

CHAPTER

6

Roots of Nonlinear Equations

INTRODUCTION

Mathematical models for a wide variety of problems in science and engineering can be formulated into equations of the form

$$f(x) = 0 \quad (6.1)$$

where x and $f(x)$ may be real, complex, or vector quantities. The solution process often involves finding the values of x that would satisfy the Eq. (6.1). These values are called the *roots* of the equation. Since the function $f(x)$ becomes zero at these values, they are also known as the zeros of the function $f(x)$.

Equation (6.1) may belong to one of the following types of equations:

1. Algebraic equations ✓
2. Polynomial equations ✓
3. Transcendental equations ✓

(Any function of one variable which does not graph as a straight line in two dimensions, or any function of two variables which does not graph as a plane in three dimensions, can be said to be *nonlinear*.) Consider the function

$$y = f(x)$$

$f(x)$ is a *linear* function, if the dependent variable y changes in direct proportion to the change in independent variable x . For example

$$y = 3x + 5$$

is a *linear* function.

On the other hand, $f(x)$ is said to be nonlinear, if the response of the dependent variable y is not in direct or exact proportion to the changes in the independent variable x . For example

$$y = x^2 + 1$$

is a nonlinear function.

There are many situations in science and engineering where the relationship between variables is *nonlinear*.

Algebraic Equations

An equation of type $y = f(x)$ is said to be *algebraic* if it can be expressed in the form

$$f_n y_n + f_{n-1} y_{n-1} + \dots + f_1 y_1 + f_0 = 0 \quad (6.2)$$

where f_i is an i th order polynomial in x . Equation (6.2) can be thought of as having a general form

$$f(x, y) = 0 \quad (6.3)$$

This implies that Eq. (6.3) portrays a dependence between the variables x and y . Some examples are:

1. $3x + 5y - 21 = 0$ (linear)
2. $2x + 3xy - 25 = 0$ (non-linear)
3. $x^3 - xy - 3y^3 = 0$ (non-linear)

These equations have an infinite number of pairs of values of x and y which satisfy them.

Polynomial Equations

Polynomial equations are a simple class of algebraic equations that are represented as follows:

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0 \quad (6.4)$$

This is called n^{th} degree polynomial and has n roots. The roots may be

1. real and different
2. real and repeated
3. complex numbers

Since complex roots appear in pairs, if n is odd, then the polynomial has at least one real root. For example, a cubic equation of the type

$$a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0$$

will have at least one real root and the remaining two may be real or complex roots. Some specific examples of polynomial equations are:

1. $5x^5 - x^3 + 3x^2 = 0$
2. $x^3 - 4x^2 + x + 6 = 0$
3. $x^2 - 4x + 4 = 0$

Transcendental Equations

(A non-algebraic equation is called a *transcendental equation*.) These include trigonometric, exponential and logarithmic functions. Examples of transcendental equation are:

1. $2 \sin x - x = 0$
2. $e^x \sin x - 1/2 x = 0$
3. $\log x^2 - 1 = 0$
4. $x - e^{1/x} = 0$

A transcendental equation may have a finite or an infinite number of real roots or may not have real root at all.

6.2 METHODS OF SOLUTION

There are a number of ways to find the roots of nonlinear equations such as those described in Section 6.1. They include:

1. Direct analytical methods
2. Graphical methods
3. Trial and error methods
4. Iterative methods

In certain cases, roots can be found by using *direct analytical methods*. For example, consider a quadratic equation such as

$$ax^2 + bx + c = 0 \quad (6.5)$$

We know that the solution of this equation is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (6.6)$$

Equation (6.6) gives the two roots of equation (6.5). However, there are equations that cannot be solved by analytical methods. For example, the simple transcendental equation

$$2 \sin x - x = 0$$

cannot be solved analytically. Direct methods for solving non-linear equations do not exist except for certain simple cases.

Graphical methods are useful when we are satisfied with approximate solution for a problem. This method involves plotting the given function and determining the points where it crosses the x -axis. These points represent approximate values of the roots of the function.

Another approach to obtain approximate solution is the trial and error technique. This method involves a series of guesses for x , each time evaluating the function to see whether it is close to zero. The value of x that causes the function value closer to zero is one of the approximate roots of the equation.

Although graphical and trial and error methods provide satisfactory approximations for many problem situations, they become cumbersome and time consuming. Moreover, the accuracy of the results are inadequate for the requirements of many engineering and scientific problems. With the advent of computers, algorithmic approaches known as *iterative methods* have become popular. An iterative technique usually begins

with an approximate value of the root, known as the *initial guess*, which is then successively corrected iteration by iteration. The process of iteration stops when the desired level of accuracy is obtained. Since iterative methods involve a large number of iterations and arithmetic operations to reach a solution, the use of computers has become inevitable to make the task simple and efficient.

In this chapter, we shall discuss a few iterative methods of solution that are commonly used. These methods are designed to determine the value of a single real root using some initial guess values. Later in the chapter, we shall also discuss methods to determine all the roots of a polynomial. Finally, we shall discuss the solution of a system of non-linear equations.



ITERATIVE METHODS

There are a number of iterative methods that have been tried and used successfully in various problem situations. All these methods typically generate a sequence of estimates of the solution which is expected to converge to the true solution. As mentioned earlier, all iterative methods begin their process of solution with one or more guesses at the solution being sought. Iterative methods, based on the number of guesses they use, can be grouped into two categories:

1. Bracketing methods
2. Open end methods

Bracketing methods (also known as *interpolation* methods) start with two initial guesses that 'bracket' the root and then systematically reduce the width of the bracket until the solution is reached. Two popular methods under this category are:

- 1. Bisection method
- 2. False position method

These methods are based on the assumption that the function changes sign in the vicinity of a root.

Open end methods (also known as *extrapolation* methods) use a single starting value or two values that do not necessarily bracket the root. The following iterative methods fall under this category:

- 1. Newton-Raphson method
- 2. Secant method
- 3. Muller's method
- 4. Fixed-point method
- 5. Bairstow's method

It may be noted that the bracketing methods require to find sign changes in the function during every iteration. Open end methods do not require this.

6.4 STARTING AND STOPPING AN ITERATIVE PROCESS

Starting the Process

Before an iterative process is initiated, we have to determine either an approximate value of root or a "search" interval that contains a root. One simple method of guessing starting points is to plot the curve of $f(x)$ and to identify a search interval near the root of interest. Graphical representation of a function cannot only provide us rough estimates of the roots, but also help us in understanding the properties of the function, thereby identifying possible problems in numerical computing. A plot of

$$f(x) = x^3 - x - 1$$

is shown in Fig. 6.1. Although $f(x)$ is a cubic function, it intersects the x -axis at only one point. This suggests that the remaining two roots are imaginary ones.

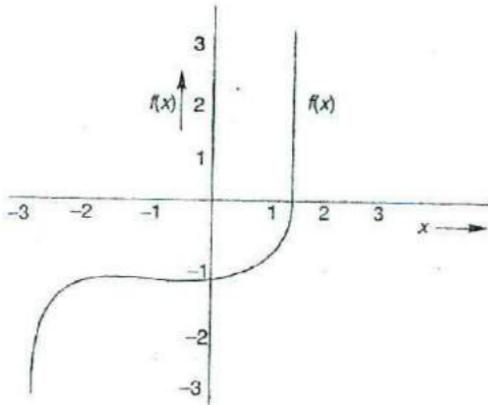


Fig. 6.1 Plot of $f(x) = x^3 - x - 1$

In the case of polynomials, many theoretical relationships between roots and coefficients are available. A few relations that might be useful for making initial guesses are described here.

Largest Possible Root For a polynomial represented by

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (6.7)$$

the largest possible root is given by

$$x_1^* = -\frac{a_{n-1}}{a_n} \quad (6.8)$$

This value is taken as the initial approximation when no other value is suggested by the knowledge of the problem at hand.

Search Bracket Another relationship that might be useful for determining the search intervals that contain the real roots of a polynomial is

$$|x^*| \leq \sqrt{\left(\frac{a_{n-1}}{a_n}\right)^2 - 2\left(\frac{a_{n-2}}{a_n}\right)} \quad (6.9)$$

where x is the root of the polynomial. Then, the maximum absolute value of the root is

$$|x_{\max}^*| = \sqrt{\left(\frac{a_{n-1}}{a_n}\right)^2 - 2\left(\frac{a_{n-2}}{a_n}\right) - 3\left(\frac{a_{n-3}}{a_n}\right)} \quad (6.10)$$

This means that no root exceeds x_{\max} in absolute magnitude and thus, all real roots lie within the interval $(-|x_{\max}^*|, |x_{\max}^*|)$.

There is yet another relationship that suggests an interval for roots. All real roots x satisfy the inequality

$$|x^*| \leq 1 + \frac{1}{|a_n|} \max \{|a_0|, |a_1|, \dots, |a_{n-1}|\} \quad (6.11)$$

where the "max" denotes the maximum of the absolute values $|a_0|, |a_1|, \dots, |a_{n-1}|$.

Example 6.1

Consider the polynomial equation

$$2x^3 - 8x^2 + 2x + 12 = 0$$

Estimate the possible initial guess values.

The largest possible root is

$$x_1^* = -\frac{-8}{2} = 4$$

That is, no root can be larger than the value 4.

All roots must satisfy the relation

$$|x^*| \leq \sqrt{\left(\frac{-8}{2}\right)^2 - 2\left(\frac{2}{2}\right)} = \sqrt{14}$$

Therefore, all real roots lie in the interval $(-\sqrt{14}, \sqrt{14})$. We can use these two points as initial guesses for the bracketing methods and one of them for the open end methods.

Stopping Criterion

An iterative process must be terminated at some stage. When? We must have an objective criterion for deciding when to stop the process. We may use one (or combination) of the following tests, depending on the behaviour of the function, to terminate the process:

1. $|x_{i+1} - x_i| \leq E_a$ (absolute error in x)
2. $\left| \frac{x_{i+1} - x_i}{x_{i+1}} \right| \leq E_r$ (relative error in x), $x \neq 0$
3. $|f(x_{i+1})| \leq E$ (value of function at root)
4. $|f(x_{i+1}) - f(x_i)| \leq E$ (difference in function values)
5. $|f(x)| \leq F_{\max}$ (large function value)
6. $|x_i| \leq XL$ (large value of x)

Here, x_i represents the estimate of the root at i th iteration and $f(x_i)$ is the value of the function at x_i .

There may be situations where these tests may fail when used alone. Sometimes even a combination of two tests may fail. A practical convergence test should use a combination of these tests. In cases where we do not know whether the process converges or not, we must have a limit on the number of iterations, like

Iterations $\geq N$ (limit on iterations).

6.5

EVALUATION OF POLYNOMIALS

All iterative methods require the evaluation of functions for which solution is sought. Since it is a recurring task, the design of an efficient algorithm for evaluating the function assumes a greater importance. While it is not possible to propose a general algorithm for evaluating transcendental functions, it is quite simple to design an algorithm for evaluating polynomials.

The polynomial is a sum of $n+1$ terms and can be expressed as

$$f(x) = \sum_{i=0}^n a_i x^i = a_0 + \sum_{i=1}^n a_i x^i \quad (6.12)$$

This can be easily implemented using a DO loop in FORTRAN. This would require $n(n + 1)/2$ multiplications and n additions.

Example 6.2

Write a FORTRAN program segment to implement Eq. (6.12).

```

SUM = AO
DO 100 I = 1, N
    SUM = SUM + A(I) * X ** I
100 CONTINUE

```

Let us consider the evaluation of a polynomial using *Horner's Rule* as follows:

$$f(x) = (((\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0) \quad (6.13)$$

Here, the innermost expression $a_n x + a_{n-1}$ is evaluated first. The resulting value constitutes a multiplicand for the expression at the next level. The number of levels equals n , the degree of polynomial. Note that this approach needs a total of n additions and n multiplications.

Horner's rule, also known as *nested multiplication*, is implemented using Algorithm 6.1. The quantities p_n , p_{n-1} , ..., p_0 are evaluated recursively. The final quantity p_0 gives the value of the function $f(x)$.

Horner's Rule

$$p_n = a_n$$

$$p_{n-1} = p_n x + a_{n-1}$$

...

...

$$p_j = p_{j+1} x + a_j$$

...

...

$$p_1 = p_2 x + a_1$$

$$f(x) = p_0 = p_1 x + a_0$$

Algorithm 6.1

Example 05

Evaluate the polynomial

$$f(x) = x^3 - 4x^2 + x + 6$$

using Horner's rule at $x = 2$.

$n = 3$, $a_3 = 1$, $a_2 = -4$, $a_1 = 1$, and $a_0 = 6$

$$p_3 = a_3 = 1$$

$$p_2 = 1 \times 2 + (-4) = -2$$

$$p_1 = (-2) \times 2 + 1 = -3$$

$$p_0 = (-3) \times 2 + 6 = 0$$

$$f(2) = 0$$

Program POLY

Program POLY shows a FORTRAN program to evaluate a polynomial of degree n using Horner's rule. This program uses a subroutine HORNER to implement Horner's algorithm.

It is an interactive program and, therefore, requests input for degree of polynomial (n), polynomial coefficients (a_i) and value of x from the user at the time of execution. Output of a sample run of the program POLY is given at the end of the program:

```

*-----*
*      PROGRAM POLY
*-----*
*-----*
* Main program
*      Program POLY evaluates a polynomial of degree n at
*      any point X using Horner's rule
*-----*
*-----*
* Functions invoked
*      NIL
*-----*
* 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
*-----*
*-----*
      INTEGER N
      REAL A, X, P

```

```

EXTERNAL HORNER
DIMENSION A(10)
WRITE(*,*) 'Input degree of polynomial, N'
READ(*,*) N
WRITE(*,*) 'Input polynomial coefficients ( A(0)
              to A(N))'
DO 100 I = 1, N+1
    READ(*,*) A(I)
100 CONTINUE
WRITE(*,*) 'Input value of X (point of evaluation)'
READ(*,*) X
* Evaluating polynomial at X using Horner's method
CALL HORNER( N,A,X,P )
* Writing the result
    WRITE(*,*) P
    STOP
END
* ----- End of main program POLY -----
* -----
* SUBROUTINE HORNER( N,A,X,P )
* -----
* Subroutine
*   HORNER computes the value of a polynomial of order
*   n at any given point x.
* -----
* Arguments
* Input
*   N - Degree of polynomial
*   A - Polynomial coefficients (array of size N+1)
*   X - Point of interest of evaluation
* Output
*   P - Value of polynomial at X
* -----
* Local Variables
*   NIL
* -----
* Functions invoked
*   NIL
* -----
* Subroutines called
*   NIL
* -----

```

POO

```

REAL A, X, P
INTEGER N
DIMENSION A(10)
P = A(N+1)
DO 111 I = N, 1, -1
    P = P*X + A(I)
111 CONTINUE
RETURN
END

```

* ----- End of Subroutine HORNER ----- *

Test Run Results

Input degree of polynomial, N

3

Input polynomial coefficients (A(0) to A(N))

12

5

6

2

Input value of X (point of evaluation)

0.125

F(X) = 12.7226600 at X = 1.250000E - 001

The polynomial used for evaluation is

$$2x^3 + 6x^2 + 5x + 12$$

and the coefficients are represented by A(1), A(2), A(3), and A(4) instead of A(0), A(1), A(2), and A(3) in the program.

MT 6.6 BISECTION METHOD

The *bisection method* is one of the simplest and most reliable of iterative methods for the solution of nonlinear equations. This method, also known as *binary chopping* or *half-interval* method, relies on the fact that if $f(x)$ is real and continuous in the interval $a < x < b$, and $f(a)$ and $f(b)$ are of opposite signs, that is,

$$\underline{f(a)f(b) < 0}$$

then there is at least one real root in the interval between a and b . (There may be more than one root in the interval).

Let $x_1 = a$ and $x_2 = b$. Let us also define another point x_0 to be the midpoint between a and b . That is,

$$\underline{x_0 = \frac{x_1 + x_2}{2}} \quad \checkmark \quad (6.14)$$

Now, there exists the following three conditions:

1. if $f(x_0) = 0$, we have a root at x_0 .

2. if $f(x_0)f(x_1) < 0$, there is a root between x_0 and x_1 .
3. if $f(x_0)f(x_2) < 0$, there is a root between x_0 and x_2 .

It follows that by testing the sign of the function at midpoint, we can deduce which part of the interval contains the root. This is illustrated in Fig. 6.2. It shows that, since $f(x_0)$ and $f(x_2)$ are of opposite sign, a root lies between x_0 and x_2 . We can further divide this subinterval into two halves to locate a new subinterval containing the root. This process can be repeated until the interval containing the root is as small as we desire.

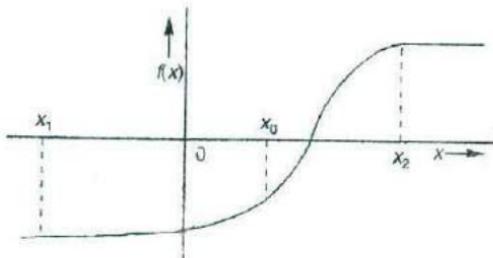


Fig. 6.2 Illustration of bisection method

Example 6.4

Find a root of the equation

using bisection method.

$$x^2 - 4x - 10 = 0$$

$$1 + 4 - 10 = -5$$

The first step is to guess two initial values that would bracket a root. Using Eq. (6.10), we can decide the maximum absolute of the solution. Thus

$$x_{\max} = \sqrt{\left(\frac{-4}{1}\right)^2 - 2\left(\frac{-10}{1}\right)} = 6$$

Therefore, we have both the roots in the interval $(-6, 6)$. The table below gives the values of $f(x)$ between -6 and 6 and shows that there is a root in the interval $(-2, -1)$ and another in $(5, 6)$.

x	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
$f(x)$	50	35	22	11	2	-5	-10	-13	-14	-13	-10	-5	2

Let us take $x_1 = -2$ and $x_2 = -1$.

Then

$$x_0 = \frac{-2 - 1}{2} = -1.5$$

$$f(-2) = 2 \text{ and } f(-1.5) = -1.75$$

Since $f(-2)f(-1.5) < 0$, the root must be in the interval $(-2, -1.5)$. The next step begins.

$$x_1 = -2, x_2 = -1.5 \text{ and } x_0 = \cancel{-1.75} / \\ f(-1.75) = 0.0625$$

Since $f(-1.75)$ and $f(-1.5)$ are of opposite sign, the root lies in the interval $(-1.75, -1.5)$. Another iteration begins.

$$x_1 = -1.75, x_2 = -1.5 \text{ and } x_0 = \cancel{-1.625} / \\ f(-1.625) = -0.859$$

Now, the root lies in the interval $(-1.75, -1.625)$

$$x_0 = \cancel{-1.6875} \\ f(-1.6875) = -0.40$$

Next $x_0 = -\frac{1.75 + 1.6875}{2} = -1.72$

$$f(-1.72) = \cancel{-0.1616}$$

Next $x_0 = -\frac{1.75 + 1.72}{2} = -1.735$

$$f(-1.735) = -0.05$$

Next $x_0 = -1.7425$

$$f(-1.7425) = +0.0063$$

The root lies between -1.735 and -1.7425 .

Approximate root is $\underline{\underline{-1.7416}}$.

An algorithm to achieve this is given in Algorithm 6.2.

Bisection Method

1. Decide initial values for x_1 and x_2 and stopping criterion, E .
2. Compute $f_1 = f(x_1)$ and $f_2 = f(x_2)$.
3. If $f_1 \times f_2 > 0$, x_1 and x_2 do not bracket any root and go to step 7;
Otherwise continue.
4. Compute $x_0 = (x_1 + x_2)/2$ and compute $f_0 = f(x_0)$
5. If $f_1 \times f_0 < 0$ then
 - set $x_2 = x_0$
 - else
 - set $x_1 = x_0$
 - set $f_1 = f_0$
6. If absolute value of $(x_2 - x_1)/x_2$ is less than error E , then
 - root = $(x_1 + x_2)/2$
 - write the value of root
 - go to step 7
 - else
 - go to step 4
7. Stop.

Algorithm 6.2

Convergence of Bisection Method

In the bisection method, we choose a midpoint x_0 in the interval between x_1 and x_2 . Depending on the sign of functions $f(x_0)$, $f(x_1)$, and $f(x_2)$, x_1 or x_2 is set equal to x_0 such that the new interval contains the root. In either case, the interval containing the root is reduced by a factor of 2. The same procedure is repeated for the new interval. If the procedure is repeated n times, then the interval containing the root is reduced to the size

$$\frac{x_2 - x_1}{2^n} = \frac{\Delta x}{2^n}$$

After n iterations, the root must lie within $\pm \Delta x/2^n$ of our estimate. This means that the error bound at n^{th} iteration is

$$E_n = \left| \frac{\Delta x}{2^n} \right|$$

Similarly,

$$E_{n+1} = \left| \frac{\Delta x}{2^{n+1}} \right| = \frac{E_n}{2} \quad (6.15)$$

That is, the error decreases linearly with each step by a factor of 0.5. The bisection method is, therefore, *linearly convergent*. Since the convergence is slow to achieve a high degree of accuracy, a large number of iterations may be needed. However, the bisection algorithm is guaranteed to converge.

Program BISECT

This program finds a root of a nonlinear equation using the bisection method. BISECT uses a subroutine, BIM to find a root in a given interval and invokes a function subprogram, F(x) to evaluate the function at the estimated root.

The subroutine subprogram BIM locates a root in the given interval $[A, B]$ using Algorithm 6.2. BIM applies the following criterion for terminating the process

$$\left| \frac{x_n - x_{n-1}}{x_n} \right| < \text{EPS}$$

That is, the relative error in the successive approximations must be less than a specified error limit.

The function subprogram $F(x)$ simply evaluates the function value at a given value of x and returns the result to the calling module. Note that by simply changing the function definition statement

$$F = x * x + x - 2$$

we can use the BISECT program to evaluate a root of any function.

Also note that the program prints out a message in case the specified interval does not bracket a root.

```

* -----
* PROGRAM BISECT
* -----
* Main program
*   This program finds a root of a nonlinear equation
*   using the bisection method
* -----
* Functions invoked
*   F
* -----
* 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
* -----
REAL A,B,ROOT,EPS,F
INTEGER S, COUNT
EXTERNAL BIM,F
PARAMETER(EPS=0.000001)
WRITE(*,*)
WRITE(*,*) ' SOLUTION BY BISECTION METHOD'
WRITE(*,*)
WRITE(*,*) 'Input starting values'
READ(*,*) A,B
CALL BIM(A,B,EPS,S,ROOT,COUNT)
IF (S.EQ.0) THEN
    WRITE(*,*)
    WRITE(*,*) 'Starting points do not bracket any root'
    WRITE(*,*) '(Check whether they bracket EVEN roots)'
    WRITE(*,*)
ELSE
    WRITE(*,*)
    WRITE(*,*) 'Root = ', ROOT
    WRITE(*,*) 'F(Root) = ', F(ROOT)
    WRITE(*,*)
    WRITE(*,*) 'ITERATIONS = ', COUNT
    WRITE(*,*)

```

```

ENDIF
STOP
END

```

* ----- End of main program ----- *

* SUBROUTINE BIM(A,B,EPS,S,ROOT,COUNT) *

* Subroutine *

* This subroutine finds a root of nonlinear equation *
in the interval [A,B] using the bisection method *

* Arguments *

* Input *

* A - Left endpoint of interval *

* B - Right endpoint of interval *

* EPS - Error bound *

* Output *

* S - Status *

* ROOT - Final Solution *

* COUNT - Number of Iterations *

* Local Variables *

* X1,X2,X0,F0,F1,F2 *

* Functions invoked *

* F,ABS *

* Subroutines called *

* NIL *

```
REAL A,B,ROOT,EPS,F,X1,X2,X0,F0,F1,F2,ABS
```

```
INTEGER S,COUNT
```

```
EXTERNAL F
```

```
INTRINSIC ABS
```

* Function values at initial points

X1 = A

X2 = B

F1 = F(A)

F2 = F(B)

-- * Test if initial values bracket a SINGLE root

IF(F1*F2 .GT. 0) THEN

S = 0

RETURN

ENDIF

* Bisect the interval and locate the root iteratively
COUNT = 1

```

111 X0 = (X1+X2)/2.0
      F0 = F(X0)
      IF (F0 .EQ. 0) THEN
        S=1
        ROOT = X0
        RETURN
      ENDIF
      IF(F1*F0 .LT.0) THEN
        X2 = X0
      ELSE
        X1 = X0
        F1 = F0
      ENDIF
* Test for accuracy and repeat the process, if necessary
      IF(ABS((X2-X1)/X2).LT.EPS) THEN
        S = 1
        ROOT = (X1+X2)/2.0
        RETURN
      ELSE
        COUNT = COUNT + 1
        GO TO 111
      ENDIF
    END
* ----- End of subroutine BIM -----
* -----
* Function subprogram F(x)
* -----
      REAL FUNCTION F(X)
      REAL X
      F = X*X+X-2
      RETURN
    END
* ----- End of function F(X) -----

```

Test Results of BISECT

The program was used to solve the equation

$$x^2 + x - 2 = 0$$

using two sets of starting points:

(0.0, 2.0) and (0.5, 2.0)

First run

```

      SOLUTION BY BISECTION METHOD
      Input starting values
      0.0 2.0

```

```

Root = 1.0000000
F(ROOT) = .0000000
ITERATIONS = 1
Stop - Program terminated.

```

Second run

```

SOLUTION BY BISECTION METHOD
Input starting values
0.5 2.0
Root      = 9.999999E-001
F(ROOT)   = -3.576279E-007
ITERATIONS = 21
Stop - Program terminated.

```

6.7 FALSE POSITION METHOD

In bisection method, the interval between x_1 and x_2 is divided into two equal halves, irrespective of location of the root. It may be possible that the root is closer to one end than the other as shown in Fig. 6.3. Note that the root is closer to x_1 . Let us join the points x_1 and x_2 by a straight line. The point of intersection of this line with the x axis (x_0) gives an improved estimate of the root and is called the *false position* of the root. This point then replaces one of the initial guesses that has a function value of the same sign as $f(x_0)$. The process is repeated with the new values of x_1 and x_2 . Since this method uses the false position of the root repeatedly, it is called the *false position method* (or *regula falsi* in Latin). It is also called the *linear interpolation method* (because an approximate root is determined by linear interpolation).

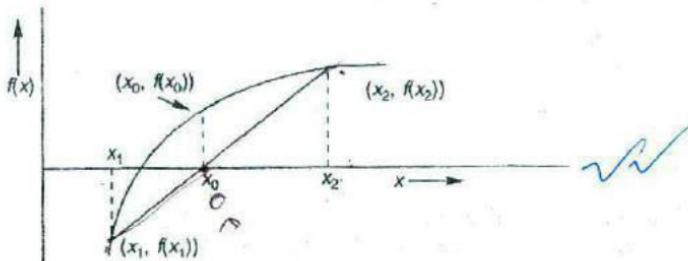


Fig. 6.3 Illustration of false position method

False Position Formula

A graphical depiction of the false position method is shown in Fig. 6.3. We know that equation of the line joining the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ is given by

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{y - f(x_1)}{x - x_1} \quad (6.16)$$

Since the line intersects the x -axis at x_0 , when $x = x_0, y = 0$, we have

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{-f(x_1)}{x_0 - x_1}$$

or

$$x_0 - x_1 = -\frac{f(x_1)(x_2 - x_1)}{f(x_2) - f(x_1)}$$

Then, we have

$$x_0 = x_1 - \frac{f(x_1)(x_2 - x_1)}{f(x_2) - f(x_1)}$$

(6.17)

This equation is known as the *false position formula*. Note that x_0 is obtained by applying a correction to x_1 .

False Position Algorithm

Having calculated the first approximate to the root, the process is repeated for the new interval, as done in the bisection method, using Algorithm 6.3.

False Position Method

$$\text{Let } x_0 = x_1 - f(x_1) \times \frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

If $f(x_0) \times f(x_1) < 0$

set $x_2 = x_0$

otherwise

set $x_1 = x_0$

Algorithm 6.3

A major difference between this algorithm and the bisection algorithm is the way x_0 is computed.

Example 6.5

Use the false position method to find a root of the function

$$f(x) = x^2 - x - 2 = 0$$

in the range $1 < x < 3$

Iteration I

Given $x_1 = 1$ and $x_2 = 3$

$$f(x_1) = f(1) = -2$$

$$f(x_2) = f(3) = 4$$

$$x_0 = x_1 - f(x_1) \times \frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

$$= 1 + 2 \times \frac{3 - 1}{4 + 2} = 1.6667$$

Iteration 2

$$f(x_0) f(x_1) = f(1.6667) f(1) = 1.7778$$

Therefore, the root lies in the interval between x_0 and x_2 . Then,

$$x_1 = x_0 = 1.6667$$

$$f(x_1) = f(1.6667) = -0.8889$$

$$f(x_2) = f(3) = 4$$

$$x_0 = 1.6667 + 0.8889 \times \frac{3 - 1.6667}{4 + 0.8889} = 1.909$$

Iteration 3

$$f(1.909) f(1.6667) = +0.2345$$

Root lies between $x_0 (= 1.909)$ and $x_2 (= 3)$

Therefore,

$$x_1 = x_0 = 1.909$$

$$x_2 = 3$$

$$x_0 = 1.909 + 0.2647 \times \frac{3 - 1.909}{4 - 0.2647}$$

$$= 1.909 + 0.2647 \times \frac{1.091}{3.7353} = 1.986$$

The estimated root after third iteration is 1.986. Remember that the interval contains a root $x = 2$. We can perform additional iterations to refine this estimate further.

Convergence of False Position Method

The false position formula is based on the linear interpolation model. In the false position iteration, one of the starting points is fixed while the other moves towards the solution. Assume that the initial points bracketing the solution are a and b and that a moves towards the solution and b is fixed as illustrated in Fig. 6.4.

Let $x_1 = a$ and x_r be the solution.
Then,

$$e_1 = x_r - x_1$$

$$e_2 = x_r - x_2$$

That is,

$$e_i = x_r - x_i$$

It can be shown that

$$e_{i+1} = e_r \times \frac{(x_r - b) f''(R)}{f'(R)} \quad (6.18)$$

where R is some point in the interval x_i and b . This shows that the process of iteration converges linearly.

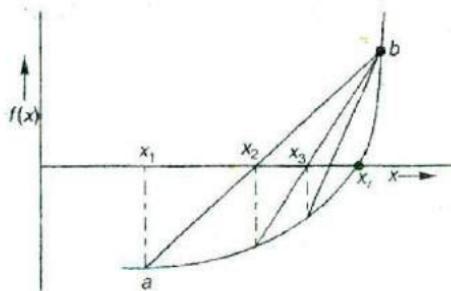


Fig. 6.4 Convergence of false position method

Program FALSE

The program FALSE finds a root of a nonlinear equation using the false position method. The program uses a function subprogram F and a subroutine, FAL to implement the method.

The function evaluates the function at any given point and the subroutine determines a root in a given interval using Algorithm 6.3.

We can use the FALSE program to identify a root of any function by changing the function statement in the function subprogram F.

```

*-----*
      PROGRAM FALSE
*-----*
* Main program
*   This program finds a root of a nonlinear equation
*   by false position method
*-----*
* Functions invoked
*   F
*-----*
* 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 = Tolerance value
*-----*
      REAL A,B,ROOT,ABS,S
      INTEGER COUNT
      EXTERNAL F,FAL
      PARAMETER (EPS = 0.00001)

```

```

      WRITE(*,*) 'Input starting values'
      READ(*,*) A,B
      WRITE(*,*) 
      WRITE(*,*) ' SOLUTION BY FALSE POSITION METHOD'
      WRITE(*,*) 
      CALL FAL(A,B,EPS,S,ROOT,COUNT)
      IF(S.EQ.0) THEN
          WRITE(*,*) 'Starting points do not bracket any
                           root'
          WRITE(*,*) 
      ELSE
          WRITE(*,*) 
          WRITE(*,*) 'Root = ', ROOT
          WRITE(*,*) 'F(ROOT) = ', F(ROOT)
          WRITE(*,*) 'NO.OF ITERATIONS = ', COUNT
          WRITE(*,*) 
      ENDIF
      STOP
      END
* ----- End of main FALSE -----
*
* -----
* SUBROUTINE FAL(A,B,EPS,S,ROOT,COUNT)
* -----
* Subroutine
*   FAL finds a root of a nonlinear equation
*
* -----
* Arguments
* Input
*   A - Left-end point of interval
*   B - Right-end point of interval
*   EPS - Error bound
* Output
*   S - Status of completion of task
*   ROOT - Final solution
*   COUNT - Number of iterations done
*
* -----
* Local Variables
*   X0,X1,X2,F0,F1,F2
*
* -----
* Functions invoked
*   F,ABS
*
* -----
* Subroutines called
*   NIL
*

```

```

REAL A,B,EPS,X0,X1,X2,F0,F1,F2,F,ABS
INTEGER S,COUNT
INTRINSIC ABS
EXTERNAL F

X1 = A
X2 = B
F1 = F(X1)
F2 = F(X2)

* Test if A and B bracket a root
IF(F1*F2 .GT. 0) THEN
    S = 0
    RETURN
ENDIF
WRITE(*,*), X1, X2
COUNT = 1
111 X0 = X1 - F1 * (X2-X1)/(F2-F1)
F0 = F(X0)
IF(F1*F0 .LT. 0) THEN
    X2 = X0
    F2 = F0
ELSE
    X1 = X0
    F1 = F0
ENDIF
WRITE(*,*), X1,X2

* Test if desired accuracy is achieved
IF(ABS((X2-X1)/X2) .LT.EPS) THEN
    S = 1
    ROOT = (X1+X2)*0.5
    RETURN
ELSE
    COUNT = COUNT+1
    GO TO 111
ENDIF
END
* ----- End of subroutine FAL -----
*----- Function subprogram F(X) -----
*
REAL FUNCTION F(X)
REAL X
F = X*X+X-2
RETURN
END

```

* ----- End of function F(X) ----- *

Test Results of FALSE

The program was used to find a root of the equation

$$x^2 + x - 2 = 0$$

using the initial values (1.5, 2.0) and (-3.0, 0.0). Test results are given below:

First run

Input starting values
1.5 2.0

SOLUTION BY FALSE POSITION METHOD

Starting points do not bracket any root

Stop - Program terminated

Second run

Input starting values
-3.0 0.0

SOLUTION BY FALSE POSITION METHOD

X1	X2
-3.0000000	-1.0000000
-3.0000000	-1.6666670
-3.0000000	-1.9090910
-3.0000000	-1.9767440
-3.0000000	-1.9941520
-3.0000000	-1.9985360
-3.0000000	-1.9996340
-3.0000000	-1.9999080
-3.0000000	-1.9999770
-3.0000000	-1.9999940
-3.0000000	-1.9999990
-3.0000000	-2.0000000
-3.0000000	-2.0000000
-2.0000000	-2.0000000

Root = -2.0000000

F(ROOT) = .0000000

NO.OF ITERATIONS = 14

Stop - Program terminated.

Note that the program outputs a message when the given set of initial values do not bracket any root. When a root is possible, the process of iteration stops when the relative error satisfies the condition

$$\left| \frac{x_n - x_{n-1}}{x_n} \right| < \text{EPS}$$

~~6.5~~ NEWTON-RAPHSON METHOD

Consider a graph of $f(x)$ as shown in Fig. 6.5. Let us assume that x_1 is an approximate root of $f(x) = 0$. Draw a tangent at the curve $f(x)$ at $x = x_1$ as shown in the figure. The point of intersection of this tangent with the x -axis gives the second approximation to the root. Let the point of intersection be x_2 . The slope of the tangent is given by

$$\tan \alpha = \frac{f(x_1)}{x_1 - x_2} = f'(x_1) \quad (6.19)$$

where $f(x_1)$ is the slope of $f(x)$ at $x = x_1$. Solving for x_2 we obtain

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \quad (6.20)$$

This is called the *Newton-Raphson formula*.

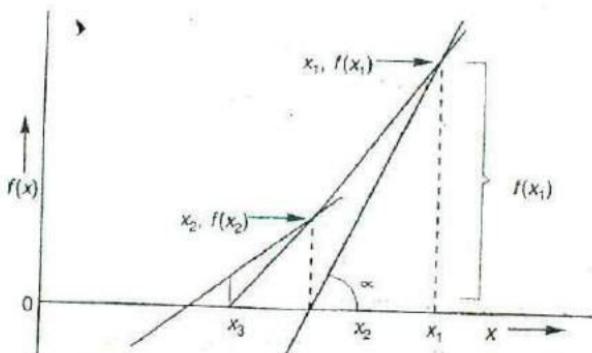


Fig. 6.5 Newton-Raphson method

The next approximation would be

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

In general,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (6.21)$$

This method of successive approximation is called the *Newton-Raphson method*. The process will be terminated when the difference between two successive values is within a prescribed limit.

The Newton-Raphson method approximates the curve of $f(x)$ by tangents. Complications will arise if the derivative $f'(x_n)$ is zero. In such cases, a new initial value for x must be chosen to continue the procedure.

Example 6.6

Derive the Newton-Raphson formula using the Taylor series expansion. Assume that x_n is an estimate of a root of the function $f(x)$. Consider a small interval h such that

$$h = x_{n+1} - x_n$$

We can express $f(x_{n+1})$ using Taylor series expansion as follows:

$$f(x_{n+1}) = f(x_n) + f'(x_n)h + f''(x_n) \frac{h^2}{2!} + \dots$$

If we neglect the terms containing the second order and higher derivatives, we get

$$f(x_{n+1}) = f(x_n) + f'(x_n)h$$

If x_{n+1} is a root of $f(x)$, then

$$f(x_{n+1}) = 0 = f(x_n) + f'(x_n)h$$

Then,

$$h = \frac{-f(x_n)}{f'(x_n)} = x_{n+1} - x_n$$

Therefore,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton-Raphson Algorithm

Perhaps the most widely used of all methods for finding roots is the Newton-Raphson method. Algorithm 6.4 describes the steps for implementing Newton-Raphson method iteratively.

Newton-Raphson Method

1. Assign an initial value to x , say x_0 .
2. Evaluate $f(x_0)$ and $f'(x_0)$.
3. Find the improved estimate of x_0

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

4. Check for accuracy of the latest estimate.

Compare relative error to a predefined value E . If $\left| \frac{x_1 - x_0}{x_1} \right| \leq E$
stop; Otherwise continue.

5. Replace x_0 by x_1 and repeat steps 3 and 4.

Algorithm 6.4

Example 6.7

Find the root of the equation

$$f(x) = x^2 - 3x + 2 \quad \checkmark$$

in the vicinity of $x = 0$ using Newton-Raphson method.

$$f'(x) = 2x - 3 \quad \checkmark$$

Let $x_1 = 0$ (first approximation)

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

$$= 0 - \frac{2}{-3} = \frac{2}{3} = 0.6667 \quad \checkmark$$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

Similarly,

$$x_3 = 0.6667 - \frac{0.4444}{-1.6667} = 0.9333$$

$$x_4 = 0.9333 - \frac{0.071}{-1.334} = 0.9959$$

$$x_5 = 0.9959 - \frac{0.0041}{-1.0082} = 0.9999$$

$$x_6 = 0.9999 - \frac{0.0001}{-1.0002} = 1.0000 \quad \checkmark$$

Since $f(1.0) = 0$, the root closer to the point $x = 0$ is 1.000.

