-1} \tag{C.10}$$

In general, we can compute $\delta^{2k} f_i$ for all positive integers k.

C.3 DIFFERENCE OPERATORS

We have seen three difference operators, namely, forward operator Δ, backward operator ∇, and central operator δ. We consider here some more operators that are often used in the manipulation of interpolation formulae.

Shift Operator

We have an operator known as *shift* or *displacement* or *translation operator* denoted by E. The displacement or shift operator is defined as

$$\boxed{Ef_i = f_{i+1}} \tag{C.11}$$

$$E^k f_i = f_{i+k} \tag{C.12}$$

If the values of x_i are equally spaced such that

$$x_{i+1} - x_i = h$$

then

$$x_{i+1} = x_i + h$$

Eq. (C.12) may be written as

$$E^k f(x_i) = f(x_i + kh) \tag{C.13}$$

Inverse Operator

An operator opposite of E is known as *inverse operator* and is defined as

$$E^{-1} f(x) = g(x) \tag{C.14}$$

Note that

$$Eg(x) = EE^{-1} f(x) = f(x)$$

$$Eg(x) = g(x + h)$$

Then

$$g(x + h) = f(x)$$

Therefore,

$$g(x) = f(x - h)$$

That is,

$$\boxed{E^{-1} f(x) = f(x - h)} \tag{C.15}$$

Similarly,

$$E^{-k} f(x) = f(x - kh) \tag{C.16}$$

Averaging Operator

The *averaging operator* μ is defined as

$$\delta f_i = \frac{1}{2}\,(f_{i+1/2} + f_{i-1/2}) \tag{C.17}$$

That is,

$$\mu f(x_i) = \frac{1}{2}\left[f\left(x_i + \frac{h}{2}\right) + f\left(x_i - \frac{h}{2}\right)\right]$$

C.4 RELATIONS BETWEEN THE OPERATORS

The difference operators are related to one another in a number of ways. We consider here a few of them.

Relation between Δ and E

We know that

$$\begin{aligned}\Delta f(x) &= f(x+h) - (f(x)\\ &= Ef(x) - 1.f(x)\\ &= (E-1)\,f(x)\end{aligned}$$

Therefore,

$$\boxed{\Delta = E - 1 \qquad \text{or} \qquad E = \Delta + 1} \tag{C.18}$$

Relation between ∇ and *E*

$$\begin{aligned}\nabla f(x) &= f(x) - f(x-h)\\ &= 1.f(x) - E^{-1}f(x)\\ &= (1 - E^{-1})\,f(x)\end{aligned}$$

Therefore

$$\boxed{\nabla = 1 - E^{-1}} \tag{C.19}$$

or

$$E = (1 - \nabla)^{-1}$$

(since $(E^{-1})^{-1} = E$)

Relation between Δ, ∇ and E

We have

$$\begin{aligned}E\nabla &= E(1 - E^{-1})\\ &= E - 1\\ &= \Delta\end{aligned}$$

Therefore

$$\boxed{E\nabla = \Delta} \tag{C.20}$$

Similarly, we can show that

$$E^{-1}\Delta = \nabla$$

Relation between E, δ and μ

We have

$$\delta f(x) = f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)$$
$$= E^{1/2}f(x) - E^{-1/2}f(x)$$

Therefore,

$$\boxed{\delta = E^{1/2} - E^{-1/2} = E^{-1/2}\Delta = E^{1/2}\nabla} \qquad \text{(C.21)}$$

Similarly, we can show that

$$\boxed{\mu = \frac{1}{2}(E^{1/2} + E^{-1/2})} \qquad \text{(C.22)}$$

From Eqs (C.21) and (C.22),

$$\mu\,\delta = \frac{1}{2}\,(E - E^{-1})$$

We know

$$\Delta + \nabla = (E - 1)(1 - E^{-1}) = E - E^{-1}$$

Then,

$$\mu\,\delta = \frac{1}{2}\,(\Delta + \nabla)$$

Example C.2

Prove that $\Delta - \nabla = \Delta\nabla$.

$$\Delta - \nabla = (E - 1) - (1 - E^{-1})$$
$$= E - 2 + E^{-1}$$
$$\Delta\nabla = (E - 1)(1 - E^{-1})$$
$$= E - 2 + E^{-1}$$

Therefore

$$\Delta - \nabla = \Delta\nabla$$

Example C.3

Show that $\Delta^2 = E^2 - 2E + 1$

$$\Delta^2 = (E - 1)(E - 1)$$
$$= E^2 - 2E + 1$$

Example C.4

Prove $\delta^2 = \Delta - \nabla$

$$\delta^2 = [E^{1/2} - E^{-1/2}]^2$$
$$= E + E^{-1} - 2$$
$$\Delta - \nabla = (E - 1) - (1 - E^{-1})$$
$$= E + E^{-1} - 2$$

Hence,

$$\delta^2 = \Delta - \nabla$$

From Example C.2, we know

$$\Delta - \nabla = \Delta\nabla$$

and, therefore,

$$\delta^2 = \Delta\nabla$$

From Example C.3, we can prove that

$$\Delta^2 = E^2 - 2E + 1$$
$$= E(E - 2 + E^{-1})$$
$$= E(\Delta - \nabla) = E\Delta\nabla$$

Similarly, the readers may attempt to prove the following relations.

1. $\mu^2 = 1 + \dfrac{\delta^2}{4}$ or $\delta^2 = 4(\mu^2 - 1)$
2. $\dfrac{\Delta}{\nabla} - \dfrac{\nabla}{\Delta} = \Delta + \nabla$
3. $\dfrac{\Delta}{\nabla} - \dfrac{\nabla}{\Delta} = 2\mu\delta$

C.5 CENTRAL DIFFERENCE INTERPOLATION FORMULAE

We have already discussed in Chapter 9 the forward and backward difference formulae. They are useful when the value required is at the beginning or at the end of the table. However, if the point of interest is located at the middle of the table, then we may use the formulae based on central differences.

Let x_0 be the middle point and f_0 the corresponding function value. Consider the values of x on either side x_0 such that

$$f_1 = f(x_0 + h), \qquad f_{-1} = f(x_0 - h)$$
$$f_2 = f(x_0 + 2h), \qquad f_{-2} = f(x_0 - 2h)$$

and so on.

We can now form a difference table with the values $f(x)$ on either side of x_0, as shown in Table C.1.

Table C.1 Central difference table

x	f *difference*	*First difference*	*Second difference*	*Third difference*	*Fourth*
$x_0 - 2h$	f_{-2}				
		$\Delta f_{-2} = \delta f_{-3/2}$			
$x_0 - h$	f_{-1}		$\Delta^2 f_{-2} = \delta^2 f_{-1}$		
		$\Delta f_{-1} = \delta f_{-1/2}$		$\Delta^3 f_{-2} = \delta^3 f_{-1/2}$	
x_0	f_0		$\Delta^2 f_{-1} = \delta^2 f_0$		$\Delta^4 f_{-2} = \delta^4 f_0$
		$\Delta f_0 = \delta f_{1/2}$		$\Delta^3 f_{-1} = \delta^3 f_{1/2}$	
$x_0 + h$	f_1		$\Delta^2 f_0 = \delta^2 f_1$		
		$\Delta f_1 = \delta f_{3/2}$			
$x_0 + 2h$	f_2				

The entries in the table are related using the relation between Δ and δ operators. We know that

$$\delta = \Delta E^{-1/2}$$

and therefore we can show that

$$\delta^2 f_{-1} = (\Delta E^{\;1/2})^2 f_{-1} = \Delta^2 E^{-1} f_{-1} = \Delta^2 f_{-2}$$

Similarly, we can prove for all the entries.

We present here the following central difference interpolation formulae which use differences close to the centre of the table.

1. Gauss forward formula
2. Gauss backward formula
3. Stirling formula
4. Bessel formula
5. Laplace-Everett formula

Gauss Forward Interpolation Formula

We have used the Newton-Gregory forward interpolation formula in Chapter 9. This is given by

$$f(x) = f_0 + p\Delta f_0 + \frac{p(p-1)}{2!}\Delta^2 f_0 + \frac{p(p-1)(p-3)}{3!}\Delta^3 f_0 + \dots \quad \text{(C.23)}$$

where

$$x = x_0 + ph \qquad \text{or} \qquad p = \frac{x - x_0}{h}$$

We can show that

$$\Delta^2 f_0 = \Delta^2 E f_{-1} = \Delta^2 (1 + \Delta) f_{-1} = \Delta^2 f_{-1} + \Delta^3 f_{-1}$$

$$\Delta^3 f_0 = \Delta^3 E f_{-1} = \Delta^3 (1+\Delta) f_{-1} = \Delta^3 f_{-1} + \Delta^4 f_{-1}$$
$$\Delta^4 f_0 = \Delta^4 f_{-1} + \Delta^5 f_{-1} \qquad \text{and so on.}$$

Substituting for $\Delta^2 f_0$, $\Delta^3 f_0$, ..., the above equivalents in Eq. (C.23) and after simplification, we obtain

$$\boxed{f(x) = f_0 + \binom{p}{1}\Delta f_0 + \binom{p}{2}\Delta^2 f_{-1} + \binom{p+1}{3}\Delta^3 f_{-1} + \binom{p+1}{4}\Delta^4 f_{-1} + \ldots} \tag{C.24}$$

where

$$\binom{m}{n} = \frac{m(m-1)(m-2)\ldots(m-n+1)}{n!}$$

Equation (C.24) is known as *Gauss forward formula*. Note that this formula involves 'odd' differences below the centre line and 'even' differences on the central line.

Gauss Backward Interpolation Formula

Gauss backward formula is obtained using the odd differences above the centre line and even differences on the central line. This is obtained by using the following substitutions in Eq. (C.23).

$$\Delta f_0 = \Delta f_{-1} + \Delta^2 f_{-1}$$
$$\Delta^2 f_0 = \Delta^2 f_{-1} + \Delta^3 f_{-1}$$
$$\Delta^3 f_0 = \Delta^3 f_{-1} + \Delta^4 f_{-1}$$
$$\Delta^3 f_{-1} = \Delta^3 f_{-2} + \Delta^4 f_{-2}$$
$$\Delta^4 f_{-1} = \Delta^4 f_{-2} + \Delta^5 f_{-2}$$

Thus, we obtain the Gauss backward interpolation formula as

$$\boxed{f(x) = f_0 + \binom{p}{1}\Delta f_{-1} + \binom{p+1}{2}\Delta^2 f_{-1} + \binom{p+1}{3}\Delta^3 f_{-2} + \binom{p+2}{4}\Delta^4 f_{-2} + \ldots} \tag{C.25}$$

Equation (C.25) is popularly known as *Gauss backward formula*.

Stirling Formula

Stirling formula is obtained by taking the average of the two Gauss formulae. Therefore, adding Eqs (C.24) and (C.25) and dividing by 2, we obtain

$$\boxed{\begin{aligned} f(x) = f_0 &+ p\left(\frac{\Delta f_0 + \Delta f_{-1}}{2}\right) + \frac{p^2}{2!}\Delta^2 f_{-1} \\ &+ \frac{p(p^2-1)}{3!}\left(\frac{\Delta^3 f_{-1} + \Delta^3 f_{-2}}{2}\right) + \frac{p^2(p^2-1)}{4!}\Delta^4 f_{-2} + \ldots \end{aligned}} \tag{C.26}$$

Equation (C.26) is known as *Stirling formula*. Note that Eq. (C.26) involves the means of the 'odd' differences just above and just below the central line and 'even' differences on the line. To use this formula, p must satisfy the condition

$$-\frac{1}{2} < p < \frac{1}{2}$$

Bessel Formula

Bessel formula is a modified form of Gauss forward formula, which is given below.

$$f(x) = f_0 + p\Delta f_0 + \frac{p(p-1)}{2!}\Delta^2 f_{-1} + \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \dots \quad \text{(C.27)}$$

We know

$$\Delta f_0 = f_1 - f_0 \quad \text{and, therefore,} \quad f_0 = f_1 - \Delta f_0 \quad \text{(C.28)}$$

$$\Delta^3 f_{-1} = \Delta^2 f_0 - \Delta^2 f_{-1}, \quad \text{then,} \quad \Delta^2 f_{-1} = \Delta^2 f_0 - \Delta^3 f_{-1} \quad \text{(C.29)}$$

Similarly,

$$\Delta^4 f_{-2} = \Delta^4 f_{-1} - \Delta^5 f_{-2} \quad \text{(C.30)}$$

Now, we rewrite Eq. (C.27) as

$$f(x) = \left(\frac{f_0}{2} + \frac{f_0}{2}\right) + p\Delta f_0 + \frac{1}{2}\frac{p(p-1)}{2!}\Delta^2 f_{-1} + \frac{1}{2}\frac{p(p-1)}{2!}\Delta^2 f_{-1}$$

$$+ \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \dots \quad \text{(C.31)}$$

Now, substituting Eqs (C.28), (C.29), and (C.30) in (C.31), we get

$$f(x) = \frac{f_0}{2} + \frac{f_1 - \Delta f_0}{2} + p\Delta f_0 + \frac{1}{2}\frac{p(p-1)}{2!}\Delta^2 f_{-1}$$

$$+ \frac{1}{2}\frac{p(p-1)}{2!}(\Delta^2 f_0 - \Delta^3 f_{-1}) + \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \dots$$

$$= \frac{f_0 + f_1}{2} + \left(p - \frac{1}{2}\right)\Delta f_0 + \frac{p(p-1)}{2!}\frac{(\Delta^2 f_0 + \Delta^2 f_{-1})}{2}$$

$$+ \frac{\left(p - \frac{1}{2}\right)p(p-1)}{3!}(\Delta^3 f_{-1}) + \dots \quad \text{(C.32)}$$

Equation (C.32) is called *Bessel formula*. Note that the Bessel formula involves odd differences below the centre line and averages of the even differences on and below the central line.