Convergence of Newton-Raphson Method

Let x_n be an estimate of a root of the function $f(x)$. If x_n and x_{n+1} are close to each other, then, using Taylor's series expansion, we can state

$$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n) + \frac{f''(R)}{2} (x_{n+1} - x_n)^2 \quad (6.22)$$

where R lies somewhere in the interval x_n to x_{n+1} and third and higher order have been dropped.

Let us assume that the exact root of $f(x)$ is x_r . Then $x_{n+1} = x_r$. Therefore $f(x_{n+1}) = 0$ and substituting these values in equation (6.22), we get

$$0 = f(x_n) + f'(x_n)(x_r - x_n) + \frac{f''(R)}{2} (x_r - x_n)^2 \quad (6.23)$$

We know that the Newton's iterative formula is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Rearranging the terms, we get

$$f(x_n) = f'(x_n)(x_n - x_{n+1})$$

Substituting this for $f(x_n)$ in Eq. (6.23) yields

$$0 = f'(x_n)(x_r - x_{n+1}) + \frac{f''(R)}{2} (x_r - x_n)^2 \quad (6.24)$$

We know that the error in the estimate x_{n+1} is given by

$$e_{n+1} = x_r - x_{n+1}$$

Similarly,

$$e_n = x_r - x_n$$

Now, equation (6.24) can be expressed in terms of these errors as

$$0 = f'(x_n) e_{n+1} + \frac{f''(R)}{2} e_n^2$$

Rearranging the terms we get,

$$e_{n+1} = -\frac{f''(R)}{2f'(x_n)} e_n^2 \quad (6.25)$$

Equation (6.25) shows that the error is roughly proportional to the square of the error in the previous iteration. Therefore, the Newton-Raphson method is said to have *quadratic convergence*.

Program NEWTON

The NEWTON shows a FORTRAN program for evaluating a root of nonlinear equations by Newton-Raphson method. The program uses two external functions, F and FD and one intrinsic function, ABS. The function F evaluates the actual function at a given value of x and FD evaluates the first derivative of the function at x.

The program employs the Algorithm 6.4 and prints out the value of a root (when it is found) and the number of iterations required to obtain the result. It also prints the value of the function at that point to check its accuracy. In case the process does not converge within a specified number of iterations, the program outputs a message accordingly.

```

PROGRAM NEWTON
*
* Main program
* This program finds a root of a nonlinear equation
* by Newton-Raphson method
*
* Functions invoked
* ABS,F,FD
*
* Subroutines used
* NIL
*

```

```

* Variables used
*   X0 - Initial value 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
*
* -----
REAL X0,XN,FX,FDX,ABS,EPS,F,FD
INTEGER COUNT, MAXIT
INTRINSIC ABS
EXTERNAL F,FD
PARAMETER(EPS = 0.000001, MAXIT = 100)
WRITE(*,*) 'Input initial value of x'
READ(*,*) X0
WRITE(*,*) ' '
WRITE(*,*) ' SOLUTION BY NEWTON-RAPHSON METHOD'
WRITE(*,*) ' '
COUNT = 1
100 FX = F(X0)
FDX = FD(X0)
XN = X0 -FX/FDX
IF(ABS((XN-X0)/XN) .LT.EPS) THEN
    WRITE(*,*) 'Root = ', XN
    WRITE(*,*) 'Function value = ', F(XN)
    WRITE(*,*) 'Number of iterations = ', COUNT
    WRITE(*,*) ' '
ELSE
    X0 = XN
    COUNT = COUNT + 1
    IF(COUNT.LE.MAXIT) THEN
        GO TO 100
    ELSE
        WRITE(*,*) ' '
        WRITE(*,*) 'SOLUTION DOES NOT CONVERGE IN'
        WRITE(*,*) MAXIT, ' ITERATIONS'
        WRITE(*,*) ' '
    ENDIF
ENDIF
STOP
END
* ----- End of main program ----- *

```

```

* -----
* Function subprogram F(x)
* -----
REAL FUNCTION F(X)
REAL X
F = X*X+X-2
RETURN
END
* ----- End of function F(X) -----
* -----
* Function subprogram FD(x)
* -----
REAL FUNCTION FD(X)
REAL X
FD = 2*X+1
RETURN
END
* ----- End of function FD(X) -----

```

Test Results of NEWTON Given below are the outputs of the test runs of the program NEWTON.

First run

```

Input initial value of x
0
SOLUTION BY NEWTON-RAPHSON METHOD
Root = 1.0000000
Function value = .0000000
Number of iterations = 6
Stop - Program terminated.

```

Second run

```

Input initial value of x
-1.0
SOLUTION BY NEWTON-RAPHSON METHOD
Root = -2.0000000
Function value = .0000000
Number of iterations = 6
Stop - Program terminated.

```

Third run

```

Input initial value of x
1.0
SOLUTION BY NEWTON-RAPHSON METHOD
Root = 1.0000000
Function value = .0000000
Number of iterations = 1
Stop - Program terminated.

```

Example 6.8

Show, through an example, that the number of correct digits approximately doubles with each iteration in Newton-Raphson method.

Given below is the output of NEWTON program for solving the equation $x^3 - 4x^2 + x + 6 = 0$, using an initial estimate of 5.0.

Iteration	Estimation	Correct digits
1	5.000000	NIL
2	4.000000	NIL
3	3.411765	1
4	3.114462	1
5	3.013215	2
6	3.000213	4
7	3.000000	7

This shows that the number of correct digits approximately doubles with each iteration near the root.

Limitations of Newton-Raphson Method

The Newton-Raphson method has certain limitations and pitfalls. The method will fail in the following situations.

1. Division by zero may occur if $f'(x_i)$ is zero or very close to zero.
2. If the initial guess is too far away from the required root, the process may converge to some other root.
3. A particular value in the iteration sequence may repeat, resulting in an infinite loop. This occurs when the tangent to the curve $f(x)$ at $x = x_{i+1}$ cuts the x -axis again at $x = x_i$.

SECANT METHOD

Secant method, like the false position and bisection methods, uses two initial estimates but does not require that they must bracket the root. For example, the secant method can use the points x_1 and x_2 in Fig. 6.6 as starting values, although they do not bracket the root. Slope of the secant line passing through x_1 and x_2 is given by

$$\frac{f(x_1)}{x_1 - x_3} = \frac{f(x_2)}{x_2 - x_3}$$

$$f(x_1)(x_2 - x_3) = f(x_2)(x_1 - x_3)$$

or

$$x_3 [f(x_2) - f(x_1)] = f(x_2)x_1 - f(x_1)x_2$$

Then

$$x_3 = \frac{f(x_2)x_1 - f(x_1)x_2}{f(x_2) - f(x_1)}$$

(6.26)

By adding and subtracting $f(x_2)x_2$ to the numerator and rearranging the terms we get

$$x_3 = x_2 - \frac{f(x_2)(x_2 - x_1)}{f(x_2) - f(x_1)} \quad (6.27)$$

Equation (6.27) is known as the *secant formula*. If the secant line represents the linear interpolation polynomial of the function $f(x)$ (with the interpolating points x_1 and x_2) then x_3 , which intercepts the x -axis, represents the approximate root of $f(x)$.

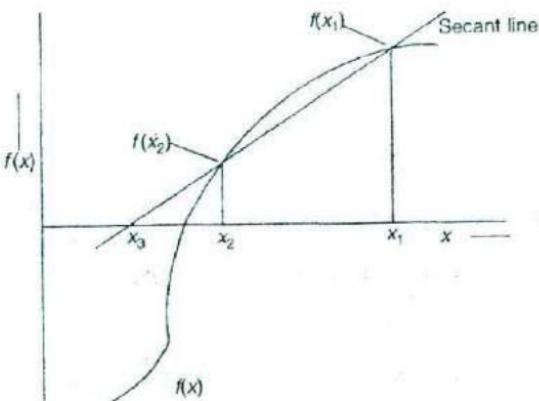


Fig. 6.6 Graphical depiction of secant method

The approximate value of the root can be refined by repeating this procedure by replacing x_1 and x_2 by x_2 and x_3 , respectively, in Eq. (6.27). That is, next approximate value is given by

$$x_4 = x_3 - \frac{f(x_3)(x_3 - x_2)}{f(x_3) - f(x_2)}$$

This procedure is continued till the desired level of accuracy is obtained. We can express the secant formula in general form as follows:

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})} \quad (6.28)$$

Note that Eqs (6.17) and (6.28) are similar and both of them use two initial estimates. However, there is a major difference in their algorithms of implementation. In Eq. (6.17), the latest estimate replaces one of the end points of the interval such that the new interval brackets the root. But, in Eq. (6.28) the values are prefaced in strict sequence, i.e., x_{i-1} is replaced by x_i and x_i by x_{i+1} . The points may not bracket the root.

Secant Algorithm

Note that the value of new approximation of the root depends on the previous two approximations and corresponding functional values. Algorithm 6.5 illustrates how this procedure is implemented to estimate a root with a given level of accuracy.

Secant Method

1. Decide two initial points x_1 and x_2 , accuracy level required, E .
2. Compute $f_1 = f(x_1)$ and $f_2 = f(x_2)$
3. Compute $x_3 = \frac{f_2 x_1 - f_1 x_2}{f_2 - f_1}$
4. Test for accuracy of x_3 .

If $\left| \frac{x_3 - x_2}{x_3} \right| > E$, then
 set $x_1 = x_2$ and $f_1 = f_2$
 set $x_2 = x_3$ and $f_2 = f(x_3)$
 go to step 3
 otherwise,
 set root = x_3
 print results
5. Stop

Algorithm 6.5

Example 6.9

Use the secant method to estimate the root of the equation

$$x^2 - 4x - 10 = 0$$

with the initial estimates of $x_1 = 4$ and $x_2 = 2$.

Given $x_1 = 4$ and $x_2 = 2$

$$f(x_1) = f(4) = -10$$

$$f(x_2) = f(2) = -14$$

(Note that these points do not bracket a root)

$$\begin{aligned} x_3 &= x_2 - \frac{f(x_2)(x_2 - x_1)}{f(x_2) - f(x_1)} \\ &= 2 - \frac{-14(2 - 4)}{-14 - (-10)} = 9. \end{aligned}$$

For second iteration,

$$x_1 = x_2 = 2$$

$$\begin{aligned} &= 2 - \frac{28}{-4} \\ &= 2 - (-7) \\ &= 2 + 7 = 9 \end{aligned}$$

$$\begin{aligned}x_2 &= x_3 = 9 \\f(x_1) &= f(2) = -14 \\f(x_2) &= f(9) = 95\end{aligned}$$

$$x_3 = 9 - \frac{35(9 - 2)}{35 + 14} = 4$$

For third iteration,

$$\begin{aligned}x_1 &= 9 \\x_2 &= 4 \\f(x_1) &= f(9) = 95 \\f(x_2) &= f(4) = -10 \\x_3 &= 4 - \frac{-10(4 - 9)}{-10 - 35} = 5.1111\end{aligned}$$

For fourth iteration,

$$\begin{aligned}x_1 &= 4 \\x_2 &= 5.1111 \\f(x_1) &= f(4) = -10 \\f(x_2) &= f(5.1111) = -4.3207 \\x_3 &= 5.1111 - \frac{-4.3207(5.1111 - 4)}{-4.3207 - 10} = 5.9563\end{aligned}$$

For fifth iteration,

$$\begin{aligned}x_1 &= 5.1111 \\x_2 &= 5.9563 \\f(x_1) &= f(5.1111) = -4.3207 \\f(x_2) &= f(5.9563) = 5.0331 \\x_3 &= 5.9563 - \frac{5.0331(5.9563 - 5.1111)}{5.0331 + 4.3207} = 5.5014\end{aligned}$$

For sixth iteration,

~~$x_1 = 5.9563$~~

$$\begin{aligned}x_1 &= 5.9563 \\x_2 &= 5.5014 \\f(x_1) &= f(5.9563) = 5.0331 \\f(x_2) &= f(5.5014) = -1.7392 \\x_3 &= 5.5014 - \frac{-1.7392(5.5014 - 5.9563)}{-1.7392 + 5.0331} = 5.6182\end{aligned}$$

The value can be further refined by continuing the process, if necessary.

Example 6.10

Compare the secant iterative formula with the Newton formula for estimating a root.

Newton formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Secant formula:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

This shows that the derivative of the function in the Newton formula $f'(x_n)$, has been replaced by the term

$$\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

in the secant formula. This is a major advantage because there is no need for the evaluation of derivatives. There are many functions whose derivatives may be extremely difficult to evaluate.

However, one drawback of the secant iterative formula is that the previous two iterates are required for estimating the new one. Another drawback of the secant method is its slower rate of convergence. It is proved later in this section that the rate of convergence of secant method is 1.618 while that of the Newton method is 2.

Convergence of Secant Method

The secant formula of iteration is

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})} \quad (6.29)$$

Let x_r be actual root of $f(x)$ and e_i the error in the estimate of x_i . Then,

$$x_{i+1} = e_{i+1} + x_r$$

$$x_i = e_i + x_r$$

$$x_{i-1} = e_{i-1} + x_r$$

Substituting these in Eq. (6.29) and simplifying, we get the error equation as

$$e_{i+1} = \frac{e_{i-1}f(x_i) - e_i f(x_{i-1})}{f(x_i) - f(x_{i-1})} \quad (6.30)$$

According to the Mean Value Theorem, there exists at least one point, say $x = R_i$, in the interval x_i and x_r such that

$$f'(R_i) = \frac{f(x_i) - f(x_r)}{x_i - x_r}$$

We know that

$$f(x_r) = 0$$

$$x_i - x_r = e_i$$

and therefore

$$f'(R_i) = \frac{f(x_i)}{e_i}$$

or

$$f(x_i) = e_i f'(R_i)$$

Similarly,

$$f(x_{i-1}) = e_{i-1} f'(R_{i-1})$$

Substituting these in the numerator of Eq. (6.30), we get

$$e_{i+1} = e_i e_{i-1} \frac{f'(R_i) - f'(R_{i-1})}{f(x_i) - f(x_{i-1})}$$

That is, we can say

$$e_{i+1} \propto e_i e_{i-1} \quad (6.31)$$

We know that the order of convergence of an iteration process is p , if

$$e_i \propto e_{i-1}^p \quad (6.32)$$

or

$$e_{i+1} \propto e_i^p \quad (6.33)$$

Substituting for e_{i+1} and e_i in Eq. (6.31), we get

$$e_i^p \propto e_{i-1}^p e_{i-1}$$

or

$$e_i \propto e_{i-1}^{(p+1)/p} \quad (6.34)$$

Comparing the relations (6.32) and (6.31), we observe that

$$p = (p + 1)/p$$

That is,

$$p^2 - p - 1 = 0$$

which has the solutions

$$p = \frac{1 \pm \sqrt{5}}{2}$$

Since p is always positive, we have

$$p = 1.618$$

It follows that the order of convergence of the secant method is 1.618 and the convergence is referred to as *superlinear convergence*.

Program SECANT

The program SECANT finds a root of a non-linear equation using two initial values supplied. The program employs a function subprogram, F, to evaluate the value of the function and a subroutine subprogram, SEC, to implement the Algorithm 6.5 for estimating the root. The subroutine uses the absolute relative error in the successive approximations for terminating the process.

```

* -----
*      PROGRAM SECANT
* -----
*      Main program
*      This program finds a root of a nonlinear
*      equation by secant method
* -----
*      Functions invoked
*          F
* -----
*      Subroutines used
*          SEC
* -----
*      Variables used
*          A - Left endpoint of interval
*          B - Right endpoint of interval
*          ROOT - Final solution
*          COUNT - Number of iterations completed
* -----
*      Constants used
*          EPS - Error bound
*          MAXIT - Maximum number of iteration
* -----
REAL A,B,ROOT,EPS,F
INTEGER COUNT,STATUS, MAXIT
EXTERNAL F,SEC
PARAMETER(EPS = 0.000001, MAXIT = 50)
WRITE(*,*)
WRITE(*,*) ' SOLUTION BY SECANT METHOD'
WRITE(*,*)
WRITE(*,*) 'Input two starting points'
READ(*,*) A,B
CALL SEC(A,B,X1,X2,EPS,ROOT,COUNT,MAXIT,STATUS)
IF( STATUS .EQ. 1 ) THEN
    WRITE(*,*)
    WRITE(*,*) ' DIVISION BY ZERO'
    WRITE(*,*)
    WRITE(*,*) ' Last X1      =' ,X1

```

```

      WRITE(*,*)' Last X2      = ',X2
      WRITE(*,*)' ITERATIONS = ',COUNT
      WRITE(*,*)
      ELSE IF( STATUS .EQ. 2 ) THEN
         WRITE(*,*)
         WRITE(*,*) 'NO CONVERGENCE IN ',MAXIT,' ITERATIONS'
         WRITE(*,*)
      ELSE
         WRITE(*,*)
         WRITE(*,*) 'Root = ', ROOT
         WRITE(*,*) 'Function value at root = ', F(ROOT)
         WRITE(*,*)
         WRITE(*,*) 'Number of iterations = ',COUNT
         WRITE(*,*)
      ENDIF
      STOP
      END
* ----- End of main program -----
*
* -----
* SUBROUTINE SEC(A,B,X1,X2,EPS,ROOT,COUNT,MAXIT,STATUS)
*
* -----
* Subroutine
* This subroutine computes a root of an equation
* using the secant method
*
* -----
* Arguments
* -----
* A - Left end point
* B - Right end point
* EPS - Tolerance bound
* MAXIT - Maximum iterations allowed
* Output
* X1 - New left point
* X2 - New right point
* ROOT - Final solution
* COUNT - Number of iterations done
* STATUS - Status of completion of the task
*
* -----
* Local Variables
* X3,F1,F2,ERROR
*
* -----
* Functions invoked
* F, ABS
*
* -----
* Subroutines called
* NIL
*

```

```

REAL A,B,X1,X2,X3,EPS,ROOT,F1,F2,F,ABS,ERROR
INTEGER COUNT,STATUS,MAXIT
INTRINSIC ABS
EXTERNAL F

* Function values at initial points
X1 = A
X2 = B
F1 = F(A)
F2 = F(B)

* Compute the root iteratively
COUNT = 1
111 IF(ABS(F1-F2) .LE. 1.E-10) THEN
    STATUS = 1
    RETURN
ENDIF

X3 = X2-F2 * (X2-X1)/(F2-F1)
ERROR = ABS((X3-X2)/X3)

* Test for accuracy
IF (ERROR .GT. EPS) THEN
* --- Test for convergence
    IF( COUNT .EQ. MAXIT ) THEN
        STATUS = 2
        RETURN
    ENDIF

    X1 = X2
    X2 = X3
    F1 = F2
    F2 = F(X3)
    COUNT = COUNT + 1
    GO TO 111
* and compute next approximation
ENDIF

ROOT = X3
STATUS = 3
RETURN
END

* ----- End of subroutine SEC -----
* -----
* Function subprogram F(x)
* -----
REAL FUNCTION F(X)
REAL X
F = X*X+X-2

```

```

    RETURN
    END
* ----- End of function F(X) -----

```

Test Results of SECANT

Given below are the outputs of test runs of SECANT

First run

```

    SOLUTION BY SECANT METHOD
    Input two starting points
    -3.0001 0
    Root = -2.0000000
    Function value at root = .0000000
    Number of iterations = 11
    Stop - Program terminated.

```

Second run

```

    SOLUTION BY SECANT METHOD
    Input two starting points
    0 -3
    Root = -2.0000000
    Function value at root = .0000000
    Number of iterations = 8
    Stop - Program terminated.

```

Note that the program incorporates a test for convergence and also a test for 'division by zero' while evaluating the secant formula (see Eq. 6.27).

6.10 FIXED POINT METHOD

Any function in the form of

$$f(x) = 0 \quad (6.35)$$

can be manipulated such that x is on the left-hand side of the equation as shown below

$$x = g(x) \quad (6.36)$$

Equations (6.35) and (6.36) are equivalent and, therefore, a root of Eq. (6.36) is also a root of Eq. (6.35). The root of equation (6.36) is given the point of intersection of the curves $y = x$ and $y = g(x)$. This intersection point is known as the *fixed point* of $g(x)$ (see Fig. 6.7).

The above transformation can be obtained either by algebraic manipulation of the given equation or by simply adding x to both sides of the equation. For example,

$$x^2 + x - 2 = 0$$

can be written as

$$x = 2 - x^2$$

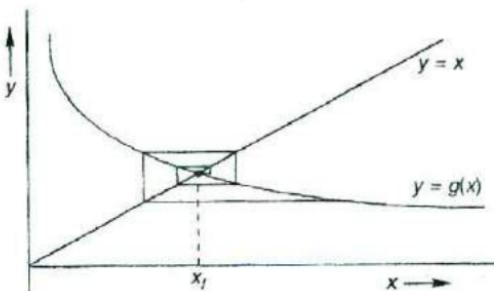


Fig. 6.7 Fixed point method

or

$$x = x^2 + x - 2 + x = x^2 + 2x - 2$$

Adding of x to both sides is normally done in situations where the original equation is not amenable to algebraic manipulations. For example,

$$\tan x = 0$$

would be put into the form of Eq. (6.36) by adding x to both sides. That is,

$$x = \tan x + x$$

The equation

$$x = g(x)$$

is known as the *fixed point equation*. It provides a convenient form for predicting the value of x as a function of x . If x_0 is the initial guess to a root, then the next approximation is given by

$$x_1 = g(x_0)$$

Further approximation is given by

$$x_2 = g(x_1)$$

This iteration process can be expressed in general form as

$$x_{i+1} = g(x_i) \quad i = 0, 1, 2, \dots \quad (6.37)$$

which is called the *fixed point iteration formula*. This method of solution is also known as the method of *successive approximations* or *method of direct substitution*.

The algorithm is simple. The iteration process would be terminated when two successive approximations agree within some specified error.

Example 6.11

Locate root of the equation

$$x^2 + x - 2 = 0$$

using the fixed point method.

The given equation can be expressed as

$$x = 2 - x^2$$

Let us start with an initial value of $x_0 = 0$

$$x_1 = 2 - 0 = 2$$

$$x_2 = 2 - 4 = -2$$

$$x_3 = 2 - 4 = -2$$

Since $x_3 - x_2 = 0$, -2 is one of the roots of the equation.
Let us assume that $x_0 = -1$. Then

$$x_1 = 2 - 1 = 1$$

$$x_2 = 2 - 1 = 1$$

Another root is 1 .

Example 6.12

Evaluate the square root of 5 using the equation

$$f(x) = x^2 - 5 = 0$$

by applying the fixed point iteration algorithm.

Let us reorganise the function as follows:

$$f(x) = x = 5/x$$

and assume $x_0 = 1$. Then,

$$x_1 = 5$$

$$x_2 = 1$$

$$x_3 = 5$$

$$x_4 = 1$$

The process does not converge to the solution. This type of divergence is known as oscillatory divergence.

Let us consider another form of $g(x)$ as shown below:

$$x = x^2 + x - 5$$

$$x_0 = 0$$

$$x_1 = -5$$

$$x_2 = 15$$

$$x_3 = 235$$

$$x_4 = 55455$$

Again it does not converge. Rather it diverges rapidly. This type of divergence is known as monotone divergence.

Let us try a third form of $g(x)$.

$$2x = 5/x + x$$

or

$$x = \frac{x + 5/x}{2}$$

$$x_0 = 1$$

$$x_1 = 3$$

$$x_2 = 2.3333$$

$$x_3 = 2.2381$$

$$x_4 = 2.2361$$

$$x_5 = 2.2361$$

$$x_6 = 2.2361$$

This time, the process converges rapidly to the solution. The square root of 5 is 2.2361 .

Convergence of Fixed Point Iteration

As stated earlier, the iteration function $g(x)$ can be formulated in different forms. Example 6.12 shows that not all forms result in convergence of solution. Convergence of the iteration process depends on the nature of $g(x)$. Figure 6.8 illustrates various patterns of behaviour of the iteration process of the fixed point method. Figures 6.8(a) and 6.8(b), show that the solution converges to the fixed point x_f during the iteration process. However, it does not happen in Fig. 6.8c and 6.8d. Notice that the process converges only when the absolute value of the slope of $y = g(x)$ curve is less than the slope of $y = x$ curve. Since the slope of $y = x$ curve is 1, the necessary condition for convergence is

$$g'(x) < 1$$

We can also notice that, in the neighbourhood of the solution, if the slope of $g(x)$ is positive, the convergence is monotone with "staircase" behaviour, and if the slope of $g(x)$ is negative, the convergence is oscillatory in behaviour. It is also clear that the closer the slope of $g(x)$ is to zero, the faster will be the convergence of the process.

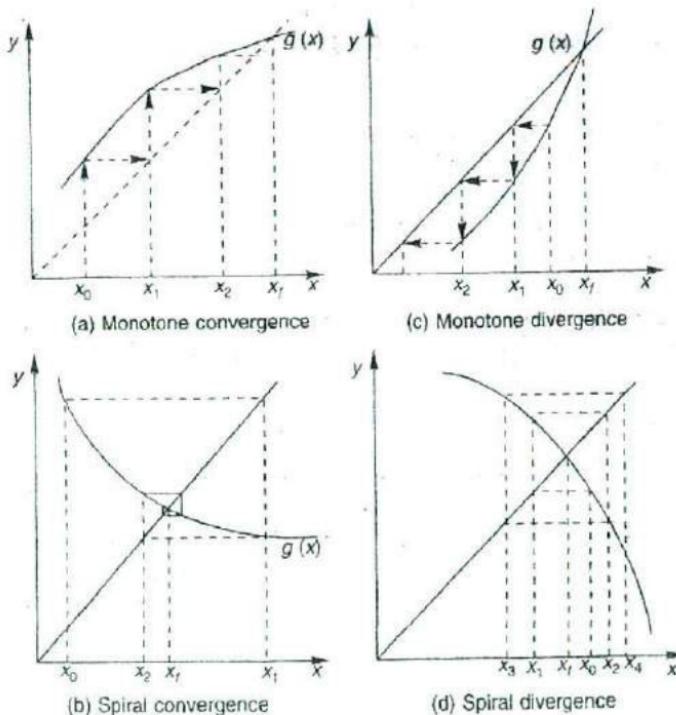


Fig. 6.8 Patterns of behaviour of fixed point iteration process

We can theoretically prove this as follows:
The iteration formula is

$$x_{i+1} = g(x_i) \quad (6.38)$$

Let x_f be a root of the equation. Then,

$$x_f = g(x_f) \quad (6.39)$$

Subtracting equation (6.38) from equation (6.39) yields

$$x_f - x_{i+1} = g(x_f) - g(x_i) \quad (6.40)$$

According to the mean value theorem, there is at least one point, say, $x = R$, in the interval x_f and x_i such that

$$g'(R) = \frac{g(x_f) - g(x_i)}{x_f - x_i}$$

This gives

$$g(x_f) - g(x_i) = g'(R)(x_f - x_i)$$

Substituting this in Eq. (6.40) yields

$$x_f - x_{i+1} = g'(R)(x_f - x_i) \quad (6.41)$$

If e_i represents the error in the i th iteration, then Eq. (6.41) becomes

$$e_{i+1} = g'(R)e_i \quad (6.42)$$

This shows that the error will decrease with each iteration only if

$$g'(R) < 1$$

Equation (6.42) implies the following:

1. Error decreases if $g'(R) < 1$
2. Error grows if $g'(R) > 1$
3. If $g'(R)$ is positive, the convergence is monotonic as in Fig. 6.8(a)
4. If $g'(R)$ is negative, the convergence will be oscillatory as in Fig. 6.8(b)
5. The error is roughly proportional to (or less than) the error in the previous step; the fixed point method is, therefore, said to be *linearly convergent*

Program FIXEDP

The program FIXEDP is the simplest of all programs discussed so far for determining a root of a nonlinear equation. The iteration process is terminated when two successive approximations agree within some specified error. The program uses a control loop to terminate the execution when the process does not converge within a specified number of iterations.

```

*-----*
      PROGRAM FIXEDP
*-----*
* Main program
*   This program finds a root of a function using
*   the fixedp point iteration method
*-----*
```

```

* Functions invoked
* G, ABS
* -----
* Subroutines used
* NIL
* -----
* Variables used
* X0 - Initial guess
* X - Estimated root
* ERROR - Relative error
* -----
* Constants used
* EPS - Error bound
* MAXIT - Maximum Iterations allowed
* -----
REAL X0,X,ERROR,G,ABS,EPS
INTEGER MAXIT
INTRINSIC ABS
EXTERNAL G
PARAMETER (EPS = 0.00001)

WRITE (*,*)
WRITE (*,*) 'SOLUTION BY FIXED POINT ITERATION METHOD'
WRITE (*,*)

WRITE(*,*) 'Input initial estimate of root'
READ (*,*) X0
WRITE(*,*) 'Maximum iterations allowed'
READ(*,*) MAXIT

WRITE(*,*) 'ITERATION      VALUE OF X      ERROR'
WRITE(*,*) '          100      1.000000E+000      1.000000E-007'

DO 100 I = 1, MAXIT
  X = G (X0)
  ERROR = ABS((X-X0)/X)
  WRITE(*,*) I,X,ERROR
  IF (ERROR .LT. EPS) THEN
    WRITE (*,*)
    STOP
  ENDIF
  X0 = X
100 CONTINUE
WRITE (*,*) 'Process does not converge to a root'
WRITE(*,*) 'Exit from loop'
STOP
END
* ----- End of main program FIXEDP -----

```

```

* -----
* Function subprogram G(X)
* -----
REAL FUNCTION G(X)
REAL X
G = 2.0-X**X
RETURN
END
* ----- End of function G(X) -----

```

Test Results of FIXEDP The outputs of the program FIXEDP for various initial values are shown below:

First Run

SOLUTION BY FIXED POINT ITERATION METHOD

Input initial estimate of root

0.0

Maximum iterations allowed

10

ITERATION	VALUE OF X	ERROR
1	2.0000000	1.0000000
2	-2.0000000	2.0000000
3	-2.0000000	.0000000

Stop - Program terminated.

Second Run

SOLUTION BY FIXED POINT ITERATION METHOD

Input initial estimate of root

1

Maximum iterations allowed

10

ITERATION	VALUE OF X	ERROR
1	1.0000000	.0000000

Stop - Program terminated.



6.11 DETERMINING ALL POSSIBLE ROOTS

All the methods discussed so far estimate only one root. What if we are interested in locating all the roots in the given interval? One option is to plot a graph of the function and then identify various independent intervals that bracket the roots. These intervals can be used to locate the various roots.

Another approach is to use an *incremental search* technique covering the entire interval containing the roots. This means that search for a root continues even after the first root is found. The procedure consists of starting at one end of the interval, say, at point a , and then searching for a root at every incremental interval till the other end, say, point b , is reached (see Fig. 6.9). The end points of each "incremental interval" can

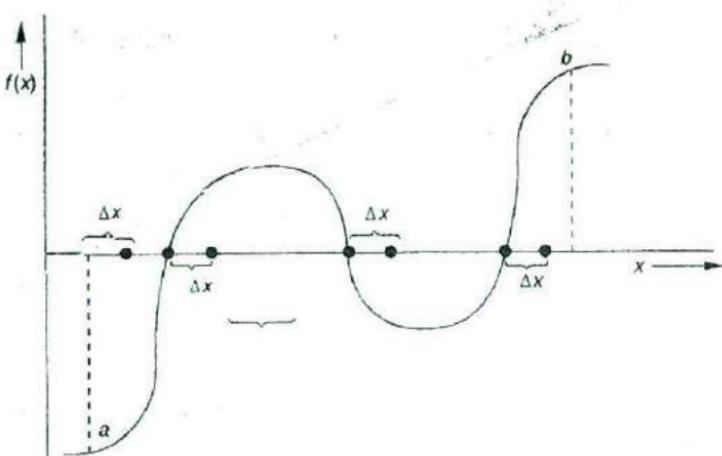


Fig. 6.9 Incremental search for all possible roots

serve as the initial points for one of the bracketing techniques discussed. Algorithm 6.6 describes the steps for implementing an incremental search technique using the bisection method for locating all roots.

A major problem is to decide the increment size. A small size may mean more iterations and more execution time. If the size is large, then there is a possibility of missing the closely spaced roots.

Determining all roots

1. Choose lower limit a and upper limit b of the interval covering all the roots.
2. Decide the size of the incremental interval Δx .
3. Set $x_1 = a$ and $x_2 = x_1 + \Delta x$.
4. Compute $f_1 = f(x_1)$ and $f_2 = f(x_2)$.
5. If $f_1 \times f_2 > 0$, the interval does not bracket any root
go to step 9.
Otherwise,
continue
6. Compute $x_0 = (x_1 + x_2)/2$ and $f_0 = f(x_0)$.
7. If $f_1 \times f_0 < 0$, then
set $x_2 = x_0$
else
set $x_1 = x_0$ and $f_1 = f_0$
8. If $|x_2 - x_1|/\Delta x < E$, then
 $\text{root} = (x_1 + x_2)/2$
write the value of root
go to step 9.
else
go to step 6.
9. If $x_2 < b$, then set $a = x_2$ and go to step 3.
10. Stop.

Algorithm 6.6

6.12 SYSTEMS OF NONLINEAR EQUATIONS

A system of equations is a set consisting of more than one equation. A system of n equations in n unknown variables is given below.

$$\begin{aligned}
 f_1(x_1, x_2, \dots, x_n) &= 0 \\
 f_2(x_1, x_2, \dots, x_n) &= 0 \\
 &\dots \\
 &\dots \\
 f_n(x_1, x_2, \dots, x_n) &= 0
 \end{aligned} \tag{6.43}$$

Equation (6.43) requires values for x_1, x_2, \dots, x_n such that they satisfy all the n equations simultaneously. If these equations can be expressed in the form

$$f(x) = a_1x_1 + a_2x_2 + \dots + a_nx_n - c = 0$$

then the system is said to be *linear*. On the other hand, if they involve variables with powers, then the system is said to be *nonlinear*.

For example,

$$x^2 + 2x - y^2 = 2$$

$$x^2 + 3xy = 4$$

is a system of nonlinear equations in two unknowns. These equations can be expressed in the form of equation (6.43) as

$$f(x, y) = x^2 + 2x - y^2 - 2 = 0 \tag{6.44}$$

$$g(x, y) = x^2 + 3xy - 4 = 0 \tag{6.45}$$

Solution of these equations requires values of x and y that could satisfy both of them simultaneously. We will discuss two methods in this section for solving such equations.

Fixed Point Method

One simple approach for solving a system of nonlinear equations is to use the fixed point iteration method. Equations (6.11) and (6.15) can be written in the form

$$x = F(x, y)$$

$$y = G(x, y)$$

We can compute x and y using some initial values of x and y on the right-hand side. The new values of x and y can again be used to compute the next set of x and y values. This process can be repeated till a desired level of accuracy in the computed values is reached. This iterative process can be represented in general form as

$$\begin{aligned}
 x_{i+1} &= F(x_i, y_i) \\
 y_{i+1} &= G(x_i, y_i)
 \end{aligned} \tag{6.46}$$

This can be implemented using the steps given in Algorithm 6.7.

Fixed point method for a system

1. Define iteration functions
 $F(x, y)$ and $G(x, y)$
2. Decide starting points x_0 and y_0 and error tolerance E
3. $x_1 = F(x_0, y_0)$
 $y_1 = G(x_0, y_0)$
4. If $|x_1 - x_0| < E$ and
 $|y_1 - y_0| < E$, then
solution obtained;
go to step 6
5. Otherwise, set
 $x_0 = x_1$
 $y_0 = y_1$
go to step 3
6. Write values of x_1 and y_1
7. Stop

Algorithm 6.7

Example 6.13

Solve the following system of nonlinear equations using fixed point method.

$$x^2 - y^2 = 3$$

$$x^2 + xy = 6$$

Iteration functions of these equations are formed as

$$x = y + \frac{3}{x+y}$$

$$y = \frac{6-x^2}{x}$$

Assume $x_0 = 1$ and $y_0 = 1$

$$x_1 = 2.5$$

$$y_1 = 5$$

$$x_2 = 5.4$$

$$y_2 = -0.1$$

$$x_3 = 0.445$$

$$y_3 = 13$$

The process does not converge. We have to solve the system by forming another set of equations for x and y .

Following an approach similar to the one discussed in Section 6.10, it can be shown that the iteration process converges if the following equations are satisfied.

$$\left| \frac{\partial F}{\partial x} \right| + \left| \frac{\partial G}{\partial x} \right| < 1$$

and

$$\left| \frac{\partial F}{\partial y} \right| + \left| \frac{\partial G}{\partial y} \right| < 1 \quad (6.47)$$

The task of forming appropriate iterative functions $F(x, y)$ and $G(x, y)$ to satisfy the above conditions may become very difficult and, therefore, the fixed point iteration process is rarely used to solve systems of nonlinear equations.

Newton-Raphson Method

The Newton-Raphson method, which was discussed in Section 6.8 for solving single nonlinear equations, can be extended to systems of nonlinear equations. Recall that a first order Taylor series of the form

$$f(x_{i+1}) = f(x_i) + (x_{i+1} - x_i) f'(x) \quad (6.48)$$

was used to derive the Newton iteration formula

$$x_{i+1} = x_i - \frac{f(x)}{f'(x)} \quad (6.49)$$

for solving one equation. For the sake of simplicity, let us again consider a two-equation nonlinear system

$$f(x, y) = 0 \quad g(x, y) = 0$$

First order Taylor series of these equations can be written as

$$f(x_{i+1}, y_{i+1}) = f(x_i, y_i) + (x_{i+1} - x_i) \left| \frac{\partial f_i}{\partial x} \right| + (y_{i+1} - y_i) \left| \frac{\partial f_i}{\partial y} \right| \quad (6.50a)$$

$$g(x_{i+1}, y_{i+1}) = g(x_i, y_i) + (x_{i+1} - x_i) \left| \frac{\partial g_i}{\partial x} \right| + (y_{i+1} - y_i) \left| \frac{\partial g_i}{\partial y} \right| \quad (6.50b)$$

If the root estimates are x_{i+1} and y_{i+1} , then

$$f(x_{i+1}, y_{i+1}) = g(x_{i+1}, y_{i+1}) = 0$$

Substituting this in Eq. (6.50) we get the following two linear equations:

$$\Delta x f_1 + \Delta y f_2 + f = 0 \quad (6.51a)$$

$$\Delta x g_1 + \Delta y g_2 + g = 0 \quad (6.51b)$$

where we denote

$$\Delta x = x_{i+1} - x_i$$

$$\Delta y = y_{i+1} - y_i$$

$$f_1 = \left| \frac{\partial f_i}{\partial x} \right|, \quad f_2 = \left| \frac{\partial f_i}{\partial y} \right|$$

$$g_1 = \left| \frac{\partial g_i}{\partial x} \right|, \quad g_2 = \left| \frac{\partial g_i}{\partial y} \right|$$

$$f = f(x_i, y_i), \quad g = g(x_i, y_i)$$

Solving for x and y , we get

$$\Delta x = -\frac{f \cdot g_2 - g \cdot f_2}{f_1 g_2 - f_2 g_1} = -\frac{Dx}{D} \quad (6.52a)$$

$$\Delta y = -\frac{g \cdot f_1 - f \cdot g_1}{f_1 g_2 - f_2 g_1} = -\frac{Dy}{D} \quad (6.52b)$$

where

$$D = \begin{vmatrix} f_1 & f_2 \\ g_1 & g_2 \end{vmatrix} = f_1 g_2 - g_1 f_2$$

is called the *Jacobian matrix*. From Eq. (6.52a) and (6.52b), we can establish the following recurring relations:

$$x_{i+1} = x_i - \frac{Dx}{D} \quad (6.53a)$$

$$y_{i+1} = y_i - \frac{Dy}{D} \quad (6.53b)$$

Equations (6.53a) and (6.53b) are similar to the single-equation Newton formula and may be called the *two-equation* Newton formula. These equations can be used iteratively and simultaneously to solve for the roots of $f(x, y)$ and $g(x, y)$.

Algorithm 6.8 lists the steps involved in implementing the Newton iteration formula for a two-equation system.

Two equation Newton-Raphson method

1. Define the functions f and g
2. Define the Jacobian elements
 f_1, f_2, g_1 and g_2
3. Decide starting points x_0 and y_0 and error tolerance E .
4. Evaluate f, g, f_1, f_2, g_1, g_2 at (x_0, y_0)
Compute Dx, Dy and D

$$x_1 = x_0 - Dx/D$$

$$y_1 = y_0 - Dy/D$$

(Contd.)

(Contd.)

5. Test for accuracy.
If $|x_1 - x_0| < E$ and
 $|y_1 - y_0| < E$, then
solution obtained;
go to step 7
6. Otherwise, set
 $x_0 = x_1$
 $y_0 = y_1$
go to step 4
7. Write results
8. Stop

Algorithm 6.8**Example 6.1**

Determine the roots of equations

$$x^2 + xy = 6$$

$$x^2 - y^2 = 3$$

using the Newton-Raphson method

Let $F(x, y) = x^2 + xy - 6$

$$G(x, y) = x^2 - y^2 - 3$$

$$f_1 = \frac{\partial F}{\partial x} = 2x + y$$

$$f_2 = \frac{\partial F}{\partial y} = y$$

$$g_1 = \frac{\partial G}{\partial x} = 2x$$

$$g_2 = \frac{\partial G}{\partial y} = -2y$$

Assume the initial guesses as

$$x_0 = 1 \quad \text{and} \quad y_0 = 1$$

Iteration 1

$$f_1 = 3, f_2 = 1$$

$$g_1 = 2, g_2 = -2$$

and therefore

$$D = -6 - 2 = -8$$

The values of functions at x_0 and y_0

$$F = 1^2 + 1 \times 1 - 6 = -4$$

$$G = 1^2 - 1^2 - 3 = -3$$

$$x_1 = 1 - \frac{(-4)(-2) - (-3)(1)}{(-8)} = 2.375$$

$$y_1 = 1 - \frac{(-3)(3) - (-4)(2)}{(-8)} = 0.875$$

Iteration 2

$$f_1 = 2 \times 2.375 + 0.875 = 5.625$$

$$f_2 = 0.875$$

$$g_1 = 4.75$$

$$g_2 = -1.75$$

$$F = (2.375)^2 + (2.375)(0.875) - 6 = 1.71187$$

$$G = (2.375)^2 - (0.875)^2 = 4.8750$$

$$D = (5.625)(-1.75) - (4.75)(0.875)$$

$$= -9.8436 - 4.1563 = -14$$

$$x_2 = 2.375 - \frac{(1.71187)(-1.75) - (4.875)(0.875)}{-14}$$

$$= 2.375 - \frac{(-3.0077) - 4.2656}{-14} = 2.375 - 0.5195$$

$$= 1.8555$$

$$y_2 = 0.875 - \frac{(4.875)(5.625) - (1.71187)(4.75)}{-14}$$

$$= 0.875 - \frac{27.4218 - 8.1638}{-14} = 2.2506$$

Continue further to obtain correct answer.

6.1.3 ROOTS OF POLYNOMIALS

We have seen that the methods discussed so far can also be used for evaluation of the roots of polynomials. However, these methods run into problems when the polynomials contain multiple or complex roots. Polynomials are the most frequently used equations in science and engineering and, therefore, require special attention in terms of evaluation of their roots. In this section, we discuss methods to determine all real (not necessarily distinct) and complex roots of polynomials. These methods are specially designed for polynomials and, therefore, cannot be used for transcendental equations.

We will try to use the following properties of n th degree polynomials:

1. There are n roots (real or complex)
2. A root may be repeated (multiple roots)
3. Complex roots occur in conjugate pairs
4. If n is odd and all the coefficients are real, then there is at least one real root
5. The polynomial can be expressed as

$$p(x) = (x - x_r) q(x)$$

where x_r is a root of $p(x)$ and $q(x)$ is the quotient polynomial of order $n - 1$.

The number of real roots can be obtained using *Descartes' rule of sign*. This rule states that

1. The number of positive real roots is equal (or less than by an even integer) to the number of sign changes in the coefficients of the equation
2. The number of negative real roots is equal (or less than by an even integer) to the number of sign changes in the coefficients, if x is replaced by $-x$

Multiple Roots

A polynomial function contains a multiple root at a point when the function is tangential to the x -axis at that point. For example, the equation

$$x^3 - 7x^2 + 15x - 9 = 0$$

has a double root at $x = 3$ (see Fig. 6.10(a)). The graph is tangent to the x -axis at this point. Similarly, the equation

$$x^4 - 10x^3 + 36x^2 - 56x + 32 = 0$$

has a triple root at $x = 2$ (see Fig. 6.10(b)). Note that the curve crosses the x -axis for odd multiple roots and turns back for the even multiple roots. This means that the bracketing methods will have problems in locating the even multiple roots. Another problem is that both $f(x)$ and its derivative $f'(x)$ become zero at the point of multiple roots. As a consequence, the methods (Newton-Raphson and secant) that use derivatives in the denominator might face the problem of division by zero near the roots.

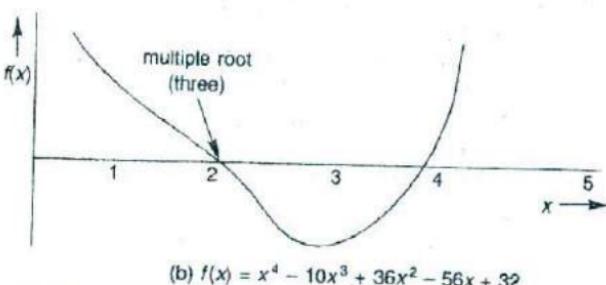
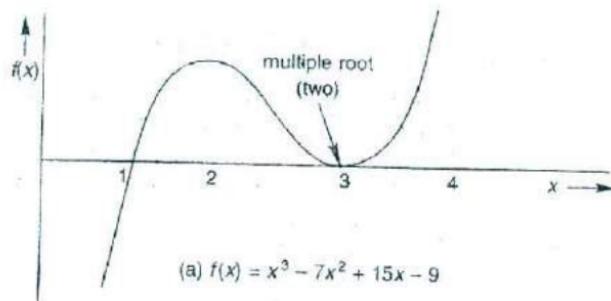


Fig. 6.10 Graph of multiple root polynomials

Deflation and Synthetic Division

We stated that a polynomial of degree n can be expressed as

$$p(x) = (x - x_r) q(x)$$

where x_r is a root of the polynomial $p(x)$ and $q(x)$ is the quotient polynomial of degree $n - 1$. Once a root is found, we can use this fact to obtain a lower degree polynomial $q(x)$ by dividing $p(x)$ by $(x - x_r)$ using a process known as *synthetic division*. The name "synthetic" is used because the quotient polynomial $q(x)$ is obtained without actually performing the division. The activity of reducing the degree of a polynomial is referred to as *deflation*.

The quotient polynomial $q(x)$ can be used to determine the other roots of $p(x)$, because the remaining roots of $p(x)$ are the roots of $q(x)$. When a root of $q(x)$ is found, a further deflation can be performed and the process can be continued until the degree is reduced to one.

Synthetic division is performed as follows:

Let

$$p(x) = \sum_{i=0}^n a_i x^i$$

and

$$q(x) = \sum_{i=0}^{n-1} b_i x^i$$

If $p(x) = (x - x_r) q(x)$, then

$$\begin{aligned} a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \\ = (x - x_r) (b_{n-1} x^{n-1} + b_{n-2} x^{n-2} + \dots + b_1 x + b_0) \end{aligned} \quad (6.54)$$

By comparing the coefficients of like powers of x on both the sides of equation (6.54), we get the following relations between them:

$$\begin{aligned} a_n &= b_{n-1} \\ a_{n-1} &= b_{n-2} - x_r b_{n-1} \end{aligned}$$

$$a_1 = b_0 - x_r b_1$$

$$a_0 = -x_r b_0$$

That is

$$a_i = b_{i-1} - x_r b_i, \quad i = n, n-1, \dots, 0$$

where $b_n = b_{n-1} = 0$.

Then

$b_{i-1} = a_i + x_r b_i, \quad i = n \dots 1$	(6.55)
$b_n = 0$	

Equation (6.55) suggests that we can determine the coefficients of $q(x)$ (i.e., b_{n-1} , b_{n-2} , ..., b_0) from the coefficients of $p(x)$ (i.e., a_n , a_{n-1} , ..., a_1) recursively. Thus, we have obtained the polynomial $q(x)$ without performing any division operation.

Example 6.18

The polynomial equation

$$p(x) = x^3 - 7x^2 + 15x - 9 = 0$$

has a root at $x = 3$. Find the quotient polynomial $q(x)$ such that

$$p(x) = (x - 3) q(x)$$

From $p(x)$, we have

$$a_3 = 1, a_2 = -7, a_1 = 15, \text{ and } a_0 = -9$$

$$b_3 = 0$$

$$b_2 = a_3 + b_3 \times 3 = 1 + 0 = 1$$

$$b_1 = a_2 + b_2 \times 3 = -7 + 3 = -4$$

$$b_0 = a_1 + b_1 \times 3 = 15 + (-12) = 3$$

Thus the polynomial $q(x)$ is

$$x^2 - 4x + 3 = 0$$

Evaluation of all real roots, including multiple roots, using Newton-Raphson method and synthetic division technique for deflation is presented in Section 6.14.

Complex Roots

Computing complex roots is much more complex than computing real multiple roots. Recall that complex roots of polynomials with real coefficients occur in conjugate pairs. This suggests that we should isolate the roots of these types by finding the appropriate quadratic factors of the original polynomial (rather than linear factors). Quadratic factors can be obtained by using the process of synthetic division.

Let us assume that

$$h(x) = x^2 - ux - v$$

is an "approximate" quadratic factor of $p(x)$. Then

$$\frac{p(x)}{h(x)} = q(x) + \frac{r(x)}{h(x)} \quad (6.56)$$

where $q(x)$ is the quotient polynomial of degree $(n - 2)$ and $r(x)$ is the remainder. Note that if $h(x)$ is an exact quadratic factor of $p(x)$, then $r(x)$ would be zero. Equation (6.56) can be rewritten as

$$\begin{aligned} p(x) &= q(x) h(x) + r(x) \\ &= q(x) (x^2 - ux - v) + r(x) \end{aligned} \quad (6.57)$$

Since $q(x)$ is a quotient polynomial, it would be of the form

$$q(x) = b_n x^{n-2} + b_{n-1} x^{n-3} + \dots + b_2 \quad (6.58)$$

Let us assume that the remainder $r(x)$ takes the form

$$r(x) = b_1(x - u) + b_0 \quad (6.59)$$

(The form of $r(x)$ is chosen for the convenience of manipulation).

The objective is to determine the factors u and v such that $r(x)$ becomes zero and, therefore, $h(x)$ becomes an exact factor of $p(x)$ given below.

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (6.60)$$

Substituting Eqs (6.58), (6.59) and (6.60) in Eq. (6.57) and comparing coefficients, we obtain the following relations:

$$b_n = a_n$$

$$b_{n-1} = a_{n-1} + u b_n$$

$$b_{n-2} = a_{n-2} + u b_{n-1} + v b_n$$

$$b_1 = a_1 + u b_2 + v b_3$$

$$b_0 = a_0 + u b_1 + v b_2$$

This can be expressed in general form as

$$b_i = a_i + u b_{i+1} + v b_{i+2} \quad (6.61)$$

where $i = n, n-1, \dots, 0$

$$b_{n+1} = b_{n+2} = 0$$

Note that all the coefficients b_i are functions of u and v which are unknown.

It is clear that $h(x)$ is a factor of $p(x)$ if and only if

$$\begin{aligned} b_1 &= a_1 + u b_2 + v b_3 = 0 \\ b_0 &= a_0 + u b_1 + v b_2 = 0 \end{aligned} \quad (6.62)$$

Note that Eq. (6.62) is a system of two nonlinear equations in two unknowns, u and v . These equations can be solved by using Newton's method discussed in Section 6.14.

Once the values of u and v are known, the roots of the equation

$$x^2 - ux - v$$

can be easily determined using the formula

$$x = \frac{u \pm \sqrt{u^2 + 4v}}{2}$$

The process can be repeated for the quotient polynomial till it becomes either a quadratic or linear polynomial which can be solved for their roots.

Purification of Roots

Purification, as the name indicates, is the process of refining the roots that do not satisfy the required accuracy conditions. These roots may be used again for testing the original problem and improving their approximations.

The Newton-Raphson method is a popular one used for purification of roots. The values of the roots obtained through other methods are used as "initial" input values to the Newton method.

6.14 MULTIPLE ROOTS BY NEWTON'S METHOD

As discussed earlier, we can locate all real roots of a polynomial by repeatedly applying Newton-Raphson method and polynomial deflation to obtain polynomials of lower and lower degrees. Algorithm 6.9 gives a step-by-step procedure to achieve this.

Note that the deflation process is performed $(n - 1)$ times where n is the degree of the given polynomial. After $(n - 1)$ deflations, the quotient is a linear polynomial of type

$$a_1 x + a_0 = 0$$

and therefore the final root is given by

$$x_r = -\frac{a_0}{a_1}$$

Evaluation of Multiple Roots

1. Obtain degree and coefficients of polynomial (n and a_i)
2. Decide an initial estimate for the first root (x_0) and error criterion

Do while $n > 1$

3. Find the root using Newton-Raphson algorithm:

$$x_r = x_0 - \frac{f(x_0)}{f'(x_0)}$$

4. Root (n) = x_r
5. Deflate the polynomial using synthetic division algorithm and make the factor polynomial as the new polynomial of order $n - 1$
6. Set $x_0 = x_r$ (initial value for next root)

End of Do

7. Root (1) = $-a_0/a_1$
8. Stop

Algorithm 6.9

Program MULTIR

The program MULTIR locates all real roots of a polynomial by repeatedly applying the Newton-Raphson method as shown in Algorithm 6.9. To achieve this, the program employs two subroutines: first, the subroutine

NEWTON to find a real root of the polynomial, and second, the subroutine DFLAT to reduce the polynomial degree by one. This process is continued till the degree of the polynomial is reduced to one. This is implemented by the DO loop DO 200 I = N, 2, -1.

The subroutine NEWTON, while evaluating a root, also implements a test for accuracy of the root obtained. In case the required accuracy is not obtained within a specified number of iterations, the execution stops after giving an appropriate message.

```

* -----
*      PROGRAM 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
* -----
REAL A,X0,XR,ROOT,EPS
INTEGER N,MAXIT,STATUS
PARAMETER( EPS=0.000001, MAXIT=50 )
DIMENSION A(11), ROOT(10)

WRITE(*,*)
WRITE(*,*)  ' EVALUATION OF MULTIPLE ROOTS'
WRITE(*,*)

WRITE(*,*)  'Input N, the degree of polynomial'
READ(*,*)  N
WRITE(*,*)  'Input poly coefficients, A(1) to A(N+1)'
READ(*,*)  (A(I), I=1, N+1)
WRITE(*,*)  'Input initial guess X'

```

```

READ(*,*) X0
WRITE(*,*)
DO 200 I = N, 2, -1
*      Find I_th root
      CALL NEWTON(N,A,X0,EPS,MAXIT,STATUS,XR)
      IF (STATUS .EQ. 2) THEN
          DO 100 J = N, I+1, -1
100      WRITE(*,*) 'ROOT',J,' =', ROOT(J)
          WRITE(*,*) 'Next root does not converge in'
          WRITE(*,*) MAXIT, ' iterations'
          WRITE(*,*)
          STOP
      ENDIF
      ROOT(I) = XR
*      Deflate the polynomial by division (X - XR)
      CALL DFLAT(N,A,XR)
      X0 = XR
*      Proceed to find next root
200 CONTINUE
*      Compute the last root
      ROOT(1) = - A(1)/A(2)
*      Write results
      WRITE(*,*) 'ROOTS OF POLYNOMIAL ARE:'
      WRITE(*,*)
      DO 300 I = 1, N
          WRITE(*,*) 'ROOT',I,' =', ROOT(I)
300 CONTINUE
      WRITE(*,*)
      STOP
END
* ----- End of main program MULTIR -----
* -----
*      SUBROUTINE NEWTON(N,A,X0,EPS,MAXIT,STATUS,XR)
* -----
*      Subroutine
*      This subroutine finds a root of the polynomial
*      using the Newton-Raphson method
* -----
*      Arguments
*      Input
*      N - Degree of polynomial
*      A - Array of polynomial coefficients

```

```

* X0 - Initial guess for a root
* EPS - Error bound
* MAXIT - Maximum iterations permitted
* Output
* STATUS - Solution status
* XR - Root obtained by Newton method
* -----
* Local Variables
* COUNT - Number of iterations performed
* FX - Value of polynomial function at X0
* FDX - Value of function derivative at X0
* -----
* Functions invoked
* ABS
* -----
* Subroutines called
* NIL
* -----

```

```

REAL A,X0,EPS,XR,ABS
INTEGER N,MAXIT,STATUS
INTRINSIC ABS
DIMENSION A(11)

COUNT = 1

* Compute the value of function at X0
100 FX = A(N+1)
    DO 111 I = N, 1, -1
        FX = FX * X0 + A(I)
111 CONTINUE
* Compute the value of derivative at X0
    FDX = A(N+1) * N
    DO 222 I = N, 2, -1
        FDX = FDX * X0 + A(I) * (I-1)
222 CONTINUE
* Compute a root XR
    XR = X0 - FX/FDX
* Test for accuracy
    IF(ABS((XR-X0)/XR).LE.EPS) THEN
        STATUS = 1
        RETURN
    ENDIF
* Test for convergence
    IF(COUNT .LT. MAXIT) THEN
        X0 = XR
        COUNT = COUNT + 1
        GOTO 100
    ENDIF

```

```

    ELSE
        STATUS = 2
        RETURN
    ENDIF

    END
* ----- End of subroutine NEWTON-----
*
* -----
* SUBROUTINE DFLAT(N,A,XR)
*
* Subroutine
*   This subroutine reduces the degree of polynomial
*   by one using synthetic division
*
* -----
* Arguments
* Input
*   N - Degree of polynomial
*   A - Array of coefficients of input polynomial
*   XR - A root of the input polynomial
* Output
*   A - coefficients of the reduced polynomial
*
* Local Variables
*   B
*
* -----
* Functions invoked
*   NIL
*
* -----
* Subroutines called
*   NIL
*
* -----
REAL A,B,XR
INTEGER N
DIMENSION A(11), B(11)

* Evaluate the coefficients of the reduced polynomial
B(N+1) = 0
DO 1 I = N, 1, -1
    B(I) = A(I+1) + XR * B(I+1)
1 CONTINUE

* Change coefficients from B array to A array
DO 2 I = 1, N+1
    A(I) = B(I)
2 CONTINUE
RETURN
END
* ----- End of subroutine DFLAT

```

Test Results of MULTIR The program was tested for evaluating the roots of the equation

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

The results of a test run are given below:

EVALUATION OF MULTIPLE ROOTS

Input N, the degree of polynomial

2

Input poly coefficients, A(1) to A(N+1)

2 -3 1

Input initial guess X

0

ROOTS OF POLYNOMIAL ARE:

ROOT 1 = 2.0000000

ROOT 2 = 1.0000000

Stop - Program terminated.

6.15 COMPLEX ROOTS BY BAIRSTOW METHOD

We have discussed in Section 6.13 that complex roots of a polynomial equation can be found by using its quadratic factors. We have also seen that if the polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

is divided by quadratic factor

$$h(x) = x^2 - ux - v$$

then the result is a polynomial

$$q(x) = b_n x^{n-2} + b_{n-1} x^{n-3} + \dots + b_2$$

with a remainder

$$r(x) = b_1(x - u) + b_0$$

The values of coefficients b_i are given by the following recurrence formula:

$b_n = a_n$	(6.63)
$b_{n-1} = a_{n-1} + ub_n$	
$b_i = a_i + ub_{i+1} + vb_{i+2}$, (for $i = n - 2$ to 0)	

We know that in order to make $h(x)$ an exact factor of $p(x)$, $r(x)$ should be zero. This implies that

$$b_1 = b_0 = 0$$

We know from the above recurrence formula that

$$b_1 = a_1 + ub_2 + vb_3 = 0$$

$$b_0 = a_0 + ub_1 + vb_2 = 0$$

The problem now is to find the solution of the system of equations

$$\begin{aligned} b_1(u, v) &= 0 \\ b_0(u, v) &= 0 \end{aligned} \quad (6.64)$$

Remember, these are nonlinear equations because coefficients b_i are functions of u and v . The strategy used to solve the system of Eqs. (6.64) is known as *Bairstow's method*. The method is similar to the Newton-Raphson approach for solving a two-equation system (discussed in Section 6.12). Using the Taylor series expansion (recall Eq. (6.51)), it can be shown that

$$\begin{aligned} \frac{\partial b_1}{\partial u} \Delta u + \frac{\partial b_1}{\partial v} \Delta v &= -b_1 \\ \frac{\partial b_0}{\partial u} \Delta u + \frac{\partial b_0}{\partial v} \Delta v &= -b_0 \end{aligned} \quad (6.65)$$

To solve these equations, we need partial derivatives of b_i coefficients. Differentiating Eq. (6.63) with respect to u , we get

$$\begin{aligned} \frac{\partial b_i}{\partial u} &= b_{i+1} + u \frac{\partial b_{i+1}}{\partial u} + v \frac{\partial b_{i+2}}{\partial u}, \quad i = n-2 \text{ to } 0 \\ \frac{\partial b_n}{\partial u} &= 0 \\ \frac{\partial b_{n-1}}{\partial u} &= b_n + u \frac{\partial b_n}{\partial u} = b_n \end{aligned} \quad (6.66)$$

For convenience, let us denote

$$c_i = \frac{\partial b_i}{\partial u}$$

Then, we have

$$\begin{aligned} c_n &= 0 \\ c_{n-1} &= b_n \\ c_i &= b_{i+1} + u c_{i+1} + v c_{i+2}, \quad i = n-2 \text{ to } 0 \end{aligned}$$

(6.67)

We need the following coefficients of c_i

$$\frac{\partial b_1}{\partial u} = c_1$$

$$\frac{\partial b_0}{\partial u} = c_0$$

c_1 and c_0 can be evaluated recursively using Eq. (6.67). Now, differentiating Eq. (6.63) with respect to v ,

$$\begin{aligned}\frac{\partial b_n}{\partial v} &= 0 \\ \frac{\partial b_{n-1}}{\partial v} &= u \frac{\partial b_n}{\partial v} = 0 \\ \frac{\partial b_i}{\partial v} &= b_{i+2} + u \frac{\partial b_{i+1}}{\partial v} + v \frac{\partial b_{i+2}}{\partial v}, \quad i = n-2 \text{ to } 0\end{aligned}\tag{6.68}$$

If we denote

$$d_i = \frac{\partial b_{i-1}}{\partial v}$$

Then, we have

$$\begin{aligned}d_n &= \frac{\partial b_{n-1}}{\partial v} = 0 \\ d_{n-1} &= \frac{\partial b_{n-2}}{\partial v} = b_n \\ d_i &= \frac{\partial b_{i-1}}{\partial v} = b_{i+1} + u \frac{\partial b_i}{\partial v} + v \frac{\partial b_{i+1}}{\partial v}\end{aligned}$$

That is,

$$d_i = b_{i+1} + u d_{i+1} + v d_{i+2}, \quad i = n-2 \text{ to } 0\tag{6.69}$$

We need the following coefficients of d_i

$\frac{\partial b_1}{\partial v} = d_2$
$\frac{\partial b_0}{\partial v} = d_1$

Again, d_2 and d_1 can be recursively evaluated using equation (6.69).

If we compare Eqs (6.67) and (6.68), it is clear that d_i values are identical to c_i values. That is

$$d_i = c_i \quad \text{for } i = n \text{ to } 0$$

Then, $d_2 = c_2$ and $d_1 = c_1$. This implies that we need not compute the coefficients d_i .

Substituting for partial derivatives in terms of c values in Eq. (6.65) we get

$$\begin{aligned}c_1 \Delta u + c_2 \Delta v &= -b_1 \\ c_0 \Delta u + c_1 \Delta v &= -b_0\end{aligned}$$

Then,

$$\Delta u = -\frac{b_1 c_1 - b_0 c_2}{c_1^2 - c_0 c_2}$$

$$\Delta v = -\frac{b_0 c_1 - b_1 c_0}{c_1^2 - c_0 c_2}$$

Now, given the initial values of u_0 and v_0 , we can estimate the values of u and v using the following recurring relations

$$u_{i+1} = u_i - \frac{b_1 c_1 - b_0 c_2}{c_1^2 - c_0 c_2} \quad (6.70a)$$

$$v_{i+1} = v_i - \frac{b_0 c_1 - b_1 c_0}{c_1^2 - c_0 c_2} \quad (6.70b)$$

Note that the main task in Bairstow's method is the evaluation of b_i and c_i coefficients using the Eqs (6.63) and (6.67). Algorithm 6.10 lists the steps to implement Bairstow's method.

Complex roots by Bairstow's method

1. Get polynomial parameters (n and a_i values)
2. Decide initial estimates, u_0 and v_0 and stopping criterion

While $n > 2$: Do

3. Compute b_i coefficients
4. Compute c_i coefficients
5. Compute

$$D = c_1 \times c_1 - c_0 \times c_2$$

$$\Delta u = -(b_1 \times c_1 - b_0 \times c_2)/D$$

$$\Delta v = -(b_0 \times c_1 - b_1 \times c_0)/D$$

$$u = u_0 + \Delta u$$

$$v = v_0 + \Delta v$$

6. Test for accuracy of u and v . If accuracy is ok, then
solution obtained;
go to step 8
7. Otherwise, set

$$u_0 = u$$

$$v_0 = v$$

go to step 3

8. Find (complex) roots of $x^2 - ux - v = 0$
write results

9. Set the coefficients of factor polynomial as a_i

$$n = n - 2$$

$$a_i = b_{i+2} \text{ (for } i = n \text{ to } 0)$$

10. Set next values for u_0 and v_0

$$u_0 = u$$

$$v_0 = v$$

End of While-Do

(Contd.)

(Contd.)

11. If $n = 2$, then

$$u = -a_1/a_2,$$

$$v = -a_0/a_2,$$

find (complex) roots

write results

else

$$\text{single root} = -a_0/a_1$$

write results

12. Stop

Algorithm 6.10**Example 6.16**

Obtain the quadratic factor of the polynomial

$$p(x) = x^3 + x + 10$$

using Bairstow's method with starting values $u = +1.8$ and $v = -1$

Given

$$a_3 = 1, a_2 = 0, a_1 = 1, a_0 = 10$$

Then

$$b_3 = 1$$

$$b_2 = a_2 + ub_3 = 0 + (+1.8) \times 1 = +1.8$$

$$b_1 = a_1 + ub_2 + vb_3 = 1 + (+1.8)(+1.8) + (-4)(1) = 0.24$$

$$\begin{aligned} b_0 &= a_0 + ub_1 + vb_2 \\ &= 10 + (+1.8)(0.24) + (-4)(1.8) = 3.232 \end{aligned}$$

$$c_3 = 0$$

$$c_2 = 1$$

$$c_1 = b_2 + uc_2 + vc_3 = +1.8 + (+1.8)(1) + (-4 \times 0) = +3.6$$

$$\begin{aligned} c_0 &= b_1 + uc_1 + vc_2 \\ &= 0.24 + (+1.8)(+3.6) + (-4 \times 1) = 3.72 \end{aligned}$$

$$D = c_1^2 - c_0 c_2 = (+3.6)^2 - 3.72 \times 1 = 9.24$$

$$\Delta u = \frac{b_1 c_1 - c_0 c_2}{D}$$

$$= -\frac{(0.24)(3.6) - (3.232) \times 1}{9.24} = 0.2563$$

$$\Delta v = -\frac{b_0 c_1 - b_1 c_0}{D}$$

$$= -\frac{(3.232)(3.6) - (0.24)(3.72)}{9.24} = -1.1616$$

$$u = 1.8 + 0.2563 = 2.0563$$

$$v = -4 - 1.1616 = -5.1616$$

Note that the true values of u and v are 2 and -5 respectively. Therefore, the estimated values are close to the true values. These values can be refined by further iterations.

Program COMPR

The program COMPR can locate all the real and complex roots of an equation. The program COMPR uses Bairstow's method to achieve this. The program logic is detailed in the Algorithm 6.10 and implemented as shown in Fig. 6.11.

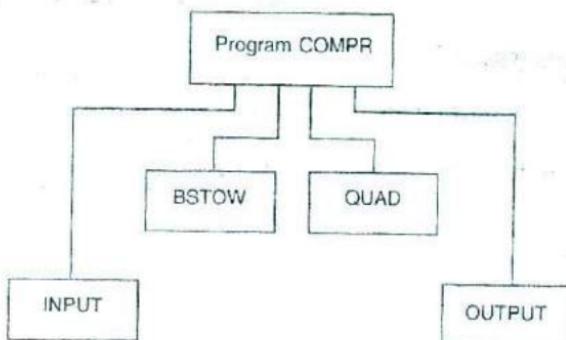


Fig. 6.11 Implementation of algorithm 6.10 to evaluate complex roots

The subprogram INPUT obtains data for polynomial and initial values of the quadratic coefficients. The subprogram BSTOW finds the quadratic factor using multivariable Newton's method and also obtains the reduced polynomial. The subprogram QUAD solves the quadratic equation, the details of which are supplied by BSTOW through the main program COMPR. Finally, the subroutine OUTPUT displays the roots of the quadratic equation.

```

PROGRAM COMPR
*
* Main program
* This program locates all the roots, both real
* and complex, using Bairstow's 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 *
* -----
INTEGER N, TYPE
REAL A,B,U0,V0,U,V,X1,X2,EPS,D0,D1,D2
PARAMETER( EPS = 1.E-6 )
DIMENSION A(11),B(11)

WRITE(*,*)
WRITE(*,*) 'EVALUATION OF COMPLEX ROOTS'
WRITE(*,*)

CALL INPUT(N,A,U0,V0)
100 IF(N.GT.2) THEN
* ---obtain a quadratic factor
    CALL BSTOW(N,A,B,U0,V0,U,V,EPS)
    D2 = 1
    D1 = -U
    D0 = -V
*---find roots of the quadratic factor
    CALL QUAD(D2,D1,D0,X1,X2,TYPE)
*---print the roots
    CALL OUTPUT(N,TYPE,X1,X2)
*---set the coefficients of the factor polynomial
    N = N-2
    DO 200 I = 1, N+1
        A(I) = B(I+2)
200 CONTINUE
*---set initial values for next quadratic factor
    U0 = U
    V0 = V
    GOTO 100
ENDIF
IF(N.EQ.2) THEN
*---polynomial is a quadratic one
    CALL QUAD(A(3),A(2),A(1),X1,X2,TYPE)

```

```

    CALL OUTPUT(N,TYPE,X1,X2)
ELSE
*----last root of an odd order polynomial
  ROOT = - A(1)/A(2)
  WRITE(*,*) 
  WRITE(*,*) 'Final root = ', ROOT
  WRITE(*,*) 

ENDIF
STOP
END

* -----End of main program COMPR -----
*
* -----
* SUBROUTINE INPUT(N,A,U0,V0)
*
* -----
* Subroutine
*   This subroutine reads polynomial details and
*   initial values of the quadratic coefficients
*
* -----
* Arguments
* Input
*   NIL
* Output
*   N - Degree of polynomial.
*   A - Polynomial coefficients
*   U0,V0 - Initial values of the quadratic factor
*
* -----
* Local Variables
* NIL
*
* -----
* Functions invoked
* NIL
*
* -----
* Subroutines called
* NIL
*
* -----
REAL A,U0,V0
INTEGER N
DIMENSION A(11)

WRITE(*,*) 'Input degree of polynomial (N)'
  READ(*,*) N
WRITE(*,*) 'Input polynomial coefficients A(N+1)
              to A(1)'
DO 11 I = N+1, 1, -1
  READ(*,*) A(I)
11 CONTINUE
WRITE(*,*) 'Give initial values U0 and V0'

```

```

READ(*,*), U0,V0
RETURN
END

* ----- End of subroutine INPUT -----
*
* -----
*      SUBROUTINE BSTOW(N,A,B,U0,V0,U,V,EPS)
*
* -----
* Subroutine
* This subroutine finds the quadratic factor using
* multivariable Newton's method and also finds the
* reduced polynomial
*
* -----
* Arguments
* Input
*   N - Degree of polynomial
*   A - Polynomial coefficients
*   U0,V0 - Initial guess for the coefficients
*           of the quadratic factor
*   EPS - Error bound
* Output
*   U,V - Computed coefficients of the quadratic
*         factor
*   B - Coefficients of the reduced polynomial
*
* Local Variables
*   D,DELU,DELV,C
*
* Functions invoked
*   ABS
*
* Subroutines called
*   NIL
*
* -----
INTEGER N
REAL A,B,U0,V0,U,V,EPS,D,DELU,DELV,C
INTRINSIC ABS
DIMENSION A(11), B(11), C(11)

COUNT = 1
100 B(N+1) = A(N+1)
      B(N) = A(N) + U0 * B(N+1)
      DO 111 I = N-1, 1, -1
          B(I) = A(I) + U0 * B(I+1) + V0 * B(I+2)
111 CONTINUE

      C(N+1) = 0
      C(N) = B(N+1)

```

```

DO 222 I = N-1, 1, -1
    C(I) = B(I+1) + U0 * C(I+1) + V0 * C(I+2)
222 CONTINUE

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(1))/D
U = U0 + DELU
V = V0 + DELV
IF( ABS(DELU/U).LE.EPS .AND. ABS(DELV/V).LE.EPS ) THEN
    RETURN
ENDIF

IF(COUNT .LT. 100) THEN
    U0 = U
    V0 = V
    COUNT = COUNT + 1
    GOTO 100
ELSE
    WRITE(*,*) ''
    WRITE(*,*)      'NO CONVERGENCE IN 100 ITERATIONS'
    WRITE(*,*) ''
    STOP
ENDIF
END

```

* ----- End of subroutine BSTOW ----- *

* ----- SUBROUTINE QUAD(A,B,C,X1,X2,TYPE) ----- *

* Subroutine
* This subroutine solves a quadratic equation of
* 2
* type AX² + BX + C
*

* Arguments
* Input

* A,B,C - Coefficients of the quadratic equation

* Output

* X1,X2 - Roots of the quadratic equation

* TYPE - Type of roots

* ----- Local Variables

* Q

* ----- Functions invoked

* SQRT,ABS

```

* Subroutines called
* NIL
*
* -----
INTEGER TYPE, IMAGE, EQUAL, UNEQUAL
REAL A,B,C,X1,X2,SQRT,ABS
INTRINSIC SQRT,ABS
PARAMETER( IMAGE = 1, EQUAL = 2, UNEQL = 3)
Q = B * B - 4 * A * C
IF(Q.LT.0) THEN
* ----- Roots are complex
    X1 = -B/(2*A)
    X2 = SQRT(ABS(Q))/(2*A)
    TYPE = IMAGE
ELSE IF(Q.EQ.0) THEN
* ----- 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)
    TYPE = UNEQL
ENDIF
RETURN
END

```

* ----- End of subroutine QUAD ----- *

SUBROUTINE OUTPUT(N,TYPE,X1,X2)

* Subroutine
* This subroutine displays the roots of the
* quadratic equation
*

* Arguments
* Input
* N - Degree of the polynomial from which
* the quadratic factor was obtained
*

* TYPE - Type of roots
*

* X1,X2 - Roots of the quadratic factor
*

* Output
*

* NIL
*

* Local Variables
*

* NIL
*

```

* -----
* Functions invoked
*   NIL
* -----
* Subroutines called
*   NIL
* -----
      INTEGER N, TYPE, IMAGE, EQUAL, UNEQL
      REAL X1,X2
      PARAMETER( IMAGE = 1, EQUAL = 2, UNEQL = 3 )
      WRITE(*,*)
      WRITE(*,*) 'Roots of quadratic factor at n = ',N
      WRITE(*,*)

      IF(TYPE .EQ. IMAGE) THEN
          WRITE(*,*) 'Root1 = ', X1, ' + ', X2,'j'
          WRITE(*,*) 'Root2 = ', X1, ' - ', X2,'j'
      ELSE IF(TYPE .EQ. EQUAL) THEN
          WRITE(*,*) 'Root1 = ', X1
          WRITE(*,*) 'Root2 = ', X1
      ELSE
          WRITE(*,*) 'Root1 = ', X1
          WRITE(*,*) 'Root2 = ', X2
      ENDIF

      RETURN
      END
* ----- End of subroutine OUTPUT -----

```

Test Results of COMPR

```

EVALUATION OF COMPLEX ROOTS
Input degree of polynomial (N)
3
Input polynomial coefficients A(N+1) to A(1)
1
0
1
10
Give initial values U0 and V0
1.8 -4.0
Roots of quadratic factor at n =      3
Root1 = 1.0000000 + 2.0000000j
Root2 = 1.0000000 - 2.0000000j
Final root = -2.0000000
Stop - Program terminated.

```

6.16 MULLER'S METHOD

Muller's method is an extension of the secant method. Muller's method uses a quadratic curve passing through three points $(x_1, f(x_1))$, $(x_2, f(x_2))$ and $(x_3, f(x_3))$ as shown in Fig. 6.12 to estimate a root of $f(x)$. One of the roots of the quadratic polynomial $p(x)$ is taken as an approximate value of the root of $f(x)$. As illustrated in Fig. 6.12, the point x_4 , one of the roots of $p(x)$, is assumed as the next approximation for the root of $f(x)$.