Observe that when $p = \frac{1}{2}$ all odd order differences vanish and we get

$$f(x) = \frac{1}{2}(f_0 + f_1) - \frac{1}{8}\left(\frac{\Delta^2 f_{-1} + \Delta^2 f_0}{2}\right) + \frac{3}{128}\left(\frac{\Delta^4 f_{-2} + \Delta^4 f_{-1}}{2}\right)$$

$$-\frac{5}{1024}\left(\frac{\Delta^6 f_{-3} + \Delta^6 f_{-2}}{2}\right) + \dots \tag{C.33}$$

Equation (C.33) is known as *formula for interpolating to haves.*

Laplace-Everett Formula

Again consider the Gauss forward formula

$$f(x) = f_0 + p\Delta f_0 + \frac{p(p-1)}{2!}\Delta^2 f_{-1} + \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \dots \tag{C.34}$$

If we eliminate the add differences from Eq. (C.34) the result will give us the Laplace-Everett formula. We know

$$\Delta f_0 = f_1 - f_0$$

$$\Delta^3 f_{-1} = \Delta^2 f_0 - \Delta^2 f_{-1}$$

$$\Delta^5 f_{-2} = \Delta^4 f_{-1} - \Delta^4 f_{-2}$$

and so on.

Substituting these in Eq. (C.34) and rearranging the terms, we get

$$f(x) = (1-p)f_0 + \binom{p}{1}f_1 - \binom{p}{1}\Delta^2 f_{-1} + \binom{p+1}{3}\Delta^2 f_0$$

$$-\binom{p+1}{5}\Delta^4 f_{-2} + \binom{p+2}{5}\Delta^4 f_{-1} + \dots \tag{C.35}$$

Letting

$$p = 1 - q \qquad \text{or} \qquad 1 - p = q$$

and simplifying, Eq. (C.35) becomes

$$f(x) = \left[qf_0 + \binom{q+1}{3}\Delta^2 f_{-1} + \dots\right] + \left[pf_1 + \binom{p+1}{3}\Delta^2 f_0 + \dots\right] \tag{C.36}$$

Equation (C.36) is known as Laplace-Everett formula. This formula involves only even differences on and below the central line. This can be used when $0 < p < 1$.

Key Terms

Averaging operator
Backward difference
Backward difference operator
Bessel formula
Central difference
Central difference operator
Displacement operator
Forward difference
Forward difference operator
Gauss backward formula
Gauss forward formula
Inverse operator
Laplace-Everett formula
Shift operator
Stirling formula
Translation operator

APPENDIX D

C Programs

D.1 INTRODUCTION

C programs are basically made of functions. As we know, a function is a unit of a program that performs a particular task. A C function is similar to a subroutine in FORTRAN. A function begins running when its name is used in the program.

All C programs need a *main* function which is named as **main()**. Therefore, the programs in this appendix begin with the main function. Since the main function does not use any arguments, its name is followed by a set of empty parentheses.

Everything inside a C function is enclosed between two curly brackets, like {...}. The left curly bracket signals the beginning of the function and the right one signals the end of the function.

Some features of C that are distinctly, different from FORTRAN are:

- C is a free-form language and therefore we may follow any format of statements that may please us.
- A comment or remark can be inserted anywhere that a space can appear in a C program provided it is preceded by /* and followed by */
- As a convention (and not as a rule), everything in C is written using lower-case letters. Thus, names of all variables and functions are written in lower-case.
- All C statements must end with a semicolon.
- C does not support statement numbers. However, we can assign a label to a statement as follows:

```
begin : sum = 0.0;
```

We may direct the control to this statement by using the **goto** statement as

```
goto begin;
```

- C language does not have STOP and END statements. While the closing bracket } represents the "END" of a function, STOP may be replaced by invoking the **exit()** function available in C.
- **Unlike FORTRAN, all variables and functions that return values must explicitly be declared for their types.**

```
int i,j,a,b; /* declaring integer variables */
float x,y,a[10]; /*declaring real variables */
```

- C supports what are known as preprocessor directives **#include** and **#define**. The **#include** directive is used to include in the program library functions and other files. We may use the **#define** directive to define constants and functions that are used in the program.
- C does not support any operator for exponentiation. Instead, it uses a function **pow(x,y)** to compute x^y.
- C uses square brackets to represent arrays variables, like, *a*[*i*]. *b*[*i*][*j*], etc.
- Unlike FORTRAN, C array elements are numbered from ZERO, not ONE. That is, the array *a*[3] will be represented in the memory as *a*[0], *a*[1], and *a*[2].
- C uses an operator & known as address operator to read the values from the keyboard. We may use this operator to obtain the address of a variable in the memory. For example, & *x* gives the address of the location of *x*.
- C defines % as the modules operator. This operator divides the first operand by the second and gives the remainder, (not the quotient) as the result.
- C defines another operator known as "star" operator * to define a pointer. The statements

```
int a;
int *p; /* p is declared as pointer */
p=&a;   /* address of a is assigned to p */
```

make **p** point to **a**. Now, the statements *p+5 and a+5 both would give the same result. That is, ***p** means the value stored at the address pointed to by **p**.

- C uses the following relational and logical operators:

==	is equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
!=	is not equal to
||	logical OR
&&	logical AND

- C has two more control statements **continue** and **break in addition** to the conventional **goto** statement. These two are used inside a loop. While the break transfers the control to the first statement outside the loop, the continue statement skips the remaining part of the loop.

D.2 FORTRAN TO C CONVERSION

The FORTRAN programs in the text have been translated almost line by line using C equivalents given in Table D.1

Table D.1 C equivalents

FORTRAN Statements	*C Equivalents*
INTEGER	int
REAL	float, double
PARAMETERS (M = 100)	#define M 100
INTRINSIC	#include <math.h>
READ(*,*) statement	scanf() statement
WRITE (*,*) statement	printf() statement
Function subprograms	Preprocessor macro definition
Statement number	Label name
GOTO <number>	goto <label>
GOTO inside a loop	break or continue
STOP inside a program	exit() function
IF (...) THEN...	if (...) ...;
IF(...) THEN ... ELSE	if (...) ... else
DO ... CONTINUE	for (...) {...}
END	}

The preprocessor directives **#define** and **#include** are always placed before the main() function. Any constants and variables declared before the main function are considered to be global and are available to all the functions defined after that point in the file.

It should be noted that the programs have been translated line by line so as to enable the readers to understand the logic easily. It is quite possible to simplify or improve the efficiency of C programs in this appendix using some of the unique features of C. This requires a complete understanding of the language.

Readers new to C language must read an introductory book on C such as *Programming in ANSI C* by E Balagurusamy before attempting to analyse and apply the C programs given in this appendix.

Table D.2 List of programs

Program No.	*Name*	*Description*
1	POLY	Program POLY evaluates a polynomial of degree n at any point x using Horner's rule.
2	BISECT	This program finds a root of a nonlinear equation using the bisection method.
3	FALSE	This program finds a root of a nonlinear equation by false position method.
4	NEWTON	This program finds a root of a nonlinear equation by Newton-Raphson method.
5	SECANT	This program finds a root of a nonlinear equation by secant method.
6	FIXEDP	This program finds a root of a function using the fixedp point iteration method.
7	MULTIR	The program finds all the real roots of a polynomial.
8	COMPR	This program locates all the roots, both real and complex, using the Bairstow method.
9	MULLER#	This program evaluates root of a polynomial using Muller's method.
10	LEG1	This program solves a system of linear equations using simple Gaussian elimination method.
11	LEG2	This program solves a system of linear equations using Gaussian elimination with partial pivoting.
12	DOLIT	This program solves a system of linear equations using Dolittle LU decomposition.
13	JACIT	This program uses the subprogram JACOBI to solve a system of equations by Jacobi iteration method.
14	GASIT	This program uses the subprogram GASEID to solve a system of equations by Gauss-Seidel iteration method.
15	LAGRAN	This program computes the interpolation value at a specified point, given a set of data points, using the Lagrange interpolation representation.
16	NEWINT	This program constructs the Newton interpolation polynomial for a given set of data points and then computes interpolation value at a specified value.
17	SPLINE	This program computes the interpolation value at a specified value, given a set of table points, using the natural cubic spline interpolation.
18	LINREG	This program fits a line Y = A + BX to a given set of data points by the method of least squares.
19	POLREG	This program fits a polynomial curve to a given set of data points by the method of least squares.
20	NUDIF#	This program computes the derivative of a tabulated function at a specified value using the Newton interpolation approach.

(Contd.)

Table D.2 *(Contd.)*

Program No.	*Name*	*Description*
21	TRAPE1	This program integrates a given function using the trapezoidal rule.
22	SIMS1	This program integrates a given function using the Simpson's 1/3 rule.
23	ROMBRG	This program performs Romberg integration by bisecting the intervals N times.
24	TRAPE2[#]	This program integrates a tabulated function using the trapezoidal rule.
25	SIMS2[#]	This program integrates a tabulated function using the Simpson's 1/3 rule. If the number of segments is odd, the trapezoidal rule is used for the last segment.
26	EULER	This program estimates the solution of the first order differential equation $y' = f(x, y)$ at the given point using Euler's method.
27	HEUN	This program solves the first order differential equation $y' = f(x, y)$ using the Heun's method.
28	POLYGN	This program solves the differential equation of type $y' = f(x, y)$ by polygon method.
29	RUNGE4	This program computes the solution of first order differential equation of type $y' = f(x, y)$ using the 4th order Runge-Kutta method.
30	MILSIM	This program solves the first order differential equation $y' = f(x, y)$ using the Milne-Simpson method.

[#] These are new programs in C and are not available in FORTRAN version in the text. Readers may try to develop FORTRAN equivalents of these programs.

Program 1 POLY

```
/*-----------------------------------------------------*
 *                                                     *
 *  Main program                                       *
 *    Program POLY evaluates a polynomial of degree n  *
 *    at any point X using Horner's rule               *
 *-----------------------------------------------------*
 *                                                     *
 *  Subroutines used                                   *
 *    horner                                           *
 *-----------------------------------------------------*
 *                                                     *
 *  Variables used                                     *
 *    n - Degree of polynomial                         *
 *    a - Array of polynomial coefficients             *
 *    x - Point of evaluation                          *
 *    p - Value of polynomial at x                     *
 *-----------------------------------------------------*
 *                                                     *
 *  Constants used                                     *
 *    NIL                                              *
 *-----------------------------------------------------*/
```

```
main( )
{
    int n,i;     /* Declaration of variables */
    float x,p,a[10];
    float horner(int n, float a[], float x);
    printf("Input degree of polynomial, n\n");
    scanf("%d", &n);
    printf("Input polynomial coefficients a(0) to a(n) \n");
    for(i=0;i<=n;i++)
      scanf("%f", &a[i]);
    printf("Input value of x (point of evaluation) \n");
    scanf("%f", &x);
/* Evaluating polynomial at x using Horner's rule */
    p = horner(n,a,x); /* Calling the function horner */
/* Writing the result */
      printf("\n");
    printf("f(x) = %f at x = %f \n", p, x);
    printf("\n");
}
/* End of main program POLY */
/* ---------------------------------------------------- */
/* Defining the function horner */
float horner(int n, float a[], float x)
/*   horner computes the value of a polynomial of order n
     at any given point x. */
    {
        int i; /* Local variables */
        float p;
        p = a[n];
        for(i=n-1;i>=0;i-)
        {
          p = p*x + a[i];
        }
        return(p);
      }
/* End of Function horner */
/* ---------------------------------------------------- */
```

Program 2 BISECT

```
/*-------------------------------------------------------*
 * Main program                                          *
 *   This program finds a root of a nonlinear            *
 *   equation using the bisection method                 *
 *-------------------------------------------------------*
 * Functions invoked                                     *
 *   Macro F(x)                                          *
 *-------------------------------------------------------*
 * Subroutines used                                      *
 *   bim()                                               *
 *-------------------------------------------------------*
 * Variables used                                        *
 *   a - Left endpoint of interval                       *
 *   b - Right endpoint of interval                      *
 *   s - Status                                          *
 *   root - Final Solution                               *
 *   count - Number of Iterations done                   *
 *-------------------------------------------------------*
 * Constants used                                        *
 *   EPS - Error bound                                   *
 *-------------------------------------------------------*/
#include <stdio.h>
#include        <math.h>
#define EPS     0.000001
#define F(x)    (x)*(x)+(x)-2

main( )
{
    int s, count;
    float a,b,root;
    printf("\n");
    printf("SOLUTION BY BISECTION METHOD \n");
    printf("\n");
    printf("Input starting values \n");
    scanf("%f %f",&a,&b);
/* Calling the subroutine bim() */
    bim(&a, &b, &root, &s, &count);
    if(s==0)
    {
      printf("\n");
      printf("Starting points do not bracket any root \n");
      printf("(Check whether they bracket EVEN roots \n");
      printf("\n");
```

```
     }
     else
     {
       printf("\nRoot = %f \n", root);
       printf("F(root) = %f \n", F(root));
       printf("\n");
       printf("Iterations = %d \n", count);
       printf("\n");
     }
}
/* End of main program */
/* ----------------------------------------------------------- */
/* Defining the subroutine bim() */
bim(float *a, float *b, float *root, int *s, int *count)
/*  This subroutine finds a root of nonlinear equation
    in the interval [a,b] using the bisection method */
{
      float x1,x2,x0,f0,f1,f2;
/*  Function values at initial points */
      x1 = *a;
      x2 = *b;
      f1 = F(x1);
      f2 = F(x2);
/*  Test if initial values bracket a SINGLE root */
    if(f1*f2 > 0)
    {
      *s = 0;
      return;                    /* Program terminated */
    }
    else
'* Bisect the interval and locate the root iteratively */
    {
      *count = 0;
      begin:                     /* Iteration begins */
      x0 = (x1 + x2)/2.0;
      f0 = F(x0);
      if(f0==0)
      {
         *s = 1;
         *root = x0;
         return;
      }
      if(f1*f0 < 0)
      {
          x2 = x0;
      }
      else
      {
          x1 = x0;
          f1 = f0;
      }
/* Test for accuracy and repeat the process, if necessary */
      if(fabs((x2-x1)/x2) < EPS)
      {
          *s = 1;
          *root = (x1+x2)/2.0;
          return;                   /* Iteration ends */
      }
      else
      {
          *count = *count + 1;
          goto begin;
      }
    }
}
/* End of subroutine bim() */
/* ---------------------------------------------------- */
```