We can write the quadratic polynomial $p(x)$ in the form

$$p(x) = a_0 + a_1(x - c) + a_2(x - c)^2 \quad (6.71)$$

Equation (6.71) is known as the shifted-power form of the polynomial and c is a constant known as the centre. If we choose $c = x_3$ then Eq. (6.71) becomes

$$p(x) = a_0 + a_1(x - x_3) + a_2(x - x_3)^2 \quad (6.72)$$

Since x_4 is a root of $p(x)$, at $x = x_4$, $p(x) = 0$ and, therefore, Eq. (6.72) becomes

$$a_2(x_4 - x_3)^2 + a_1(x_4 - x_3) + a_0 = 0$$

Solving the quadratic equation for $(x_1 - x_3)$ we get

$$x_4 - x_3 = \frac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2a_0}} \quad (6.73)$$

This is one of the forms of quadratic formula, chosen here to minimise error due to any subtractive cancellation. The constants a_0 , a_1 and a_2 can be obtained in terms of known function values $f(x_1)$, $f(x_2)$, and $f(x_3)$ as follows:

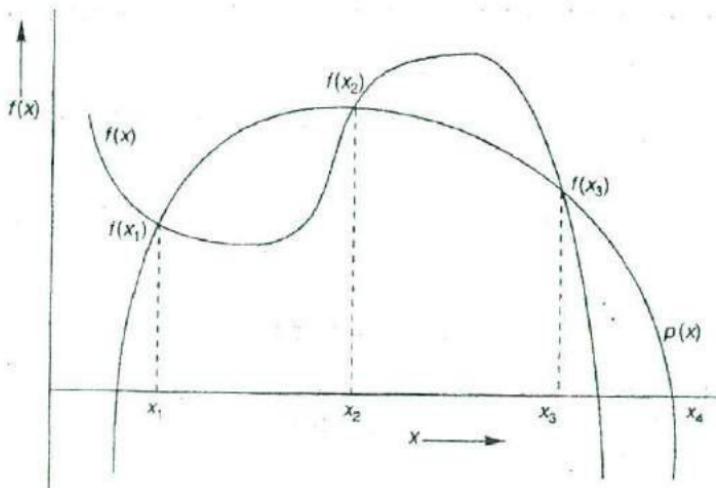


Fig. 6.12 Illustration of Muller's method

At $x = x_1, x_2$ and x_3 , we have

$$a_2(x_1 - x_3)^2 + a_1(x_1 - x_3) + a_0 = p(x_1) = f(x_1)$$

$$a_2(x_2 - x_3)^2 + a_1(x_2 - x_3) + a_0 = p(x_2) = f(x_2)$$

$$a_2(x_3 - x_3)^2 + a_1(x_3 - x_3) + a_0 = p(x_3) = f(x_3)$$

Letting $h_1 = x_1 - x_3$ and $h_2 = x_2 - x_3$, and denoting $f_i = f(x_i)$, we get

$$a_2 h_1^2 + a_1 h_1 + a_0 = f_1$$

$$a_2 h_2^2 + a_1 h_2 + a_0 = f_2$$

$$0 + 0 + a_0 = f_3$$

Since $a_0 = f_3$, we can obtain a_1 and a_2 by solving the equations

$$a_2 h_1^2 + a_1 h_1 = f_1 - f_3 = d_1$$

$$a_2 h_2^2 + a_1 h_2 = f_2 - f_3 = d_2$$

This results in

$$a_1 = \frac{d_2 h_1^2 - d_1 h_2^2}{h_1 h_2 (h_1 - h_2)}$$

$$a_2 = \frac{d_1 h_2 - d_2 h_1}{h_1 h_2 (h_1 - h_2)}$$

Equation (6.73) can be written as

$$x_4 = x_3 + h_4$$

where

$$h_4 = \frac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2 a_0}}$$

The sign in the denominator of h_4 is chosen such that h_4 is as small in magnitude as possible so that x_3 is close to x_4 . That is, the magnitude of $(a_1 \pm \sqrt{a_1^2 - 4a_2 a_0})$ should be large.

This process is then repeated using x_2, x_3 and x_4 as the initial three points to obtain the next approximation x_5 .

$$x_5 = x_4 + h_5$$

The process is continued till $f(x_i)$ is within the specified accuracy. Algorithm 6.11 lists the steps in detail for computing a root by Muller's method

Muller's Method

- Decide the initial three points and stopping criterion
- Compute $f_1 = f(x_1), f_2 = f(x_2), f_3 = f(x_3)$
- Compute

$$h_1 = x_1 - x_3, h_2 = x_2 - x_3$$

$$d_1 = f_1 - f_3, d_2 = f_2 - f_3$$

(Contd.)

(Contd.)

4. Compute parameters a_0, a_1, a_2
 $a_0 = f_3$

$$a_1 = \frac{d_2 h_1^2 - d_1 h_2^2}{h_1 h_2 (h_1 - h_2)}$$

$$a_2 = \frac{d_1 h_2 - d_2 h_1}{h_1 h_2 (h_1 - h_2)}$$

5. Compute h

$$h = \frac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2 a_0}}$$

(choose the sign in the denominator such that its magnitude is the largest. That is, if a_1 is positive use + sign, otherwise, - sign)

6. Compute $x_4 = x_3 + h$
7. Compute $f_4 = f(x_4)$
8. If $f(x_4)$ satisfies the given criterion, then
root is obtained,
go to step 10
9. Otherwise, set

$$x_1 = x_2, x_2 = x_3, x_3 = x_4 \text{ and}$$

$$f_1 = f_2, f_2 = f_3, f_3 = f_4, \text{ then}$$
go to step 3
10. Write the value of root (x_4)
11. Stop

Algorithm 6.11

Example 6.17

Solve the Leonardo equation

$$f(x) = x^3 + 2x^2 + 10x - 20 = 0$$

by Muller's method

Let us assume the three starting points as

Iteration 1

$$x_1 = 0, x_2 = 1, x_3 = 2$$

$$f_1 = -20$$

$$f_2 = -7$$

$$f_3 = 16$$

$$h_1 = x_1 - x_3 = -2$$

$$h_2 = x_2 - x_3 = -1$$

$$d_1 = f_1 - f_3 = -36$$

$$d_2 = f_2 - f_3 = -23$$

$$\begin{aligned} D &= h_1 h_2 (h_1 - h_2) \\ &= 2(-2 + 1) = -2 \end{aligned}$$

$$a_1 = \frac{(-23)(-2)^2 - (-36)(-1)^2}{-2} = 28$$

$$a_2 = \frac{(-36)(-1) - (-23)(-2)}{-2} = 5$$

$$\begin{aligned} h &= \frac{-2 \times 16}{28 \pm \sqrt{28^2 - 4(5)(16)}} \\ &= -\frac{32}{49.540659} \quad (\text{choosing + sign}) \\ &= -0.645934 \\ x_4 &= x_3 + h = 1.3540659 \end{aligned}$$

Iteration 2

$$x_1 = 1$$

$$x_2 = 2$$

$$x_3 = 1.3540659$$

$$h_1 = x_1 - x_3 = -0.3540659$$

$$h_2 = x_2 - x_3 = 0.645934$$

$$f_1 = -7$$

$$f_2 = 16$$

$$f_3 = f(1.3540659) = -0.3096797$$

$$d_1 = f_1 - f_3 = -6.6903202$$

$$d_2 = f_2 - f_3 = 16.3096797$$

$$D = h_1 h_2 (h_1 - h_2) = 0.2287031$$

$$a_1 = \frac{d_2 h_1^2 - d_1 h_2^2}{D} = 21.145459$$

$$a_2 = \frac{d_1 h_2 - d_2 h_1}{D} = 6.3540717$$

$$a_0 = f_3 = -0.3096797$$

$$h = \frac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2 a_0}} = \frac{0.6193594}{42.47622} = 0.0145813$$

$$x_4 = x_3 + h = 1.3686472$$

This process can be continued to obtain better accuracy. The correct answer is 1.368808107.

Complex Roots

Note that, in Example 6.17, we obtained a real root of the polynomial. In some cases, we may encounter complex approximations while solving Eq. (6.73). However, in such cases, the imaginary component will normally be small in magnitude and it can be neglected.

In case we are interested in the complex roots as well, we can obtain these by implementing the Muller algorithm using complex arithmetic (which is supported by FORTRAN).

Multiple Roots

The algorithm can be modified to find more than one root by incorporating the *deflation* procedure using the following equation as discussed in Section 6.13:

$$f'(x) = \frac{f(x)}{x - z_1}$$

Program MULLER

Design and development of a program to implement Muller's method is left to the reader as an exercise.

6.17 SUMMARY

In this chapter, we defined various forms of nonlinear equations and stated a number of approaches to find the roots of such equations. We discussed in detail the following iterative methods to evaluate a root:

- Bisection method (also known as interval halving method)
- False position method (also called linear interpolation method)
- Newton-Raphson method
- Secant method
- Fixed point method (also known as method of direct substitution)
- Muller's method

We also discussed the solution of a system of nonlinear equations using

- Fixed point method
- Newton-Raphson method

We further presented two methods to find the roots of polynomials:

- Newton-Raphson method with synthetic division
- Bairstow's method (for real as well as complex roots)

We discussed the process of converging of iterative methods and proved that

- Newton-Raphson method converges with order of 2
- Bisection method converges linearly
- False position method is linearly convergent
- Secant method follows superlinear convergence

We presented FORTRAN programs and test results for the following methods:

- Bisection method
- False position method
- Newton Raphson method (single root)
- Secant method
- Fixed point method
- Newton-Raphson method (multiple roots)
- Bairstow's method (for complex roots)

Key Terms

<i>Algebraic equation</i>	<i>Monotone divergence</i>
<i>Analytical method</i>	<i>Muller's method</i>
<i>Bairstow's method</i>	<i>Newton-Raphson formula</i>
<i>Binary chopping method</i>	<i>Newton-Raphson method</i>
<i>Bisection method</i>	<i>Nonlinear</i>
<i>Bracketing method</i>	<i>Open end method</i>
<i>Complex number</i>	<i>Polynomial equation</i>
<i>Complex root</i>	<i>Purification</i>
<i>Convergence</i>	<i>Quadratic convergence</i>
<i>Deflation</i>	<i>Quadratic equation</i>
<i>Descartes' rule</i>	<i>Real root</i>
<i>Direct substitution method</i>	<i>Regula falsi</i>
<i>Extrapolation method</i>	<i>Repeated roots</i>
<i>False position method</i>	<i>Roots</i>
<i>Fixed point equation</i>	<i>Search bracket</i>
<i>Fixed point method</i>	<i>Secant formula</i>
<i>Graphical method</i>	<i>Secant method</i>
<i>Half-interval method</i>	<i>Shifted-power form</i>
<i>Horner's rule</i>	<i>Spiral convergence</i>
<i>Incremental search</i>	<i>Spiral divergence</i>
<i>Interpolation method</i>	<i>Stopping criterion</i>
<i>Iterative function</i>	<i>Successive approximations</i>
<i>Iterative method</i>	<i>Superlinear convergence</i>
<i>Jacobian matrix</i>	<i>Synthetic division</i>
<i>Linear</i>	<i>Trial and error</i>
<i>Linear interpolation</i>	<i>Transcendental equation</i>
<i>Linearly convergent</i>	<i>Zeros</i>
<i>Monotone convergence</i>	

REVIEW QUESTIONS

- ✓ 1. What is a nonlinear equation? Give an example from real-life problems.
- ✓ 2. What is an algebraic equation? Give two examples.

3. Polynomial equations are a simple class of algebraic equations. Explain.
- ✓ 4. What is a transcendental equation? What are its characteristics?
5. What is meant by direct analytical method of solution? What are its limitations?
6. When do we seek the help of graphical method for solving a nonlinear equation?
7. What is an iterative technique? How is it implemented on a computer?
- ✓ 8. Describe the concept applied in the bracketing methods used for solving nonlinear equations.
- ✓ 9. How do we decide initial guess values for solving a polynomial equation using
 - (a) open end methods, and
 - (b) bracketing methods?
10. What is meant by stopping criterion? State some of the tests that can be used for terminating an iterative process.
11. What is Horner's rule? How does it improve the accuracy of evaluation of a polynomial?
- ✓ 12. Explain the principle of bisection method with the help of an illustration.
- ✓ 13. Explain the principle of false position method.
- ✓ 14. State the Newton-Raphson formula and explain how it is used to obtain a real root.
- ✓ 15. Explain the limitations of using Newton-Raphson method.
- ✓ 16. Note that the secant formula and the false position formula are similar. Then what is the difference between these two methods?
- ✓ 17. How does the secant method compare with the Newton-Raphson method?
- ✓ 18. Discuss the situations where the fixed-point iteration process may not converge to a solution.
19. Describe an algorithm to determine all possible roots of an equation.
20. State the limitations of using the fixed-point approach for solving a system of nonlinear equations.
21. State the Descartes' rule to estimate the number of real roots of a polynomial.
22. What is synthetic division? How is it used to obtain the multiple roots of a polynomial?
23. What is deflation?
24. What is meant by purification of roots? How is it done?
25. Muller's method is an extension of secant method. Explain.
26. Compare, in a tabular form, the order of convergence of various iterative methods used for solving nonlinear equations.

REVIEW EXERCISES

- ✓ 1. Evaluate the following polynomials using Horner's rule:
- $f(x) = 2x^3 + 4x^2 - 2x + 5$ at $x = 3$
 - $f(x) = x^3 - 2x^2 + 5x + 10$ at $x = 5$
 - $f(x) = x^4 + 2x^2 - 16x + 5$ at $x = 2$
- ✓ 2. Prove that the bisection method is linearly convergent.
- ✓ 3. How would you decide the two initial values that are required for using the bisection method?
- ✓ 4. Find a root of each of the following equations using the bisection method.

- $e^x - x - 2 = 0$
- $\sin x - 2x + 1 = 0$
- $\log x - \cos x = 0$
- $x \tan x - 1 = 0$
- $x^3 - x - 3 = 0$
- $4x^3 - 2x - 6 = 0$
- $x^4 - 2x^3 - x - 3 = 0$

✓ 5. Derive the false position formula for evaluating a root of a nonlinear equation.

✓ 6. Use the false position formula repeatedly and obtain roots of the following equations :

- $x - e^{-x} = 0$
- $\sin x - x + 2 = 0$
- $x^3 - 4x^2 + x + 6 = 0$
- $3x^2 + 6x - 45 = 0$
- $4x^3 - 2x - 6 = 0$

✓ 7. Derive the Newton-Raphson iterative formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

for solving $f(x) = 0$

- Show that the Newton-Raphson method converges to solution quadratically.
- Obtain the Newton's iterative formula for evaluating the square root of a number. Use this formula to find the square root of 3.
- Derive a recursive formula for finding the n th root of a number, say A.
- Show that Newton's formula for finding the reciprocal of A is

$$x_{n+1} = x_n (2 - Ax_n)$$

✓ 12. Find the Newton-Raphson formula for the following functions:

- $f(x) = x^2 - 2x - 1$
- $f(x) = x^3 - x - 3$
- $f(x) = x^3 - 3x - 2$
- $f(x) = \cos x$

V

(e) $f(x) = xe^{-x}$

(f) $f(x) = x \tan x - 1$

13. Apply Newton's method to find the roots of the following equations:

(a) $e^{-x} - x = 0$

(b) $\log x - \cos x = 0$

(c) $\tan x - x = 0$

(d) $x - 1.5 \sin x - 2.5 = 0$

14. Compute a root of each of the following equations using Newton-Raphson method.

(a) $x^2 - 5x + 6 = 0$

$x_0 = 5$

(b) $x^3 - 1.2x^2 + 2x - 2.4 = 0$

$x_0 = 2$

(c) ~~$x^3 - 4x^2 + x + 6 = 0$~~

$x_0 = 5$

(d) $x^4 + 3x^3 - 2x^2 - 12x - 8 = 0$

$x_0 = 1$

(e) ~~$x^5 - 3x^2 - 100 = 0$~~

$x_0 = 2$

15. Derive the secant formula. How is it different from the false position formula.

16. Prove that the rate of convergence of secant method is better than that of bisection method or false position method.

17. Use the secant method to compute a root of the following equations:

(a) $4x^3 - 2x - 6 = 0$

(d) $e^x - 3x = 0$

(b) $x^2 - 5x + 6 = 0$

(e) $x - e^x + 2 = 0$

(c) $x \sin x - 1 = 0$

(f) $x^5 - 3x^2 - 100 = 0$

18. Derive a condition under which the error in the fixed-point iteration method will decrease with each iteration.

19. Use the fixed-point iteration method to evaluate a root of the equation

$$x^2 - x - 1 = 0$$

using the following forms of $g(x)$:

(a) $x = x^2 - 1$

(b) $x = 1 + 2x - x^2$

(c) $x = \frac{1}{2}(1 + 3x - x^2)$

starting with (i) $x_0 = 1$ and (ii) $x_0 = 2$. Discuss the results.

20. Find the square root of 0.75 by writing $f(x) = x^2 - 0.75$ and solving the equation

$$x = x^2 + x - 0.75$$

by the method of fixed-point iteration. Assume an initial value of $x_0 = -0.8$. Try with an initial value of $x_0 = 0.8$. Comment on the results.

21. Use a suitable method to find to three decimal places the roots of the following equations.

(a) $x^2 - x - 6 = 0$

(b) $x^2 + 2x - 0.5 = 0$

(c) $x^2 - 10 \times \log x = 0$

(d) $x^3 - 2x^2 - 3x + 10 = 0$

22. Solve the system of equations

$$x^2 + y^2 = 5$$

$$x^2 - y^2 = 1$$

using (a) fixed-point method and (b) two equation Newton-Raphson method. Assume $x_0 = 1$ and $y_0 = 1$.

23. Use Newton's method to solve the following systems of equations:

(a) $3x^2 - 2y^2 = 1$

$$x^2 - 2x + y^2 + 2y = 8$$

(Assume $x_0 = -1$ and $y_0 = 1$)

(b) $x^3 - y^2 + 1 = 0$

$$x^2 - 2x + y^3 - 2 = 0$$

(Assume $x_0 = 1$ and $y_0 = 1$)

24. The polynomial

$$p(x) = x^3 - 6x^2 + 11x - 6 = 0$$

has a root at $x = 2$. Find the quotient polynomial $q(x)$ such that

$$p(x) = (x - 2) q(x)$$

25. A box open at the top is made from a rectangular piece of plywood measuring 5 by 8 metres by removing square pieces from the corners. What will be the size of square pieces removed if the volume of the box is to be 20 cubic metres?

26. The supply and demand functions of a product are

$$Q_s = p^2 - 500$$

$$Q_d = p^2 - 60p + 1500$$

Determine the market equilibrium price which occurs when $Q_s = Q_d$.

27. Use Muller's method to find a root of the following equations:

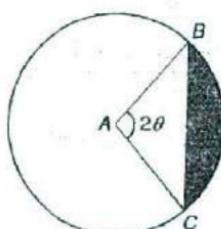
(a) $x^3 - x - 2$, $x_1 = 1$, $x_2 = 1.2$ and $x_3 = 1.4$

(b) $1 + 2x - \tan x$, $x_1 = 1.5$, $x_2 = 1.4$ and $x_3 = 1.3$

28. Use Baird's method to estimate the roots of

$$f(x) = x^4 - 2x^3 + 4x^2 - 4x + 4$$

29. In the figure shown below, estimate the angle θ in radians (to two decimal places) using Newton's method (or any other method). Area of triangle ABC equals area shaded.



Also show that there is only one answer in the interval 0 and $\pi/2$.

30. The equation $x \tan x - 1$ occurs in the theory of vibrations.
 (a) How many roots does it have in the interval 0 and $\pi/2$
 (b) Estimate them to two decimal places.

31. The flux equation of an iron core electric circuit is given by

$$f(\phi) = 10 - 2.1\phi - 0.01\phi^3$$

The steady state value of flux is obtained by solving the equation $f(\phi) = 0$. Use a suitable method to estimate the steady state ϕ .

32. The state of an imperfect gas is given by van der Waals' equation

$$\left(p + \frac{\alpha}{v^2} \right) (v - \beta) = RT$$

or

$$pv^3 - (\beta p + RT)v^2 + \alpha v - \alpha\beta = 0$$

Solve the equation for v (molar volume) given the following:

$$p \text{ (pressure)} = 1.1$$

$$T \text{ (temperature)} = 250^\circ \text{ K}$$

$$R \text{ (gas constant)} = 0.082$$

$$\alpha = 3.6$$

$$\beta = 0.043$$

Use any suitable method.

PROGRAMMING PROJECTS

1. Develop a program to compute all the roots of a polynomial using the bisection method. Use Algorithm 6.6. Test the program for

$$x^3 - 6x^2 + 11x - 6 = 0$$
2. Modify the above program to use Newton-Raphson method instead of bisection method and test the program.
3. Write a program to solve a system of nonlinear equations using
 - (a) fixed-point method (Algorithm 6.7)
 - (b) Newton-Raphson method (Algorithm 6.8)
4. Write a program for computing a real root of an equation using Muller's method. (Algorithm 6.11).
5. Modify the program in Project 4 to implement the Muller algorithm using complex data type supported in FORTRAN to compute complex roots.
6. Design a menu-driven program to compute a root of a given equation. The menu will provide the choices of methods that a user can select, depending on the nature of equation.

CHAPTER

Direct Solution of Linear Equations

7.1

NEED AND SCOPE

Analysis of linear equations is significant for a number of reasons. First, mathematical models of many of the real world problems are either linear or can be approximated reasonably well using linear relationships. Second, the analysis of linear relationship of variables is generally easier than that of nonlinear relationships.

A linear equation involving two variables x and y has the standard form

$$ax + by = c \quad (7.1)$$

where a , b , and c are real numbers and a and b cannot both equal zero. Notice that the exponent (power) of variables is one. The equation becomes nonlinear if any of the variables has the exponent other than one. Similarly, equations containing terms involving a product of two variables are also considered nonlinear.

Some examples of linear equations are:

$$4x + 7y = 15$$

$$-x - 2/3y = 0$$

$$3u - 2v = -1/2$$

Some examples of nonlinear equations are:

$$2x - xy + y = 2$$

$$x^2 + y^2 = 25$$

$$x + \sqrt{x} = 6$$

In practice, linear equations occur in more than two variables. A linear equation with n variables has the form

$$a_1x_1 + a_2x_2 + a_3x_3 + \dots + a_nx_n = b \quad (7.2)$$

here a_i ($i = 1, 2, \dots, n$) are real numbers and at least one of them is not zero. The main concern here is to solve for x_i ($i = 1, 2, \dots, n$), given the values of a_i and b . Note that an infinite set of x_i values will satisfy the above equation. There is no unique solution. If we need a unique solution of an equation with n variables (unknowns), then we need a set of n such independent equations. This set of equations is known as *system of simultaneous equations* (or simply, system of equations).

A system of n linear equations is represented generally as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ &\vdots \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n &= b_n \end{aligned} \quad (7.3)$$

In matrix notation, Eq. (7.3) can be expressed as

$$Ax = b \quad (7.4)$$

where A is an $n \times n$ matrix, b is an n vector, and x is a vector of n unknowns.

The techniques and methods for solving systems of linear algebraic equations belong to two fundamentally different approaches:

1. Elimination approach
2. Iterative approach

Elimination approach, also known as *direct method*, reduces the given system of equations to a form from which the solution can be obtained by simple substitution. We discuss the following elimination methods in this chapter:

1. Basic Gauss elimination method
2. Gauss elimination with pivoting
3. Gauss-Jordan method
4. LU decomposition methods
5. Matrix inverse method

The solution of direct methods do not contain any truncation errors. However, they may contain roundoff errors due to floating point operations.

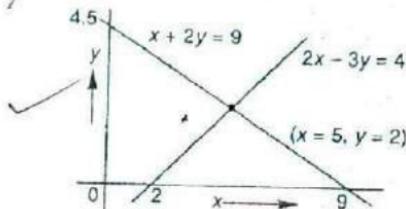
Iterative approach, as usual, involves assumption of some initial values which are then refined repeatedly till they reach some accepted level of accuracy. Iterative methods are discussed in Chapter 8.

EXISTENCE OF SOLUTION

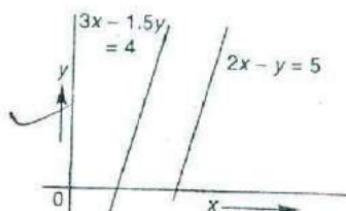
In solving systems of equations, we are interested in identifying values of the variables that satisfy all equations in the system simultaneously.

Given an arbitrary system of equations, it is difficult to say whether the system has a solution or not. Sometimes there may be a solution but it may not be unique. There are four possibilities:

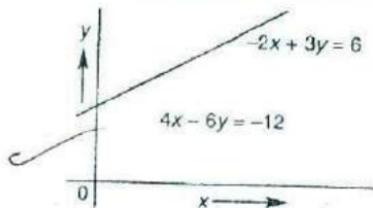
1. System has a unique solution
2. System has no solution
3. System has a solution but not a unique one (i.e., it has infinite solutions)
4. System is ill-conditioned



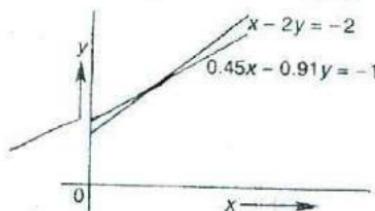
(a) System with unique solution



(b) System with no solution



(c) System with infinite solutions



(d) Ill-conditioned system

Fig. 7.1 Various forms of a system of two linear equations

Unique Solution

Consider the system

$$x + 2y = 9$$

$$2x - 3y = 4$$

The system has a solution

$$x = 5 \quad \text{and} \quad y = 2$$

Since no other pair of values of x and y would satisfy the equation, the solution is said to be *unique*. The system is illustrated in Fig. 7.1(a).

No Solution

The equations

$$2x - y = 5$$

$$3x - 3/2y = 4$$

have no solution. These two lines are parallel as shown in Fig. 7.1(b) and, therefore, they never meet. Such equations are called *inconsistent* equations.

No Unique Solution

The system

$$-2x + 3y = 6$$

$$4x - 6y = -12$$

has many different solutions. We can see that these are two different forms of the same equation and, therefore, they represent the same line (Fig. 7.1(c)). Such equations are called *dependent equations*.

The systems represented in Figures 7.1(b) and 7.1(c) are said to be *singular systems*.

III-Conditioned Systems

There may be a situation where the system has a solution but it is very close to being singular. For example, the system

$$x - 2y = -2$$

$$0.45x - 0.91y = -1$$

has a solution but it is very difficult to identify the exact point at which the lines intersect (Fig. 7.1(d)). Such systems are said to be *ill-conditioned*. Ill-conditioned systems are very sensitive to roundoff errors and, therefore, may pose problems during computation of the solution.

Let us consider a general form of a system of linear equations of size $m \times n$.

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

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

$$\vdots \quad \vdots \quad \vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m$$

In order to effect a unique solution, the number of equations m should be equal to the number of unknowns, n . If $m < n$, the system is said to be *under determined* and a unique solution for all unknowns is not possible. On the other hand, if the number of equations is larger than the number of unknowns, then the set is said to be *over determined*, and a solution may or may not exist.

The system is said to be *homogeneous* when the constants b_i are all zero.

7.3

SOLUTION BY ELIMINATION

Elimination is a method of solving simultaneous linear equations. This method involves elimination of a term containing one of the unknowns in all but one equation. One such step reduces the order of equations by one. Repeated elimination leads finally to one equation with one unknown. Some rules that are useful in manipulation of the equations are:

1. An equation can be multiplied or divided by a constant.

2. One equation can be added or subtracted from another equation.
3. Equations can be written in any order.

For example, the system

$$2x + y = 4$$

$$5x - 2y = 1$$

can be written in different forms as follows:

$$1. \quad 4x + 2y = 8$$

$$5x - 2y = 1$$

$$2. \quad -3x + 3y = 3$$

$$2x + y = 4$$

$$3. \quad 5x - 2y = 1$$

$$2x + y = 4$$

Consider a general form of three linear equations:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

(7.5)

We have three unknowns and three equations. Our objective is to modify this set to the following form:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a'_{21}x_1 + a'_{22}x_2 + 0 = b'_2$$

$$a'_{31}x_1 + a'_{32}x_2 + 0 = b'_3$$

This represents a new set of equations with x_3 eliminated in the last two equations. The last two equations represent a set with two unknowns.

This system can be further transformed into the form

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a''_{21}x_1 + a''_{22}x_2 + 0 = b''_2$$

$$a''_{31}x_1 + 0 + 0 = b''_3$$

Now, the last equation has only one unknown and, therefore, its value can be obtained as

$$x_1 = \frac{b''_3}{a''_{31}}$$

By substituting this in the second equation, we can obtain the value of x_2 . Finally, x_3 can be solved using the computed values of x_1 and x_2 in the first equation.

Remember that the three-equation system (Eq. (7.5)) can also be transformed into the following form:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$0 + a''_{22}x_2 + 0 = b''_2$$

$$a''_{31}x_1 + 0 + a''_{33}x_3 = b''_3$$

Note that a prime indicates that the coefficients have been modified.

The elimination process basically involves the addition of multiples of one equation to other equations so as to set the coefficients of one of the variables in these (other) equations to zero. Example 7.1 illustrates this process.

Solve the following system of equations by the process of elimination.

$$\begin{array}{r} 3x + 2y + z = 10 \\ 2x + 3y + 2z = 14 \\ \hline x + 2y + 3z = 14 \end{array}$$

The elimination process involves the following steps:

Step 1: Elimination of x from second and third equations

Multiply first equation by 2/3 and subtract the result from the second equation. This gives

$$\frac{5}{3}y + \frac{4}{3}z = 22/3$$

or $5y + 4z = 22$

Similarly, multiply first equation by 1/3 and subtract the result from the third equation. This gives

$$y + 2z = 8$$

After step 1, we have the following first derived system:

$$3x + 2y + z = 10$$

$$5y + 4z = 22$$

$$y + 2z = 8$$

Step 2: Elimination of y from the third equation in the derived system

Multiply second equation in the derived system by 1/5 and subtract the result from the third. This results in

$$6z = 18$$

The system now has been reduced to an upper triangular form:

$$3x + 2y + z = 10$$

$$5y + 4z = 22$$

$$6z = 18$$

The derivation of this upper triangular system of equations is called the *forward elimination process*.

We can now solve these equations as follows:

$$z = 18/6 = 3$$

Then,

$$5y + 4 \times 3 = 22$$

Therefore,

$$y = (22 - 4 \times 3)/5 = 2$$

Finally,

$$3x + 2 \times 2 + 3 = 10$$

$$x = (10 - 7)/3 = 1$$

Computation of unknowns from the upper triangular system, as illustrated here, is known as *back substitution*.

7.4

BASIC GAUSS ELIMINATION METHOD

We have seen in Example 7.1 how to solve a system of three equations using the process of elimination. This approach can be extended to systems with more equations. However, the numerous calculations that are required for larger systems make the method complex and time consuming for manual implementation. Therefore, we need to use computer-based techniques for solving large systems. *Gaussian elimination* is one such technique.

Gauss elimination method proposes a systematic strategy for reducing the system of equations to the upper triangular form using the *forward elimination* approach and then for obtaining values of unknowns using the *back substitution* process. The strategy, therefore, comprises two phases:

1. *Forward elimination phase*: This phase is concerned with the manipulation of equations in order to eliminate some unknowns from the equations and produce an upper triangular system.
2. *Back substitution phase*: This phase is concerned with the actual solution of the equations and uses the back substitution process on the reduced upper triangular system.

Let us consider a general set of n equations in n unknowns:

$$\begin{aligned} a_{11} x_1 + a_{12} x_2 + \dots + a_{1n} x_n &= b_1 \\ a_{21} x_1 + a_{22} x_2 + \dots + a_{2n} x_n &= b_2 \\ &\vdots &&\vdots \\ a_{n1} x_1 + a_{n2} x_2 + \dots + a_{nn} x_n &= b_n \end{aligned} \quad \checkmark \tag{7.6}$$

Let us also assume that a solution exists and that it is unique. Algorithm 7.1 illustrates the steps involved in implementing Gauss elimination strategy for such a general system.

Gauss elimination (basic) method

1. Arrange equations such that $a_{11} \neq 0$
2. Eliminate x_1 from all but the first equation. This is done as follows:
 - (i) Normalise the first equation by dividing it by a_{11} .
 - (ii) Subtract from the second Eq. a_{21} times the normalised first equation.

(Contd.)

The result is

$$\left[a_{21} - a_{21} \frac{a_{11}}{a_{11}} \right] x_1 + \left[a_{22} - a_{21} \frac{a_{12}}{a_{11}} \right] x_2 + \dots = b_2 - a_{21} \frac{b_{11}}{a_{11}}$$

We can see that

$$a_{21} - a_{21} \frac{a_{11}}{a_{11}} = 0$$

Thus, the resultant equation does not contain x_1 . The new second equation is

$$0 + a'_{22} x_2 + \dots + a'_{2n} x_n = b'_2$$

- (iii) Similarly, subtract from the third Eq. a_{31} times the normalised first equation.

The result would be

$$0 + a'_{32} x_2 + \dots + a'_{3n} x_n = b'_3$$

If we repeat this procedure till the n th equation is operated on, we will get the following new system of equations:

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

$$a'_{22} x_2 + \dots + a'_{2n} x_n = b'_2$$

...

...

...

$$a'_{n2} x_2 + \dots + a'_{nn} x_n = b'_n$$

The solution of these equations is the same as that of the original equations.

3. Eliminate x_2 from the third to the last equation in the new set. Again, we assume that $a'_{22} \neq 0$.

(i) Subtract from the third equation a'_{32} times the normalised second equation.

(ii) Subtract from the fourth equation, a'_{42} times the normalised second equation,
and so on.

This process will continue till the last equation contains only one unknown, namely, x_n . The final form of the equations will look like this:

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

$$a'_{22} x_2 + \dots + a'_{2n} x_n = b'_2$$

...

$$a_{nn}^{(n-1)} x_n = b_n^{(n-1)}$$

This process is called *triangularisation*. The number of primes indicate the number of times the coefficient has been modified.

(Contd.)

(Contd.)

4. Obtain solution by back substitution.

The solution is as follows:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

This can be substituted back in the $(n-1)^{\text{th}}$ equation to obtain the solution for x_{n-1} . This back substitution can be continued till we get the solution for x_1 .

Algorithm 7.1

Note that the relation for obtaining the coefficients of the k^{th} derived system has the general form

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)} \quad (7.7)$$

where

$$i = k+1 \text{ to } n$$

$$j = 1 \text{ to } n$$

$$a_{ij}^{(0)} = a_{ij} \quad \text{for } i = 1 \text{ to } n, \quad j = 1 \text{ to } n$$

The k^{th} equation, which is multiplied by the factor a_{ik}/a_{kk} , is called the *pivot equation* and a_{kk} is called the pivot element. The process of dividing the k^{th} equation by a_{ik}/a_{kk} is referred to as *normalisation*.

Similarly, the relation for obtaining the k^{th} unknown x_k has the general form

$$x_k = \frac{1}{a_{kk}^{(k-1)}} \left[b_k^{(k-1)} - \sum_{j=k+1}^n a_{kj}^{(k-1)} x_j \right] \quad (7.8)$$

where

$$k = n-1 \text{ to } 1$$

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

Example 7.2

Solve the following 3×3 system using the basic Gauss elimination method.

$$\begin{aligned} 3x_1 + 6x_2 + x_3 &= 16 \\ 2x_1 + 4x_2 + 3x_3 &= 13 \\ x_1 + 3x_2 + 2x_3 &= 9 \end{aligned}$$

After the first step of elimination using multiplication factor $2/3$ and $1/3$, we obtain the new system as follows:

$$3x_1 + 6x_2 + x_3 = 16$$

$$0 + 0 + 7x_3 = 7$$

$$0 + 3x_2 + 5x_3 = 11$$

At this point $a_{22} = 0$ and, therefore, the elimination procedure breaks down. We need to reorder the equations as shown below:

$$3x_1 + 6x_2 + x_3 = 16$$

$$3x_2 + 5x_3 = 11$$

$$7x_3 = 7$$

Note that the process of elimination is complete and the solution is:

$$x_3 = 1, x_2 = 2, \text{ and } x_1 = 1$$

Computational Effort

Computational effort is one of the parameters used to decide the efficiency of a method. Here we estimate the computational effort required in terms of arithmetic operations. The number of operations required for eliminating x_k from the equations below the k th row are:

Multiplications : $(n - k + 1)(n - k)$

Subtractions : $(n - k + 1)(n - k)$

Divisions : $(n - k + 1)$

The total operations required in Gauss elimination method is, therefore,

$$\text{Multiplications} = \sum_{k=1}^{n-1} (n - k + 1)(n - k) = \frac{1}{3}n(n^2 - 1)$$

$$\text{Subtractions} = \sum_{k=1}^{n-1} (n - k + 1)(n - k) = \frac{1}{3}n(n^2 - 1)$$

$$\text{Divisions} = \sum_{k=1}^n (n - k + 1) = \frac{1}{2}n(n - 1)$$

For back substitution, we are evaluating the x values from x_n to x_1 . For evaluating the value of x_k , we require

$n - k$ multiplications

$n - k$ subtractions

1 division

Therefore, the total operations required for back substitution process are

$$\text{Multiplications} = \sum_{k=1}^n (n - k) = \frac{1}{2}n(n - 1)$$

$$\text{Subtractions} = \sum_{k=1}^n (n - k) = \frac{1}{2}n(n - 1)$$

$$\text{Divisions} = \sum_{k=1}^n 1 = n$$

Total operations required for both the stages are given in Table 7.1.

Table 7.1 Computational effort required

	<i>Elimination process</i>	<i>Substitution process</i>	<i>Both stages</i>
Multiplication	$\frac{1}{3} n (n^2 - 1)$	$\frac{1}{2} n (n - 1)$	$\frac{(n - 1)n (2n + 5)}{6}$
Subtraction	$\frac{1}{3} n (n^2 - 1)$	$\frac{1}{2} n (n - 1)$	$\frac{(n - 1)n (2n + 5)}{6}$
Division	$\frac{1}{2} n (n^2 - 1)$	n	$\frac{n(n + 1)}{2}$

We can thus conclude that the number of multiplications and subtractions grows proportional to $n^3/3$ and the number of divisions proportional to $n^2/2$.

Program LEG1

The basic Gauss elimination technique enumerated in Algorithm 7.1 is implemented by the program LEG1. The driver program LEG1 uses a separate subprogram GAUSS1 to implement the computational part of the algorithm.

LEG1 obtains the input data from the user and then calls the subprogram GAUSS1 to solve the specified system of linear equations. It finally prints the results when they are received from the subprogram.

The subprogram GAUSS1 receives the details of the equation from the driver program, determines whether the pivot is zero or not, performs the elimination process (if it is not zero), computes x values (by back substitution), and finally sends the results to the driver program.

Note that when the pivot value is near zero, appropriate message is sent to the driver to inform the user accordingly.

```

*-----*
*      PROGRAM 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
* -----
* Constants used
*   STATUS - Solution status
* -----
REAL A,B,X
INTEGER STATUS,N
EXTERNAL GAUSS1
DIMENSION A(10,10), B(10), X(10)
WRITE(*,*)
WRITE(*,*) 'SOLUTION BY SIMPLE GAUSS METHOD'
WRITE(*,*)

WRITE(*,*) 'What is the size of the system(n)?'
READ(*,*) N
WRITE(*,*) 'Input coefficients a(i,j), row-wise,
+           one row on each line'
DO 20 I = 1, N
    READ(*,*) (A(I,J),J=1,N)
20 CONTINUE

WRITE(*,*) 'Input vector b'
READ(*,*) (B(I), I = 1, N)
CALL GAUSS1(N,A,B,X,STATUS)

IF (STATUS .NE. 0) THEN
    WRITE(*,*)
    WRITE(*,*) 'SOLUTION VECTOR X'
    WRITE(*,*) (X(I), I = 1, N)
    WRITE(*,*)
ELSE
    WRITE(*,*) 'SINGULAR MATRIX, NO SOLUTION'
    WRITE(*,*) 'REORDER EQUATIONS'
    WRITE(*,*) ENDIF
STOP
END
* ----- End of main program LEG1 -----
* -----
*          SUBROUTINE GAUSS1(N,A,B,X,STATUS)
* -----

```

218 Numerical Methods

```
* Subroutine
*   This subroutine solves a set of n linear
*   equations by Gauss elimination method
*
* -----
* Arguments
* Input
*   N - Number of equations
*   A - Matrix of coefficients
*   B - Right side vector
* Output
*   X - Solution vector
* STATUS - Solution status
*
* -----
* Local Variables
*   PIVOT, FACTOR, SUM
*
* -----
* Functions invoked
*   NIL
*
* -----
* Subroutines called
*   NIL
*
* -----
REAL A,B,X,PIVOT,FACTOR,SUM
INTEGER STATUS,N
DIMENSION A(10,10), B(10), X(10)
* ----- Elimination begins -----
DO 33 K = 1, N-1
    PIVOT = A(K,K)
    IF(PIVOT .LT. 0.000001) THEN
        STATUS = 0
        RETURN
    ENDIF
    STATUS = 1
    DO 22 I = K+1, N
        FACTOR = A(I,K)/PIVOT
        DO 11 J = K+1, N
            A(I,J) = A(I,J) - FACTOR * A(K,J)
11    CONTINUE
        B(I) = B(I) - FACTOR * B(K)
22    CONTINUE
33    CONTINUE
* ----- Back substitution begins -----
X(N) = B(N)/A(N,N)
DO 55 K = N-1,1,-1
    SUM = 0
    DO 44 J = K+1,N
```

```

        SUM = SUM + A(K,J) * X(J)
44    CONTINUE
        X(K) = (B(K) - SUM)/A(K,K)
55    CONTINUE
        RETURN
        END
* ----- End of subroutine GAUSS1 -----

```

Test Run Results

SOLUTION BY SIMPLE GAUSS METHOD
 What is the size of the system(n)?
 3
 Input coefficients a(i,j), row-wise, one row on each line
 2 1 3
 4 4 7
 2 5 9
 Input vector b
 1 1 3
 SOLUTION VECTOR X
 -5.000000E-001 -1.0000000 1.0000000
 Stop - Program terminated.

7.5 GAUSS ELIMINATION WITH PIVOTING

In the basic Gauss elimination method, the element a_{ij} when $i = j$ is known as a pivot element. Each row is normalised by dividing the coefficients of that row by its pivot element. That is

$$a_{kj} = \frac{a_{kj}}{a_{kk}} \quad j = 1, \dots, n$$

If $a_{kk} = 0$, k th row cannot be normalised. Therefore, the procedure fails. One way to overcome this problem is to interchange this row with another row below it which does not have a zero element in that position (see Example 7.2).

From the given set of equations, it is possible to reorder the equations such that a_{11} is not zero. But subsequently, the values of a_{kk} are continuously modified during the elimination process and, therefore, it is not possible to predict their values beforehand.

The reordering of the rows is done such that a_{kk} of the row to be normalised is not zero. There may be more than one non-zero values in the k th column below the element a_{kk} . The question is: which one of them is to be selected? It can be proved that roundoff errors would be reduced if the absolute value of the pivot element is large. Therefore, it is suggested that the row with zero pivot element should be interchanged with the row having the largest (absolute value) coefficient in that position. In general, the reordering of equations is done to improve accuracy, even if the pivot element is not zero.

The procedure of reordering involves the following steps:

1. Search and locate the largest absolute value among the coefficients in the first column
2. Exchange the first row with the row containing that element
3. Then eliminate the first variable in the second equation as explained earlier
4. When the second row becomes the pivot row, search for the coefficients in the second column from the second row to the n th row and locate the largest coefficient. Exchange the second row with the row containing the large coefficient.
5. Continue this procedure till $(n - 1)$ unknowns are eliminated.

This process is referred to as *partial pivoting*. There is an alternative scheme known as *complete pivoting* in which, at each stage, the largest element in any of the remaining rows is used as the pivot. Figure 7.2 illustrates the partial and complete pivoting strategies. Algorithm 7.2 shows the implementation steps for partial pivoting.

Complete pivoting requires a lot of overhead and, therefore, it is not generally used (though it may yield slightly improved numerical stability).

Gauss eliminating with partial pivoting

1. Input n , a_{ij} and b_i values.
2. Beginning from the first equation,
 - (i) check for the pivot element
 - (ii) if it is the largest among the elements below it, obtain the derived system
 - (iii) otherwise, identify the largest element and make it the pivot element
 - (iv) interchange the original pivot equation with the one containing the largest element so that the later becomes the new pivot equation
 - (v) obtain the derived system
 - (vi) continue the process till the system is reduced to triangular form
3. Compute x_i values by back substitution.
4. Print results.

Algorithm 7.2

Example 7.1

Solve the following system of equations using partial pivoting technique

$$2x_1 + 2x_2 + x_3 = 6$$

$$4x_1 + 2x_2 + 3x_3 = 4$$

$$x_1 + x_2 + x_3 = 0$$

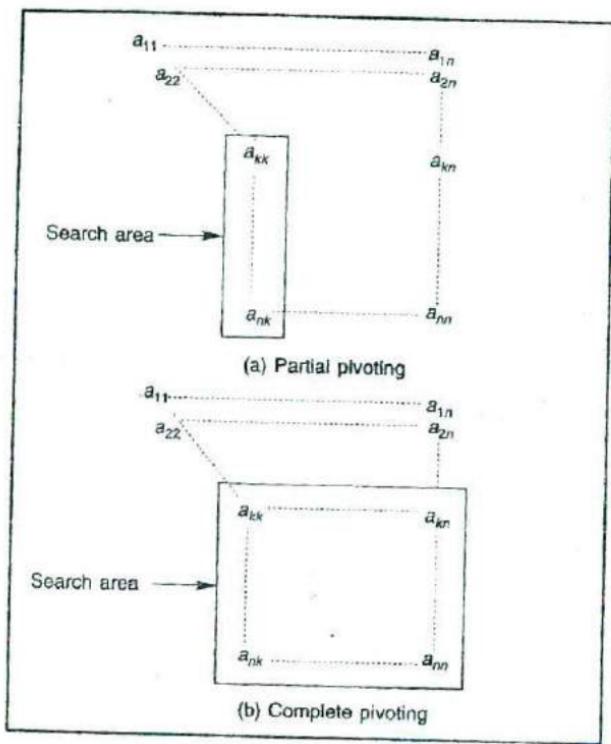


Fig. 7.2 Pivoting strategies

The forward elimination process using partial pivoting is shown below in tabular form. The process involves two steps of elimination and, in both the steps, the rows are interchanged. Note that the absolute value of $-3/2$ is greater than 1.

Original system

	2	2	1	6	Interchange
4	2	3	4		
1	-1	1	0		

Modified original system

4	2	3	4	pivot
2	2	1	6	
1	-1	1	0	

First derived system

4	2	3	4	
1	-1/2	4		Interchange
-3/2	1/4	-1		

Modified first derived system

4	2	3	4	
-3/2	1/4	-1		pivot
1	-1/2	4		

Second and final derived system 4 2 3 4
 -3/2 1/4 -1
 -1/3 10/3

The solution is

$$x_3 = -10$$

$$x_2 = -1$$

$$x_1 = 9$$

Program LEG2

Program LEG2 is designed to solve a system of linear equations using Gauss elimination with partial pivoting. The modular structure of the program is shown in Fig. 7.3.

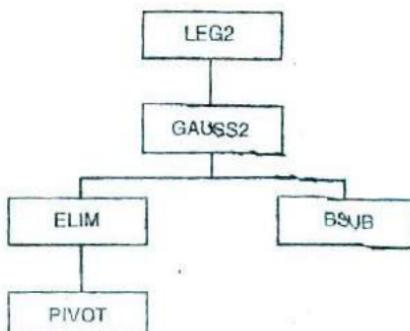


Fig. 7.3 Modular structure of LEG2

The master program LEG2, while reading data from the user and printing solution vector, depends on the services of the subprogram GAUSS2 for implementing the actual solution procedure given in Algorithm 7.2. GAUSS2, in turn, uses the services of two other subprograms, namely, ELIM, to perform forward elimination, and BSUB, to obtain the solution vector using the back substitution approach.

The subprogram PIVOT undertakes the task of partial pivoting by identifying the pivot element and then rearranging the rows such that the equation containing the pivot element becomes the pivot equation.

```

*-----*
*      PROGRAM 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
*
REAL A,B,X
INTEGER N
EXTERNAL GAUSS2
DIMENSION A(10,10), B(10), X(10)

WRITE(*,*)
WRITE(*,*) 'GAUSS METHOD WITH PARTIAL PIVOTING'
WRITE(*,*)

WRITE(*,*) 'What is the size of the system(n)?'
READ(*,*) N

WRITE(*,*) 'Input coefficients a(i,j), row-wise'
WRITE(*,*) 'one row on each line'
DO 20 I = 1, N
    READ(*,*) (A(I,J),J=1,N)
20 CONTINUE
WRITE(*,*) 'Input vector b'
READ(*,*) (B(I), I = 1, N)
CALL GAUSS2(N,A,B,X)

WRITE(*,*)
WRITE(*,*) 'SOLUTION VECTOR X'
WRITE(*,*)
WRITE(*,*) (X(I), I = 1, N)
WRITE(*,*)

STOP
END

* ----- End of main program LEG1 -----
*
* -----
* SUBROUTINE GAUSS2(N,A,B,X)
*
* -----
* Subroutine
* This subroutine solves a system of linear
* equations using Gauss elimination method with
* partial pivoting
*
```

```

* Arguments
* Input
*   A - Coefficient matrix
*   B - Right Side vector
*   N - Size of the system
* Output
*   X - Solution vector
* -----
* Local Variables
*   NIL
* -----
* Functions invoked
*   NIL
* -----
* Subroutines called
*   ELIM, BSUB
* -----
      REAL A,B,X
      INTEGER N
      EXTERNAL ELIM, BSUB
      DIMENSION A(10,10), B(10), X(10)

* Forward elimination
      CALL ELIM(N,A,B)
* Solution by back substitution
      CALL BSUB(N,A,B,X)
      RETURN
      END

* ----- End of subroutine GAUSS2 -----
* -----
      SUBROUTINE ELIM(N,A,B)
* -----
* Subroutine
*   This subroutine performs forward elimination
*   incorporating partial pivoting technique
* -----
* Arguments
* Input
*   A - Coefficient matrix
*   B - Right side vector
*   N - System size
* Output
*   A - Modified A
*   B - Modified B
* -----

```

```

* Local Variables
* FACTOR
* -
* . . . . . functions invoked
* . . . . . NIL
* -
* . . . . . Subroutines called
* PIVOT
* -
REAL A,B,X,FACTOR
INTEGER N
EXTERNAL PIVOT
DIMENSION A(10,10),B(10)
DO 33 K = 1, N-1
    CALL PIVOT (N,A,B,K)
    DO 22 I = K+1, N
        FACTOR = A(I,K)/A(K,K)
        DO 11 J = K+1, N
            A(I,J) = A(I,J) - FACTOR * A(K,J)
11    CONTINUE
        B(I) = B(I) - FACTOR * B(K)
22    CONTINUE
33    CONTINUE
    RETURN
END
* ----- End of subroutine ELIM -----
* -
SUBROUTINE PIVOT(N,A,B,K)
* -
* Subroutine
* This subroutine performs the task of partial
* pivoting (reordering of equations)
* -
* Arguments
* Input
*   N - System size
*   A - Coefficients matrix
*   B - Right side vector
*   K - Row under consideration for pivoting
* Output
*   A - Modified A (after pivoting)
*   B - Modified B (after pivoting)
* -
* Local Variables
*   LARGE,TEMP,P
* -

```

```

* Functions invoked
*   ABS
*
* -----
* Subroutines called
*   NIL
*
* -----
    REAL LARGE,TEMP,A,B
    INTEGER P,N,K
    INTRINSIC ABS
    DIMENSION A(10,10), B(10)

* Find pivot P

    P = K
    LARGE = ABS(A(K,K))
    DO 11 I = K+1, N
        IF(ABS(A(I,K)) .GT. LARGE) THEN
            LARGE = ABS(A(I,K))
            P = I
        ENDIF
11    CONTINUE
* Exchange rows P and K
    IF(P.NE.K) THEN
        DO 22 J = K,N
            TEMP = A(P,J)
            A(P,J) = A(K,J)
            A(K,J) = TEMP
22    CONTINUE
        TEMP = B(P)
        B(P) = B(K)
        B(K) = TEMP
    ENDIF
    RETURN
END
* -----End of subroutine PIVOT-----
* -----
* SUBROUTINE BSUB(N,A,B,X)
*
* -----
* Subroutine
* This subroutine obtains the solution vector X
* by back substitution
*
* -----
* Arguments
* Input
*   N - System size
*   A - Coefficient matrix (after elimination)
*   B - Right side vector (after elimination)

```

```

* Output
*      X - Solution vector
*
* -----
* Local Variables
    SUM
-
* Functions invoked
*      NIL
*
* -----
* Subroutines called
*      NIL
*
* -----
      INTEGER N
      REAL A,B,X, SUM
      DIMENSION A(10,10), B(10), X(10)
      X(N) = B(N)/A(N,N)
      DO 55 K = N-1, 1, -1
          SUM = 0
          DO 46 J = K+1, N
              SUM = SUM + A(K,J) * X(J)
44     CONTINUE
      X(K) = (B(K) - SUM)/A(K,K)
55     CONTINUE
      RETURN
      END
* ----- End of subroutine BSUB -----

```

Test Run Results

```

GAUSS METHOD WITH PARTIAL PIVOTING
What is the size of the system(n)?
3
Input coefficients a(i,j), row-wise
one row on each line
2 2 1
4 2 3
1 1 1
Input vector b
6 4 0
SOLUTION VECTOR X
      5.0000000      1.0000000      -6.0000000
Stop - Program terminated.

```

7.6 GAUSS-JORDAN METHOD

Gauss-Jordan method is another popular method used for solving a system of linear equations. Like Gauss elimination method, Gauss-Jordan method also uses the process of elimination of variables, but there is a major difference between them. In Gauss elimination method, a variable is eliminated from the rows below the pivot equation. But in Gauss-Jordan method, it is eliminated from all other rows (both below and above). This process thus eliminates all the off-diagonal terms producing a diagonal matrix rather than a triangular matrix. Further, all rows are normalised by dividing them by their pivot elements. This is illustrated in Fig. 7.4. Consequently, we can obtain the values of unknowns directly from the b vector, without employing back-substitution. Algorithm 7.3 enumerates the Gauss-Jordan elimination steps.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

The diagram shows two stages of row reduction. An arrow points from the original system to a lower triangular system where the first row has been divided by its pivot element (a₁₁). Another arrow points from this to a diagonal system where each row has been divided by its pivot element (a₁₁, a₂₂, and a₃₃ respectively).

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b''_2 \\ b'''_3 \end{bmatrix}$$

Result of Gauss elimination

Result of Gauss-Jordan elimination

Fig. 7.4 Comparison of Gauss and Gauss-Jordan methods of elimination

Gauss-Jordan elimination

1. Normalise the first equation by dividing it by its pivot element.
2. Eliminate x_1 term from all the other equations.
3. Now, normalise the second equation by dividing it by its pivot element.
4. Eliminate x_2 from all the equations, above and below the normalised pivotal equation.
5. Repeat this process until x_n is eliminated from all but the last equation.
6. The resultant b vector is the solution vector.

Algorithm 7.3

The Gauss-Jordan method requires approximately 50 per cent more arithmetic operations compared to Gauss method. Therefore, this method is rarely used.

Example 7.4

Solve the system

$$2x_1 + 4x_2 - 6x_3 = -8$$

$$\begin{aligned}x_1 + 3x_2 + x_3 &= 10 \\2x_1 - 4x_2 - 2x_3 &= -12\end{aligned}$$

using Gauss-Jordan method.

Step 1: Normalise the first equation by dividing it by 2 (pivot element).

The result is:

$$\begin{aligned}x_1 + 2x_2 - 3x_3 &= -4 \\x_1 + 3x_2 + x_3 &= 10 \\2x_1 - 4x_2 - 2x_3 &= -12\end{aligned}$$

Step 2: Eliminate x_1 from the second equation, subtracting 1 time the first equation from it. Similarly, eliminate x_1 from the third equation by subtracting 2 times the first equation from it. The result is:

$$\begin{aligned}x_1 + 2x_2 - 3x_3 &= -4 \\0 + x_2 + 4x_3 &= 14 \\0 - 8x_2 + 4x_3 &= -4\end{aligned}$$

Step 3: Normalise the second equation. (Note that it is already in normalised form.)

Step 4: Following similar approach, eliminate x_2 from first and third equations. This gives

$$\begin{aligned}x_1 + 0 - 11x_3 &= -32 \\0 + x_2 + 4x_3 &= 14 \\0 + 0 + 36x_3 &= 108\end{aligned}$$

Step 5: Normalise the third equation

$$\begin{aligned}x_1 + 0 - 11x_3 &= -32 \\0 + x_2 + 4x_3 &= 14 \\0 + 0 + x_3 &= 3\end{aligned}$$

Step 6: Eliminate x_3 from the first and second equations. We get

$$\begin{aligned}x_1 + 0 + 0 &= 1 \\0 + x_2 + 0 &= 2 \\0 + 0 + x_3 &= 3\end{aligned}$$

Computational Effort

The Gauss-Jordan method requires only the elimination process. To eliminate x_k from all but the k th equation, we need to undertake the following tasks:

1. Divide the coefficients $x_{k+1}, x_{k+2}, \dots, x_n$ and b_k by the coefficient of x_k .
2. Subtract suitable multiples of the k th equation from the other $(n-1)$ equations to eliminate x_k from these equations.

These tasks require:

- $(n-k+1)$ divisions
- $(n-1)(n-k+1)$ multiplications
- $(n-1)(n-k+1)$ subtractions

Therefore, the total operations required in order to complete the elimination process are:

$$\text{Multiplications} = \sum_{k=1}^n (n-1)(n-k+1) = \frac{1}{2} n(n^2 - 1)$$

$$\text{Subtractions} = \sum_{k=1}^n (n-k+1) = \frac{1}{2} n(n^2 + 1)$$

$$\text{Divisions} = \sum_{k=1}^n (n-1)(n-k+1) = \frac{1}{2} n(n-1)$$

We see that the number of multiplications and subtractions is approximately equal to $(1/2)n^3$ and the number of divisions is $(1/2)n^2$. Computational efforts required by the Gauss and Gauss-Jordan methods are given in Table 7.2.

Table 7.2 Comparison of computational effort

	Gauss method	Gauss-Jordan method
Multiplication	$\frac{1}{3}n^3$	$\frac{1}{2}n^3$
Subtraction	$\frac{1}{3}n^3$	$\frac{1}{2}n^3$
Divisions	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$

It shows that the Gauss method requires only two-third of the number of multiplications or subtractions that the Gauss-Jordan method requires: i.e., the Gauss-Jordan method requires 50 per cent more multiplications and subtractions as pointed out earlier.



7.7 TRIANGULAR FACTORISATION METHODS

The coefficient matrix A of a system of linear equations can be factorised (or decomposed) into two triangular matrices L and U such that

$$A = LU \quad (7.9)$$

where

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

and

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

L is known as lower triangular matrix and **U** is known as upper triangular matrix.

Once **A** is factorised into **L** and **U**, the system of equations

$$\mathbf{Ax} = \mathbf{b}$$

can be expressed as follows

$$(\mathbf{LU}) \mathbf{x} = \mathbf{b}$$

or

$$\mathbf{L}(\mathbf{Ux}) = \mathbf{b} \quad (7.10)$$

Let us assume that

$$\boxed{\mathbf{Ux} = \mathbf{z}} \quad (7.11)$$

where **z** is an unknown vector. Substituting Eq. (7.11) in equation (7.10), we get

$$\boxed{\mathbf{Lz} = \mathbf{b}} \quad (7.12)$$

Now we can solve the system

$$\mathbf{Ax} = \mathbf{b}$$

in two stages:

1. Solve the equation

$$\mathbf{Lz} = \mathbf{b}$$

for **z** by forward substitution

2. Solve the equation

$$\boxed{\mathbf{Ux} = \mathbf{z}}$$

for **x** using **z** (found in stage 1) by back substitution.

The elements of **L** and **U** can be determined by comparing the elements of the product of **L** and **U** with those of **A**. The process produces a system of n^2 equations with $n^2 + n$ unknowns (l_{ij} and m_{ij}) and, therefore, **L** and **U** are not unique. In order to produce unique factors, we should reduce the number of unknowns by n .

This is done by assuming the diagonal elements of **L** or **U** to be unity. The decomposition with **L** having unit diagonal values is called the *Doolittle LU decomposition* while the other one with **U** having unit diagonal elements is called the *Crout LU decomposition*.

Doolittle Algorithm

We can solve for the components of **L** and **U**, given **A**, as follows:

$$\mathbf{A} = \mathbf{LU}$$

implies that

$$a_{ij} = l_{i1} u_{1j} + l_{i2} u_{2j} + \dots + l_{ii} u_{ij} \quad \text{for } i < j \quad (7.13)$$

$$a_{ij} = l_{i1} u_{1j} + l_{i2} u_{2j} + \dots + l_{ii} u_{jj} \quad \text{for } i = j \quad (7.14)$$

$$a_{ij} = l_{i1} u_{1j} + l_{i2} u_{2j} + \dots + l_{ij} u_{jj} \quad \text{for } i > j \quad (7.15)$$

where $u_{ij} = 0$ for $i > j$ and $l_{ij} = 0$ for $i < j$

The Dolittle algorithm assumes that all the diagonal elements of \mathbf{L} are unity. That is

$$l_{ii} = 1, \quad i = 1, 2, \dots, n.$$

Using equations (7.13), (7.14) and (7.15), we can successively determine the elements of \mathbf{U} and \mathbf{L} as follows:

If $i \leq j$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \quad j = 1, 2, \dots, n$$

where $u_{11} = a_{11}$, $u_{12} = a_{12}$, $u_{13} = a_{13}$

Similarly, if $i > j$

$$l_{ij} = \frac{l}{u_{ij}} \times \left[a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right] \quad j = 1, 2, \dots, i-1$$

where $l_{11} = l_{22} = l_{33} = 1$ and $l_{11} = a_{11}/u_{11}$ for $i = 2$ to n .

Note that, for computing any element, we need the values of elements in the previous columns as well as the values of elements in the column above that element, as illustrated in Fig. 7.5. This suggests that we should compute the elements, *column by column from left to right* within each column from *top to bottom*.

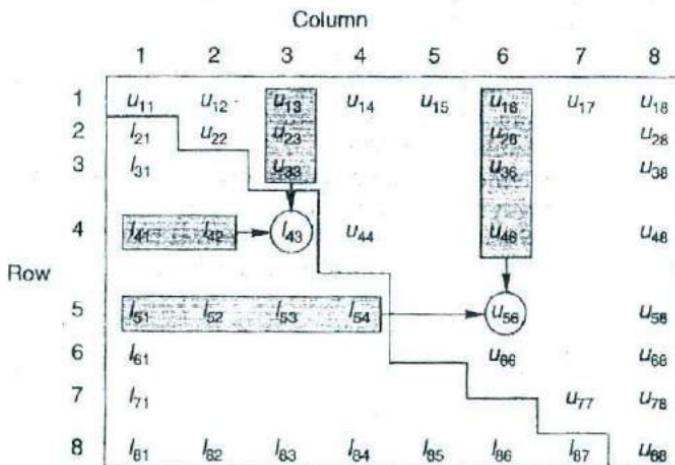


Fig. 7.5 Pictorial view of Dolittle algorithm of LU decomposition

Algorithm 7.4 lists steps involved in LU decomposition and its application to the solution of linear equations.

Note:

1. There is no need to store 1's on the diagonal of **L** matrix.
2. There is also no need to store 0's of **L** or **U**. Consequently, the values of **L** can be stored in the zero space of **U**.
3. Further, each element of a_{ij} is used only once (and never used again).

It is clear that we can "overwrite" **A** with **L** and **U** and save memory. This means the corresponding l_{ij} or u_{ij} can be stored in the location of a_{ij} .

Doolittle LU decomposition and solution

1. Given n, A, b

2. Set $u_{1j} = a_{1j}$ for $j = 1$ to n

Set $l_{1j} = 1$ for $j = 1$ to n

Set $l_{11} = a_{11}/u_{11}$ for $i = 2$ to n

3. For each $j = 2$ to n do:

- (i) For $i = 2$ to j

$$\text{Compute } u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$$

Repeat i

- (ii) For $i = j+1$ to n

$$\text{Compute } l_{ij} = \frac{1}{u_{jj}} \times \left[a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right]$$

Repeat i

Repeat j

4. Set $z_1 = b_1$

5. For $i = 2$ to n

$$\text{Set sum} = \sum_{j=1}^{i-1} l_{ij} z_j$$

Set $z_i = b_i - \text{sum}$

Repeat i

6. Set $x_n = z_n / u_{nn}$

7. For $i = n-1$ to 1

$$\text{Set sum} = \sum_{j=i+1}^n u_{ij} x_j$$

$$\text{Set } x_i = (z_i - \text{sum}) / u_{ii}$$

Repeat i

8. Write results

Example 7.5

Solve the system

$$\begin{aligned}3x_1 + 2x_2 + x_3 &= 10 \\2x_1 + 3x_2 + 2x_3 &= 14 \\x_1 + 2x_2 + 3x_3 &= 14\end{aligned}$$

by using Dolittle LU decomposition method

Factorisation

For $i = 1$, $l_{11} = 1$ and

$$u_{11} = a_{11} = 3$$

$$u_{12} = a_{12} = 2$$

$$u_{13} = a_{13} = 1$$

For $i = 2$

$$l_{21} = \frac{a_{21}}{u_{11}} = \frac{2}{3} \quad \text{and} \quad l_{22} = 1$$

$$u_{22} = a_{22} - l_{21} u_{12} = 3 - \frac{2}{3} \times 2 = \frac{5}{3}$$

$$u_{23} = a_{23} - l_{21} u_{13} = 2 - \frac{2}{3} \times 1 = \frac{4}{3}$$

For $i = 3$

$$l_{31} = \frac{a_{31}}{u_{11}} = \frac{1}{3}$$

$$l_{32} = \frac{a_{32} - l_{31} u_{12}}{u_{22}}$$

$$= \frac{2 - 1/3 \times 2}{5/3} = \frac{4}{5}$$

$$l_{33} = 1$$

$$u_{33} = a_{33} - l_{31} u_{13} - l_{32} u_{23}$$

$$= 3 \frac{1}{3} \times 1 - \frac{4}{5} \times \frac{4}{3} = \frac{24}{15}$$

Thus, we have

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 1/3 & 4/5 & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 5/3 & 4/3 \\ 0 & 0 & 24/15 \end{bmatrix}$$

Forward Substitution

Solving $\mathbf{Lz} = \mathbf{b}$ by forward substitution, we get

$$\begin{aligned} z_1 &= b_1 = 10 \\ z_2 &= b_2 - l_{21} z_1 \\ &= 14 - 2/3 \times 10 = 22/3 \\ z_3 &= b_3 - l_{31} z_1 - l_{32} z_2 \\ &= 14 - 1/3 \times 10 - 4/5 \times 22/3 = \frac{72}{15} \end{aligned}$$

Back Substitution

Solving $\mathbf{Ux} = \mathbf{z}$ by back substitution, we get

$$\begin{aligned} x_3 &= \frac{27/15}{24/15} = 3 \\ x_2 &= \frac{z_2 - u_{23} x_3}{u_{22}} \\ &= \frac{(22/3) - (4/3) \times 3}{5/3} = 2 \\ x_1 &= \frac{z_1 - u_{12} x_2 - u_{13} x_3}{u_{11}} \\ &= \frac{10 - 2 \times 2 - 1 \times 3}{3} = 1 \end{aligned}$$

Program DOLIT

The Dolittle LU decomposition method for solving a system of linear equations may be implemented on a computer using the program DOLIT. The DOLIT program solves a problem with the help of two subprograms, LUD and SOLVE.

The subprogram LUD decomposes the given coefficient matrix using the Dolittle algorithm and the resultant \mathbf{L} and \mathbf{U} matrices are supplied back to the main program DOLIT. Note that when it fails to decompose the matrix, a message to that effect is sent to the main program for necessary action.

The subprogram SOLVE receives the right side vector \mathbf{B} and the decomposed matrices \mathbf{L} and \mathbf{U} from the main program and then obtains the solution vector \mathbf{X} employing both the forward and backward substitution techniques.

* ----- *
PROGRAM 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 - Factorisation status
* -----
* Constants used
*   YES,NO
* -----
      INTEGER N,YES,NO,FACT
      REAL A,U,L,B,X
      EXTERNAL LUD,SOLVE
      PARAMETER( YES = 1, NO = 0 )
      DIMENSION A(10,10),U(10,10),L(10,10),B(10),X(10)
      WRITE(*,*)
      WRITE(*,*) 'SOLUTION BY DOLITTLE METHOD '
      WRITE(*,*)

* Read input data
      WRITE(*,*) 'What is the size of A?'
      READ(*,*) N
      WRITE(*,*) 'Input coefficients a(i,j), row-wise, '
      +           'one row on each line'
      DO 10 I = 1, N
      READ(*,*) (A(I,J), J=1,N)
10  CONTINUE
      WRITE(*,*) 'Input vector B on one line'
      READ(*,*) (B(I), I=1, N)
* LU factorisation
      CALL LUD(N,A,U,L,FACT)
      IF( FACT .EQ. YES ) THEN
*   Print LU matrices
      * Print matrix U
          WRITE(*,*) 
          WRITE(*,*) 'MATRIX U'
          DO 20 I = 1,N

```

```

        WRITE(*,111) (U(I,J),J=1,N)
20    CONTINUE
* Print matrix L
    WRITE(*,*) 
    WRITE(*,*) 'MATRIX L'
    DO 30 I=1,N
        WRITE(*,111) (L(I,J),J=1,N)
30    CONTINUE
ELSE
    WRITE(*,*) 
    WRITE(*,*) 'FACTORISATION NOT POSSIBLE'
    WRITE(*,*) 
    STOP
ENDIF
* Solve for X
    CALL SOLVE(N,U,L,B,X)
    WRITE(*,*) 
    WRITE(*,*) 'SOLUTION VECTOR X'
    WRITE(*,*) 
    WRITE(*,111) (X(I), I=1,N)
    WRITE(*,*) 
111 FORMAT(3F15.6)
    STOP
END
* -----End of main program DOLIT -----
*
* -----
SUBROUTINE LUD(N,A,U,L,FACT)
*
* Subroutine
* This subroutine decomposes the matrix A into
* L and U matrices using Dolittle algorithm
*
* -----
* Arguments
* Input
*     N - System size
*     A - Coefficient matrix of the original system
* Output
*     U - Decomposed upper triangular matrix
*     L - Decomposed lower triangular matrix
*     FACT - Fact about decomposition (yes or no)
*
* -----
* Local Variables
*     SUM
*
```

```

* Functions invoked
*   NIL
*
* -----
* Subroutines called
*   NIL
*
* -----
INTEGER N,YES,NO,FACT
REAL A,U,L,SUM
PARAMETER( YES = 1, NO = 0 )
DIMENSION A(10,10), U(10,10), L(10,10)

* Initialise U and L matrices
DO 1 I = 1,N
    DO 1 J = 1,N
        U(I,J) = 0.0
        L(I,J) = 0.0
1 CONTINUE

* Compute the elements of U and L
DO 10 J = 1,N
    U(1,J) = A(1,J)
10 CONTINUE

DO 20 I = 1,N
    L(I,I) = 1.0
20 CONTINUE

DO 30 I = 2,N
    L(I,1) = A(I,1)/U(1,1)
30 CONTINUE

DO 100 J = 2,N
    DO 50 I = 2,J
        SUM = A(I,J)
        DO 40 K = 1,I-1
            SUM = SUM - L(I,K) * U(K,J)
40    CONTINUE
        U(I,J) = SUM
50    CONTINUE

IF( U(J,J) .LE. 1.E-6 ) THEN
    FACT = NO
    RETURN
ENDIF

DO 70 I = J+1,N
    SUM = A(I,J)
    DO 60 K = 1,J-1
        SUM = SUM - L(I,K) * U(K,J)
60    CONTINUE

```

```

L(I,J) = SUM/U(J,J)
70    CONTINUE
100 CONTINUE
FACT = YES
RETURN
END
* ----- End of subroutine LUD -----
*
* SUBROUTINE SOLVE(N,U,L,B,X)
*
* Subroutine
*   This subroutine obtains the solution vector X
*   using the coefficients of L and U matrices
*
* Arguments
* Input
*   N - System size
*   U - Upper triangular matrix
*   L - Lower triangular matrix
*   B - Right side vector
* Output
*   X - Solution vector
*
* Local Variables
*   SUM, Z(vector)
*
* Functions invoked
*   NIL
*
* Subroutines called
*   NIL
*
INTEGER N
REAL U,L,SUM,B,X,Z
DIMENSION U(10,10),L(10,10),B(10),X(10),Z(10)
* Forward substitution
Z(1) = B(1)
DO 20 I = 2,N
  SUM = 0.0
  DO 10 J = 1,I-1
    SUM = SUM + L(I,J) * Z(J)
10  CONTINUE
  Z(I) = B(I) - SUM
20 CONTINUE
* Back substitution
X(N) = Z(N)/U(N,N)

```