Program 3 FALSE

```
/* --------------------------------------------------- *
 * Main program                                        *
 *   This program finds a root of a nonlinear equation *
 *   by false position method                          *
 * --------------------------------------------------- *
 * Functions invoked                                   *
 *   Macro F(x)                                        *
 * --------------------------------------------------- *
 * Subroutines used                                    *
 *   fal()                                             *
 * --------------------------------------------------- *
 * Variables used                                      *
 *   a - Left endpoint of interval                     *
 *   b - Right endpoint of interval                    *
 *   s - Status                                        *
 *   root - Final solution                             *
 *   count - Number of iterations completed            *
 * --------------------------------------------------- *
 * Constants used                                      *
 *   EPS - Error bound                                 *
 * --------------------------------------------------- */
```

```
#include  <math.h>
#define EPS      0.000001
#define F(x)     (x)*(x)+(x)-2
main( )
{
      int s,count;
      float a,b,root;
      printf("\n");
      printf("SOLUTION BY FALSE POSITION METHOD \n");
      printf("\n");
      printf("Input starting values \n");
      scanf("%f %f", &a, &b);
/* Calling the function fal() */
      fal(&a,&b,&s,&root,&count);
      if(s==0)
      {
          printf("\n");
          printf("Starting points do not bracket
            any root \n");
          printf("\n");
      }
      else
      {
          printf("\n");
          printf("Root = %f \n", root);
          printf("F(root) = %f \n", F(root));
          printf("\n NO. OF ITERATIONS = %d \n", count);
      }
}
/* End of main() program */
/* ---------------------------------------------------- */
/* Defining the subroutine fal() */
fal(float *a, float *b, int *s, float *root, int *count)
/* fal finds a root of a nonlinear equation */
{
      float x1,x2,x0,f0,f1,f2;
      x1 = *a;
      x2 = *b;
      f1 = F(x1);
      f2 = F(x2);
/* Test if a and b bracket a root */
```

```
      if(f1*f2 > 0)
      {
          *s = 0;
          return;                 /* Program terminated */
      }
      else
      {
          printf("       n         x1           x2 \n");
      }
      *count = 1;
      begin:                       /* Iteration begins */
      x0 = x1-f1*(x2-x1)/(f2-f1);
      f0 = F(x0);
      if(f1*f2 < 0)
      {
          x2 = x0;
          f2 = f0;
      }
      else
      {
          x1 = x0;
          f1 = f0;
      }
      printf("%5d %15.6f %15.6f \n", *count, x1, x2);
/* Test whether the desired accuracy has been achieved */
      if(fabs((x2-x1)/x2) < EPS)
      {
          *s = 1;
          *root = (x1+x2)*0.5;
          return;                    /* Iteration ends */
      }
      else
      {
          *count = *count + 1;
          goto begin;
      }
}
/* End of subroutine fal() */
/* ---------------------------------------------- */
```

Program 4 NEWTON

```
/* ----------------------------------------------- *
 * Main program                                    *
 *  This program finds a root of a nonlinear equation *
 *  by Newton-Raphson method                       *
 * ---------------------------------------------------------- *
 * Functions invoked                                          *
 *   Macros F(x), FD(x), Library function fabs()              *
 * ---------------------------------------------------------- *
 * Subroutines used                                           *
 *   NIL                                                      *
 * ---------------------------------------------------------- *
 * Variables used                                             *
 *   x0 - Initial vale of x                                   *
 *   xn - New value of x                                      *
 *   fx - Function value at x                                 *
 *   fdx - Value of function derivative at x                  *
 *   count - Number of iterations done                        *
 * ---------------------------------------------------------- *
 *   Constants used                                           *
 *     EPS - Error bound                                      *
 *     MAXIT - Maximum number of iterations permitted         *
 * ---------------------------------------------------------- */

#include <math.h>
#define EPS 0.000001
#define MAXIT 20
#define F(x)  (x)*(x)+(x)-2
#define FD(x)  2*(x)+1

main( )
{
     int count;
     float x0, xn, fx, fdx;

     printf("\n");
     printf("Input initial value of x \n");
     scanf("%f", &x0);
     printf("\n");
     printf("      SOLUTION BY NEWTON-RAPHSON METHOD \n");
     printf("\n");

     count = 1;
     begin:                          /* Iteration begins */
     fx = F(x0);
     fdx = FD(x0);
     xn = x0 - fx/fdx;

     if(fabs((xn-x0)/xn) < EPS)      /* Iteration ends */
     {
        printf("Root = %f \n", xn);
        printf("Function value = %f \n", F(xn));
        printf("Number of iterations = %d \n", count);
        printf("\n");
```

```
   }
   else
 {
    x0 = xn;
    count = count + 1;
    if(count < MAXIT)
    {
        goto begin;
    }
    else
    {
        printf("\n SOLUTION DOES NOT CONVERGE \n");
        printf("IN %d ITERATIONS \n", MAXIT);
    }
   }
}
/* End of main() program */
/* -------------------------------------------------- */
```

Program 5 SECANT

```
/* ------------------------------------------------ *
*                                                   *
* Main program                                      *
*   This program finds a root of a nonlinear        *
*   equation by secant method                       *
* ------------------------------------------------- *
*                                                   *
* Functions invoked                                 *
*   Macro F(x)                                      *
* ------------------------------------------------- *
*                                                   *
* Subroutines used                                  *
*   sec()                                           *
* ------------------------------------------------- *
*                                                   *
*Variables used                                     *
*      a - Left endpoint of interval                *
*      b - Right endpoint of interval               *
*     x1 - New left point                           *
*     x2 - New right point                          *
*   root - Final solution                           *
*  count - Number of iterations completed           *
* ------------------------------------------------- *
*                                                   *
* Constants used                                    *
*   EPS - Error bound                               *
*   MAXIT - Maximum number of iterations permitted  *
* ------------------------------------------------- */

#include  <math.h>
#define   EPS     0.000001
```

```
#define   MAXIT   50
#define   F(x)    (x)*(x)+(x)-2
main( )
{
     float a,b,root,x1,x2;
     int count,status;

     printf("\n");
     printf("         SOLUTION BY SECANT METHOD \n");
     printf("\n");
     printf("Input two starting points \n");
     scanf("%f %f", &a, &b);

     sec(&a, &b, &x1, &x2, &root, &count, &status);

     if(status == 1)
     {
        printf("\n");
        printf("       DIVISION BY ZERO \n");
        printf("\nLast x1 = %f \n", x1);
        printf("\nLast x2 = %f \n", x2);
        printf("\nNO. OF ITERATIONS = %d \n", count);
        printf("\n");
     }
     else if(status == 2)
     {
        printf("\n");
        printf("NO CONVERGENCE IN %d ITERATIONS \n", MAXIT);
        printf("\n");
     }
     else
     {
        printf("\n");
        printf("Root = %f \n", root);
        printf("Function value at root = %f \n", F(root));
        printf("\n");
        printf("NO. OF ITERATIONS = %d \n", count);
        printf("\n");
     }
}
/* End of main() program */
/* -------------------------------------------- */
/* Defining subroutine sec() */
sec(float *a, float *b, float *x1, float *x2, float
     *root, int *count, int *status)
```

```
/*  This subroutine computes a root of an equation using
    the secant method. */
{
      float x3,f1,f2,error;
/*  Function values at initial points */
      *x1 = *a;
      *x2 = *b;
      f1 = F(*a);
      f2 = F(*b);
/*  Compute the root iteratively */
      *count = 1;
      begin:            /* Iteration process begins */
      if(fabs(f1-f2) <= 1.E-10)
      {
          *status = 1;
          return;              /* Program terminated */
      }
      x3 = *x2 - f2*(*x2-*x1)/(f2-f1);
      error = fabs((x3-*x2)/x3);
/* Test for accuracy */
      if(error > EPS)
      {
/*  Test for convergence */
         if(*count == MAXIT)
         {
             *status = 2;
             return;          /* Program terminated */
         }
         else
         {
             *x1 = *x2;
         }
         *x2 = x3;
         f1 = f2;
         f2 = F(x3);
         *count = *count + 1;
         goto begin;    /* Compute next approximation */
   }
   else
   {
         *root = x3;
         *status = 3;
         return;                   /* Iteration ends */
    }
}
/* End of subroutine sec() */
/* ---------------------------------------------- */
```

Program 6 FIXEDP

```
/* ---------------------------------------------- *
 * Main program                                   *
 *  This program finds a root of a function using *
 *  the fixedp point iteration method             *
 * ---------------------------------------------- *
 * Functions invoked                              *
 *  Library function fabs() and macro G(x)        *
 * ---------------------------------------------- *
 * Subroutines used                               *
 *  NIL                                           *
 * ---------------------------------------------- *
 * Variables used                                 *
 *  x0 - Initial guess                            *
 *  x - Estimated root                            *
 *  error - Relative error                        *
 * ---------------------------------------------- *
 *Constants used                                  *
 *  EPS - Error bound                             *
 *  MAXIT - Maximum iterations allowed            *
 * ---------------------------------------------- */
# include <math.h>
# define EPS  0.000001
# define G(x)  2.0-(x)*(x)

main( )
{
    int MAXIT, i;
    float x0, x, error;
    printf("\n       SOLUTION BY FIXED-POINT METHOD \n");
    printf("\n");
    printf("Input initial estimate of a root \n");
    scanf("%f", &x0);
    printf("Maximum iterations allowed \n");
    scanf("%d", &MAXIT);
    printf("\n   ITERATION   VALUE OF X   ERROR \n");
 /*Iteration process begins*/
    for(i=1;i<=MAXIT;i++)
    {
        x = G(x0);
```

```
      error = fabs((x-x0)/x);
      printf("%10d  %10.7f  %10.7f \n", i, x, error);
      if (error < EPS)
          goto end;   /*Iteration process ends*/
      else
      x0 = x;
    }
    printf("\nProcess does not converge to a root.\n");
    printf("Exit from iteration loop.\n");
    end:
    ;
}
/* End of main() program*/
/* ---------------------------------------------- */
```

Program 7 MULTIR

```
/* ---------------------------------------------- *
 * Main program                                    *
 *  The program finds all the real roots of a polynomial *
 * ---------------------------------------------- *
 * Functions invoked                               *
 *  NIL                                            *
 * ---------------------------------------------- *
 * Subroutines used                                *
 *  newton()                                       *
 *  dflat()                                        *
 * ---------------------------------------------- *
 * Variables used                                  *
 *  n - Degree of polynomial                       *
 *  a - Polynomial coefficients A(N+1)             *
 *  x0 - Initial guess                             *
 *  xr - Root obtained by Newton method            *
 *  root - Root Vector                             *
 *  status - Solution status                       *
 * ---------------------------------------------- *
 * Constants used                                  *
 *  EPS - Error bound                              *
 *  MAXIT - Maximum iterations permitted           *
 * ---------------------------------------------- *

#include <math.h>
#define EPS 0.000001
#define MAXIT 50

void main( )
{
    int n, status, i, j;
```

```
     float a[11], root[10], x0, xr;
     void dflat(int n, float a[11], float xr);
     void newton(int n, float a[11], float x0, int *status,
                  float *xr);
     printf("\n");
     printf("\n     EVALUATION OF MULTIPLE ROOTS \n");
     printf("\n");
     printf("Input degree of polynomial, n. \n");
     scanf("%d", &n);
     printf("\nInput poly coefficients, a(1) to a(n+1). \n");
     for(i = 1;  i <= n+1; i++)
        scanf("%f",  &a[i]);
     printf("\nInput initial guess of x \n");
     scanf ("%f",  &x0);
     printf("\n");
/* Process of root searching begins */
     for(i=n;i>=2;i--)
     /* Find ith root by Newton Method */
  {
     newton(n,a,x0,&status,&xr);
     if(status == 2)
     {
          for(j=n;j>=i+1;j--)
        printf("root %d = %f \n",j, root[j]);
          printf("\nNext root does not converge in \n");
          printf("%d iterations \n",  MAXIT);
          printf("\n");
          goto end;    /* Processing ends */
     }
      root[i]  =  xr;
/* Deflate the polynomial by division (x-xr) */
     dflat(n,a,xr);
     x0 = xr;
/* Proceed to next root */
  }  /* End of for loop */
/* Compute the last root */
     root[1]  = -a[1]/a[2];
/* Write results */
     printf("\n  ROOTS OF POLYNOMIAL ARE: \n");
     printf("\n");
     for (i=1;i<=n;i++)
        printf("ROOT %d = %f \n",  i, root[i]);
     printf("\n");
     end:
     printf("END");
}
/* End of main() program */
/* ----------------------------------------------- */
/* Defining the subroutine newton() */
void newton(int n, float a[11], float x0, int *status,
              float *xr)
/*  This subroutine finds a root of the polynomial using
    the Newton-Raphson method */
{
      int i, count;
      float fx;  /* Value of polynomial at x0 */
      float fdx; /* Value of polynomial derivative at x0 */
      count = 1;
/* Compute the value of function at x0 */
      begin:
      fx = a[n+1];
      for(i=n;i>=1;i--)
      fx = fx * x0 + a[i];
/* Compute the value of derivative at x0 */
      fdx = a[n+1] * n;
      for(i=n;i>=2;i--)
      fdx = fdx * x0 + a[i] * (i-1);
/* Compute a root xr */
      *xr  =  x0-fx/fdx;
/* Test for accuracy */
      if(fabs((*xr-x0)/(*xr))  <= EPS)
      {
      *status  = 1;
      return;
      }
/*  Test for convergence */
      if(count<MAXIT)
      {
          x0 = *xr;
          count = count + 1;
          goto begin;
      }
      else
      {
            *status = 2;
            return;
          }

}
/* End of subroutine newton() */
/* ------------------------------------------------------ */
/* Defining the subroutine dflat() */
void dflat(int n,  float a[11],  float xr)
/*  This subroutine reduces the degree of polynomial by
    one using synthetic division */
{
      float b[11];
      int i;
/* Evaluate the coefficients of the reduced polynomial */
      b[n+1]   = 0;
      for(i=n;i>=1;i--)
      b[i] = a[i+1] + xr * b[i+1];
/* Change coefficients from b array to a array */
      for(i=1;i<=n+1;i++)
      a[i] = b[i];
}
/*  End of subroutine dflat() */
/* ------------------------------------------------------ */
```