```

DO 40 I = N-1,1,-1
    SUM = 0.0
    DO 30 J = I+1,N
        SUM = SUM + U(I,J) * X(J)
30    CONTINUE
        X(I) = (Z(I) - SUM)/U(I,I)
40    CONTINUE
    RETURN
END
* -----End of subroutine SOLVE -----

```

Test Run Results

SOLUTION BY DOLITTLE METHOD

What is the size of A?

3

Input coefficients a(i,j), row-wise, one row on each line

3 2 1

2 3 2

1 2 3

Input vector B on one line

10 14 14

MATRIX U

3.000000	2.000000	1.000000
.000000	1.666667	1.333333
.000000	.000000	1.600000

MATRIX L

1.000000	.000000	.000000
.666667	1.000000	.000000
.333333	.800000	1.000000

SOLUTION VECTOR X

1.000000	2.000000	3.000000
----------	----------	----------

Stop - Program terminated.

Crout Algorithm

Another approach to LU decomposition is *Crout algorithm*. As mentioned earlier, Crout decomposition algorithm assumes unit diagonal values for **U** matrix and the diagonal elements of **L** matrix may assume any values as shown below.

$$\begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} l & u_{12} & \dots & u_{1n} \\ 0 & 1 & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & l \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

We can use an approach that is similar to the one used in Dolittle decomposition to evaluate the elements of L and U.

Cholesky Method

In case A is symmetric, the LU decomposition can be modified so that the upper factor is the transpose of the lower one (or vice versa). That is, we can factorise A as

$$A = LL^T$$

or

$$A = U^T U \quad (7.16)$$

Just as for Dolittle decomposition, by multiplying the terms of Eq. (7.16) and setting them equal to each other (see Eqs (7.13), (7.14) and (7.15)), the following recurrence relations can be obtained.

$$u_{ij} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2} \quad (i = 1 \text{ to } n)$$

$$u_{ij} = \frac{1}{u_{ii}} \left[a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj} \right] \quad (j > i)$$

(7.17)

This decomposition is called the *Cholesky's factorisation* or the *method of square roots*. Algorithm 7.5 lists the basic steps for computing the elements U, column by column.

Cholesky's factorisation

1. Given n, A
2. Set $u_{11} = \sqrt{a_{11}}$
3. Set $u_{ij} = a_{ij} / u_{11}$ for $i = 2$ to n
4. For $j = 2$ to n
 - For $i = 2$ to j
 - sum = a_{ij}
 - For $k = 1$ to $i-1$
 - sum = sum - $u_{ki} u_{kj}$
 - Repeat k
 - set $u_{ij} = \text{sum} / u_{ii}$ if $i < j$
 - set $u_{ij} = \sqrt{\text{sum}}$ if $i = j$
 - Repeat i
 - Repeat j
 5. End of factorisation

Example 7.6

Factorise the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 8 & 22 \\ 3 & 22 & 82 \end{bmatrix}$$

using Cholesky's algorithm

For $i = 1$, according Eq. (7.17)

$$u_{11} = \sqrt{1} = 1$$

$$u_{12} = \frac{a_{12}}{u_{11}} = \frac{2}{1} = 2$$

$$u_{13} = \frac{a_{13}}{u_{11}} = \frac{3}{1} = 3$$

For $i = 2$

$$u_{22} = \sqrt{a_{22} - u_{12}^2} = \sqrt{8 - 4} = 2$$

$$u_{23} = \frac{a_{23} - u_{12}u_{13}}{u_{22}} = \frac{22 - 2 \times 3}{2} = \frac{16}{2} = 8$$

For $i = 3$

$$\begin{aligned} u_{33} &= \sqrt{a_{33} - u_{13}^2 - u_{23}^2} \\ &= \sqrt{82 - 9 - 64} = \sqrt{9} = 3 \end{aligned}$$

Thus, we have

$$\mathbf{U} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & 8 \\ 0 & 0 & 3 \end{bmatrix}$$

7.5**ROUNDOFF ERRORS AND REFINEMENT**

In all the direct methods, only one estimate of x_i is produced. As we know, methods use a large number of floating point operations and, therefore, introduce roundoff errors in the final solution. We have no indication how accurate the solution is.

One way to check this is to substitute the answer back into the original equations to see whether a substantial error has occurred. In case the error is beyond the acceptable limit, the solution can be improved by a technique known as *iterative refinement*.

Let us suppose $\mathbf{x}^{(1)}$ is the solution of the system

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

Substituting $\mathbf{x}^{(1)}$ back in the original equation, we get

$$\mathbf{A}\mathbf{x}^{(1)} = \mathbf{b}'$$

Since $\mathbf{x}^{(1)}$ is not exact, \mathbf{b}' is not equal to \mathbf{b} . If we define

$$\mathbf{r}^{(1)} = \mathbf{b}' - \mathbf{b}$$

then we have

$$\mathbf{r}^{(1)} = \mathbf{A}\mathbf{x}^{(1)} - \mathbf{b} \quad (7.18)$$

where \mathbf{r} is known as *residual vector*. If we can use this information to compute the error, then we can correct the approximate solution with this error.

If we assume that \mathbf{x}^* is the exact solution and e is the error in \mathbf{x} , then

$$\mathbf{x}^* = \mathbf{x}^{(1)} - e^{(1)}$$

or

$$\mathbf{x}^{(1)} = \mathbf{x}^* + e^{(1)} \quad (7.19)$$

Substituting this in Eq. (7.18), we get

$$\begin{aligned} \mathbf{r}^{(1)} &= \mathbf{A}(\mathbf{x}^* + e^{(1)}) - \mathbf{b} \\ &= \mathbf{A}\mathbf{x}^* + \mathbf{A}e^{(1)} - \mathbf{b} \end{aligned}$$

Since $\mathbf{A}\mathbf{x}^* = \mathbf{b}$, this results in

$$\boxed{\mathbf{A}e^{(1)} = \mathbf{r}^{(1)}} \quad (7.20)$$

We can now obtain $e^{(1)}$ by solving Eq. (7.20) and then estimate the next improved solution as

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - e^{(1)}$$

If we need further improvement, we can repeat the process by calculating $e^{(2)}$ using

$$\mathbf{A}e^{(2)} = \mathbf{r}^{(2)}$$

where

$$\mathbf{r}^{(2)} = \mathbf{A}\mathbf{x}^{(2)} - \mathbf{b}$$

We get the next estimate as

$$\mathbf{x}^{(3)} = \mathbf{x}^{(2)} - e^{(2)}$$

This process can be repeated as many times as we wish to achieve a desired accuracy. Algorithm 7.6 lists the steps for implementing the iterative refinement process.

Iterative refinement

1. Obtain LU factorisation of \mathbf{A}
2. Compute the solution \mathbf{x} by forward and back substitutions
3. Find the residual vector \mathbf{r} using

$$\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}$$
4. Compute the error using

$$\mathbf{A}e = \mathbf{r}$$

 by forward and back substitutions

(Contd.)

(contd.)

5. Set $\mathbf{x} = \mathbf{x} - \mathbf{e}$
6. If e is sufficiently small
stop
otherwise
go to step 3

Algorithm 7.6**7.9****ILL-CONDITIONED SYSTEMS**

As pointed out in the beginning of the chapter, arriving at a proper solution depends on the condition of the system. Systems where small changes in the coefficient result in large deviations in the solution are said to be *ill-conditioned systems*. A wide range of answers can satisfy such equations. This means that a completely erroneous set of answers may produce zero (or near zero) residuals. This is illustrated in Example 7.7.

Ill-conditioned systems are very sensitive to roundoff errors. These errors during computing process may induce small changes in the coefficients which, in turn, may result in a large error in the solution.

We can decide the condition of a system either graphically or mathematically. Graphically, if two lines appear almost parallel, then we can say the system is ill-conditioned, since it is hard to decide just at which point they intersect.

The problem of ill-condition can be mathematically described as follows: consider a two equation system

$$a_{11}x_1 + a_{12}x_2 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$

If these two lines are almost parallel, their slopes must by nearly equal. That is

$$\frac{a_{11}}{a_{12}} \approx \frac{a_{21}}{a_{22}}$$

Alternatively,

$$a_{11}a_{22} \approx a_{12}a_{21}$$

or

$$a_{11}a_{22} - a_{12}a_{21} \approx 0$$

Note that $a_{11}a_{22} - a_{12}a_{21}$ is the determinant of the coefficient matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

This shows that the determinant of an ill-conditioned system is very small or nearly equal to zero.

In partial pivoting technique, we try to interchange the rows so that the largest element becomes the pivot element. This is done basically to avoid a division by zero or nearly zero point. Even the largest element in that column may happen to be zero (or nearly zero). Such situations arise when the systems are ill-conditioned. Solution of these systems may not be meaningful.

Example 7.7

Solve the following equations

$$2x_1 + x_2 = 25$$

$$2.001x_1 + x_2 = 25.01$$

and thereby discuss the effect of ill-conditioning.

$$x_1 = \frac{25 \times 1 - 25.01 \times 1}{2 \times 1 - 2.001 \times 1} = 10$$

$$x_2 = \frac{25.01 \times 2 - 25 \times 2.001}{2 \times 1 - 2.001 \times 1} = 5$$

Let us change the coefficient of x_1 in the second equation to 2.0005. Now the values of x_1 and x_2 are

$$x_1 = \frac{25 - 25.01}{2 - 2.0005} = 20$$

$$x_2 = \frac{25.01 \times 2 - 25 \times 2.0005}{2 - 2.0005} = -15$$

Compare the results. A small change in one of the coefficients has resulted in a large change in the result.

If we substitute these values back into the equations, we get the residuals as

$$r_1 = 40 - 15 - 25 = 0$$

$$r_2 = 40.02 - 15 - 25.01 = 0.01$$

The first equation is satisfied exactly and the residual of the second is small. It appears as if the results are correct. This illustrates the effect of roundoff errors on ill-conditioned systems.

7.10 MATRIX INVERSION METHOD

Another way to obtain the solution of an equation of type

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad (7.21)$$

is by using matrix algebra. Multiply each side of Eq. (7.21) by the inverse of \mathbf{A} . This yields

$$\mathbf{A}^{-1} \mathbf{A} \mathbf{x} = \mathbf{A}^{-1} \mathbf{b} \quad (7.22)$$

since $\mathbf{A}^{-1} \mathbf{A} = \mathbf{I}$, the identity matrix, equation (7.22) becomes

$$\boxed{\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}} \quad (7.23)$$

Equation (7.23) gives the solution for \mathbf{x} .

This approach becomes useful when we need to solve Eq. (7.21) for different sets of \mathbf{b} values while \mathbf{A} remains the same.

Computing Matrix Inverse

Although the Gauss-Jordan method is more complicated compared to Gauss elimination method, this method provides a simple approach for obtaining the inverse of a matrix.

This is done as follows:

1. Augment the coefficient matrix \mathbf{A} with an identity matrix as shown below:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array} \right]$$

2. Apply the Gauss-Jordan method to the augmented matrix to reduce \mathbf{A} to an identity matrix. The result will be as shown below:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & a'_{11} & a'_{12} & a'_{13} \\ 0 & 1 & 0 & a'_{21} & a'_{22} & a'_{23} \\ 0 & 0 & 1 & a'_{31} & a'_{32} & a'_{33} \end{array} \right]$$

The right-hand side of the augmented matrix is the inverse of \mathbf{A} . Now, we can obtain the solution as follows:

$$x_1 = a'_{11} \times b_1 + a'_{12} \times b_2 + a'_{13} \times b_3$$

$$x_2 = a'_{21} \times b_1 + a'_{22} \times b_2 + a'_{23} \times b_3$$

$$x_3 = a'_{31} \times b_1 + a'_{32} \times b_2 + a'_{33} \times b_3$$

Condition Number

The inverse matrix can also be used to decide whether a system is ill-conditioned. Let us define a matrix \mathbf{C} as

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{A}^{-1} \quad (7.24)$$

If \mathbf{C} is close to identity matrix, then the system is well-conditioned. If not, it indicates ill-conditioning.

Equation (7.24) can be expressed using the concept of matrix norm as follows:

$$\text{cond}(\mathbf{A}) = ||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}|| \quad (7.25)$$

where $\text{cond}(\mathbf{A})$ is called the *condition number* and $||\mathbf{A}||$ is the "norm" of the matrix \mathbf{A} . The norm is defined as follows

$$||\mathbf{A}|| = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$$

This is known as *row-sum norm*. In this norm, the sum of the absolute values of the elements for each row is computed and the largest of these is taken as the norm.

The smaller the condition number, the better is matrix \mathbf{A} suited to numerical computation.

7.11 SUMMARY

In this chapter we studied systems of linear equations. Among the two popular approaches available for solving these equations, we considered the elimination (also known as direct) methods in detail. They include:

- Gauss elimination method (basic)
- Gauss elimination with pivoting
- Gauss-Jordan method
- LU decomposition method using Dolittle algorithm
- Matrix inverse method

We also stated that other LU decomposition techniques, such as Crout algorithm and Cholesky's factorisation, may be applied to solve the equations.

Direct methods introduce roundoff errors. We presented an iterative refinement procedure for improving the final result.

Computer programs with test results have been given for the following methods:

- Basic Gauss elimination method
- Gauss elimination with partial pivoting
- Dolittle LU decomposition method

Key Terms

Back substitution
Basic Gauss' elimination
Cholesky's algorithm
Cholesky's factorisation
Complete pivoting
Condition number
Crout algorithm
Crout LU decomposition

Lower triangular matrix
LU decomposition
Matrix inversion
Matrix norm
Method of square roots
Modular structure
Nonlinear
Normalisation

(Contd.)

<i>Decomposition</i>	<i>Over-determined</i>
<i>Dependent equations</i>	<i>Partial pivoting</i>
<i>Direct method</i>	<i>Pivot element</i>
<i>Dolittle LU decomposition</i>	<i>Pivot equation</i>
<i>Elimination approach</i>	<i>Pivoting</i>
<i>Forward elimination</i>	<i>Residual vector</i>
<i>Gauss elimination</i>	<i>Row-sum norm</i>
<i>Gauss-Jordan method</i>	<i>Simultaneous equations</i>
<i>Homogeneous equations</i>	<i>Singular systems</i>
<i>Ill-conditioned system</i>	<i>Triangularisation</i>
<i>Inconsistent equations</i>	<i>Under-determined</i>
<i>Infinite solutions</i>	<i>Unique solution</i>
<i>Iterative refinement</i>	<i>Upper triangular matrix</i>
<i>Linear</i>	<i>Zero residuals</i>

REVIEW QUESTIONS

1. Describe the two basic approaches that are employed for solving a system of linear equations.
2. What are the four possible solution conditions of a system of linear equations? Explain each one of them with an illustration.
3. Explain under-determined and over-determined systems.
4. What is meant by homogenous equations?
5. State some basic rules that are used in the elimination method of solving simultaneous linear equations.
6. Explain the basic concepts used in the Gauss elimination approach.
7. What is triangularisation of equations? How does it help obtain the solution?
8. What is pivoting? Distinguish between partial pivoting and complete pivoting.
9. How does pivoting improve accuracy of solution?
10. Compare critically Gauss elimination and Gauss-Jordan methods of solving simultaneous equations.
11. Show that Gauss-Jordan takes about 50% more operations than Gauss elimination for the case of three equations.
12. What is Dolittle decomposition? How is it different from Crout decomposition?
13. What is Cholesky's factorisation?
14. What is iterative refinement? How is it used to improve the accuracy of results?
15. What is meant by ill-conditioned systems?
16. Can we solve an ill-conditioned system? If yes, how?
17. What is condition number of a system? How is it computed?

REVIEW EXERCISES

~~Step 1~~
~~Step 2~~
~~Step 3~~
~~NWB process:~~
 Solve the following system of equations using simple elimination.

$$x + y + z = 6$$

$$2x - y + 3z = 4$$

$$4x + 5y - 10z = 13$$

~~Q~~ Show that the following system of equations has no solution.

$$-2x + y + 3z = 12$$

$$x + 2y + 5z = 4$$

$$6x - 3y - 8z = 24$$

~~Q~~ Show that the following system of equations has infinite number of solutions.

$$x + y + z = 20$$

$$2x - 3y + z = -5$$

$$3x - 2y + 2z = 15$$

~~Q~~ Solve the following systems of equations by simple Gauss elimination.

~~NWB~~

$$2x_1 + 3x_2 + 4x_3 = 5$$

$$3x_1 + 4x_2 + 5x_3 = 6$$

$$4x_1 + 5x_2 + 6x_3 = 7$$

$$(b) \quad 2x_1 + 3x_2 + 4x_3 = 5$$

$$3x_1 + 4.5x_2 + 5x_3 = 6$$

$$4x_1 + 5x_2 + 6x_3 = 7$$

$$(c) \quad x_1 + 2x_2 + 3x_3 = 8$$

$$2x_1 + 4x_2 + 9x_3 = 8$$

$$4x_1 + 3x_2 + 2x_3 = 2$$

~~✓~~ 5. Solve the systems in Exercise 4 using partial pivoting.
 6. Solve the systems in Exercise 4 using complete pivoting.

7. Using Gauss elimination with partial pivoting, solve the following sets of equations.

$$(a) \quad 2x_1 + x_2 + x_3 - 2x_4 = 0$$

$$4x_1 + 2x_3 + x_4 = 8$$

$$3x_1 + 2x_2 + 2x_3 = 7$$

$$x_1 + 3x_2 + 2x_3 = 3$$

$$(b) \quad x_1 + x_2 - 2x_3 = 3$$

$$4x_1 - 2x_2 + x_3 = 5$$

$$3x_1 - x_2 + 3x_3 = 8$$

8. Solve the following systems of equations by Gauss-Jordan method

Note:

$$(a) \begin{aligned}x_1 + 2x_2 - 3x_3 &= 4 \\ 2x_1 + 4x_2 - 6x_3 &= 8 \\ x_1 - 2x_2 + 5x_3 &= 4\end{aligned}$$

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

9. Find the Dolittle LU decompositions of the coefficient matrices of the systems in Exercises 7 and 8.

10. Solve the systems in Exercises 7 and 8 using the matrices L and U found in exercise 9 by forward and backward substitutions.

11. Find the Cholesky decomposition of the matrix

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

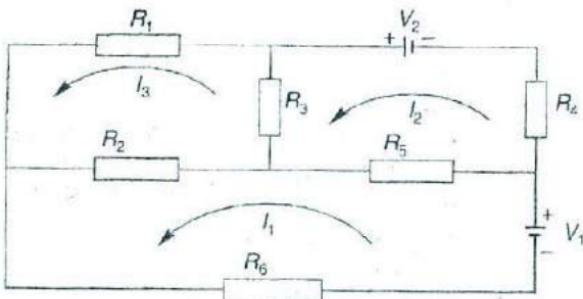
12. Find the inverse of the following matrices using Gauss-Jordan elimination technique

$$(a) \begin{bmatrix} 2 & 3 & 4 \\ 4 & 2 & 3 \\ 3 & 4 & 2 \end{bmatrix}$$

$$(b) \begin{bmatrix} 1 & 2 & -3 \\ 2 & 4 & -6 \\ -1 & -2 & 3 \end{bmatrix}$$

13. Find the condition numbers of the coefficient matrices of systems in Exercise 4.

14. Consider the following electrical network connecting six resistors and two batteries:



Ohm's law states that the voltage across a resistor equals the current through it multiplied by its resistance. Using this law, we can set up the following equations:

$$\begin{aligned}R_6I_1 + R_5(I_1 - I_2) + R_2(I_1 - I_3) &= V_1 \\ R_4I_2 + R_3(I_2 - I_3) + R_5(I_2 - I_1) &= V_2 \\ R_1I_3 + R_2(I_3 - I_1) + R_3(I_3 - I_2) &= 0\end{aligned}$$

Assuming $R_1 = R_2 = R_3 = 2$, $R_4 = R_5 = R_6 = 3$ and $V_1 = V_2 = 5$, Solve the system of equations for currents I_1 , I_2 and I_3 using Gauss elimination or Gauss-Jordan method.

15. A company produces four different products. They are processed through four different departments A , B , C and D . The table below gives the number of hours that each department spends on each product.

Department	Products			
	P ₁	P ₂	P ₃	P ₄
D1	2	3	1	2
D2	1	2	2	4
D3	3	4	4	5
D4	3	2	2	3

Total production hours available each month in each department is as follows:

Department	D1	D2	D3	D4
Hours	265	260	352	250

Formulate the appropriate system of linear equations to determine the quantities of the four products that can be produced in each month, so that all the hours available in all departments are fully utilised. Determine how much time each department spends for each product.

PROGRAMMING PROJECTS

1. Program LEG2 solves a system of linear equations using Gauss elimination with partial pivoting. Modify the program to implement complete pivoting.
2. Develop a program to factorise a matrix using Cholesky's algorithm.
3. Design and develop a program to implement the Gauss-Jordan elimination method for solving a system of linear equations.
4. Write a program to implement the Crout decomposition solution of linear equations.
5. Construct a program to implement the iterative refinement process as given in Algorithm 7.6.

Iterative Solution of Linear Equations

8.1 NEED AND SCOPE

Direct methods discussed in the previous chapter pose some problems when the systems grow larger or when most of the coefficients are zero. They require prohibitively large number of floating point operations and, therefore, not only become time consuming but also severely affect the accuracy of the solution due to roundoff errors. In such cases, iterative methods provide an alternative. For instance, ill-conditioned systems can be solved by iterative methods without facing the problem of roundoff errors.

The following three iterative methods are discussed in this chapter:

- 1. Jacobi iteration method
- 2. Gauss-Seidel iteration method
- 3. Successive over relaxation method

Like all other iterative processes, these methods introduce truncation errors and, therefore, it is important to understand the magnitude of this error as well as the rate of convergence of the iteration process.

8.2 JACOBI ITERATION METHOD

Jacobi method is one of the simple iterative methods. The basic idea behind this method is essentially the same as that for the fixed point method discussed in Chapter 6. Recall that an equation of the form

$$f(x) = 0$$

can be rearranged into a form

$$x = g(x)$$

The function $g(x)$ can be evaluated iteratively using an initial approximation x as follows:

$$x_{i+1} = g(x_i) \quad \text{for } i = 0, 1, 2, \dots$$

Jacobi method extends this idea to a system of equations. It is a direct substitution method where the values of unknowns are improved by substituting directly the previous values.

Let us consider a system of n equations in n unknowns.

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

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

$$\vdots \quad \vdots \quad \vdots$$

$$a_{n1} x_1 + a_{n2} x_2 + \dots + a_{nn} x_n = b_n$$

(8.1)

We rewrite the original system as

$$x_1 = \frac{b_1 - (a_{12} x_2 + a_{13} x_3 + \dots + a_{1n} x_n)}{a_{11}}$$

$$x_2 = \frac{b_2 - (a_{21} x_1 + a_{23} x_3 + \dots + a_{2n} x_n)}{a_{22}}$$

$$\vdots \quad \vdots \quad \vdots$$

$$x_n = \frac{b_n - (a_{n1} x_1 + a_{n2} x_2 + \dots + a_{n-1} x_{n-1})}{a_{nn}}$$

(8.2)

Now, we can compute x_1, x_2, \dots, x_n by using initial guesses for these values. These new values are again used to compute the next set of x values. The process can continue till we obtain a desired level of accuracy in the x values.

In general, an iteration for x_i can be obtained from the i th equation as follows

$$x_i^{(k+1)} = \frac{b_i - (a_{i1} x_1^{(k)} + a_{i2} x_2^{(k)} + a_{i3} x_3^{(k)} + \dots + a_{in} x_n^{(k)})}{a_{ii}} \quad (8.3)$$

The computational steps of Jacobi iteration process are given in Algorithm 8.1.

Jacobi iteration method

1. Obtain n , a_{ij} and b_i values.
2. Set $x_{0j} = b_j/a_{jj}$ for $j = 1, \dots, n$
3. Set key = 0
4. For $i = 1, 2, \dots, n$
 - (i) Set sum = b_i
 - (ii) For $j = 1, 2, \dots, n$ ($j \neq i$)

Set sum = sum - $a_{ij}x_{0j}$

Repeat j
 - (iii) Set $x_i = \text{sum}/a_{ii}$
 - (iv) If key = 0 then

if $\left| \frac{x_i - x_{0i}}{x_i} \right| > \text{error}$ then

set key = 1

Repeat i
5. If key = 1 then

set $x_{0i} = x_i$

go to step-3
6. Write results

Algorithm 8.1**Example 8.1**

Obtain the solution of the following system using the Jacobi iteration method

$$2x_1 + x_2 + x_3 = 5$$

$$3x_1 + 5x_2 + 2x_3 = 15$$

$$2x_1 + x_2 + 4x_3 = 8$$

First, solve the equations for unknowns on the diagonal. That is

$$x_1 = \frac{5 - x_2 - x_3}{2}$$

$$x_2 = \frac{15 - 3x_1 - 2x_3}{5}$$

$$x_3 = \frac{8 - 2x_1 - x_2}{4}$$

If we assume the initial values of x_1 , x_2 and x_3 to be zero, then we get

$$x_1^{(1)} = \frac{5}{2} = 2.5$$

$$x_2^{(1)} = \frac{15}{5} = 3$$

$$x_3^{(1)} = \frac{8}{4} = 2$$

(Note that these values are nothing but $x_i^1 = b_i/a_{ii}$)

For the second iteration, we have

$$x_1^{(2)} = \frac{5 - 3 - 2}{2} = 0$$

$$x_2^{(2)} = \frac{15 - 3 \times 2.5 - 2 \times 2}{5} = \frac{3.5}{5} = 0.7$$

$$x_3^{(2)} = \frac{8 - 2 \times 2.5 - 3}{4} = 0$$

After third iteration,

$$x_1^{(3)} = \frac{5 - 0.7}{2} = 2.15$$

$$x_2^{(3)} = \frac{15 - 3 \times 0 - 2 \times 0}{5} = 3$$

$$x_3^{(3)} = \frac{8 - 2 \times 0 - 0.7}{4} = 1.825$$

After fourth iteration,

$$x_1^{(4)} = \frac{5 - 3 - 1.825}{2} = 0.0875$$

$$x_2^{(4)} = \frac{15 - 3 \times 2.15 - 2 \times 1.825}{4} = 1.225$$

$$x_3^{(4)} = \frac{8 - 2 \times 2.15 - 3}{4} = 0.175$$

The process can be continued till the values of x reach a desired level of accuracy.

Program JACIT

The program JACIT solves a system of n linear equations using the Jacobi iteration method as detailed in Algorithm 8.1. The main program reads interactively the system specifications and displays the results on the screen. The solution algorithm is implemented through the subroutine JACOBI.

The subprogram JACOBI, while computing the solution vector \mathbf{X} , tests for the accuracy as well as the convergence. The computing process stops either when the desired accuracy is achieved or when the process does not converge within a specified number of iterations.

```

* -----
*      PROGRAM 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 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
* -----
REAL A,B,X,EPS
INTEGER N,COUNT,MAXIT,STATUS
PARAMETER(EPS=0.000001,MAXIT = 50)
DIMENSION A(10,10), B(10), X(10)

WRITE(*,*)
WRITE(*,*)      'SOLUTION BY JACOBI ITERATION'
WRITE(*,*)
WRITE(*,*)      'What is the size of the system(n)?'
READ(*,*)      N
WRITE(*,*)      'Input coefficients a(i,j), row-wise',
WRITE(*,*)      'one row on each line'
DO 20 I = 9, N
  READ(*,*) (A(I,J),J=1,N)
20 CONTINUE
WRITE(*,*)      'Input vector b'

```

```

READ(*,*) (B(I), I = 1, N)
CALL JACOBI(N,A,B,X,EPS,COUNT,MAXIT,STATUS)
IF (STATUS .EQ. 2) THEN
    WRITE(*,*)
    WRITE(*,*) 'NO CONVERGENCE IN', MAXIT,
    'ITERATIONS'
    WRITE(*,*)
ELSE
    WRITE(*,*)
    WRITE(*,*) 'SOLUTION VECTOR X'
    WRITE(*,*)
    WRITE(*,*) (X(I), I = 1, N)
    WRITE(*,*)
    WRITE(*,*) 'ITERATIONS = ', COUNT
    WRITE(*,*)
ENDIF
STOP
END

```

```

* ----- End of main program JACIT -----
*
* -----
* SUBROUTINE JACOBI(N,A,B,X,EPS,COUNT,MAXIT,STATUS)
*
* -----
* Subroutine
*   This subroutine solves a system of n linear
*   equations using the Jacobi iteration method
*
* -----
* Arguments
* Input
*   N - Number of equations
*   A - Matrix of coefficients of the equations
*   B - Right side vector
*   EPS - Error bound
*   MAXIT - Maximum iterations allowed
* Output
*   X - Solution vector
*   COUNT - Number of iterations done
*   STATUS - Convergence status
*
* -----
* Local Variables
*   XO, SUM
*
* -----
* Functions invoked
*   ABS
*
* -----
* Subroutines called
*   NIL

```

```

* -----
      INTEGER N,KEY,COUNT,MAXIT,STATUS
      REAL A,B,X,XO,EPS
      DOUBLE PRECISION SUM
      INTRINSIC ABS
      DIMENSION A(10,10),B(10),X(10),XO(10)
* Initial values of X
      DO 10 I=1,N
         XO(I) = B(I)/A(I,I)
10   CONTINUE
      COUNT = 1
99   KEY = 0
* Computing values of X(I)
      DO 30 I = 1,N
         SUM = B(I)
         DO 20 J = 1,N
            IF(I.EQ.J) GOTO 20
            SUM = SUM - A(I,J) * XO(J)
20   CONTINUE
         X(I) = SUM/A(I,I)
         IF(KEY .EQ. 0) THEN
* Testing for accuracy
            IF(ABS((X(I)-XO(I))/X(I)) .GT. EPS) THEN
               KEY = 1
            ENDIF
            ENDIF
30   CONTINUE
         IF(KEY.EQ.1) THEN
* Testing for convergence
            IF(COUNT .EQ. MAXIT) THEN
               STATUS = 2
               RETURN
            ELSE
               STATUS = 1
               DO 40 I = 1,N
                  XO(I) = X(I)
40   CONTINUE
            ENDIF
            COUNT = COUNT+1
            GO TO 11
         ENDIF
         RETURN
      END
* ----- End of subroutine JACOBI -----

```

Test Run Results The program was used to solve the following system of equations:

$$3x_1 + x_2 = 5$$

$$x_1 - 3x_2 = 5$$

The interactive computer output is given below:

SOLUTION BY JACOBI ITERATION

What is the size of the system(n)?

2

Input coefficients a(i,j), row-wise
one row on each line

3 1

1 -3

Input vector b

5 5

SOLUTION VECTOR X

2.0000000 -9.999998E-001

ITERATIONS = 14

Stop - Program terminated.

Now, rearrange the equations as shown below and then use program JACIT to solve the system.

$$x_1 - 3x_2 = 5$$

$$3x_1 + x_2 = 5$$

The output now is as given below:

SOLUTION BY JACOBI ITERATION

What is the size of the system(n)?

2

Input coefficients a(i, j), row-wise
one row on each line

1 -3

3 1

Input vector b

5 5

NO CONVERGENCE IN 50 ITERATIONS

Stop - Program terminated.

Note that the same two equations, when their positions are interchanged, do not produce required results even after 50 iterations. Convergence is discussed in Section 8.5.

8.3 GAUSS-SEIDEL METHOD

Gauss-Seidel method is an improved version of Jacobi iteration method. In Jacobi method, we begin with the initial values

$$x_1^{(0)}, x_2^{(2)}, \dots, x_n^{(0)}$$

and obtain next approximation

$$x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)}$$

Note that, in computing $x_2^{(1)}$, we used $x_1^{(0)}$ and not $x_1^{(1)}$ which has just been computed. Since, at this point, both $x_1^{(0)}$ and $x_1^{(1)}$ are available, we can use $x_1^{(1)}$ which is a better approximation for computing $x_2^{(1)}$. Similarly, for computing $x_3^{(1)}$, we can use $x_1^{(1)}$ and $x_2^{(1)}$ along with $x_4^{(0)}, \dots, x_n^{(0)}$. This idea can be extended to all subsequent computations. This approach is called the *Gauss-Seidel* method.

The Gauss-Seidel method uses the most recent values of x as soon as they become available at any point of iteration process. During the $(k+1)$ th iteration of Gauss-Seidel method, x_i takes the form

$$x_i^{(k+1)} = \frac{b_i - (a_{i1}x_1^{(k+1)} + \dots + a_{i(i-1)}x_{i-1}^{(k+1)} + a_{i(i+1)}x_{i+1}^{(k)} + \dots + a_{in}x_n^{(k)})}{b_i} \quad (8.4)$$

When $i = 1$, all superscripts in the right-hand side become (k) only. Similarly, when $i = n$, all become $(k+1)$. Figure 8.1 illustrates pictorially the difference between the Jacobi and Gauss-Seidel method.

x_1	=	$(b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$	
x_2	=	$(b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$	Iteration 1
x_3	=	$(b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$	
 ↓			
x_1	=	$(b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$	
x_2	=	$(b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$	Iteration 2
x_3	=	$(b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$	

(a) Jacobi method

x_1	=	$(b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$	
x_2	=	$(b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$	Iteration 1
x_3	=	$(b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$	
 ↓			
x_1	=	$(b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$	
x_2	=	$(b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$	Iteration 2
x_3	=	$(b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$	

(b) Gauss-Seidel method

Fig. 8.1 Comparison of Jacobi and Gauss-Seidel methods

Example 8.2

Obtain the solution of the following system using Gauss-Seidel iteration method

$$2x_1 + x_2 + x_3 = 5$$

$$3x_1 + 5x_2 + 2x_3 = 15$$

$$2x_1 + x_2 + 4x_3 = 8$$

$$x_1 = (5 - x_2 - x_3)/2$$

$$x_2 = (15 - 3x_1 - 2x_3)/5$$

$$x_3 = (8 - 2x_1 - x_2)/4$$

Assuming initial value as $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$

<i>Iteration 1</i>	$x_1 = (5 - 0 - 0)/2$	= 2.5
	$x_2 = (15 - 3 \times 2.5 - 0)/5$	= 1.5
	$x_3 = (8 - 2 \times 2.5 - 1.5)/4$	= 0.4 (rounded to one decimal)

<i>Iteration 2</i>	$x_1 = (5 - 1.5 - 0.4)/2$	= 1.6
	$x_2 = (15 - 3 \times 1.6 - 2 \times 0.4)/5$	= 1.9
	$x_3 = (8 - 2 \times 1.6 - 1.9)/4$	= 0.7

We can continue this process until we get $x_1 = 1.0$, $x_2 = 2.0$ and $x_3 = 1.0$ (correct answers)

Algorithm

Gauss-Seidel algorithm is a simple modification of the algorithm of the Jacobi method. Note that once a new value of $x_i^{(k+1)}$ has been calculated and compared with the previous values of $x_i^{(k)}$, the previous value is no longer required and, therefore, the previous value can be replaced by the new one. This implies that we need not use two vectors (one to store previous values and another to store new values) for storing x values. We need to use only one vector x that stores always the latest values of x . This is illustrated in Algorithm 8.2

Gauss-Seidel method

1. Obtain n , a_{ij} and b_i values
2. Set $x_i = b/a_{ii}$ for $i = 1$ to n
3. Set key = 0
4. For $i = 1$ to n
 - (i) Set sum = b ,
 - (ii) For $j = 1$ to n ($j \neq i$)
 - Set sum = sum - $a_{ij} x_j$
 - Repeat j

(Contd.)

(iii) Set dummy = sum / a_{ii}
(iv) If key = 0 then

$$\text{if } \left| \frac{\text{dummy} - x_i}{\text{dummy}} \right| > \text{error} \text{ then}$$

set key = 1
(v) Set $x_i = \text{dummy}$
Repeat i
5. If key = 1 then
go to step 3
6. Write results

Algorithm 8.2**Program GASIT**

Like JACIT, the program GASIT also solves a system of n linear equations but employs the Gauss-Seidel iteration method as detailed in Algorithm 8.2. The iteration algorithm is implemented with the help of a subprogram called GASEID.

```

* -----
*      PROGRAM 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
*
* -----
REAL A,B,X,EPS
INTEGER N,COUNT,MAXIT,STATUS
PARAMETER(EPS=0.000001,MAXIT=50)
DIMENSION A(10,10), B(10), X(10)

WRITE(*,*)
WRITE(*,*)      'SOLUTION BY GAUSS-SEIDEL ITERATION'
WRITE(*,*)

WRITE(*,*) 'What is the size of the system(n)?'
READ(*,*) N
WRITE(*,*)      'Input coefficients a(i,j), row-wise'
WRITE(*,*)      'one row on each line'

DO 20 I = 1,N
    READ(*,*) (A(I,J),J=1,N)
20 CONTINUE

WRITE(*,*)      'Input vector b'
READ(*,*) (B(I), I = 1, N)

CALL GASEID(N,A,B,X,EPS,COUNT,MAXIT,STATUS)

IF (STATUS .EQ. 2) THEN
    WRITE(*,*)
    WRITE(*,*) 'NO CONVERGENCE IN', MAXIT,
               'ITERATIONS'
    WRITE(*,*)
ELSE
    WRITE(*,*)
    WRITE(*,*) 'SOLUTION VECTOR X'
    WRITE(*,*)
    WRITE(*,*) (X(I), I = 1, N)
    WRITE(*,*)
    WRITE(*,*) 'ITERATIONS = ', COUNT
    WRITE(*,*)
ENDIF

STOP
END

```

```

* ----- End of main program GASIT -----
* -----
SUBROUTINE GASEID(N,A,B,X,EPS,COUNT,MAXIT,STATUS)
* -----

```

```

* Subroutine
* This subroutine solves a system of linear
* equations using Gauss-Seidel iteration algorithm
*
* -----
* Arguments
* Input
*   N - Number of equations
*   A - Coefficient matrix
*   B - Right side vector
*   EPS - Error bound
*   MAXIT - Maximum iterations allowed
* Output
*   X - Solution vector
*   COUNT - Number of iterations done
* STATUS - Status of convergence
*
* -----
* Local Variables
*   DUMMY,SUM,KEY
*
* -----
* Functions invoked
*   ABS
*
* -----
* Subroutines called
*   NIL
*
* -----
INTEGER N,KEY,COUNT,MAXIT,STATUS
REAL A,B,X,EPS,DUMMY
DOUBLE PRECISION SUM
DIMENSION A(10,10),B(10),X(10),X0(10)
INTRINSIC ABS

* Initial values of X
DO 10 I = 1,N
    X(I) = B(I) / A(I,I)
10 CONTINUE

COUNT = 1
11 KEY = 0

* Computing X(I) values
DO 30 I = 1,N
    SUM = B(I)
    DO 20 J = 1,N
        IF(I.EQ.J) GOTO 20
        SUM = SUM - A(I,J) * X(J)
20 CONTINUE
30 X(I) = SUM

```

```

20      CONTINUE
        DUMMY = SUM/A(I,I)
        IF(KEY .EQ. 0) THEN
*          Testing for accuracy
        IF(ABS((DUMMY - X(I))/DUMMY) .GT. EPS) THEN
            KEY = 1
        ENDIF
        ENDIF
        X(I) = DUMMY
30      CONTINUE
        IF(KEY .EQ. 1) THEN
*          Testing for convergence
        IF(COUNT .EQ. MAXIT) THEN
            STATUS = 2
            RETURN
        ELSE
            STATUS = 1
            COUNT = COUNT + 1
            GOTO 11
        ENDIF
        ENDIF
        RETURN
    END
* ----- End of subroutine CASEID-----*

```

Test Run Results The program was used to solve two different sets of equations and the results are as follows:

First set

```

        SOLUTION BY GAUSS-SEIDEL ITERATION
What is the size of the system(n)?
3
Input Coefficients a(i,j), row-wise
one row on each line
2 1 3
4 4 7
2 5 9
Input vector b
1 1 3
SOLUTION VECTOR X
-4.999992E-001 -9.999992E-001 9.999993E-001
ITERATIONS = 38
Stop - Program terminated.

```

Second set

SOLUTION BY GAUSS-SEIDEL ITERATION

What is the size of the system(n)?

3

Input coefficients a(i,j), row-wise
one row on each line

7 63 0

3 30 0

2 28 10

Input vector b

13.3 3.9 6.9

NO CONVERGENCE IN

50 ITERATIONS

Stop - Program terminated.

METHOD OF RELAXATION

Relaxation method represents a slightly modified version of the Gauss-Seidel method. The modification is aimed at faster convergence. The basic idea is to take the change produced in a Gauss-Seidel iteration step and extrapolate the new value by a factor r of this change. The new relaxation value is given by

$$\begin{aligned}x_{ir}^{(k+1)} &= x_i^{(k)} + r(x_i^{(k+1)} - x_i^{(k)}) \\&= rx_i^{(k+1)} + (1 - r)x_i^{(k)}\end{aligned}\quad (8.5)$$

The parameter r is called the *relaxation parameter*. This step is applied "successively" to each component of vector \mathbf{x} during iteration process and, therefore, the method is known as *successive relaxation method*.

The parameter r may be assigned a value between 0 and 2. We have the following possibilities:

$0 < r < 1$ under-relaxation

$r = 1$ no relaxation ($x_{ir}^{(k+1)} = x_i^{(k+1)}$)

$1 < r < 2$ over-relaxation

For values of r between 1 and 2, an extra weight is placed on the present value and Eq. (8.5) really represents an extrapolation. The intention here is to push the estimate closer to the solution. This method, when $1 < r < 2$, is popularly known as *successive over-relaxation* (or SOR) method. It is also known as *simultaneous over-relaxation* method.

The SOR technique can be easily implemented by a simple modification of the Gauss-Seidel algorithm. The relaxation value is obtained using Eq. (8.5) at the end of evaluation of each value of \mathbf{x} . The extrapolated value becomes the new value of \mathbf{x} for the next cycle. Equation (8.5) can be simply implemented as

$$x_i^{(k+1)} = r x_i^{(k+1)} + (1 - r) x_i^{(k)} \quad (8.6)$$

That is, the old value of $x_i^{(k+1)}$ is replaced by the new value of $x_i^{(k+1)}$. The implementation of this step is shown in Algorithm 8.3.

The choice of value of r depends on the problem and is often decided empirically.

SOR method

Algorithm is the same as Algorithm 8.2, except the statement

(iii) Set dummy = sum/a_{ii}
is replaced by a pair of statements

Set dummy = sum/a_{ii}

Set dummy = r × dummy + (1 - r) x_i

Algorithm 8.3

8.5

CONVERGENCE OF ITERATION METHODS

Condition for Convergence

We know that the iteration methods presented here are based on the basic idea of the fixed point method discussed in Chapter 6. We have shown that sufficient condition for convergence for solving one nonlinear equation is

$$|G'(x)| < 1$$

and for two nonlinear equations, $F(x, y)$ and $G(x, y)$, are

$$\left| \frac{\partial F}{\partial x} \right| + \left| \frac{\partial G}{\partial x} \right| < 1 \quad (8.7)$$

$$\left| \frac{\partial F}{\partial y} \right| + \left| \frac{\partial G}{\partial y} \right| < 1 \quad (8.8)$$

These conditions apply to linear equations as well. Therefore, we can use these conditions in the Jacobi and Gauss-Seidel iteration methods.

For the sake of simplicity, let us consider a two-equation linear system. We can express the Gauss-Seidel algorithm as follows:

$$\begin{aligned} x_1 &= F(x_1, x_2) = \frac{1}{a_{11}} (b_1 - a_{12}x_2) \\ &= \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} x_2 \end{aligned} \quad (8.9)$$

$$\begin{aligned}x_2 &= G(x_1, x_2) = \frac{1}{a_{22}}(b_2 - a_{21}x_1) \\&= \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1\end{aligned}\quad (8.10)$$

The partial derivatives of these equations are

$$\frac{\partial F}{\partial x_1} = 0, \quad \frac{\partial F}{\partial x_2} = -\frac{a_{12}}{a_{11}}$$

and

$$\frac{\partial G}{\partial x_1} = -\frac{a_{21}}{a_{22}}, \quad \frac{\partial G}{\partial x_2} = 0$$

Substituting these values in Eqs (8.7) and (8.8), we get

$$\left| \frac{a_{21}}{a_{22}} \right| < 1 \quad \text{and} \quad \left| \frac{a_{12}}{a_{11}} \right| < 1$$

This means that

$$|a_{11}| > |a_{12}| \quad (8.11)$$

and

$$|a_{22}| > |a_{21}| \quad (8.12)$$

That is, the absolute value of diagonal element must be greater than that of the off-diagonal element for each row.

The above derivation can be extended to a general system of n equations to show that

$$\boxed{|a_{ii}| > \sum_{j=1}^n |a_{ij}|, \quad i \neq j} \quad (8.13)$$

For each row, the absolute value of the diagonal element should be greater than the sum of absolute values of the other elements in the equation. Remember that this condition is sufficient, but not necessary, for convergence. Some systems may converge even if this condition is not satisfied.

Systems that satisfy the condition Eq. (8.13) are called *diagonally dominant* systems. Convergence of such systems are guaranteed.

Rate of Convergence

Consider the iterative Eqs (8.9) and (8.10). At $(k+1)$ th iteration, we have

$$x_1^{(k+1)} = \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} x_2^k \quad (8.14)$$

$$x_2^{(k+1)} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}} x_1^{k+1} \quad (8.15)$$

Substituting for $x_1^{(k+1)}$ in Eq. (8.15), we get

$$x_2^{(k+1)} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}} \left[\frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} x_2^k \right] \quad (8.16)$$

Similarly, we have

$$x_2^{(k+2)} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}} \left[\frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} x_2^{(k+1)} \right] \quad (8.17)$$

Subtracting Eq. (8.16) from Eq. (8.17), we get

$$x_2^{(k+2)} - x_2^{(k+1)} = \frac{a_{12}a_{21}}{a_{11}a_{22}} (x_2^{(k+1)} - x_2^{(k)})$$

If we denote the errors as

$$e_2^{(k+1)} = x_2^{(k+2)} - x_2^{(k+1)}$$

$$e_2^{(k)} = x_2^{(k+1)} - x_2^{(k)}$$

Then

$$e_2^{(k+1)} = \frac{a_{12}a_{21}}{a_{11}a_{22}} e_2^{(k)} \quad (8.18)$$

If we want the error to decrease with successive iterations, then we should have the coefficients such that

$$\frac{a_{12}a_{21}}{a_{11}a_{22}} < 1 \quad (8.19)$$

This also conforms with Eqs (8.11) and (8.12).

Example 8.3

Solve the equations

$$3x_1 + x_2 = 5$$

$$x_1 - 3x_2 = 5$$

by the Gauss-Seidel method

First, we rearrange the equations in the form

$$x_1 = 1/3(5 - x_2)$$

$$x_2 = 1/3(x_1 - 5)$$

Assuming initial values as $x_1^0 = 0$, and $x_2^0 = 0$

$$x_1^{(1)} = 5/3$$

Remember, the new value of x_1 should be used in the calculation of new x_2 . Therefore

$$x_2^{(1)} = -10/9$$

Similarly,

$$x_1^{(2)} = \frac{1}{3} \left(5 + \frac{10}{9} \right) = \frac{55}{27}$$

The table below shows the values of x_1 and x_2 rounded to 4 decimal places.

Iteration	x_1	x_2	True error in x_1	True error in x_2
0	0.0000	0.0000	2.0000	1.0000
1	1.6667	-1.1111	0.3333	0.1111
2	2.0370	-0.9877	0.0370	0.0123
3	1.9959	-1.0014	0.0041	0.0014
4	2.0005	-0.9999	0.0005	0.0001
5	2.0000	-1.0000	0.0000	0.0000

The process converges to the solution ($x_1 = 2$, $x_2 = -1$) in five iterations. Note that the given system is *diagonally dominant*. The convergence is graphically illustrated in Fig. 8.2

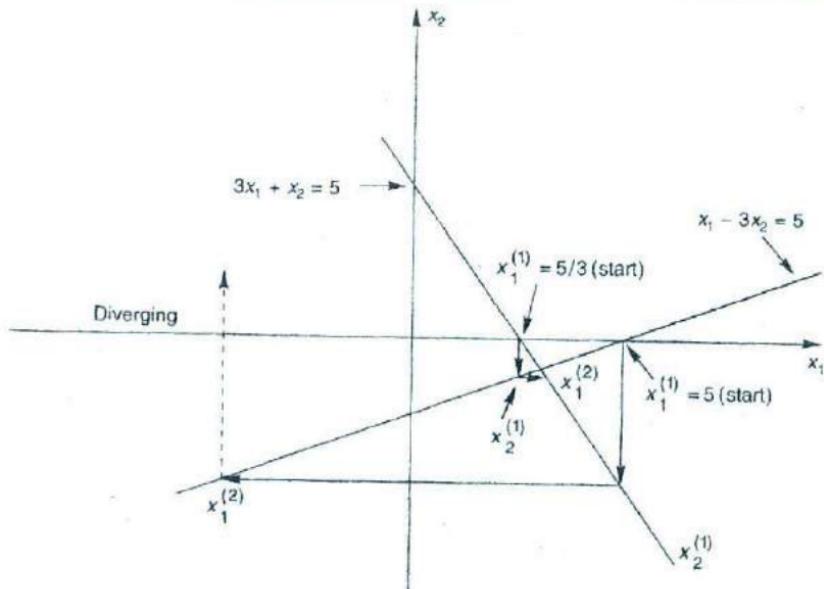


Fig. 8.2 Pictorial representation of Gauss-Seidel convergence

Example 8.4

Solve the equations

$$x_1 - 3x_2 = 5$$

$$3x_1 + x_2 = 5$$

by the Gauss-Seidel method

Note that the system contains the same two equations as in Example 8.3, except they are interchanged.

The iterative equations are

$$x_1 = 5 + 3x_2$$

$$x_2 = 5 - 3x_1$$

As before, we start with $x_1^0 = 0$ and $x_2^0 = 0$. Then,

$$x_1^{(1)} = 5 \quad \text{and} \quad x_2^{(1)} = -10$$

$$x_1^{(2)} = -25 \quad \text{and} \quad x_2^{(2)} = 80$$

$$x_1^{(3)} = 245 \quad \text{and} \quad x_2^{(3)} = -730$$

It is clear that the process does not converge towards the solution. Rather, it diverges (see Fig. 8.2). The result will be the same even if we start with the initial values very close to the solution (except the solution itself). Readers may try with $x_1^0 = 2.5$ and $x_2^0 = -1.2$.

From Examples 8.3 and 8.4 we observe the following:

1. Iteration process converges when

$$\left| \frac{a_{21}}{a_{22}} \right| < 1 \quad \text{and} \quad \left| \frac{a_{12}}{a_{11}} \right| < 1$$

2. The process does not converge for the same set of equations when their order is changed. That is, when

$$\frac{a_{12}a_{21}}{a_{11}a_{22}} > 1$$

the process does not converge

3. When it converges, the errors in x_1 and x_2 decrease by a factor of

$$\frac{a_{11}a_{22}}{a_{12}a_{21}} = 9$$

at each iteration

4. Stronger the diagonal elements, faster the convergence.

8.6 SUMMARY

Iterative methods provide an alternative to the direct methods for solving linear equations. These methods are particularly suitable for solving ill-conditioned systems. We considered the following three iterative methods:

- Jacobi method
- Gauss-Seidel method
- Successive Over Relaxation (SOR) method

We also presented FORTRAN programs along with test results for the Jacobi and Gauss-Seidel methods.

We have shown that a sufficient condition for convergence is that, for each row, the absolute value of the diagonal element should be greater than the sum of absolute values of the other elements in the equation.

Key Terms

Diagonally dominant system

Gauss-Seidel iteration

Jacobi iteration

Relaxation parameter

Successive over relaxation

Successive relaxation method

REVIEW QUESTIONS

1. State the two popular approaches available for solving a system of linear equations.
2. What are the limitations and pitfalls of using direct methods for solving a system of linear equations?
3. State the two important factors that are to be considered while applying iterative methods.
4. The basic idea behind the Jacobi iterative method is essentially the same as that of fixed point method used for solving nonlinear equations. Explain.
5. Gauss-Seidel method is similar in principle to Jacobi method. Then, what is the difference between them?
6. Show that, for a two-equation system

$$a_{11}x_1 + a_{12}x_2 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$

a sufficient condition for convergence of the iteration process is

$$\left| \frac{a_{12}a_{21}}{a_{11}a_{22}} \right| < 1$$

7. Explain the basic concept used in the relaxation method.
8. What is relaxation parameter?

9. What is meant by over-relaxation and under-relaxation?
 10. Give an algorithm for solving a system of linear equations using the successive over-relaxation (SOR) method.

REVIEW EXERCISES

- ~~1. Solve the set of equations given below by Jacobi method.~~

~~$$3x_1 - 6x_2 + 2x_3 = 15$$~~

~~$$4x_1 - x_2 + z = 2$$~~

~~$$x_1 - 3x_2 + 7z = 22$$~~

- ~~2. Solve the system of equations~~

~~$$2x - y + 2z = 6$$~~

~~$$2x - y + z = 3$$~~

~~$$x + 3y - z = 4$$~~

~~by using Jacobi method.~~

- ~~3. Solve the systems given in Exercises 1 and 2 by Gauss-Seidel iteration. Compare the rate of convergence in both the cases.~~

- ~~4. Solve the pair equations~~

~~$$x_1 + 2x_2 = 5$$~~

~~$$3x_1 + x_2 = 5$$~~

~~by applying Jacobi method to the equations~~

~~$$x_1 = 5 - 2x_2$$~~

~~$$x_2 = 5 - 3x_1$$~~

~~Observe the divergence.~~

- ~~5. Solve the equations in Exercise 4 applying Gauss-Seidel method. Compare the divergence with that of earlier one.~~

- ~~6. Interchange the order of equations given in Exercise 4 and then solve them~~

~~(a) using Jacobi method~~

~~(b) using Gauss-Seidel method~~

~~Compare the convergence.~~

- ~~7. Solve the system of equations~~

~~$$3x_1 - 2x_2 = 5$$~~

~~$$-x_1 + 2x_2 - x_3 = 0$$~~

~~$$-2x_2 + x_3 = -1$$~~

~~by applying~~

~~(a) Jacobi method~~

~~(b) Gauss-Seidel method, and~~

~~(c) Successive over-relaxation method with $r = 1.4$~~

~~Comment on the results.~~

8. Solve the following equations by Gauss-Seidel method

$$2x - 7y - 10z = -17$$

$$5x + y + 3z = 14$$

$$x + 10y + 5z = 7$$

Assume suitable initial values.

9. Monthly faculty salary in three departments of an institute is given below. Assuming that the salary for a particular category is same in all the departments, calculate the salary of each category of faculty.

Department	Number of Faculty			Total Salary (in '000)
	Professor	Asst. Professor	Lecturer	
A	2	2	4	60
B	3	1	2	50
C	1	4	3	60

10. Mr. Ram has invested a sum of Rs 20,000 in three types of fixed deposits with an interest rate of 10%, 11% and 12%. He earns an annual interest of Rs 2,220 from all the three types of deposits. If sum of the amounts with 11% and 12% interest rates is four times the amount earning 10% interest, what is the amount invested in each type.

PROGRAMMING PROJECTS

1. Develop a menu-driven, user-friendly single program which provides options for using either Jacobi method or Gauss-Seidel method.
2. Modify the Gauss-Seidel iteration program to incorporate the successive over relaxation method to improve the speed of convergence.

Curve Fitting: Interpolation

9.1 INTRODUCTION

Scientists and engineers are often faced with the task of estimating the value of dependent variable y for an intermediate value of the independent variable x , given a table of discrete data points (x_i, y_i) , $i = 0, 1, \dots, n$. This task can be accomplished by constructing a function $y(x)$ that will pass through the given set of points and then evaluating $y(x)$ for the specified value of x . The process of construction of $y(x)$ to fit a table of data points is called *curve fitting*. A table of data may belong to one of the following two categories:

1. *Table of values of well-defined functions:* Examples of such tables are logarithmic tables, trigonometric tables, interest tables, steam tables, etc.
2. *Data tabulated from measurements made during an experiment:* In such experiments, values of the dependent variable are recorded at various values of the independent variable. There are numerous examples of such experiments—the relationship between stress and strain on a metal strip, relationship between voltage applied and speed of a fan, relationship between time and temperature raise in heating a given volume of water, relationship between drag force and velocity of a falling body, etc., can be tabulated by suitable experiments.

In category 1, the table values are accurate because they are obtained from well-behaved functions. This is not the case in category 2 where the relationship between the variables is not well-defined. Accordingly, we have two approaches for fitting a curve to a given set of data points.

In the first case, the function is constructed such that it passes through all the data points. This method of constructing a function and estimating values at non-tabular points is called interpolation. The functions are known as interpolation polynomials.

In the second case, the values are not accurate and, therefore, it will be meaningless to try to pass the curve through every point. The best strategy would be to construct a single curve that would represent the general trend of the data, without necessarily passing through the individual points. Such functions are called approximating functions. One popular approach for finding an approximate function to fit a given set of experimental data is called least-squares regression. The approximating functions are known as least-squares polynomials.

Figure 9.1 shows an approximate linear function and an interpolation polynomial for a set of data. Note that although the interpolation poly-

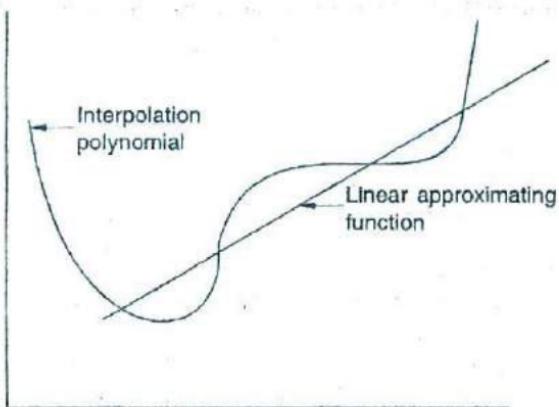


Fig. 9.1 Curve fitting to a set of points

nomial passes through all the points, the curve oscillates widely at the end and beyond the range of data. The linear approximating curve which does not pass through any of the points appears to represent the trend of data adequately. The straight line gives a much better idea of likely values beyond the table points.

In this chapter, we discuss various methods of interpolation. They include:

1. Lagrange interpolation
2. Newton's interpolation
3. Newton-Gregory forward interpolation
4. Spline interpolation

Before we discuss these methods, we introduce various forms of polynomials that are used in deriving interpolation functions. Least-squares regression techniques are discussed in the next chapter.

9.1**POLYNOMIAL FORMS**

The most common form of an n th order polynomial is

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n \quad (9.1)$$

This form, known as the *power form*, is very convenient for differentiating and integrating the polynomial function and, therefore, are most widely used in mathematical analysis. However, there are situations where this form has been found inadequate, as illustrated by Example 9.1.

Example 9.1

Consider the power form of $p(x)$ for $n = 1$,

$$p(x) = a_0 + a_1 x$$

Given that

$$p(100) = +3/7$$

$$p(101) = -4/7$$

obtain the linear polynomial $p(x)$ using four-digit floating point arithmetic. Verify the polynomial by substituting back the values $x = 100$ and $x = 101$.

$$p(100) = a_0 + 100 a_1 = +0.4286$$

$$p(101) = a_0 + 101 a_1 = -0.5714$$

Then, we get

$$a_1 = -1$$

$$a_0 = 100.4 \text{ (only four significant digits)}$$

Therefore,

$$p(x) = 100.4 - x$$

using this polynomial, we obtain

$$p(100) = 0.4$$

$$p(101) = -0.6$$

Compare these results with the original values of $p(100)$ and $p(101)$. We have lost three decimal digits.

Example 9.1 shows that the polynomials obtained using the power form may not always produce accurate results. In order to overcome such problems, we have alternative forms of representing a polynomial. One of them is the *shifted power form* as shown below:

$$p(x) = a_0 + a_1 (x - C) + a_2 (x - C)^2 + \dots + a_n (x - C)^n$$

(9.2)

where C is a point somewhere in the interval of interest. This form of representation significantly improves the accuracy of the polynomial evaluation. This is illustrated by Example 9.2.

Example 9.2

Repeat Example 9.1 using the shifted power form and four-digit arithmetic.

Shifted power form of first order $p(x)$ is

$$p(x) = a_0 + a_1(x - C)$$

Let us choose the centre C as 100. Then

$$p(x) = a_0 + a_1(x - 100)$$

This gives,

$$p(100) = a_0 = 3/7 = 0.4286$$

$$p(101) = 0.4286 + a_1(101 - 100) = -0.5714$$

$$a_1 = -1$$

Thus the linear polynomial becomes

$$p(x) = 0.4286 - (x - 100)$$

Using this polynomial, we obtain

$$p(100) = 0.4286$$

$$p(101) = -0.5714$$

Note the improvement in the results.

Note that Eq. (9.2) is the *Taylor expansion* of $p(x)$ around the point C , when the coefficients a_i are replaced by appropriate function derivatives. It can be easily verified that

$$a_i = \frac{p^{(i)}(C)}{i!} \quad i = 0, 1, 2, \dots, n$$

where $p^{(i)}(C)$ is the i th derivative of $p(x)$ at C .

There is a third form of $p(x)$ known as *Newton form*. This is a generalised shifted power form as shown below:

$$\begin{aligned} p(x) &= a_0 + a_1(x - C_1) + a_2(x - C_1)(x - C_2) + a_3(x - C_1) \\ &\quad (x - C_2)(x - C_3) + \dots + a_n(x - C_1)(x - C_2) \dots (x - C_n) \end{aligned}$$

(9.3)

Note that Eq. (9.3) reduces to shifted power form when $C_1 = C_2 = C_3 = \dots = C_n$ and to simple power form when $C_i = 0$ for all i . The Newton form plays an important role in the derivation of an interpolating polynomial as seen in Section 9.5.

Polynomials can also be expressed in the form

$$\begin{aligned} p_2(x) &= b_0(x - x_1)(x - x_2) \\ &\quad + b_1(x - x_0)(x - x_2) \\ &\quad + b_2(x - x_0)(x - x_1) \end{aligned}$$

In general form,

$$P_n(x) = \sum_{i=0}^n b_i \prod_{j=0, j \neq i}^n (x - x_j) \quad (9.4)$$

LINEAR INTERPOLATION

The simplest form of interpolation is to approximate two data points by a straight line. Suppose we are given two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. These two points can be connected linearly as shown in Fig. 9.2. Using the concept of similar triangles, we can show that

$$\frac{f(x) - f(x_1)}{x - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

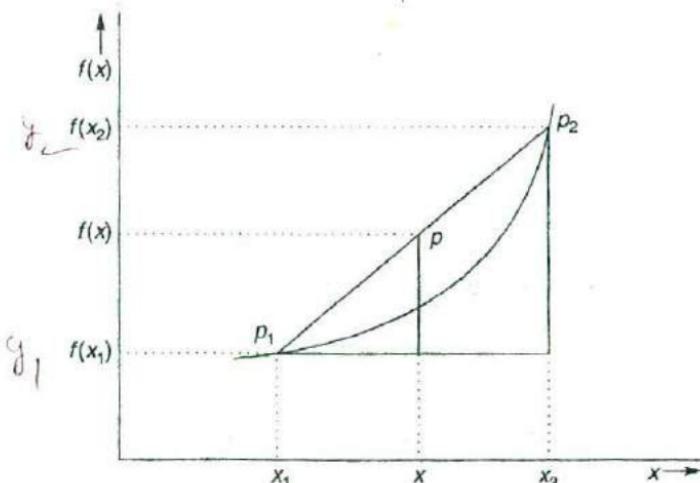


Fig. 9.2 Graphical representation of linear interpolation

Solving for $f(x)$, we get

$$f(x) = f(x_1) + (x - x_1) \frac{f(x_2) - f(x_1)}{x_2 - x_1} \quad (9.5)$$

Equation (9.5) is known as *linear interpolation formula*. Note that the term

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

represents the slope of the line. Further, note the similarity of equation (9.5) with the *Newton form* of polynomial of first-order.

$$C_1 = x_1$$

$$a_0 = f(x_1)$$

$$a_1 = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

The coefficient a_1 represents the first derivative of the function.

Example 9.3

The table below gives square roots for integers.

x	1	2	3	4	5
f(x)	1	1.4142	1.7321	2	2.2361

Determine the square root of 2.5

The given value of 2.5 lies between the points 2 and 3. Therefore,

$$x_1 = 2, \quad f(x_1) = 1.4142$$

$$x_2 = 3, \quad f(x_2) = 1.7321$$

Then

$$\begin{aligned} f(2.5) &= 1.4142 + (2.5 - 2.0) \frac{1.7321 - 1.4142}{3.0 - 2.0} \\ &= 1.4142 + (0.5)(0.3179) \\ &= 1.5732 \end{aligned}$$

The correct answer is 1.5811. The difference is due to the use of a linear model to a nonlinear one.

Now, let us repeat the procedure assuming $x_1 = 2$ and $x_2 = 4$.

$$f(x_1) = 1.4142$$

$$f(x_2) = 2.0$$

Then,

$$f(2.5) = 1.4142 + (2.5 - 2.0) \frac{2.0 - 1.4142}{4.0 - 2.0}$$

$$= 1.4142 + (0.5)(0.2929)$$

$$= 1.5607$$

Notice that the error has increased from 0.0079 to 0.0204. In general, the smaller the interval between the interpolating data points, the better will be the approximation.

The results could be improved considerably by using higher-order interpolation polynomials. We shall demonstrate this in the next section.

LAGRANGE INTERPOLATION POLYNOMIAL

In this section, we derive a formula for the polynomial of degree n which takes specified values at a given set of $n+1$ points.

Let x_0, x_1, \dots, x_n denote n distinct real numbers and let f_0, f_1, \dots, f_n be arbitrary real numbers. The points $(x_0, f_0), (x_1, f_1), \dots, (x_n, f_n)$ can be imagined to be data values connected by a curve. Any function $p(x)$ satisfying the conditions

$$p(x_k) = f_k \quad \text{for } k = 0, 1, \dots, n$$

is called an *interpolation function*. An interpolation function is, therefore, a curve that passes through the data points as pointed out in Section 9.1.

Let us consider a second-order polynomial of the form

$$\begin{aligned} p_2(x) &= b_1(x - x_0)(x - x_1) \\ &\quad + b_2(x - x_1)(x - x_2) \\ &\quad + b_3(x - x_2)(x - x_0) \end{aligned} \tag{9.6}$$

If $(x_0, f_0), (x_1, f_1)$ and (x_2, f_2) are the three interpolating points, then we have

$$p_2(x_0) = f_0 = b_2(x_0 - x_1)(x_0 - x_2)$$

$$p_2(x_1) = f_1 = b_3(x_1 - x_2)(x_1 - x_0)$$

$$p_2(x_2) = f_2 = b_1(x_2 - x_0)(x_2 - x_1)$$

Substituting for b_1, b_2 and b_3 in Eq. (9.6), we get

$$\begin{aligned} p_2(x) &= f_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \\ &\quad + f_1 \frac{(x - x_2)(x - x_0)}{(x_1 - x_2)(x_1 - x_0)} \\ &\quad + f_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \end{aligned} \tag{9.7}$$

Equation (9.7) may be represented as

$$p_2(x) = f_0 l_0(x) + f_1 l_1(x) + f_2 l_2(x)$$

$$= \sum_{i=0}^2 f_i l_i(x)$$

where

$$l_i(x) = \prod_{j=0, j \neq i}^2 \frac{(x - x_j)}{(x_i - x_j)}$$

In general, for $n+1$ points we have n th degree polynomial as

$$p_n(x) = \sum_{i=0}^n f_i l_i(x) \quad (9.8)$$

where

$$l_i(x) = \prod_{j=0, j \neq i}^n \frac{(x - x_j)}{(x_i - x_j)} \quad (9.9)$$

Equation (9.8) is called the *Lagrange interpolation polynomial*. The polynomials $l_i(x)$ are known as *Lagrange basis polynomials*. Observe that

$$l_i(x_j) = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

Now, consider the case $n = 1$

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}$$

$$l_1(x) = \frac{x - x_0}{x_1 - x_0}$$

Therefore,

$$\begin{aligned} p_1(x) &= f_0 \frac{x - x_1}{x_0 - x_1} + f_1 \frac{x - x_0}{x_1 - x_0} \\ &= \frac{f_0(x - x_1) - f_1(x - x_0)}{x_0 - x_1} \\ &= f_0 + \frac{f_1 - f_0}{x_1 - x_0} (x - x_0) \end{aligned}$$

This is the *linear interpolation formula*.

Example 9.3

Consider the problem in Example 9.3. Find the square root of 2.5 using the second order Lagrange interpolation polynomial.

Let us consider the following three points:

$$x_0 = 2, \quad x_1 = 3, \quad \text{and} \quad x_2 = 4$$

Then

$$f_0 = 1.4142, \quad f_1 = 1.7321, \quad \text{and} \quad f_2 = 2$$

For $x = 2.5$, we have

$$l_0(2.5) = \frac{(2.5 - 3.0)(2.5 - 4.0)}{(2.0 - 3.0)(2.0 - 4.0)} = 0.3750$$

$$l_1(2.5) = \frac{(2.5 - 2.0)(2.5 - 4.0)}{(3.0 - 2.0)(3.0 - 4.0)} = 0.7500$$

$$l_2(2.5) = \frac{(2.5 - 2.0)(2.5 - 3.0)}{(4.0 - 2.0)(4.0 - 3.0)} = -0.125$$

$$\begin{aligned} p_2(2.5) &= (1.4142)(0.3750) + (1.7321)(0.7500) + (2.0)(-0.125) \\ &= 0.5303 + 1.2991 - 0.250 = 1.5794 \end{aligned}$$

The error is 0.0017 which is much less than the error obtained in Example 9.3

Example 9.5

Find the Lagrange interpolation polynomial to fit the following data.

i	0	1	2	3
x_i	0	1	2	3
$e^{x_i} - 1$	0	1.7183	6.3891	19.0855

Use the polynomial to estimate the value of $e^{1.5}$.

Lagrange basis polynomials are

$$l_0(x) = \frac{(x - 1)(x - 2)(x - 3)}{(0 - 1)(0 - 2)(0 - 3)}$$

$$= \frac{x^3 - 6x^2 + 11x - 6}{-6}$$

$$l_1(x) = \frac{(x - 0)(x - 2)(x - 3)}{(1 - 0)(1 - 2)(1 - 3)}$$

$$= \frac{x^3 - 5x^2 + 6x}{-6}$$

$$\begin{array}{l} x_0 = 0 \\ x_1 = 1 \\ x_2 = 2 \\ x_3 = 3 \end{array}$$

$$\begin{array}{l} l_0 = 0 \\ l_1 = 1.7183 \\ l_2 = 6.3891 \\ l_3 = 19.0855 \end{array}$$

$$\begin{array}{l} l_0 = 0 \\ l_1 = 1.7183 \\ l_2 = 6.3891 \\ l_3 = 19.0855 \end{array}$$

$$l_2(x) = \frac{(x-0)(x-1)(x-3)}{(0-0)(2-1)(2-3)}$$

$$= \frac{x^3 - 4x^2 + 3x}{-2}$$

$$l_3(x) = \frac{(x-0)(x-2)(x-3)}{(3-0)(3-1)(3-2)}$$

$$= \frac{x^3 - 3x^2 + 2x}{6}$$

$$p(1.5) = 0.9375$$

The interpolation polynomial is

$$p(x) = f_0 l_0(x) + f_1 l_1(x) + f_2 l_2(x) + f_3 l_3(x)$$

$$= 0 + \frac{1.7183(x^3 - 5x^2 + 6x)}{2}$$

$$+ \frac{6.3891(x^3 - 4x^2 + 3x)}{2}$$

$$+ \frac{19.0856(x^3 - 3x^2 + 2x)}{6}$$

$$\frac{5.0732x^3 - 6.3621x^2 + 11.5987x}{6}$$

$$= 0.8455x^3 - 1.0604x^2 + 1.9331x$$

$$p(1.5) = 3.3677$$

$$e^{1.5} = p(1.5) + 1 = 4.3677$$

Points to be noted about Lagrange polynomial:

1. It requires $2(n+1)$ multiplications/divisions and $2n+1$ additions and subtractions
2. If we want to add one more data point, we have to compute the polynomial from the beginning. It does not use the polynomial already computed. That is, $p_{k+1}(x)$ does not use $p_k(x)$ which is already available

Program LAGRAN

Program LAGRAN computes the interpolation value at a specified point, given a set of data points, using the Lagrange interpolation polynomial representation.

```

* -----
*      PROGRAM 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
*   XN - 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
* -----

```

```

INTEGER N,MAX
REAL X,F,FP,LF,SUM
PARAMETER(MAX = 10)
DIMENSION X(MAX),F(MAX)

WRITE(*,*)    'Input number of data points(N)'
READ(*,*) N
WRITE(*,*)    'Input data points X(I) and Function',
+           'values F(I)'
WRITE(*,*)    'one set in each line'
DO 10 I = 1,N
    READ(*,*) X(I), F(I)
10 CONTINUE

WRITE(*,*)    'Input X value at which'
WRITE (*,*)    'interpolation is required'
READ(*,*) XP

SUM = 0.0
DO 30 I = 1,N
    LF = 1.0
    DO 20 J = 1,N

```

```

    IF (I.NE.J) THEN
        LF = LF * (XP - X(J)) / (X(I) - X(J))
    ENDIF
20    CONTINUE
    SUM = SUM + LF * F(I)
30    CONTINUE
    FP = SUM

    WRITE(*,*)
    WRITE(*,*) 'LAGRANGIAN INTERPOLATION'
    WRITE(*,*)
    WRITE(*,*) 'Interpolated Function Value'
    WRITE(*,*) 'at X = ', XP, ' is', FP
    WRITE(*,*)

    STOP
    END

```

* ----- End of main LAGRAN ----- *

Test Run Results The program was used to compute the function value at $x = 2.5$ for the following table of data points:

x	2	3	4
f	1.4142	1.7321	2.0

The results are shown below:

```

Input number of data points(N)
3
Input data points X(I) and Function values F(I)
one set in each line
2 1.4142
3 1.7321
4 2.0
Input X value at which
interpolation is required
2.5
LAGRANGIAN INTERPOLATION
Interpolated Function Value
at X = 2.5000000 is 1.5794000
Stop - Program terminated.

```

NEWTON INTERPOLATION POLYNOMIAL

We have seen that, in Lagrange interpolation, we cannot use the work that has already been done if we want to incorporate another data point

in order to improve the accuracy of estimation. It is therefore necessary to look for some other form of representation to overcome this drawback.

Let us now consider another form of polynomial known as *Newton form* which was discussed in Section 9.2. The Newton form of polynomial is

$$p_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \dots + a_n(x - x_0)(x - x_1)\dots(x - x_{n-1}) \quad (9.10)$$

where the interpolation points x_0, x_1, \dots, x_{n-1} act as centres.

To construct the interpolation polynomial, we need to determine the coefficients a_0, a_1, \dots, a_n . Let us assume that $(x_0, f_0), (x_1, f_1), \dots, (x_{n-1}, f_{n-1})$ are the interpolating points. That is,

$$p_n(x_k) = f_k \quad k = 0, 1, \dots, n-1$$

Now, at $x = x_0$, we have (using Eq. (9.10))

$$p_n(x_0) = a_0 = f_0 \quad (9.11)$$

Similarly, at $x = x_1$,

$$p_n(x_1) = a_0 + a_1(x_1 - x_0) = f_1$$

Substituting for a_0 from Eq. (9.11), we get

$$a_1 = \frac{f_1 - f_0}{x_1 - x_0} \quad (9.12)$$

At $x = x_2$,

$$p_n(x_2) = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = f_2$$

Substituting for a_0 and a_1 from Eqs. (9.11) and (9.12) and rearranging the terms, we get

$$a_2 = \frac{[(f_2 - f_1)/(x_2 - x_1)] - [(f_1 - f_0)/(x_1 - x_0)]}{x_2 - x_0} \quad (9.13)$$

Let us define a notation

$$f[x_k] = f_k$$

$$f[x_k, x_{k+1}] = \frac{f[x_{k+1}] - f[x_k]}{x_{k+1} - x_k}$$

$$f[x_k, x_{k+1}, x_{k+2}] = \frac{f[x_{k+1}, x_{k+2}] - f[x_k, x_{k+1}]}{x_{k+2} - x_k}$$

$$f[x_k, x_{k+1}, \dots, x_i, x_{i+1}] = \frac{f[x_{k+1}, \dots, x_{i+1}] - f[x_k, \dots, x_i]}{x_{i+1} - x_k} \quad (9.14)$$

These quantities are called *divided differences*. Now we can express the coefficients a_i in terms of these divided differences.

$$a_0 = f_0 = f[x_0]$$

$$a_1 = \frac{f_1 - f_0}{x_1 - x_0} = f[x_0, x_1]$$

$$a_2 = \frac{\frac{f_2 - f_1}{x_2 - x_1} - \frac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0}$$

$$= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

$$= f[x_0, x_1, x_2]$$

Thus,

$$a_n = f[x_0, x_1, \dots, x_n]$$

(9.15)

Note that a_1 represents the *first divided difference* and a_2 the *second divided difference* and so on.

Substituting for a_i coefficients in equation (9.10), we get

$$\boxed{P_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \dots + f[x_0, x_1, \dots, x_n](x - x_0)(x - x_1)\dots(x - x_{n-1})}$$

This can be written more compactly as

$$\boxed{P_n(x) = \sum_{i=1}^n f[x_0, \dots, x_i] \prod_{j=0}^{i-1} (x - x_j)}$$
(9.16)

Equation (9.16) is called *Newton's divided difference interpolation polynomial*.

Example 9.6

Given below is a table of data for $\log x$. Estimate $\log 2.5$ using second order Newton interpolation polynomial.

i	0	1	2	3
x_i	1	2	3	4
$\log x_i$	0	0.3010	0.4771	0.6021

Second order polynomials require only three data points. We use the first three points

$$a_0 = f[x_0] = 0,$$

$$a_1 = f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{0.3010}{2 - 1} = 0.3010$$

$$a_2 = f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

$$f[x_1, x_2] = \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{0.4771 - 0.3010}{3 - 2} = 0.1761$$

Therefore,

$$a_2 = \frac{0.1761 - 0.3010}{3 - 1} = -0.06245$$

$$p_2(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1)$$

$$= 0 + 0.3010(x - 1) + (-0.06245)(x - 1)(x - 2)$$

$$p_2(2.5) = 0.3010 \times 1.5 - (0.06245)(1.5)(0.5)$$

$$= 0.4515 - 0.0468$$

$$= 0.4047$$

Note that, in Example 9.6, had we used a linear polynomial, we would have obtained the result as follows:

$$p_1(x) = a_0 + a_1(x - x_0)$$

$$p_1(2.5) = 0 + 0.3010(1.5) = 0.4515$$

This shows that $p_2(2.5)$ is obtained by simply adding a correction factor due to third data point. That is

$$\begin{aligned} p_2(x) &= p_1(x) + a_2(x - x_0)(x - x_1) \\ &= p_1(x) + \Delta_2 \end{aligned}$$

If we want to improve the results further, we can apply further correction by adding another data point. That is

$$p_3(x) = p_2(x) + \Delta_3$$

where

$$\Delta_3 = a_3(x - x_0)(x - x_1)(x - x_2)$$

This shows that the Newton interpolation formula provides a very convenient form for interpolation at an increasing number of interpolation points. Newton formula can be expressed recursively as follows:

$$p_{k+1}(x) = p_k(x) + f[x_0, \dots, x_{k+1}] \phi_k(x)(x - x_k) \quad (9.17)$$

where $p_k(x) = f[x_0, \dots, x_k] \phi_k(x) = \sum_{i=0}^k a_i \phi_i(x)$

and $\phi_i(x) = (x - x_0)(x - x_1) \dots (x - x_{i-1})$

9.3 DIVIDED DIFFERENCE TABLE

We have seen that the coefficients of Newton divided difference interpolation polynomial are evaluated using the divided differences at the interpolating points. We have also seen that a higher-order divided difference is obtained using the lower-order differences. Finally, the first-order divided differences use the given interpolating points (i.e., x_k and f_k values). For example, consider the second-order divided difference

$$\begin{aligned} a_2 &= f[x_0, x_1, x_2] \\ &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} \end{aligned}$$

where $f[x_1, x_2]$ and $f[x_0, x_1]$ are first-order divided differences and are given by

$$\begin{aligned} f[x_0, x_1] &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1 - f_0}{x_1 - x_0} \\ f[x_1, x_2] &= \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{f_2 - f_1}{x_2 - x_1} \end{aligned}$$

This shows that, given the interpolating points, we can obtain recursively a higher-order divided difference, starting from the first-order differences. While this can be conveniently implemented in a computer, we can generate a *divided difference table* for manual computing. A divided difference table for five data points is shown in Fig. 9.3. A particular entry in the table is obtained as follows:

$$f[x_1, x_2, x_3, x_4] = \frac{f[x_2, x_3, x_4] - f[x_1, x_2, x_3]}{x_4 - x_1}$$

<i>i</i>	x_i	$f[x_i]$	First difference	Second difference	Third difference	Fourth difference
0	x_0	$f[x_0]$				
1	x_1	$f[x_1]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$	$f[x_0, \dots, x_4]$
2	x_2	$f[x_2]$	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	$f[x_1, x_2, x_3, x_4]$	
3	x_3	$f[x_3]$	$f[x_2, x_3]$	$f[x_2, x_3, x_4]$		
4	x_4	$f[x_4]$	$f[x_3, x_4]$			

Fig. 9.3 Divided difference table

Draw the two diagonals from the entry to be calculated through its neighbouring entries to the left. If these lines terminate at $f(x_i)$ and $f(x_j)$,

then divide the difference of the neighbouring entries by the corresponding difference $x_j - x_i$. The result is the desired entry. This is illustrated in Fig. 9.3. for the entry $f[x_1, x_2, x_3, x_4]$.

When the table is completed, the entries at the top of each column represent the divided difference coefficients.

Example 9.7

Given the following set of data points, obtain the table of divided differences. Use the table to estimate the value of $f(1.5)$.

i	0	1	2	3	4
x_i	1	2	3	4	5
$f(x_i)$	0	7	26	63	124

The divided difference table is given below:

i	x_i	$f(x_i)$	First difference	Second difference	Third difference	Fourth difference
0	1	0				
1	2	7	7			
2	3	26	19	12		
3	4	63	37	24	6	
4	5	124	61			0

The value of polynomial at $x = 1.5$ is computed as follows:

$$p_0(1.5) = 0$$

$$p_1(1.5) = 0 + 7(1.5 - 1) = 3.5$$

$$p_2(1.5) = 3.5 + 12(1.5 - 1)(1.5 - 2) = 0.5$$

$$p_3(1.5) = 0.5 + 6(1.5 - 1)(1.5 - 2)(1.5 - 3) = 2.25$$

$$p_4(1.5) = 2.25 + 0 = 2.25$$

The function value at $x = 1.5$ is 2.25

Note that $p_3(1.5) = p_4(1.5)$. This implies that correct results can be obtained using the third-order interpolation polynomial. It also illustrates that we can compute $f(1.5)$ in stages (recursively) using interpolation polynomials in increasing order. Computation is terminated when two consecutive estimates are approximately equal or their difference is within a specified limit.

It is clear that the computational effort required in adding one more data point to the estimation process is very much reduced due to the recursive nature of computation.

Let us have a close look at the divided difference table of Example 9.7. Notice the constant values under the column "third difference" and zero value under the column "fourth difference". Recall that the first divided difference is given by

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

This is nothing but the finite divided difference approximation of the first derivative of the function. Similarly, $f[x_0, x_1, x_2]$ is the second derivative and so on. Since the third derivative is constant, the function $f(x)$ should be a third-degree polynomial. In fact, the function used in Example 9.7 is

$$f(x) = x^3 - 1$$

and therefore

$$\frac{d'''f}{dx} = 6$$

and the fourth derivative is zero.

Program NEWINT

Program NEWINT constructs the Newton interpolation polynomial for a given set of data points and then computes the interpolation value at a specified value.