Program 8 COMPR

```
* ------------------------------------------------------ *
* Main program                                           *
*   This program locates all the roots, both real        *
*   and complex, using the Bairstow method               *
* ------------------------------------------------------ *
* Functions invoked                                      *
*   NIL                                                  *
* ------------------------------------------------------ *
* Subroutines used                                       *
*   input,bstow,quad,output                              *
* ------------------------------------------------------ *
* Variables used                                         *
*   n - Degree of polynomial                             *
*   a - Array of coefficients of polynomial              *
*   u0,v0 - Initial values of coefficients of the        *
*           quadratic factor                             *
*   u,v   - Computed(values of coefficients of the   *
*           quadratic factor                          *
*   b - Coefficients of the reduced polynomial       *
*   x1,x2 - Roots of the quadratic factor            *
*   type  - Type of roots (real,imaginary or equal)  *
* ------------------------------------------------- *
* Constants used                                     *
*   EPS - Error bound                                *
* ------------------------------------------------- */

#include <math.h>
#define  EPS     0.000001
#define  image   1
#define  equal   2
#define  unequal 3

main( )
{
     int n, i;
     float a[11], b[11], u0, v0, u, v, x1, x2, d0, d1,
        d2, root, type, status;

     printf("\n");
     printf("         EVALUATION OF COMPLEX ROOTS \n");
     printf("\n");

/* Get input data */
     printf("Input degree of polynomial, n \n");
     scanf("%d",&n);
     printf("\n Input coefficients a(n+1) to a(1) \n");
     for(i=n+1;i>=1;i--)
        scanf("%f",&a[i]);
     printf("\n Give initial values u0 and v0 \n");
     scanf("%f %f",&u0,&v0);

     begin:
     if(n > 2)
     {

     /* Obtain a quadratic factor */

bstow(n,a[11],b[11],u0,v0,&u,&v,&status);
     if(status == 1)
     {
          d2 = 1;
          d1 = -u;
          d0 = -v;
     }
     else
     {
        printf("\n No Convergence in 100 iterations \n");
        goto end;
     }
     /* Find roots of the quadratic factor */
       quad(d2,d1,d0,&x1,&x2,&type);
     /* Print the roots */
      output(n,type,x1,x2);
     /* Set the coefficients of the factor polynomial */
        n = n-2;
        for(i=1;i<=n+1;i++)
                 a[i] = b[i+2];
     /* Set initial values for next quadratic factor */
        u0 = u;
        v0 = v;
        goto begin;
     }        /* endif */
     if(n == 2)          /* polynomial is quadratic */
     {
        quad(a[3],a[2],a[1],&x1,&x2,&type);
        output(n,type,x1,x2);
     }
     else       /* last root of an odd order polynomial */
     {
        root = -a[1]/a[2];
        printf("\n");
        printf("Final root = %f \n", root);
        printf("\n");
     }
     end:
     printf("End");
}
/* End of main() program */
/* ----------------------------------------------- */
/* Defining the subroutine bstow() */
bstow(int n,float a[11],float b[11],float u0,float v0,
      float *u,float *v,float *status)
/* This subroutine finds the quadratic factor using
   multivariable Newton's method and also finds the
   reduced polynomial */
```

```
{
    float d,delu,delv,c[11];
    int count,i;
    count = 1;
    begin:
    b[n+1] = a[n+1];
    b[n] = a[n] + u0*b[n+1];
    for(i=n-1;i>=1;i--)
        b[i] = a[i] + u0 * b[i+1] + v0 * b[i+2];
    c[n+1] = 0;
    c[n] = b[n+1];
    for(i=n-1;i>=1;i--)
      c[i] = b[i+1] + u0 * c[i+1] + v0 * c[i+2];
    d = c[2] * c[2] - c[1] * c[3];
    delu = -(b[2] * c[2] - b[1] * c[3])/d;
    delv = -(b[1] * c[2] - b[2] * c[2])/d;
    *u = u0 + delu;
    *v = v0 + delv;
    if(fabs(delu/*u) <= EPS && fabs(delv/*v) <= EPS)
    {
      *status = 1;
      return;
    }
    if(count < 100)
    {
      u0 = *u;
      v0 = *v;
      count = count + 1;
      goto begin;
    }
    else
    {
      *status = 2;
      return;
    }
}
/* End of subroutine bstow() */
/* ---------------------------------------------- */
/* Define the subroutine quad() */
quad(float a,float b,float c,float *x1,float *x2,float
*type)
```

```
/*  This subroutine solves a quadratic equation of type
    a(x*x) + bx + c */
{
      float q;

      q = b*b - 4*a*c;
      if(q < 0.0)                  /* roots are complex */
    {
      *x1 = -b/(2*a);
      *x2 = sqrt(fabs(q))/(2*a);
      *type = image;
    }
    else if (q == 0.0)    /* roots are real and equal */
    {
      *x1 = -b/(2*a);
      *x2 = *x1;
      *type = equal;
    }
     else                  /* roots are real and unequal */
    {
      *x1 = (-b+sqrt(q))/(2*a);
      *x2 = (-b-sqrt(q))/(2*a);
    }
    return;
}
/* End of subroutine quad() */
/* ---------------------------------------------------------- */

/*  Defining output() routine */
output(int n, int type, float x1, float x2)
/* This subroutine displays the roots of the quadratic
    equation */
{
   printf("\n");
   printf("Roots of quadratic factor at n = %d \n", n);
   printf("\n");

   if(type == image)
   {
     printf("Root1 = %f+%fj \n", x1, x2);
     printf("Root2 = %f-%fj \n", x1, x2);
   }
   else if(type == equal)
   {
     printf("Root1 = %f \n", x1);
     printf("Root2 = %f \n", x2);
   }
   else               /* Type == unequal */
```

```
  {
    printf("Root1 = %f \n", x1);
    printf("Root2 = %f \n", x2);
  }
  return;
}
/*  End of subroutine output() */
/* ---------------------------------------------------- */
```

Program 9 MULLER

```
/* ---------------------------------------------------- *
 * Main program                                         *
 *   This program evaluates root of a polynomial using  *
 *   Muller's method                                    *
 * ---------------------------------------------------- *
 * Functions invoked                                    *
 *   NIL                                                *
 * ---------------------------------------------------- *
 * Subroutines used                                     *
 *   F(x)                                               *
 * ---------------------------------------------------- *
 * Variables used                                       *
 *   x1,x2,x3 - initial values                          *
 *   f1,f2,f3 - function values at x1,x2,x3             *
 *   a0,a1,a2 - coefficients of quadratic polynomial    *
 *   hi -   xi-x3                                       *
 *   di -  function difference fi-f3                    *
 * ---------------------------------------------------- *
 * Constants used                                       *
 *   EPS  -  Error bound                                *
 * ---------------------------------------------------- */

#include <math.h>
#define EPS 0.000001

main( )
{
    float F(float x);
    float x1,x2,x3,x4,f1,f2,f3,f4,h1,h2,d1,d2,a0,a1,a2,h;
    printf("\nInput three initial points \n");
    scanf("%f %f %f", &x1,&x2,&x3);

    f1 = F(x1);
    f2 = F(x2);
    f3 = F(x3);

    begin:
    h1 = x1-x3;
```

```
      h2 = x2-x3;
      d1 = f1-f3;
      d2 = f2-f3;
/*  Compute parameters a0,a1,a2 */
      a0 = f3;
      a1 = (d2*h1*h1-d1*h2*h2)/(h1*h2*(h1-h2));
      a2 = (d1*h2-d2*h1)/(h*h2*(h1-h2));
/*  Compute h */
      if(a1>0.0)
         h=(-2.0*a0)/(a1+sqrt(a1*a1-4*a2*a0));
      else
         h=(-2.0*a0)/(a1-sqrt(a1*a1-4*a2*a0));
/* Compute x4 and f4 */
      x4 = x3+h;
      f4 = F(x4);
/* Test for accuracy */
      if(f4<=EPS) /* root obtained */
      {
         printf("\n\nROOT BY MULLER'S METHOD\n\n");
         printf("Root=%f\n", x4);
         printf("\n");
      }
      else
      {
         x1 = x2;
         x2 = x3;
         x3 = x4;
         f1 = f2;
         f2 = f3;
         f3 = f4;
         goto begin;
      }
}
/* End of main() program */
/* --------------------------------------------- */
/* Defining the subroutine F(x) */
float F(float x)
{
    float f;
    f = x*x*x+2*x*x+10*x-20;
    return (f);
}
/* End of subroutine () */
/* --------------------------------------------- */
```

Program 10 LEG1

```
/*------------------------------------------------------*
 * Main program                                          *
 *   This program solves a system of linear equations   *
 *   using simple Gaussian elimination method            *
 *------------------------------------------------------*
 * Functions invoked                                     *
 *   NIL                                                 *
 *------------------------------------------------------*
 * Subroutines used                                      *
 *   GAUSS1                                              *
 *------------------------------------------------------*
 * Variables used                                        *
 *   n - Number of equations in the system               *
 *   a - Matrix of coefficients                          *
 *   b - Right side vector                               *
 *   x - Solution vector                                 *
 *------------------------------------------------------*
 *   status - Solution status                            *
 *------------------------------------------------------*/

main( )
{
    int status,n,i,j;
    float a[10][10], b[10], x[10];

    printf("\n    SOLUTION BY SIMPLE GAUSS METHOD \n");

    printf("What is the size of the system (n)? \n");
    scanf("%d", &n);

    printf("Input coefficients a(i,j), row-wise, \n");
    printf("one row on each line. \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
          scanf("%f", &a[i][j]);
    printf("\nInput vector b \n");
    for(i=1;i<=n;i++)
      scanf("%f", &b[i]);

/* obtain solution by simple Gauss elimination method */

/* call the subroutine gauss1() */

    gauss1(n,a,b,x,&status);

    if(status != 0)
    {
      printf("\nSOLUTION VECTOR X \n");
      for(i=1;i<=n;i++)
           printf("%10.6f", x[i]);
      printf("\n");
    }
    else
    {
      printf("Singular matrix, reorder equations. \n");
    }
  }
  /* End of main() program */
  /* ---------------------------------------------------- */
  /* Defining subroutine gauss1() */
  gauss1(int n, float a[10][10], float b[10], float x[10],
         int *status)
  /* This subroutine solves a set of n linear equations
     by Gauss elimination method */
  {
     int i,j,k;
     float pivot, factor, sum;
  /* ---------- Elimination begins ----------- */
     for(k=1;k<=n-1;k++)
     {
       pivot = a[k][k];
       if(pivot < 0.000001)
       {
          *status = 0;
          return;
       }
       *status = 1;
       for(i=k+1;i<=n;i++)
       {
          factor = a[i][k] /pivot;
          for(j=k+1;j<=n;j++)
          {
             a[i][j] = a[i][j] - factor * a[k][j];
          }
          b[i] = b[i] - factor * b[k];
       }
     }
  /* ------ Back substitution begins ------------- */
     x[n] = b[n] / a[n][n];
     for(k=n-1;k>=1;k--)
  {
    sum = 0.0;
    for(j=k+1;j<=n;j++)
        sum = sum + a[k][j] * x[j];
    x[k] = (b[k] - sum) / a[k][k];
  }
  return;
}
/* End of subroutine gauss1 () */
/*-------------------------------------------------*/
```

Program 11 LEG2

```
/*-------------------------------------------------*
 * Main program                                    *
 *   This program solves a system of linear equations *
 *   using Gaussian elimination with partial pivoting *
 *-------------------------------------------------*
 * Functions invoked                               *
 *   NIL                                           *
 *-------------------------------------------------*
 * Subroutines used                                *
 *   Gauss2                                        *
 *-------------------------------------------------*
 * Variables used                                  *
 *   n - Number of equations                       *
 *   a - Coefficients matrix                       *
 *   b - Right side vector                         *
 *   x - Solution vector                           *
 *-------------------------------------------------*
 * Constants used                                  *
 *   NIL                                           *
 *-------------------------------------------------*/

main( )
{
    int i,j,n;
    float a[10][10], b[10], x[10];

    printf("\n  GAUSS METHOD WITH PARTIAL PIVOTING \n");

    printf("\nWhat is the size n of the system? \n");
    scanf("%d", &n);
    printf("\nInput coefficients a(i,j), row-wise \n");
    printf("one row on each line \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
           scanf("%f",&a[i][j]);
    printf("\nEnter vector b \n");
    for(i=1;i<=n;i++)
    scanf("%f", &b[i]);
    gauss2(n,a,b,x);
    printf("\n          SOLUTION VECTOR X \n");
    printf("\n");
    for(i=1;i<=n;i++)
        printf("\t %f", x[i]);
    printf("\n");
}
/* End of main() program */
/* ------------------------------------------------ */
/* Defining subroutine gauss2() */
gauss2(int n, float a[10][10],float b[10], float x[10])
/* This subroutine solves a system of linear equations using
   Gauss elimination method with partial pivoting */
{
/* Forward elimination */
    elim(n,a,b);
/* Solution by back substitution */
    bsub(n,a,b,x);
    return;
} /* Endof routine gauss2() */
/* ------------------------------------------------ */
/* Defining the subroutine elim() */
[illegible]
/* This subroutine performs forward elimination
    incorporating partial pivoting technique */
{
      int i,j,k;
      float factor;
      for(k=1;k<=n-1;k++)
      {
        pivot(n,a,b,k);
        for(i=k+1;i<=n;i++)
        {
          factor = a[i][k] / a[k][k];
          for(j=k+1;j<=n;j++)
          {
            a[i][j] = a[i][j]-factor * a[k][j];
         }
         b[i] = b[i]-factor*b[k];
      }
   }
   return;
}  /* End of elim() routine */
/* -------------------------------------------- */
/* Defining subroutine pivot() */
# include <math.h>
pivot(int n, float a[10][10], float b[10], int k)
/* This subroutine performs the task of partial pivoting
   (reordering of equations) */
{
     int p,i,j;
     float large, temp;
/* Find pivot p */
     p = k;
     large = fabs(a[k][k]);
     for(i=k+1;i<=n;i++)
     {
          if(fabs(a[i][k])>large)
          {
               large = fabs(a[i][k]);
               p = i;
          }
     }
/* Exchange rows p and k */
     if(p!=k)
     {
          for(j=k;j<=n;j++)
          {
               temp = a[p][j];
               a[p][j] = a[k][j];
               a[k][j] = temp;
          }
          temp = b[p];
          b[p] = b[k];
          b[k] = temp;
     }
     return;
}
/* End of subroutine pivot() */
/* -------------------------------------------- */
/* Defining subroutine bsub() */

bsub(int n, float a[10][10], float b[10], float x[10])

/* This subroutine obtains the solution vector x by back
substitution */

{
      int i,j,k;
      float sum;
      x[n] = b[n] / a[n][n];
      for(k=n-1;k>=1;k--)
      {
        sum = 0.0;
        for(j=k+1;j<=n;j++)
              sum = sum + a[k][j] * x[j];
        x[k] = (b[k]-sum) / a[k][k];
      }
      return;
}
/* End of subroutine bsub() */
/* ----------------------------------------------- */
```

Program 12 DOLIT

```
/* ----------------------------------------------- *
 *  Main program                                   *
 *    This program solves a system of linear equations *
 *    using Dolittle LU decomposition              *
 * ----------------------------------------------- *
 *  Functions invoked                              *
 *    NIL                                          *
 * ----------------------------------------------- *
 *  Subroutines used                               *
 *    LUD, SOLVE                                   *
 * ----------------------------------------------- *
 *  Variables used                                 *
 *    n - System size                              *
 *    a - Coefficient matrix of the system         *
 *    b - Right side vector                        *
 *    l - Lower triangular matrix                  *
 *    u - Upper triangular matrix                  *
 *    fact - Factorization status                  *
 * ----------------------------------------------- *
 *  Constants used                                 *
 *    YES,NO                                       *
 * ----------------------------------------------- */
```

```
# define YES 1
# define NO  0

main( )
{
     int n, fact, i,j;
     float a[10][10], u[10][10], l[10][10], b[10], x[10];

     printf("\n      SOLUTION BY DOLITTLE METHOD \n\n");

/* Read input data */
     printf("\nWhat is size of A? \n");
     scanf("%d", &n);
     printf("Type coefficients a(i,j), row by row \n");
     for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
           scanf("%f", &a[i][j]);
        printf("\nType vector b on one line \n");
        for(i=1;i<=n;i++)
           scanf("%f", &b[i]);
/* LU factorization */

      lud(n,a,u,l,&fact);

      if(fact == YES)          /* Print LU matrices */
      {

/* Print U matrix */
      printf("\nMATRIX U \n");
      for(i=1;i<=n;i++)
      {
      for(j=1;j<=n;j++)
      {
        printf("%15.6f", u[i][j]);
      }
      printf("\n");
      }
/* Print L matrix */
      printf("\nMATRIX L \n");
      for(i=1;i<=n;i++)
      {
         for(j=1;j<=n;j++)
         {
            printf("%15.6f", l[i][j]);
         }
         printf("\n");
       }
```

```
/* Solve for x */
      solve(n,u,l,b,x);
      printf("\nSOLUTION VECTOR X \n\n");
      for(i=1;i<=n;i++)
           printf("%15.6f \n", x[i]);
      printf("\n");
      }
      else
      {
      printf("\n       FACTORIZATION NOT POSSIBLE \n");
      }
}
/* End of main() program */
/* ----------------------------------------------------- */
/* Defining the subroutine lud() */
lud(int n, float a[10][10], float u[10][10], float
l[10][10], int *fact)
/*  This subroutine decomposes the matrix A into L and U
    matrices using DOLITTLE algorithm */
{
        int i,j,k;
        float sum;
/* Initialize U and L matrices */
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
      {
        u[i][j] = 0.0;
        l[i][j] = 0.0;
      }
/* Compute the elements of U and L */
    for(j=1;j<=n;j++)
      u[1][j] = a[1][j];
    for(i=1;i<=n;i++)
      l[i][i] = 1.0;
    for(i=1;i<=n;i++)
       l[i][1] = a[i][1] / u[1][1];
    for(j=2;j<=n;j++)
    {
      for(i=2;i<=j;i++)
      {
        sum = a[i][j];
        for(k=1;k<=i-1;k++)
              sum = sum - l[i][k] * u[k][j];
        u[i][j] = sum;
      }
      if(u[j][j]<=1.E-6)
      {
        *fact = NO;
        return;
      }
      for(i=j+1;i<=n;i++)
      {
        sum = a[i][j];
        for(k=1;k<=j-1;k++)
              sum = sum - l[i][k] * u[k][j];
        l[i][j] = sum / u[j][j];
      }
    }
    *fact = YES;
    return;
} /* End of subroutine lud() */
/* ------------------------------------------------------- */
/* Defining the subroutine solve() */
solve(int n, float u[10][10], float l[10][10], float
b[10], float x[10])
/*  This subroutine obtains the solution vector X using the
    coefficients of L and U matrices */
{
    int i,j;
    float sum, z[10];
 /* Forward substitution */
     z[1] = b[1];
     for(i=2;i<=n;i++)
     {
       sum = 0.0;
       for(j=1;j<=i-1;j++)
             sum = sum + l[i][j] * z[j];
       z[i] = b[i] - sum;
     }
 /* Back substitution */
       x[n] = z[n]/u[n][n];
       for(i=n-1;i>=1;i--)
       {
         sum = 0.0;
         for(j=i+1;j<=n;j++)
               sum = sum + u[i][j] * x[j];
        x[i] = (z[i]-sum)/u[i][i];
      }
      return;
}
/* End of subroutine solve() */
/* ---------------------------------------------- */
```

Program 13 JACIT

```
/* ---------------------------------------------- *
 * Main program                                   *
 *    This program uses the subprogram JACOBI to solve *
 *    a system of equations by Jacobi iteration method *
 * ---------------------------------------------- *
 * Functions invoked                              *
 *    NIL                                         *
 * ---------------------------------------------- *
 * Subroutines used                               *
 *    JACOBI                                      *
 * ---------------------------------------------- *
 * Variables used                                 *
 *    a - Coefficient matirx                      *
 *    b - Right side vector                       *
 *    n - System size                             *
 *    x - Solution vector                         *
 *    count - Number of iterations completed      *
 *    status - Convergence status                 *
 * ---------------------------------------------- *
 * Constants used                                 *
 *    EPS - Error bound                           *
 *    MAXIT - Maximum iterations permitted        *
 * ---------------------------------------------- */
#include <math.h>
#define EPS 0.000001
#define MAXIT 100

main( )
{
    int i,j,n,count,status;
    float a[10][10], b[10], x[10];

    printf("\n      SOLUTION BY JACOBI ITERATION \n");
    printf("\nWhat is the size n of the system? \n");
    scanf("%d", &n);
    printf("\nInput coefficients a(i,j), row by row \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
```

```
            scanf("%f", &a[i][j]);
    printf("\nInput vector b\n");
    for(i=1;i<=n;i++)
      scanf("%f", &b[i]);
    jacobi(n,a,b,x, &count, &status);
    if(status==2)
    {
      printf("\nNo convergence in %d iterations", MAXIT);
      printf("\n\n");
    }
    else
    {
      printf("\n                    SOLUTION VECTOR X \n\n");
      for(i=1;i<=n;i++)
          printf("%15.6f", x[i]);
      printf("\n\nIterations = %d", count);
    }
}
/* End of main() program */
/* ----------------------------------------------------------- */
/* Defining the routine jacobi() */
jacobi(int n, float a[10][10], float b[10], float x[10], int
*count, int *status)
/*  This subroutine solves a system of n linear equations using
    the Jacobi iteration method */
{
      int i,j,key;
      float sum, x0[10];
 /* Initial values of x */
      for(i=1;i<=n;i++)
        x0[i] = b[i] / a[i][i];
 /* Jacobi iteration begins */
      *count = 1;
      begin:
      key = 0;
 /* Computing values of x(i) */
      for(i=1;i<=n;i++)
      {
        sum = b[i];
        for(j=1;j<=n;j++)
        {
            if(i==j)
                continue;
            sum = sum - a[i][j] * x0[j];
        }
        x[i] = sum / a[i][i];
        if(key == 0)
        {
/* Testing for accuracy */
            if(fabs((x[i]-x0[i])/x[i]) > EPS)
                key = 1;
        }
    }
/* Testing for convergence */
     if(key == 1)
     {
       if(*count==MAXIT)
       {
           *status = 2;
           return;
     }
     else
     {
           *status = 1;
           for(i=1;i<=n;i++)
               x0[i] = x[i];
     }
     *count = *count+1;
     goto begin;
   }
   return;
}
/* End of subroutine jacobi() */
/* ------------------------------------------------------ */
```

Program 14 GASIT

```
/*-------------------------------------------------------- 
 * Main program                                           *
 *   This program uses the subprogram GASEID to solve a  *
 *   system of equations by Gauss-Seidel iteration method *
 *--------------------------------------------------------*
 * Functions invoked                                      *
 *   NIL                                                  *
 *--------------------------------------------------------*
```

```
                                                            *
* Subroutines used                                          *
*    GASEID                                                 *
*-----------------------------------------------------------*
* Variables used                                            *
*    a - Coefficient matrix                                 *
*    b - Right side vector                                  *
*    n - System size                                        *
*    x - Solution vector                                    *
*    count - Number of iterations completed                 *
*    status - Convergence status                            *
*-----------------------------------------------------------*
* Constants used                                            *
*    EPS - Error bound                                      *
*    MAXIT - Maximum iterations permitted                   *
*-----------------------------------------------------------*/

# define MAXIT 50
# define EPS 0.000001

main( )
{
    float a[10][10], b[10], x[10];
    int i,j,n,count,status;
    printf("\n    SOLUTION BY GAUSS-SEIDEL ITERATION \n");
    printf("\nWhat is the size n of the system? \n");
    scanf("%d", &n);
    printf("\nInput coefficients a(i,j), row by row \n");
    printf("one row on each line \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
          scanf("%f", &a[i][j]);
    printf("\nInput vector b\n");
    for(i=1;i<=n;i++)
      scanf("%f", &b[i]);

    gaseid(n,a,b,x, &count, &status);

    if(status==2)
    {
      printf("\nNo convergence in %d iterations \n", MAXIT);
      printf("\n\n");
    }
    else
    {
      printf("\n                SOLUTION VECTOR X \n\n");
    for(i=1;i<=n;i++)
         printf("%15.6f", x[i]);
    printf("\n\nIterations = %d", count);
```

```
      }
}
* End of main() program */
/* ------------------------------------------------------ */
/*  Defining the routine gaseid() */
gaseid(int n, float a[10][10], float b[10], float x[10], int
        *count, int *status)
/*  This subroutine solves a system of linear equations using
    the Gauss-Seidel iteration algorithm */
{
      int i,j,key;
      float sum, x0[10];
/* Initial values of x */
      for(i=1;i<=n;i++)
        x0[i] = b[i] / a[i][i];
/* Gaseid iteration begins */
      *count = 1;
      begin:
      key = 0;
/* Computing values of x(i) */
      for(i=1;i<=n;i++)
      {
        sum = b[i];
        for(j=1;j<=n;j++)
        {
            if(i==j)
            continue;
            sum = sum - a[i][j] * x0[j];
        }
        x[i] = sum / a[i][i];
        if(key==0)
        {

            /* Testing for accuracy */
            if(fabs((x[i]-x0[i])/x[i])>EPS)
                key = 1;
        }
      }
/* Testing for convergence */
     if(key==1)
     {
         if(*count==MAXIT)
         {
            *status = 2;
            return;              /* Program terminated */
        }
        else
        {
            *status = 1;
            for(i=1;i<=n;i++)
              x0[i] = x[i];
        }
        *count = *count+1;
        goto begin;
    }
    return;
}
* End of routine gaseid() */
/* ---------------------------------------------------- */
```

Program 15 LAGRAN

```
/* ---------------------------------------------------- *
 * -Main program                                         *
 *    This program computes the interpolation value at a *
 *    specified point, given a set of data points, using *
 *    the Lagrange interpolation representation          *
 * ---------------------------------------------------- *
 *  Functions invoked                                    *
 *    NIL                                                *
 * ---------------------------------------------------- *
 *  Subroutines used                                     *
 *    NIL                                                *
 * ---------------------------------------------------- *
 *  Variables used                                       *
 *    n    - Number of data sets                         *
 *    x(i) - Data points                                 *
 *    f(i) - Function values at data points             *
 *    xp   - Point at which interpolation is required    *
 *    fp   - Interpolated value at XP                    *
 *    lf   - Lagrangian factor                           *
 * ---------------------------------------------------- *
 *  Constants used                                       *
 *    MAX - Maximum number of data points permitted      *
 * ---------------------------------------------------- 

#define MAX 10

main( )
{
        int n,i,j;
        float x[MAX],f[MAX],fp,lf,sum,xp;

        printf("\nInput number of data points, n \n");
        scanf("%d", &n);

        printf("\nInput data points x(i) and values f(i) \n");
        printf("(one set in each line)\n")
        for(i=1;i<=n;i++)
             scanf("%f %f", &x[i], &f[i]);

        printf("\nInput x at which interpolation
          is required. \n");
        scanf("%f", &xp);

        sum = 0.0;
        for(i=1;i<=n;i++)
        {
          lf = 1.0;
          for(j=1;j<=n;j++)
          {
                if(i!=j)
                   lf = lf*(xp-x[j])/(x[i]-x[j]);
          }
          sum = sum + lf * f[i];
        }
        fp = sum;
        printf("\n      LAGRANGIAN INTERPOLATION \n\n");

        printf("Interpolated function value \n");
        printf("at x=%f is %f \n", xp, fp);
}
/* End of main() program */

/* --------------------------------------------------- */
```

Program 16 NEWINT

```
/* ------------------------------------------------- *
 * Main program                                      *
 *    This program constructs the Newton interpolation *
 *    polynomial for a given set of data points and then *
 *    computes interpolation value at a specified value *
 * ------------------------------------------------- *
 * Functions invoked                                 *
 *    NIL                                            *
 * ------------------------------------------------- *
 * Subroutines used                                  *
 *    NIL                                            *
 * ------------------------------------------------- *
 *                                                              *
 * Variables used                                               *
 *     n  - Number of data points                               *
 *     x  - Array of independent data points                    *
 *     f  - Array of function values                            *
 *     xp - Desired point for interpolation                     *
 *     fp - Interpolation value at XP                           *
 *     f  - Difference table                                    *
 *     a  - Array of coefficients of interpolation              *
 *          polynomial                                          *
 * ------------------------------------------------------------ *
 * Constants used                                               *
 *   NIL                                                        *
 * ------------------------------------------------------------ */

main( )
{
      int i,j,n;
      float xp, fp, sum, pi, x[10], f[10], a[10], d[10][10];

      printf("\nInput number of data points \n");
      scanf("%d", &n);
      printf("\nInput values of x and f(x), \n");
      printf("One set on each line \n");

      for(i=1;i<=n;i++)
      scanf("%f %f", &x[i], &f[i]);

 /* Construct difference table */
      for(i=1;i<=n;i++)
        d[i][1] = f[i];

      for(j=2;j<=n;j++)
        for(i=1;i<=n-j+1;i++)
             d[i][j] = (d[i+1][j-1]-d[i][j-1])/(x[i+j-1]-x[i]);

 /* Set the coefficients of interpolation polynomial */
      for(j=1;j<=n;j++)
         a[j] = d[1][j];

 /* Compute interpolation value */
      printf("\nInput xp where interpolation is required \n");
      scanf("%f", &xp);
      sum = a[1];
      for(i=2;i<=n;i++)
      {
        pi = 1.0;
        for(j=1;j<=i-1;j++)
             pi = pi * (xp-x[j]);
        sum = sum + a[i] * pi;
```

```
       }
       fp = sum;
/* Write results */
       printf("\n");
       printf("       NEWTON INTERPOLATION \n");
       printf("\n");
       printf("Interpolated value \n");
       printf("at x = %f is %f \n", xp, fp);
}
/* End of main() program */
/* ------------------------------------------------------ */
```

Program 17 SPLINE

```
/* ---------------------------------------------------- *
 *  Main program                                        *
 *    This program computes the interpolation value at  *
 *    a specified value, given a set of table points,   *
 *    using the natural cubic spline interpolation      *
 * ---------------------------------------------------- *
 *  Functions invoked                                   *
 *    NIL                                               *
 * ---------------------------------------------------- *
 *  Subroutines used                                    *
 *    GAUSS                                             *
 * ---------------------------------------------------- *
 *  Variables used                                      *
 *    n  - Number of data points.                       *
 *    x  - N by 1 array of data points.                 *
 *    f  - N by 1 array of function values              *
 *    xp - Point at which interpolation is required     *
 *    fp - Interpolation value at XP                    *
 *    a  - Array of second derivatives (N-2 by 1)       *
 *    d  - Array representing (N-2 by 1)                *
 *    c  - Matrix (N-2 by N-2) representing the         *
 *           coefficients of second derivatives         *
 *    h  - Array of distances between data points       *
 *           ( h(i) = x(i) - x(i-1) )                   *
 *    df - Array of differences of functions            *
 * ---------------------------------------------------- *
 *  Constants used                                      *
 *    MAX - Maximum number of table points permitted    *
 * ---------------------------------------------------- */
#define S 10
main( )
{
      int i,j,n,m;
      float x[S],f[S],a[S],d[S],c[S][S],h[S],df[S],u[S],
            xp,fp,q1,q2,q3;
/* Reading input data */
      printf("\nInput number of data points, n \n");
      scanf("%d", &n);
      printf("\nInput data points x(i) and function \n");
      printf("values f(i), one set on each line \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i],&f[i]);
      printf("\nInput point of interpolation \n");
      scanf("%f", &xp);
/* Compute distances between data points and function
   differences */
      for(i=2;i<=n;i++)
      {
        h[i] = x[i] - x[i-1];
        df[i] = f[i] - f[i-1];
      }
/* Initialize C matrix */
      for(i=2;i<=n-1;i++)
        for(j=2;j<=n-1;j++)
             c[i][j] = 0.0;
/* Compute diagonal elements of c */
      for(i=2;i<=n-1;i++)
        c[i][i] = 2.0 * (h[i] - h[i+1]);
/* Compute off-diagonal elements of c */
      for(i=3;i<=n-1;i++)
      {
        c[i-1][i] = h[i];
        c[i][i-1] = h[i];
      }
/* Compute elements of d array */
      for(i=2;i<=n-1;i++)
        d[i] = (df[i+1]/h[i+1] - df[i]/h[i]) * 6.0;
/*  Compute elements of a using Gaussian elimination.
    Change array subscripts from 2 to n-1 to 1 to n-2
    before calling gauss() subroutine. */
      m = n-2;
      for(i=1;i<=m;i++)
      {
        d[i] = d[i+1];
        for(j=1;j<=m;j++)
             c[i][j] = c[i+1][j+1];
      }
      gauss(m,c,d,a);

/*  Compute the coefficients of natural cubic spline */
      for(i=n-1;i>=2;i--)
        a[i] = a[i-1];
      a[1] = 0.0;
      a[n] = 0.0;

/*  Locate the domain of xp */
      for(i=2;i<=n;i++)
      {
        if(xp <= x[i])
             break;
      }

/* Compute interpolation value at xp */
      u[i-1] = xp - x[i-1];
      u[i] = xp - x[i];
      q1 = h[i]*h[i] * u[i] - u[i]*u[i]*u[i];
      q2 = u[i-1]*u[i-1]*u[i-1] - h[i]*h[i] * u[i-1];
      q3 = f[i]*u[i-1] - f[i-1]*u[i];
      fp = (a[i-1]*q1 + a[i]*q2)/(6.0 * h[i]) + q3/h[i];

/* Write results */
      printf("\n         SPLINE INTERPOLATION \n \n");
      printf("Interpolation value = %f \n", fp);
}
/* End of main() program */
/* ---------------------------------------------------- */
/*  Defining gauss() subroutine */
gauss(int n, float a[10][10], float b[10], float x[10])
/* This subroutine solves a set of n linear equations using
   Gauss elimination method */
{
      int i,j,k;
      float pivot, factor, sum;
/* ----------- Elimination begins ------------ */
      for(k=1;k<=n-1;k++)
      {
        pivot = a[k][k];
        for(i=k+1;i<=n;i++)
```

```
         {
           factor = a[i][k] / pivot;
           for(j=k+1;j<=n;j++)
              a[i][j] = a[i][j] - factor * a[k][j];
           b[i] = b[i] - factor * b[k];
         }
       }
/* Back substitution begins */
      x[n] = b[n]/a[n][n];
      for(k=n-1;k>=1;k-)
      {
        sum = 0.0;
        for(j=k+1;j<=n;j++)
             sum = sum + a[k][j] * x[j];
        x[k] = (b[k]-sum)/a[k][k];
      }
      return;
}
/*  End of subroutine gauss() */
/* ---------------------------------------------------- */
```

Program 18 LINREG

```
/*---------------------------------------------------- *
 *  Main program                                        *
 *    This program fits a line Y = A + BX  to a given  *
 *    set of data points by the method least squares   *
 *---------------------------------------------------- *
 *  Functions invoked                                   *
 *    Library function fabs()                           *
 *---------------------------------------------------- *
 *  Subroutines used                                    *
 *    NIL                                               *
 *---------------------------------------------------- *
 *  Variables used                                      *
 *    x,y - Data arrays                                 *
 *    n - Number of data sets                           *
 *    sumx - Sum of x values                            *
 *    sumy - Sum of y values                            *
 *    sumxx - Sum of squares of x values                *
 *    sumxy - Sum of products of x and y                *
 *    xmean - Mean of x values                          *
 *    ymean - Mean of y values                          *
 *    a - y intercept of the line                       *
 *    b - Slope of the line                             *
 *---------------------------------------------------- *
 *  Constants used                                          *
 *     MAX - Limit for number of data points                *
 *     EPS - Error limit                                    *
 * -------------------------------------------------------- *

#include <math.h>
#define MAX 10
#define EPS 0.000001

main( )
{
      int i,n;
      float x[10], y[10];
      float sumx, sumy, sumxx, sumxy, xmean, ymean,
            denom, a, b;

      printf("\n     LINEAR REGRESSION \n \n");

/* Reading data values */
      printf("\nInput number of data points, n \n");
      scanf("%d", &n);
      printf("\nInput x and y values \n");
      printf("One set on each line \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i], &y[i]);

/* Computing constants a and b of the linear equation */
      sumx = 0.0;
      sumy = 0.0;
      sumxx = 0.0;
      sumxy = 0.0;
      for(i=1;i<=n;i++)
      {
        sumx = sumx + x[i];
        sumy = sumy + y[i];
        sumxx = sumxx + x[i] * x[i];
        sumxy = sumxy + x[i] * y[i];
      }
      xmean = sumx/n;
      ymean = sumy/n;
      denom = n*sumxx - sumx * sumx;
      if(fabs(denom) > EPS)
      {
        b = (n*sumxy - sumx * sumy)/denom;
        a = ymean - b * xmean;
        printf("\n   LINEAR REGRESSION LINE y = a+bx \n");
        printf("\nThe coefficients are: \n");
```

```
         printf("a = %f \n", a);
         printf("b = %f \n", b);
      }
      else
      {
         printf("\n NO SOLUTION \n");
      }
}
/*  End of main() program */
/* ---------------------------------------------- */
```

Program 19 POLREG

```
/* --------------------------------------------- *
 *  Main program                                 *
 *    This program fits a polynomial curve to a given  *
 *    set of data points by the method of        *
 *    least squares                              *
 * --------------------------------------------- *
 *  Functions invoked                            *
 *    NIL                                        *
 * --------------------------------------------- *
 *  Subroutines used                             *
 *    normal, gauss                              *
 * --------------------------------------------- *
 *  Variables used                               *
 *    x,y - Arrays of data values                *
 *    n - Number of data points                  *
 *    mp - Order of the polynomial under construction  *
 *    m - Number of polynomial coefficients      *
 *    c - Coefficient matrix of normal equations *
 *    b - Right side vector of normal equations  *
 *    a - Array of coefficients of the polynomial *
 * --------------------------------------------- *
 *  Constants used                               *
 *    MAX - Maximum number of data points        *
 * --------------------------------------------- */

#include <math.h>
#define MAX 10

main( )
{
      int n,mp,m,i;
      float x[MAX],y[MAX],c[MAX][MAX],a[MAX],b[MAX];

      printf("\n     POLYNOMIAL REGRESSION \n\n");
```

```
/* Reading values */
     printf("Input number of data points n \n");
     scanf("%d",&n);
     printf("Input order of polynomial, mp required \n");
     scanf("%d",&mp);
     printf("Input data values, x and y \n");
     printf("one set on each line \n");
     for(i=1;i<=n;i++)
       scanf("%f %f", &x[i], &y[i]);
/* Testing the order */
     if(n <= mp)
     {
       printf("\n  REGRESSION IS NOT POSSIBLE \n");
       goto stop;
     }
/* Number of polynomial coefficients */
     m = mp + 1;
/* Computation of elements of c and b */
     normal(x,y,c,b,n,m);
/* Computation of coefficients a(1) to a(m) */
     gauss(m,c,b,a);
/* Printing of coefficients a(i) */
     printf("\n    POLYNOMIAL COEFFICIENTS \n\n");
     for(i=1;i<=m;i++)
       printf("%15.6f", a[i]);
     printf("\n");
     stop:
     printf("END");
}
/* End of main() program */
/* ---------------------------------------------------*/
/* Defining the subroutine normal() */
normal(float x[MAX], float y[MAX], float c[MAX][MAX],
       float b[MAX], int n, int m)
/* This subroutine computes the coefficients of normal
   equations */
{
     int i,j,k,l1,l2;
     for(j=1;j<=m;j++)
     {
       for(k=1;k<=m;k++)
```

```
        {
            c[j][k] = 0.0;
            l1 = k+j-2;
            for(i=1;i<=n;i++)
              c[j][k] = c[j][k] + pow(x[i],l1);
        }
      }
      for(j=1;j<=m;j++)
      {
        b[j] = 0.0;
        l2 = j-1;
        for(i=1;i<=n;i++)
             b[j] = b[j]+y[i] * pow(x[i],l2);
      }
      return;
}
/* End of subroutine normal() */
/* ------------------------------------------------ */
/* Defining gauss() subroutine */
gauss(int n, float a[MAX][MAX], float b[MAX], float x[MAX])
/* This subroutine solves a set of n linear equations by Gauss
   Elimination method */
{
      int i,j,k;
      float pivot, factor, sum;
/* Elimination begins */
      for(k=1;k<=n-1;k++)
      {
          pivot = a[k][k];
          for(i=k+1;i<=n;i++)
          {
             factor = a[i][k]/pivot;
             for(j=k+1;j<=n;j++)
                  a[i][j] = a[i][j] - factor * a[k][j];
             b[i] = b[i] - factor * b[k];
          }
      }
/* Back substitution begins */
      x[n] = b[n]/a[n][n];
      for(k=n-1;k>=1;k--)
      {
        sum = 0.0;
        for(j=k+1;j<=n;j++)
```

```
            sum = sum + a[k][j] * x[j];
        x[k] = (b[k]-sum)/a[k][k];
      }
      return;
}
/* End of subroutine gauss() */
/* ---------------------------------------------------------- */
```

Program 20 NUDIF

```
/* ---------------------------------------------------------- *
 * Main program                                               *
 *   This program computes the derivative of a tabulated      *
 *   function at a specified value using the Newton           *
 *   interpolation approach                                   *
 * ---------------------------------------------------------- *
 * Functions invoked                                          *
 *   NIL                                                      *
 * ---------------------------------------------------------- *
 * Subroutines used                                           *
 *   NIL                                                      *
 * ---------------------------------------------------------- *
 * Variables used                                             *
 *   n - Number of function values given                      *
 *   x - Array of values                                      *
 *   f - Array of function values                             *
 *   a - Array of coefficients of Newton polynomial           *
 *   d - Difference table                                     *
 *   xp - Desired point of differentiation                    *
 *   dif - Derivative of the function at XP                   *
 * ---------------------------------------------------------- *
 * Constants used                                             *
 *   NIL                                                      *
 * ---------------------------------------------------------- */

main( )
{
      int i,j,k,n;
      float x[10],f[10],a[10],d[10][10],xp,dif,sum,p;

/* Reading input data */
      printf("\nInput number of data points \n");
      scanf("%d", &n);
      printf("\nInput values of x and f(x), \n");
      printf("one set on each line \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i], &f[i]);
 /* Constructing difference table d */
       for(i=1;i<=n;i++)
         d[i][1] = f[i];
       for(j=2;j<=n;j++)
         for(i=1;i<=n-j+1;i++)
              d[i][j] = (d[i+1][j-1] - d[i][j-1])/(x[i+j-1] -
 x[i]);
 /* Set the coefficients of Newton interpolating polynomial */
       for(j=1;j<=n;j++)
         a[j] = d[1][j];
 /* Compute derivative at a given x = xp */
       printf("\nInput xp where derivative is required \n");
       scanf("%f", &xp);
       dif = a[2];
       for(k=3;k<=n;k++)
       {
         sum = 0.0;
         for(i=1;i<=k-1;i++)
         {
             p = 1.0;
             for(j=1;j<=k-1;j++)
             {
               if(i == j)
                   continue;
               p = p * (xp - x[j]);
           }
           sum = sum + p;
         }
        dif = dif + a[k] * sum;
      }
/* Write results */
      printf("\nNUMERICAL DIFFERENTIATION USING ");
      printf("NEWTON POLYNOMIAL \n\n");
      printf("DERIVATIVE at x = %f is %f \n", xp,dif);
}
/* End of main() program */
/* ---------------------------------------------------- */
```

Program 21 TRAPE1

```
/* ---------------------------------------------------- *
 * Main program                                         *
 *   This program integrates a given function           *
 *   using the trapezoidal rule                         *
 * ---------------------------------------------------- *
```

```
                                                                 *
 * Functions invoked                                             *
 *   NIL                                                         *
 * --------------------------------------------------------------*
 * Subroutines used                                              *
 *   F(x)                                                        *
 * --------------------------------------------------------------*
 * Variables used                                                *
 *     a - Lower limit of integration                            *
 *     b - Upper limit of integration                            *
 *     h - Segment width                                         *
 *     n - Number of segments                                    *
 *   ict - Value of integral                                     *
 * --------------------------------------------------------------*
 * Constants used                                                *
 *   NIL                                                         */
 * ---------------------------------------------------------------

#include <math.h>

Main( )
{
      int n,i;
      float a,b,h,sum,ict;
      float F(float x):

      printf("\nGive initial value of x \n");
      scanf("%f",&a);
      printf("\nGive final value of x \n");
      scanf("%f", &b);
      printf("\nWhat is the segment width? \n");
      scanf("%f", &h);

      n = (b-a)/h;

      sum = (F(a) + F(b))/2.0;
      for(i=1;i<=n-1;i++)
      {
        sum = sum + F(a+i*h);
      }

      ict = sum * h;

      printf("\n");
      printf("Integration between %f and %f \n", a,b);
      printf("When h = %f is %f \n", h, ict);
      printf("\n");
}
/* End of main() program */
/* ---------------------------------------------------------------*/
```

```
float F(float x)
{
      float f;
      f = 1.0-exp(-x/2.0);
      return (f);
}
/* End of subroutine F(x) */
/* ------------------------------------------------- */
```

Program 22 SIMS1

```
/* -------------------------------------------------- *
 * Main program                                       *
 *   This program integrates a given function         *
 *   using the Simpson's 1/3 rule                     *
 * -------------------------------------------------- *
 * Functions invoked                                  *
 *   Macro F(x)                                       *
 * -------------------------------------------------- *
 * Subroutines used                                   *
 *   NIL                                              *
 * -------------------------------------------------- *
 * Variables used                                     *
 *   a - Lower limit of integration                   *
 *   b - Upper limit of integration                   *
 *   h - Segment width                                *
 *   n - Number of segments                           *
 * ics - Value of the integral                        *
 * -------------------------------------------------- *
 * Constants used                                     *
 *   NIL                                              *
 * -------------------------------------------------- */
#include <math.h>
#define F(x) 1 - exp(-(x)/2.0)
main( )
{
      int n,m,i;
      float a,b,h,sum,ics,x,f1,f2,f3;
      printf("\nInitial value of x \n");
      scanf("%f", &a);
      printf("\nFinal value of x \n");
      scanf("%f", &b);
      printf("\nNumber of segments (EVEN number) \n");
      scanf("%d", &n);
```

```
    h = (b-a)/n;
    m = n/2;
    sum = 0.0;
    x = a;
    f1 = F(x);
    for(i=1;i<=m;i++)
    {
      f2 = F(x+h);
      f3 = F(x+2*h);
      sum = sum + f1 + 4*f2 + f3;
      f1 = f3;
      x = x + 2*h;
    }
    ics = sum * h/3.0;
    printf("\nIntegral from %f to %f \n", a,b);
    printf("When h = %f is %f \n", h, ics);
}
/* End of main() program */
/* ----------------------------------------------------- */
```

Program 23 ROMBRG

```
/* ---------------------------------------------------- *
 * Main program                                         *
 *   This program performs Romberg integration          *
 *   by bisecting the intervals N times                 *
 * ---------------------------------------------------- *
 * Functions invoked                                    *
 *   Macro F(x), Library function fabs()                *
 * ---------------------------------------------------- *
 * Subroutines used                                     *
 *   NIL                                                *
 * ---------------------------------------------------- *
 * Variables used                                       *
 *   a - Starting point of the interval                 *
 *   b - End point of the interval                      *
 *   h - Width of the interval                          *
 *   n - Number of times bisection is done              *
 *   m - Number of trapezoids                           *
 *   r - Matrix of Romberg integral values              *
 * ---------------------------------------------------- *
 * Constants used                                       *
 *   EPS - Error bound                                  *
 * ---------------------------------------------------- */
```

```
#include <math.h>
#define EPS 0.000001
#define F(x) 1.0/(x)
main( )
{
      int i,j,k,m,n;
      float a,b,h,sum,x,r[10][10];
      printf("\nInput end points of the interval \n");
      scanf("%f %f", &a,&b);
      printf("\nInput maximum number of times");
      printf("\nthe subintervals are bisected \n");
      scanf("%d", &n);
/* Compute area using entire interval as one trapezoidal */
      h = b - a;
      r[1][1] = h * (F(a)+F(b))/2.0;
      printf("\n%15.6f \n", r[1][1]);
/* Process of Romberg integration begins */
      for(i=2;i<=n+1;i++)
      {
        m = pow(2,(i-2)); /* trapezoidal for ith refinement */
        h = h/2;          /* bisect step-size */
/* Use recursive trapezoidal rule for m strips */
        sum = 0.0;
        for(k=1;k<=m;k++)
        {
          x = a + (2*k-1)*h;
          sum = sum + F(x);
        }
        r[i][1] = r[i-1][1]/2.0 + h*sum;
/* Compute Richardson's improvements */
      for(j=2;j<=i;j++)
      {
         r[i][j]=r[i][j-1]+(r[i][j-1]-r[i-1][j-1])/(pow(4,j-1)-1);
      }
/* Write results of improvements for ith refinement */
      for(j=1;j<=i;j++)
          printf("%15.6f", r[i][j]);
      printf("\n");
/* Test for accuracy */
      if(fabs(r[i-1][i-1] - r[i][i]) < EPS)
      /* Stop further refinement */
      {
            printf("\n");
            printf("ROMBERG INTEGRATION = %f \n", r[i][i]);
            printf("\n");
            goto stop;
          }
          else
          {
            continue;
          }
        }
/* Write final result */
      printf("\nROMBERG INTEGRATION = %f \n", r[n+1][n+1]);
      printf("(Normal exit from loop) \n");

      stop:
      printf("End");
}
/* End of main() program */
/* ---------------------------------------------------------*/
```

Program 24 TRAPE2

```
/* ------------------------------------------------------- *
 * Main program                                             *
 *   This program integrates a tabulated function          *
 *   using the trapezoidal rule                             *
 * ------------------------------------------------------- *
 * Functions invoked                                        *
 *   Library function fabs()                                *
 * ------------------------------------------------------- *
 * Subroutines used                                         *
 *   NIL                                                    *
 * ------------------------------------------------------- *
 * Variables used                                           *
 *     n - Number of table points                           *
 *     x - Array of independent data points                 *
 *     y - Array of function values                         *
 *     a - Lower limit of integration                       *
 *     b - Upper limit of integration                       *
 *     h - Distance between points                          *
 *    n1 - Position of A in the table                       *
 *    n2 - Position of B in the table                       *
 *   ict - Value of integral                                *
 * ------------------------------------------------------- *
 * Constants used                                           *
 *   NIL                                                    *
 * ------------------------------------------------------- */
```

```
#include <math.h>
#define MAX 15
main( )
{
      int n,n1,n2,i;
      float  a,b,h,sum,ict,x[MAX],y[MAX];
/* Reading table values */
      printf("Number of data points \n");
      scanf("%d", &n);
      printf("\nInput table values, set by set \n");
      for(i=1;i<=n;i++)
      scanf("%f %f", &x[i],&y[i]);
/* Reading the limits of integration */
      printf("Initial value of x \n");
      scanf("%f",&a);
      printf("Final value of x \n");
      scanf("%f", &b);
      printf("\nWhat is segment width? \n");
      scanf("%f", &h);
/* Computing the position of initial and final values */
      n1 = (int)(fabs(a-x[1])/h+1.5);
      n2 = (int)(fabs(b-x[1])/h+1.5);
/* Evaluating the integral */
      sum = 0.0;
      for(i=n1;i<=n2-1;i++)
        sum = sum + y[i] + y[i+1];
      ict = sum * h/2.0;
      printf("\nIntegral from %f to %f is %f\n", a,b,ict);
}
/*  End of main() program */
/* -------------------------------------------------- * /
```

Program 25 SIMS2

```
/ * ------------------------------------------------ *
 *  Main program                                      *
 *    This program integrates a tabulated function    *
 *    using the Simpson's 1/3 rule.If the number of   *
 *    segments is odd, the trapezoidal rule is used   *
 *    for the last segment.                           *
 * -------------------------------------------------- *
 *  Functions invoked                                 *
 *    Library function fabs()                         *
 * -------------------------------------------------- *
 *  Subroutines used                                          *
 *     NIL                                                    *
 * ---------------------------------------------------------- *
 *  Variables used                                            *
 *      n - Number of table points                            *
 *      x - Array of independent values                       *
 *      y - Array of function values                          *
 *      a - Lower limit of integration                        *
 *      b - Upper limit of integration                        *
 *      h - Distance between independent values               *
 *     n1 - Position of A in the table                        *
 *     n2 - Position of B in the table                        *
 *     i1 - Area computed using Simpson's rule                *
 *     i2 - Area of the last segment (by trapezoidal rule)    *
 *    ics - Value of integral                                 *
 * ---------------------------------------------------------- *
 *  Constants used                                            *
 *     NIL                                                    *
 * ---------------------------------------------------------- */

#include <math.h>
main( )
{
      int  i,n,n1,n2,m,l;
      float  x[15],y[15],a,b,h,sum,ics,i1,i2;

/* Reading table values */
      printf("Input number of data points \n");
      scanf("\%d", &n);
      printf("\nInput table values, set by set \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i],&y[i]);

/* Reading the limits of integration */
      printf("Input initial value of x \n");
      scanf("%f", &a);
      printf("Input final value of x \n");
      scanf("%f", &b);
      printf("What is segment width? \n");
      scanf("%f", &h);

/* Computing the position of initial and final values */
      n1 = (int)(fabs(a-x[1])/h+1.5);
      n2 = (int)(fabs(b-x[1])/h+1.5);

/* Testing for even intervals */
      m = n2 - n1;
      if(m%2 == 0) /* m is even */
      {
        i2 = 0.0;
        l = n2 - 2;
      }
      else /* m is odd */
      {
      /* Use trapezoidal rule for the last strip */
        i2 = (y[n2-1]+y[n2])*h/2.0;
        l = n2 - 3;
      }
/* Use Simpson's rule for l strips */
      sum = 0.0;
      for(i=n1;i<=l;i=i+2)
        sum = sum + y[i] + 4*y[i+1] + y[i+2];
      i1 = sum*h/3.0;
/* Integral is sum of i1 and i2 */
      ics = i1 + i2;
/* Writing the results */
      printf("\nIntegral from %f to %f is %f \n", a,b,ics);
}
/* End of main() program */
/* ---------------------------------------------- */
```

Program 26 EULER

```
/* ---------------------------------------------- *
 * Main program                                   *
 *   This program  estimates  the solution of the first *
 *   order differential equation y' = f(x,y) at a given *
 *   point using Euler's method                   *
 * ---------------------------------------------- *
 * Functions invoked                              *
 *   NIL                                          *
 * ---------------------------------------------- *
 * Subroutines used                               *
 *   func()                                       *
 * ---------------------------------------------- *
 * Variables used                                 *
 *      x - Initial value of independent variable *
 *      y - Initial value of dependent variable   *
 *     xp - Point of solution                     *
 *      h - Incremental step-size                 *
 *      n - Number of computational steps required *
 *     dy - Incremental Y in each step            *
 * ---------------------------------------------- *
 * Constants used                                    *
 *    NIL                                            *
 * ------------------------------------------------- */
main( )
{
      int i,n;
      float x,y,xp,h,dy;
      float func(float, float);

      printf("\n      SOLUTION BY EULER'S METHOD \n \n");

/* Reading initial data                              */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);

/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);

/* Compute y recursively at each step */
      for(i=1;i<=n;i++)
      {
        dy = h * func(x,y);
        x = x + h;
        y = y + dy;
        printf("%5d %10.6f %10.6f \n", i,x,y);
      }

/* Write the final result */
      printf("\nValue of y at x = %f is %f \n", x,y);
}
/*  End of main() program */

/* ------------------------------------------------- */

/* Defining subroutine func() */

float func(float x, float y)

{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/*  End of subroutine func() */

/* ------------------------------------------------- */
```

Program 27 HEUN

```
/* ----------------------------------------------------- *
 *  Main program                                         *
 *    This program solves the first order differential  *
 *    equation y' = f(x,y) using the Heun's method      *
 * ----------------------------------------------------- *
 *  Functions invoked                                    *
 *    NIL                                                *
 * ----------------------------------------------------- *
 *  Subroutines used                                     *
 *    func()                                             *
 * ----------------------------------------------------- *
 *  Variables used                                       *
 *    x  - Initial value of independent variable        *
 *    y  - Initial value of dependent variable          *
 *    xp - Point of solution                             *
 *    h  - Step-size                                     *
 *    n  - Number of steps                               *
 * ----------------------------------------------------- *
 *  Constants used                                       *
 *    NIL                                                *
 * ----------------------------------------------------- */

main( )
{
      int i,n;
      float x,y,xp,h,m1,m2;
      float func(float, float);

      printf("\n   SOLUTION BY HEUN'S METHOD \n \n");

/* Reading initial data */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);

/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);

/* Compute y recursively at each step */
      for(i=1;i<=n;i++)
      {
        m1 = func(x,y);
        m2 = func(x+h,y+m1*h);
        x = x + h;
        y = y + 0.5*h*(m1+m2);
        printf("%5d %10.6f %10.6f \n", i,x,y);
      }
/* Write the final result */
      printf("\nValue of y at x = %f is %f \n", x,y);
}
/* End of main() program */
/* --------------------------------------------------- */
/* Defining subroutine func() */
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/* End of subroutine func() */
/* --------------------------------------------------- */
```

Program 28 POLYGN

```
/* --------------------------------------------------- *
 *                                                      *
 * Main program                                         *
 *   This program solves the differential equation     *
 *   of type y' = f(x,y) by polygon method             *
 * --------------------------------------------------- *
 *                                                      *
 * Functions invoked                                    *
 *   NIL                                                *
 * --------------------------------------------------- *
 *                                                      *
 * Subroutines used                                     *
 *   func()                                             *
 * --------------------------------------------------- *
 *                                                      *
 * Variables used                                       *
 *   x - Initial value of the independent variable     *
 *   y - Initial value of the dependent variable       *
 *   xp - Point of solution                             *
 *   h - Incremental step-size                          *
 *   n - Number of computational steps required        *
 * --------------------------------------------------- *
 *                                                      *
 * Constants used                                       *
 *   NIL                                                *
 * --------------------------------------------------- */

main( )
{
      int i,n;
      float x,y,xp,h,m1,m2;
      float func(float, float);
      printf("\n  SOLUTION BY POLYGON METHOD \n \n");
/* Reading initial data */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);
/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);
/* Compute y recursively at each step */
      for(i=1;i<=n;i++)
      {
        m1 = func(x,y);
        m2 = func(x+0.5*h,y+0.5*h*m1);
        x = x + h;
        y = y + m2*h;
        printf("%5d %10.6f %10.6f \n", i,x,y);
      }
/* Write the final result */
      printf("\nValue of y at x = %f is %f \n", x,y);
}
/*  End of main() program */
/* ---------------------------------------------- */
/*  Defining subroutine func() */
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/*  End of subroutine func() */
/* ---------------------------------------------- */
```

Program 29 RUNGE4

```
/* ---------------------------------------------- *
  *  Main program                                  *
  *    This program computes the solution of first order  *
  *    differential equation of type  y' = f(x,y)  using  *
  *    the 4th order Runge-Kutta method            *
  * ---------------------------------------------- *
```

```
 *  Functions invoked                                          *
 *    Nil                                                      *
 * ----------------------------------------------------------- *
 *  Subroutines used                                           *
 *    func()                                                   *
 * ----------------------------------------------------------- *
 *  Variables used                                             *
 *      x - Initial value of independent variable              *
 *      y - Initial value of dependent variable                *
 *     xp - Point of solution                                  *
 *      h - Step-size                                          *
 *      n - Number of steps                                    *
 * ----------------------------------------------------------- *
 * Constants used                                              *
 *    NIL                                                      *
 * ----------------------------------------------------------- */

main( )
{
      int i,n;
      float x,y,xp,h,m1,m2,m3,m4;
      float func(float, float);

      printf("\n   SOLUTION BY 4th ORDER RK METHOD\n\n");

/* Input initial data */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);

/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);

/* Compute y at each step */
      printf("\n");
      printf(" ----------------------------------\n");
      printf("   STEP       X          Y         \n");
      printf(" ----------------------------------\n");
      for(i=1;i<=n;i++)
      {
         m1 = func(x,y);
         m2 = func(x+0.5*h, y+0.5*m1*h);
         m3 = func(x+0.5*h, y+0.5*m2*h);
         m4 = func(x+h, y+m3*h);
         x = x+h;
         y = y + (m1+2.0*m2 + 2.0*m3 + m4) * h/6.0;
       printf("%5d %15.6f %15.6f \n", i,x,y);
      }
      printf(" ------------------------------ \n");
/* Write the final value of y */
      printf("\nValue of y at x = %f is %f \n", x,y);

}
/*  End of main() program */
/* ------------------------------------------------- */
/* Defining subroutine func() */
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/*  End of subroutine func() */
* ------------------------------------------------- *
```

Program 30 MILSIM

```
/* ------------------------------------------------ *
 *  Main program                                     *
 *    This program solves the first order differential *
 *    equation y' = f(x,y) using Milne-Simpson method *
 * ------------------------------------------------ *
 *  Functions invoked                                *
 *    NIL                                             *
 * ------------------------------------------------ *
 *  Subroutines used                                 *
 *    func()                                         *
 * ------------------------------------------------ *
 *  Variables used                                   *
 *    x(1) - Initial value of independent variable   *
 *    y(1) - Initial value of dependent variable     *
 *      xp - Point of solution                       *
 *       n - Number of steps                         *
 *       h - Step-size                               *
 *       x - Array of independent variable           *
 *       y - Array of dependent variable             *
 * ------------------------------------------------ *
 *  Constants used                                   *
 *    NIL                                            *
 * ------------------------------------------------ *
```

```
main( )
{
      int i,n;
      float h,x[15],y[15],xp,m1,m2,m3,m4,f2,f3,f4,f5;
      float func(float, float);
/* Reading initial values */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x[1],&y[1]);
      printf("Input value of x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);
/* Compute number of computation required */
      n = (int)((xp-x[1])/h+0.5);
/* We require four starting points for Milne-Simpson
method.
Initial
    values form the first point. Remaining three points are
obtained
    using 4th order RK method */
      printf("\nINITIAL VALUES ARE %10.6f %10.6f", x[1],y[1]);
/* Computing three more points by RK method */
      printf("\n\nTHREE VALUES BY RK METHOD \n");
      for(i=1;i<=3;i++)
      {
        m1 = func(x[i],y[i]);
        m2 = func(x[i]+0.5*h, y[i]+0.5*m1*h);
        m3 = func(x[i]+0.5*h, y[i]+0.5*m2*h);
        m4 = func(x[i]+h, y[i]+m3*h);
        x[i+1] = x[i] + h;
        y[i+1] = y[i] + (m1+2.0*m2 + 2.0*m3 + m4)*h/6.0;
        printf("\n%5d %10.6f %10.6f", i, x[i+1],y[i+1]);
      }
/* Computing values by Milne-Simpson method */
      printf("\n\nVALUES OBTAINED BY MILNE-SIMPSON METHOD \n");
      for(i=4;i<=n;i++)
      {
        f2 = func(x[i-2], y[i-2]);
        f3 = func(x[i-1], y[i-1]);
        f4 = func(x[i], y[i]);
        /* Predicted value of y (by Milne's formula)*/
           y[i+1] = y[i-3] + 4.0*h/3.0 * (2.0*f2-f3+2.0*f4);
        x[i+1] = x[i] + h;
        f5 = func(x[i+1], y[i+1]);
        /* Corrected value of y (by Simpson's formula) */
        y[i+1] = y[i-1]+h/3.0 * (f3+4.0*f4+f5);
        printf("\n%5d %10.6f %10.6f", i, x[i+1], y[i+1]);
      }
      printf("\n\nValue of y at x = %f is %f", x[n+1], y[n+1]);
}
/*  End of main() program */
/* ------------------------------------------------ */
/* Defining subroutine func()*/
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/* End of subroutine func() */
/* ------------------------------------------------ */
```

APPENDIX E

Bibliography

E.1 NUMERICAL COMPUTING

1. Antia, H M, *Numerical Methods for Scientists and Engineers*, Tata McGraw-Hill Publishing Company, New Delhi, 1991.
2. Atkinson, L V, P J Harley and J D Hudson, *Numerical Methods with FORTRAN 77*, Addison-Wesley Publishing Company, 1989.
3. Buchanan, J L and P R Turner, *Numerical Methods and Analysis*, McGraw-Hill Book Company, 1992.
4. Chapra, S C and R P Canale, *Numerical Methods for Engineers*, 2nd Edition, McGraw-Hill Book Company, 1989.
5. Constantinides, A, *Applied Numerical Methods with Personal Computers*, McGraw-Hill Book Company, 1987.
6. Conte, S D and Carl de Boor, *Elementary Numerical Analysis*, 3rd Edition, McGraw-Hill Book Company, 1981.
7. Gerald, C F and P O Wheatly, *Applied Numerical Analysis*, 5th Edition, Addison-Wesley Publishing Company, 1994.
8. Griffiths, D V and I M Smith, *Numerical Methods for Engineers*, Oxford University Press, 1991.
9. Ledermann, W (Chief Editor), *Handbook of Applied Mathematics* (Volume III), John Wiley & Sons, 1981.
10. Mathews, J H, *Numerical Methods for Mathematics, Science and Engineering*, 2nd Edition, Prentice-Hall of India, 1994.
11. Scheid, F, *Numerical Analysis* (Schaum's Series), McGraw-Hill Publishing Company, 1990.
12. Yakowitz, S and F Szidarovszky, *An Introduction to Numerical Computation*, 2nd Edition, Macmillan Publishing Company, 1990.

E.2 PROGRAMMING

1. Balagurusamy, E, *FORTRAN for Beginners*, Tata McGraw-Hill Publishing Company, New Delhi, 1985.
2. Balagurusamy, E, *Programming in ANSI C*, 2nd Edition, Tata McGraw-Hill Publishing Company, New Delhi, 1989.
3. Chamberland, L, *FORTRAN 90-A Reference Guide*, Prentice-Hall, 1995.
4. Etter, D M, *Structured FORTRAN 77 for Engineers and Scientists*, 4th Edition, Benjamin/Cummings Publishing Company, 1993.
5. Kochan, S G, *Programming in C*, Hayden Book Company, 1983.
6. McCracken, D D and W I Salman, *Computing for Engineers and Scientists with FORTRAN 77*, 2nd Edition, John Wiley & Sons, 1988.
7. Nyhoff, L and S Leestma, *FORTRAN 77 and Numerical Methods for Engineers and Scientists*, Prentice-Hall, 1995.

Index