```

* -----
*      PROGRAM 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
*      D - Difference table
*      A - Array of coefficients of interpolation
*            polynomial
* -----
* Constants used
*      NIL
* -----
*      INTEGER N
REAL XP,FP,SUM,PI,X,F,A,D
DIMENSION X(10),F(10),A(10),D(10,10)

WRITE(*,*) 'Input number of data points'
READ(*,*) N
WRITE(*,*)      'Input the values of X and F(x),',
+                  'one set on each line'
DO 10 I = 1,N
    READ(*,*) X(I), F(I)
10   CONTINUE

* Construct difference table D
DO 20 I = 1,N
    D(I,1) = F(I)
20   CONTINUE

DO 40 J = 2,N
    DO 30 I = 1, N-J+1
        D(I,J) = (D(I+1,J-1)-D(I,J-1))/(X(I+J-1)-X(I))
30   CONTINUE
40   CONTINUE

* Set the coefficients of interpolating polynomial
DO 50 J = 1,N
    A(J) = D(1,J)
50   CONTINUE

* Compute interpolation value
WRITE(*,*) 'Input XP where interpolation is
            required'
READ(*,*) XP

SUM = A(1)
DO 70 I = 2,N
    PI = 1.0
    DO 60 J = 1, I-1
        PI = PI *(XP-X(J))
60   CONTINUE
70   CONTINUE

```

```

60  CONTINUE
    SUM = SUM + A(I) * PI
70  CONTINUE

    FP = SUM

* Write results

    WRITE(*,*) 
    WRITE(*,*) 'NEWTON INTERPOLATION'
    WRITE(*,*) 
    WRITE(*,*) 'Interpolated Function Value'
    WRITE(*,*) 'at X = ', XP, ' is', FP
    WRITE(*,*) 

    STOP
    END
* ----- End of main NEWINT -----

```

Test Run Results Let us use the same table values that were used for testing the program LAGRAN. Test run results are given below:

```

Input number of data points
3
Input the values of X and F(x), one set on each line
2 1.4142
3 1.7321
4 2.0
Input XP where interpolation is required
2.5
Newton Interpolation
Interpolated Function Value
at X = 2.5000000 is 1.5794000
Stop - Program terminated.

```

INTERPOLATION WITH EQUIDISTANT POINTS

In this section, we consider a particular case where the function values are given at a sequence of equally spaced points. Most of the engineering and scientific tables are available in this form. We often use such tables to estimate the value at a non-tabular point. Let us assume that

$$x_k = x_0 + kh$$

where x_0 is the reference point and h is the step size. The integer k may take either positive or negative values depending on the position of the reference point in the table. We also assume that we are going to use *simple differences* rather than *divided differences*. For this purpose, we define the following:

The first forward difference Δf_i is defined as

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

The second forward difference is defined as

$$\Delta^2 f_i = \Delta f_{i+1} - \Delta f_i$$

In general,

$$\boxed{\Delta^j f_i = \Delta^{j-i} f_{i+1} - \Delta^{j-1} f_i} \quad (9.18)$$

We can now express the simple forward differences in terms of the divided differences. We know that

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1 - f_0}{h}$$

Therefore,

$$f_1 - f_0 = h f[x_0, x_1]$$

Then

$$\Delta f_0 = f_1 - f_0 = h f[x_0, x_1]$$

Similarly,

$$\Delta f_1 = h f[x_1, x_2]$$

Now,

$$\begin{aligned}\Delta^2 f_0 &= \Delta f_1 - \Delta f_0 \\&= h f[x_1, x_2] - h f[x_0, x_1] \\&= h \{f[x_1, x_2] - f[x_0, x_1]\} \\&= h \cdot 2h \cdot f[x_0, x_1, x_2] \\&= 2h^2 f[x_0, x_1, x_2]\end{aligned}$$

In general, by induction,

$$\Delta^j f_i = j! h^j f[x_i, x_{i+1}, \dots, x_{i+j}]$$

Therefore,

$$f[x_0, x_1, \dots, x_j] = \frac{\Delta^j f_0}{j! h^j}$$

Substituting this in the Newton's divided difference interpolation polynomial (Eq. (9.16)) we get,

$$\boxed{p_n(x) = \sum_{j=0}^n \frac{\Delta^j f_0}{j! h^j} \prod_{k=0}^{j-1} (x - x_k)} \quad (9.19)$$

Let us set

$$x = x_0 + sh \quad \text{and} \quad p_n(s) = p_n(x)$$

We know that

$$x_k = x_0 + kh$$

Thus we get

$$x - x_k = (s - k)h$$

Substituting this in Eq. (9.19), we get

$$\begin{aligned} p_n(s) &= \sum_{j=0}^n \frac{\Delta^j f_0}{j! h^j} \prod_{k=0}^{j-1} (s - k)h \\ &= \sum_{j=0}^n \frac{\Delta^j f_0}{j! h^j} [s(s-1)\dots(s-j+1)] h^j \end{aligned}$$

Thus,

$$p_n(s) = \sum_{j=0}^n \binom{s}{j} \Delta^j f_0 \quad (9.20)$$

where

$$\binom{s}{j} = \frac{s(s-1)\dots(s-j+1)}{j!}$$

is the binomial coefficient. Equations (9.19) and (9.20) are known as *Gregory-Newton forward difference formula*.

Forward Difference Table

The coefficients $\Delta^j f_i$ can be conveniently obtained from the *forward difference table* shown in Fig. 9.4. According to Eq. (9.18), each entry is merely the difference between the two diagonal entries immediately on its left. That is

$$\Delta^j f_i = \Delta^{j-1} f_{i+1} - \Delta^{j-1} f_i$$

The differences which appear on the top of each column correspond to the differences of equation (9.20).

x	f	Δf	$\Delta^2 f$	$\Delta^3 f$	$\Delta^4 f$	$\Delta^5 f$	$\Delta^6 f$
x_0	f_0						
		Δf_0					
x_1	f_1		$\Delta^2 f_0$				
		Δf_1		$\Delta^3 f_0$			
x_2	f_2		$\Delta^2 f_1$		$\Delta^4 f_0$		
		Δf_2		$\Delta^3 f_1$		$\Delta^5 f_0$	
x_3	f_3		$\Delta^2 f_2$		$\Delta^4 f_1$		
		Δf_3		$\Delta^3 f_2$			
x_4	f_4		$\Delta^2 f_3$				
		Δf_4					
x_5	f_5		-				

Fig. 9.4 Forward difference table

As pointed out earlier, difference tables can be used not only to estimate the value of the function at a non-tabular point but can also be used to decide on the degree of the interpolating polynomial that is most appropriate to the given data points.

Example 9.8

Estimate the value of $\sin \theta$ at $\theta = 25^\circ$ using the Newton-Gregory forward difference formula with the help of the following table.

θ	10	20	30	40	50
$\sin \theta$	0.1736	0.3420	0.5000	0.6428	0.7660

In order to use the Newton-Gregory forward difference formula, we need the values of $\Delta^f f_0$. These coefficients can be obtained from the difference table given below. The required coefficients are boldfaced.

θ	$\sin \theta$	Δf	$\Delta^2 f$	$\Delta^3 f$	$\Delta^4 f$	$\Delta^5 f$
10	0.1736					
		0.1684				
20	0.3420		-0.0104			
		0.1580		0.0048		
30	0.5000		-0.0152		-0.0004	
		0.1428		0.0044		
40	0.6428		-0.0196			
		0.1232				
50	0.7660					

$$x_0 = \theta_0 = 10$$

$$h = 10$$

Therefore,

$$s = \frac{x - x_0}{h} = \frac{25 - 10}{10} = 1.5$$

Using Eq. (9.20), we have

$$p_1(s) = 0.1736 + (1.5)(0.1684) = 0.4262$$

$$p_2(s) = 0.4262 + \frac{(1.5)(0.5)(-0.0104)}{2} = 0.4223$$

$$p_3(s) = 0.4223 + \frac{(1.5)(0.5)(-0.5)(0.0048)}{6} = 0.4220$$

$$p_4(s) = 0.4220 + \frac{(1.5)(0.5)(-0.5)(-1.5)(-0.0004)}{24} = 0.4220$$

Thus,

$$\sin 25 = 0.4220$$

which is accurate to four decimal places.

Backward Difference Table

If the table is too long and if the required point is close to the end of the table, we can use another formula known as *Newton-Gregory backward difference formula*. Here, the reference point is x_n , instead of x_0 . Therefore, we have

$$x = x_n + sh$$

$$x_k = x_n - kh$$

$$x - x_k = (s + k)h$$

Then, the Newton-Gregory backward difference formula is given by

$$p_n(s) = f_n + s \nabla f_n + \frac{s(s+1)}{2!} \nabla^2 f_n + \dots \\ + \frac{s(s+1) \dots (s+n-1)}{n!} \nabla^n f_n \quad (9.21)$$

For a given table of data, the backward difference table will be identical to the forward difference table. However, the reference point will be below the point for which the estimate is required. This implies that the value of s will be negative for backward interpolation. The coefficients $\nabla^j f_i$ can be obtained from the backward difference table shown in Fig. 9.5.

x	f	∇f	$\nabla^2 f$	$\nabla^3 f$	$\nabla^4 f$	$\nabla^5 f$	$\nabla^6 f$
x_0	f_0						
		∇f_1					
x_1	f_1		$\nabla^2 f_2$				
		∇f_2		$\nabla^3 f_3$			
x_2	f_2		$\nabla^2 f_3$		$\nabla^4 f_4$		
		∇f_3		$\nabla^3 f_4$		$\nabla^5 f_5$	
x_3	f_3		$\nabla^2 f_4$		$\nabla^4 f_5$		
		∇f_4		$\nabla^3 f_5$			
x_4	f_4		$\nabla^2 f_5$				
		∇f_5					
x_5	f_5						

Fig. 9.5 Backward difference table

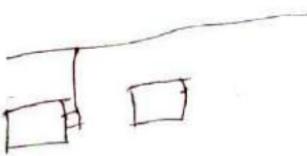
Example 9.9

Repeat the estimation of $\sin 25$ in Example 9.8 using Newton's backward difference formula

$$s = \frac{(x - x_n)}{h} = \frac{25 - 50}{10} = -2.5$$

Using Eq. (9.21), we get

$$\begin{aligned}
 p_4(2.5) &= 0.7660 + (-2.5)(0.1232) \\
 &\quad + \frac{(-2.5)(-1.5)(-0.0196)}{2} \\
 &\quad + \frac{(-2.5)(-1.5)(-0.5)(0.0044)}{6} \\
 &\quad + \frac{(-2.5)(-1.5)(-0.5)(0.5)(-0.0004)}{24} \\
 &= 0.4200
 \end{aligned}$$



9.8 SPLINE INTERPOLATION

So far we have discussed how an interpolation polynomial of degree n can be constructed and used given a set of values of functions. There are situations in which this approach is likely to face problems and produce incorrect estimates. This is because the interpolation takes a global rather than a local view of data. It has been proved that when n is large compared to the order of the "true" function, the interpolation polynomial of degree n does not provide accurate results at the ends of the range. This is illustrated in Fig. 9.6. Note that the interpolation polynomial contains undesirable maxima and minima between the data points. This only shows that increasing the order of polynomials does not necessarily increase the accuracy.

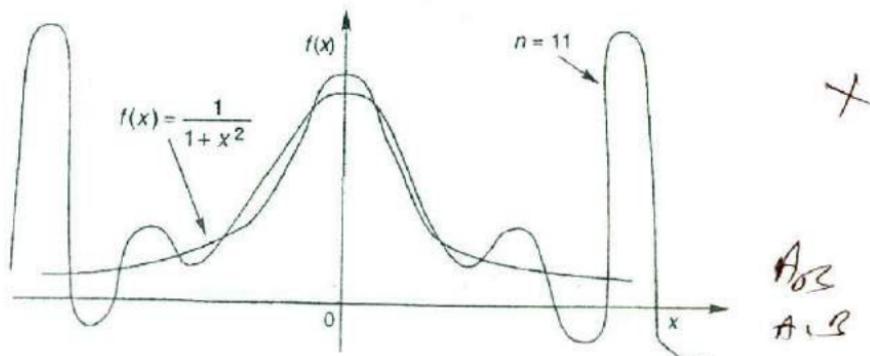


Fig. 9.6 Interpolation polynomial of degree 11 of the function $\frac{1}{1+x^2}$

One approach to overcome this problem is to divide the entire range of points into subintervals and use local low-order polynomials to interpolate each subinterval. Such polynomials are called *piecewise polynomials*. Subintervals are usually taken as $[x_i, x_{i+1}]$, $i = 0, 1, \dots, n$ as illustrated in Fig. 9.7.

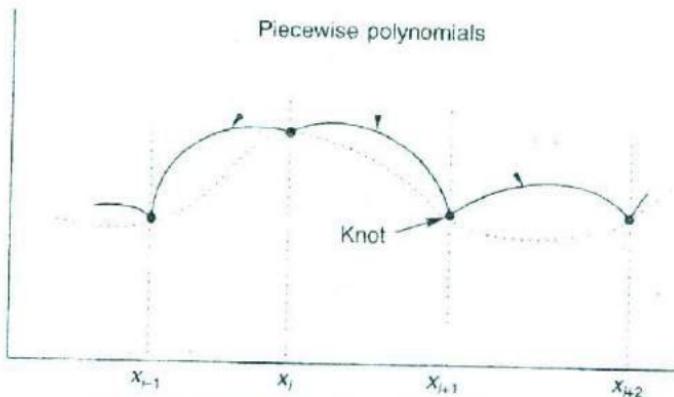


Fig. 9.7 Piecewise polynomial interpolation

Notice that the piecewise polynomials shown in Fig. 9.7 exhibit discontinuity at the interpolating points (which connect these polynomials). It is possible to construct piecewise polynomials that prevent such discontinuities at the connecting points. Such piecewise polynomials are called *spline functions* (or simply *splines*). Spline functions, therefore, look smooth at the connecting points as shown in Fig. 9.8. The connecting points are called *knots* or *nodes* (because this is where the polynomial pieces are tied together).

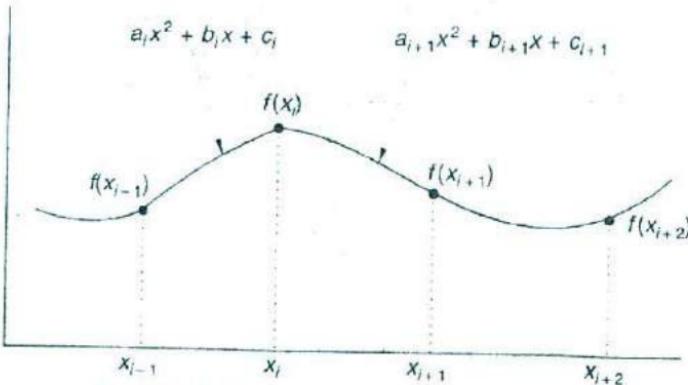


Fig. 9.8 Second degree spline polynomials

A spline function $s(x)$ of degree m must satisfy the following conditions:

- $s(x)$ is a polynomial of degree *atmost* m in each of the subintervals $[x_i, x_{i+1}]$, $i = 0, 1, \dots, n$.
- $s(x)$ and its derivatives of orders $1, 2, \dots, m-1$ are continuous in the range $[x_0, x_n]$.

According to the first condition, each interval will have a different polynomial of degree m or less. The set of all polynomials form a *spline interpolation polynomial*, if $s(x_i) = f_i$, for $i = 0, 1, \dots, n$. The process of constructing such polynomials for a given set of function points is known as *spline interpolation*.

Example 9.10

State whether the following piecewise polynomials are splines or not.

$$(i) f(x) = \begin{cases} x + 1 & -1 \leq x \leq 0 \\ 2x + 1 & 0 \leq x \leq 1 \\ 4 - x & 1 \leq x \leq 2 \end{cases}$$

$$(ii) f(x) = \begin{cases} x^2 + 1 & 0 \leq x \leq 1 \\ 2x^2 & 1 \leq x \leq 2 \\ 5x - 2 & 2 \leq x \leq 3 \end{cases}$$

$$(iii) f(x) = \begin{cases} x & 0 \leq x \leq 1 \\ x^2 - x + 1 & 1 \leq x \leq 2 \\ 3x - 3 & 2 \leq x \leq 3 \end{cases}$$

Case (i)

Given,

$$n = 4, \quad x_0 = -1, \quad x_1 = 0, \quad x_2 = 1, \quad x_3 = 2$$

$$f_1(x) = x + 1$$

$$f_2(x) = 2x + 1$$

$$f_3(x) = 4 - x$$

Then,

$$f_1(x_1) = 0 + 1 = 1$$

$$f_2(x_1) = 2 \times 0 + 1 = 1$$

$$f_2(x_2) = 2 + 1 = 3$$

$$f_3(x_2) = 4 - 1 = 3$$

Note that

$$f_1(x_1) = f_2(x_1) \quad \text{and} \quad f_2(x_2) = f_3(x_2)$$

Therefore, the piecewise polynomials are continuous and $f(x)$ is a linear spline. Note that the first-derivative is not continuous and, therefore, $f(x)$ is not a second-degree spline.

Case (ii)

Given,

$$n=4, \quad x_0 = 0, \quad x_1 = 1, \quad x_2 = 2, \quad x_3 = 3$$

$$f_1(x) = x^2 + 1, \quad f_1'(x) = 2x$$

$$f_2(x) = 2x^2, \quad f_2'(x) = 4x$$

$$f_3(x) = 5x - 2, \quad f_3'(x) = 5$$

Then,

$$f_1(x_1) = 1 + 1 = 2, \quad f_1'(x_1) = 2$$

$$f_2(x_1) = 2 \times 1 = 2, \quad f_2'(x_1) = 4$$

$$f_2(x_2) = 2 \times 4 = 8, \quad f_2'(x_2) = 8$$

$$f_3(x_2) = 5 \times 2 - 2 = 8, \quad f_3'(x_2) = 5$$

Polynomials are continuous but their derivatives are not. Therefore, $f(x)$ is not a spline.

Case (iii)

Given,

$$n=4, \quad x_0 = 0, \quad x_1 = 1, \quad x_2 = 2, \quad x_3 = 3$$

$$f_1(x) = x, \quad f_1'(x) = 1, \quad f_1''(x) = 0$$

$$f_2(x) = x^2 - x + 1, \quad f_2'(x) = 2x - 1, \quad f_2''(x) = 2$$

$$f_3(x) = 3x - 3, \quad f_3'(x) = 3, \quad f_3''(x) = 0$$

Then,

$$f_1(x_1) = 1 \quad f_1'(x_1) = 1$$

$$f_2(x_1) = 1 \quad f_2'(x_1) = 1$$

$$f_2(x_2) = 3 \quad f_2'(x_2) = 3$$

$$f_3(x_2) = 3 \quad f_3'(x_2) = 3$$

Since both the polynomials and their first derivatives are continuous in the given interval, $f(x)$ is a second-degree spline. Note that the second derivatives are not continuous.

Cubic Splines

The concept of splines originated from the mechanical drafting tool called "spline" used by designers for drawing smooth curves. It is a slender flexible bar made of wood or some other elastic material. These curves resemble cubic curves and hence the name "cubic spline" has been given to the piecewise cubic interpolating polynomials. Cubic splines are popular because of their ability to interpolate data with smooth curves. It is believed that a cubic polynomial spline always appears smooth to the eyes.

We consider here the construction of a cubic spline function which would interpolate the points $(x_0, f_0), (x_1, f_1), \dots (x_n, f_n)$. The cubic spline $s(x)$ consists of $(n - 1)$ cubics corresponding to $(n - 1)$ subintervals. If we denote such cubic by $s_i(x)$, then

$$s(x) = s_i(x), \quad i = 1, 2, \dots, n$$

As pointed out earlier, these cubics must satisfy the following conditions:

1. $s(x)$ must interpolate f at all the points x_0, x_1, \dots, x_n , i.e., for each i

$$s(x_i) = f_i \quad (9.22)$$

2. The function values must be equal at all the interior knots

$$s_i(x_i) = s_{i+1}(x_i) \quad (9.23)$$

3. The first derivatives at the interior knots must be equal

$$s'_i(x_i) = s'_{i+1}(x_i) \quad (9.24)$$

4. The second derivatives at the interior knots must be equal

$$s''_i(x_i) = s''_{i+1}(x_i) \quad (9.25)$$

5. The second derivative at the end points are zero

$$s''(x_0) = s''(x_n) = 0$$

Step 1

Let us first consider the second derivatives. Since $s_i(x)$ is a cubic function, its second derivative $s''_i(x)$ is a straight line. This straight line can be represented by a first-order Lagrange interpolating polynomial. Since the line passes through the points $(x_i, s''_i(x_i))$ and $(x_{i-1}, s''_i(x_{i-1}))$, we have,

$$s''_i(x) = s''_i(x_{i-1}) \frac{x - x_i}{x_{i-1} - x_i} + s''_i(x_i) \frac{x - x_{i-1}}{x_i - x_{i-1}} \quad (9.26)$$

The unknowns $s''_i(x_{i-1})$ and $s''_i(x_i)$ are to be determined. For the sake of simplicity, let us denote

$$s''_i(x_{i-1}) = a_{i-1} \quad \text{and} \quad s''_i(x_i) = a_i$$

$$x - x_i = u_i$$

$$x_i - x_{i-1} = h_i = u_{i-1} - u_i$$

Then, Eq. (9.26) becomes

$$\begin{aligned} s''_i(x) &= a_{i-1} \frac{u_i}{-h_i} + a_i \frac{u_{i-1}}{h_i} \\ &= \frac{a_i u_{i-1} - a_{i-1} u_i}{h_i} \end{aligned} \quad (9.27)$$

Step 2

Now we can obtain $s_i(x)$ by integrating Eq. (9.27) twice. Thus

$$s_i(x) = \frac{a_i u_{i-1}^3 - a_{i-1} u_i^3}{6h_i} + C_1x + C_2 \quad (9.28)$$

where C_1 and C_2 are constants of integration [observe that $du/dx = 1$ and, therefore, differentiation and integration with respect to x and with respect to u_i will be equivalent]. The linear part $C_1x + C_2$ can be expressed as

$$b_1(x - x_{i-1}) + b_2(x - x_i)$$

with suitable choice of b_1 and b_2 .

Therefore,

$$\begin{aligned} C_1x + C_2 &= b_1(x - x_{i-1}) + b_2(x - x_i) \\ &= b_1u_{i-1} + b_2u_i \end{aligned}$$

Then, Eq. (9.28) becomes,

$$s_i(x) = \frac{a_i u_{i-1}^3 - a_{i-1} u_i^3}{6h_i} + b_1u_{i-1} + b_2u_i$$

Step 3

Now, we must determine the coefficients b_1 and b_2 . We know that, by condition 1,

$$s(x_i) = f_i \quad \text{and} \quad s(x_{i-1}) = f_{i-1}$$

At $x = x_i$,

$$u_i = 0, u_{i-1} = h_i$$

$$f_i = \frac{a_i h_i^2}{6} + b_1 h_i$$

Similarly, at $x = x_{i-1}$

$$u_{i-1} = 0, \quad u_i = -h_i$$

and therefore

$$f_{i-1} = \frac{a_{i-1} h_i^2}{6} - b_2 h_i$$

Thus, we get

$$b_1 = \frac{f_i}{h_i} - \frac{a_i h_i}{6} \quad (9.29a)$$

$$b_2 = -\frac{f_{i-1}}{h_i} + \frac{a_{i-1} h_i}{6} \quad (9.29b)$$

Substituting for b_1 and b_2 in Eq. (9.29) and after rearrangement of terms, we get

$$s_i(x) = \frac{a_{i-1}}{6h_i} (h_i^2 u_i - u_i^3) + \frac{a_i}{6h_i} (u_{i-1}^3 - h_i^2 u_{i-1}) \\ + \frac{1}{h_i} (f_i u_{i-1} - f_{i-1} u_i) \quad (9.30)$$

Note that Eq. (9.30) has only two unknowns, a_{i-1} and a_i .

Step 4

The final step is to evaluate these constants. This can be done by invoking the condition

$$s_i'(x_i) = s_{i+1}'(x_i)$$

Differentiating Eq. (9.30) we get

$$s_i'(x) = \frac{a_{i-1}}{6h_i} (h_i^2 - 3u_i^2) \\ + \frac{a_i}{6h_i} (3u_{i-1}^2 - h_i^2) \\ + \frac{1}{h_i} (f_i - f_{i-1})$$

Achish

Setting $x = x_i$,

$$s_i'(x_i) = \frac{a_{i-1}h_i}{6} + \frac{a_ih_i}{3} + \frac{f_i - f_{i-1}}{h_i}$$

NUB

Similarly,

$$s_{i+1}'(x_i) = -\frac{a_i h_{i+1}}{3} - \frac{a_{i+1} h_{i+1}}{6} + \frac{f_{i+1} - f_i}{h_{i+1}}$$

Since

$$s_i'(x_i) = s_{i+1}'(x_i)$$

We have

$$h_i a_{i-1} + 2(h_i + h_{i+1})a_i + h_{i+1}a_{i+1} = 6 \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i} \quad (9.31)$$

Equation (9.31), when written for all interior knots ($i = 1, \dots, n-1$), we get $n-1$ simultaneous equations containing $n+1$ unknowns (a_0, a_1, \dots, a_n). Now, applying the condition that the second derivatives at the end points are zero, we get

$$a_0 = a_n = 0$$

Thus, we have $n - 1$ equations with $n - 1$ unknowns which can be easily solved.

Note

The cubic splines with zero second derivatives at the end points are called the *natural cubic splines*. This is because the splines are assumed to take their natural straight line shape outside the intervals of approximations.

The system of $n - 1$ equations contained in Eq. (9.31) can be expressed as

$$\begin{bmatrix} 2(h_1 + h_2) & h_2 & 0 & \cdots & 0 & 0 & 0 \\ h_2 & 2(h_2 + h_3) & h_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & h_{n-1} & 2(h_{n-1} + h_n) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ \vdots \\ D_{n-1} \end{bmatrix} \quad (9.32)$$

where

$$D_i = 6 \left[\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i} \right]$$

$$h_i = x_i - x_{i-1}$$

$$i = 1, 2, \dots, n - 1$$

Example 9.11

Given the data points

i	0	1	2
x_i	4	9	16
f_i	2	3	4

estimate the function value f at $x = 7$ using cubic splines.

$$h_1 = x_1 - x_0 = 9 - 4 = 5$$

$$h_2 = x_2 - x_1 = 16 - 9 = 7$$

$$f_0 = 2, \quad f_1 = 3, \quad f_2 = 4$$

From Eq. (9.31), we have, for $i = 1$,

$$h_1 a_0 + 2(h_1 + h_2)a_1 + h_2 a_2 = 6 \left[\frac{f_2 - f_1}{h_2} - \frac{f_1 - f_0}{h_1} \right]$$

We know that $a_0 = a_2 = 0$. Thus,

$$2(5 + 7)a_1 = 6 \left[\frac{1}{7} - \frac{1}{5} \right]$$

Therefore,

$$a_1 = \frac{(6)(-2)}{(35)(24)} = -0.0143$$

Since $n = 3$, there are two cubic splines, namely,

$$s_1(x) \quad x_0 \leq x \leq x_1$$

$$s_2(x) \quad x_1 \leq x \leq x_2$$

The target point $x = 7$ is in the domain of $s_1(x)$ and, therefore, we need to use only $s_1(x)$ for estimation.

From Eq. (9.30)

$$s_1(x) = \frac{a_1(u_0^3 - h_1^2 u_0)}{6h_1} + \frac{1}{h_1}(f_1 u_0 - f_0 u_1)$$

$$u_0 = x - x_0 \quad \text{and} \quad u_1 = x - x_1$$

Upon substitution of specific values,

$$s_1(7) = -\frac{0.0143}{6 \times 5} [(7 - 4)^3 - 5^2(7 - 4)]$$

$$= +\frac{1}{5} [3(7 - 4) - 2(7 - 9)]$$

$$= 2.6229$$

Algorithm

Note that Eq. (9.32) form a *tridiagonal* system which is relatively simple to solve using Gauss elimination method. A detailed solution procedure to evaluate spline functions is given in Algorithm 9.1

Natural cubic spline

1. Provide input data.
2. Compute step lengths and form function differences.
3. Obtain the coefficients of the tridiagonal matrix.
4. Compute the right-hand side (D array) of the system.
5. Compute the elements a_i using Gauss elimination method.

(Contd.)

(Contd.)

6. Evaluate the coefficients of natural cubic splines
7. Evaluate the spline function at the point of interest.
8. Print results.

Algorithm 9.1

Program SPLINE

Natural cubic splines interpolation uses Gauss elimination method to implement its algorithm. Program SPLINE, therefore, calls for the help of GAUSS subprogram to compute the array of second derivatives.

```

* ----- *
*          PROGRAM 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 right side of (9.32)
*       (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
* -----

```

```

INTEGER N,MAX
REAL XP,FP,X,F,A,D,C,H,DF,U
PARAMETER (MAX=10)
DIMENSION X(MAX),F(MAX),A(MAX),D(MAX),C(MAX,MAX),
+ H(MAX),DF(MAX),U(MAX)

* Read input data

      WRITE(*,*) 'Input number of data points n'
      READ(*,*) N
      WRITE(*,*) 'Input data points X(I) and function'
      WRITE(*,*) 'values F(I), one set in each line'
      DO 5 I = 1,N
          READ(*,*) X(I), F(I)
5     CONTINUE

      WRITE(*,*) 'Input XP'
      READ(*,*) XP

* Compute distances between data points
* and function differences

      DO 10 I = 2,N
          H(I) = X(I) - X(I-1)
          DF(I) = F(I) - F(I-1)
10    CONTINUE

* Initialise C matrix

      DO 30 I = 2,N-1
          DO 20 J = 2, N-1
              C(I,J) = 0.0
20      CONTINUE
30      CONTINUE

* Compute diagonal elements of C

      DO 40 I = 2,N-1
          C(I,I) = 2.0 * (H(I)+H(I+1))
40      CONTINUE

* Compute off_diagonal elements of C

      DO 50 I = 3,N-1
          C(I-1,I) = H(I)
          C(I,I-1) = H(I)
50      CONTINUE

* Compute elements of D array

      DO 60 I = 2,N-1
          D(I) = (DF(I+1)/H(I+1) - DF(I)/H(I)) * 6.0

```

310 Numerical Methods

60 CONTINUE

* Compute elements of A using Gaussian elimination
* Change array subscripts from 2 to n-1 to 1 to n-1
* before calling GAUSS

M = N-2

DO 80 I = 1,M

 D(I) = D(I+1)

 DO 70 J = 1,M

 C(I,J) = C(I+1,J+1)

70 CONTINUE

80 CONTINUE

CALL GAUSS(M,C,D,A)

* Compute the coefficients of natural cubic spline

DO 90 I = N-1,2,-1

 A(I) = A(I-1)

90 CONTINUE

A(1) = 0.0

A(N) = 0.0

* Locate the domain of XP

I = 2

100 IF(XP .LE. X(I)) GO TO 110

I = I+1

GO TO 100

* Compute interpolation value at XP

* Use equation (9.30)

110 U(I-1) = XP - X(I-1)

U(I) = XP - X(I)

Q1 = H(I)**2 * U(I) - U(I)**3

Q2 = U(I-1)**3 - H(I)**2 * 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

WRITE(*,*)

WRITE(*,*) 'SPLINE INTERPOLATION'

WRITE(*,*)

WRITE(*,*) 'Interpolation value = ',FP

WRITE(*,*)

STOP

END

* ----- End of main SPLINE ----- *

```

* -----
*      SUBROUTINE GAUSS(N,A,B,X)
*
*      Subroutine
*      This subroutine solves a set of n linear
*      equations using Gauss elimination method
*
* -----
*      Arguments
*      Input
*          N - Number of equations
*          A - Matrix of coefficients
*          B - Right side vector
*      Output
*          X - Solution vector
*
* -----
*      Local Variables
*          PIVOT, FACTOR, SUM
*
* -----
*      Functions invoked
*          NIL
*
* -----
*      Subroutines called
*          NIL
*
* -----
*      INTEGER N
*      REAL A,B,X,PIVOT,FACTOR,SUM
*      DIMENSION A(10,10), B(10), X(10)
*
* ----- Elimination begins -----
*
      DO 33 K = 1, N-1
          PIVOT = A(K,K)
          DO 22 I = K+1, N
              FACTOR = A(I,K)/PIVOT
              DO 11 J = K+1, N
                  A(I,J) = A(I,J) - FACTOR * A(K,J)
11      CONTINUE
                  B(I) = B(I) - FACTOR * B(K)
22      CONTINUE
33      CONTINUE
*
* ----- Back substitution begins -----
*
      X(N) = B(N)/A(N,N)
      DO 55 K = N-1, 1, -1
          SUM = 0
          DO 44 J = K+1, N

```

```

        SUM = SUM + A(K,J) * X(J)
44      CONTINUE
        X(K) = (B(K) - SUM)/A(K,K)
55      CONTINUE
        RETURN
        END

```

* ----- End of subroutine GAUSS ----- *

Test Run Results Program SPLINE was tested using the table of data points given in Example 9.11.

Results are given below:

```

Input number of data points n
3
Input data points, X(I) and function
values F(I), one set in each line
4 2
9 3
16 4
Input XP
7
SPLINE INTERPOLATION
Interpolation Value = 2.6228570
Stop - Program terminated.

```

Equidistant Knots

Most often the knots are equally spaced. This would simplify the solution considerably. If the knots are equally spaced,

$$h_1 = h_2 = \dots = h_n = h.$$

Substituting this in equations (9.11) and dividing throughout by h , we get

$$\begin{bmatrix} 4 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 1 & 4 & 1 & \cdots & : & : & : \\ 0 & 1 & 4 & 1 & : & : & : \\ : & \cdots & \cdots & \cdots & : & : & : \\ : & \cdots & \cdots & \cdots & 4 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & 1 & 4 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ d_{n-1} \end{bmatrix} \quad (9.33)$$

where

$$d_i = \frac{D_i}{h} = \frac{6(f_{i+1} - 2f_i + f_{i-1})}{h^2}$$

$$= \frac{6}{h^2} \Delta^2 f_{i-1}$$

$$= 12 f[x_{i-1}, x_i, x_{i+1}] \quad i = 1, 2, \dots, n - 1$$

Example 9.12

Given the table of values

i	0	1	2	3
x_i	1	2	3	4
$f(x_i)$	0.5	0.3333	0.25	0.20

estimate the value of $f(2.5)$ using cubic spline functions

$$h_1 = h_2 = h_3 = 1$$

Since $n = 4$, we have three intervals and three cubics and, therefore, only a_1 and a_2 are to be determined. From Eq. (9.33), we have

$$\begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

$$d_1 = \frac{6}{h^2} (f_2 - 2f_1 + f_0)$$

$$= 6 (0.25 - 2 \times 0.3333 + 0.5)$$

$$= 0.5004$$

$$d_2 = \frac{6}{h^2} (f_3 - 2f_2 + f_1)$$

$$= 6 (0.2 - 2 \times 0.25 + 0.3333)$$

$$= 0.1998$$

Solving for a_1 and a_2

$$a_1 = \frac{d_1 \times 4 - d_2 \times 1}{15}$$

$$= \frac{0.5004 \times 4 - 0.1998}{15} = 0.1201$$

$$a_2 = \frac{d_2 \times 4 - d_1 \times 1}{15}$$

$$= \frac{0.1998 \times 4 - 0.5004}{15} = 0.0199$$

314 Numerical Methods

The target point $x = 2.5$ is in the domain of $s_2(x)$. Using Eq. (9.30),

$$\begin{aligned}s_2(x) &= \frac{a_1}{6}(u_2 - u_2^3) + \frac{a_2}{6}(u_1^3 - u_1) + (f_2 u_1 - f_1 u_2) \\&= \frac{a_1}{6}[(x - x_2) - (x - x_2)^3] + \frac{a_2}{6}[(x - x_1)^3 \\&\quad - (x - x_1)] + [f_2(x - x_1) - f_1(x - x_2)]\end{aligned}$$

Upon substitution of values, we get

$$\begin{aligned}s_2(2.5) &= \frac{0.1201}{6}[(2.5 - 3) - (2.5 - 3)^3] \\&\quad + \frac{0.0199}{6}[(2.5 - 2)^3 - (2.5 - 2)] \\&\quad + (0.25)(2.5 - 2) - 0.3333(2.5 - 3) \\&= -0.0075 - 0.0012 + 0.125 + 0.1667 \\&= 0.2829\end{aligned}$$

CHEBYSHEV INTERPOLATION POLYNOMIAL

Recall that the truncation error in approximating a function $f(x)$ by an interpolating polynomial $p_n(x)$ with interpolation points x_i , $i = 0, 1, \dots, n$ is

$$f(x) - p_n(x) = w_n(x) \frac{f^{(n+1)}(\theta)}{(n+1)!}$$

where

$$w_n(x) = (x - x_0)(x - x_1) \dots (x - x_n)$$

and θ is some point in the interval of interest. One of the goals while applying an interpolation polynomial is to minimise the truncation error. Since $f^{(n+1)}(\theta)$ is not in our control, we can try to minimise the absolute value of $w_n(x)$. This can be done by choosing a proper set of interpolating points x_i in the given interval (a, b) .

Chebyshev Points

The Russian mathematician Chebyshev showed that the error bound is minimum when the interpolation points are chosen as follows:

$$x_k = \frac{a+b}{2} + \frac{a-b}{2} \cos \left[\frac{2k+1}{2(n+1)} \pi \right] \quad k = 0, 1, \dots, n \quad (9.34)$$

These values are called *Chebyshev nodes* (or *points*). We can evaluate function values at these points. That is

$$f_k = f(x_k)$$

Now, we can apply the Lagrange interpolation method to the Chebyshev points and the corresponding function values to obtain an interpolation polynomial known as *Lagrange - Chebyshev interpolation polynomial*.

Chebyshev Polynomials

Another approach to construct the interpolation polynomial $p_n(x)$ is to use Chebyshev polynomials as basis polynomials. That is

$$\begin{aligned} p_n(x) &= C_0 T_0(t) + C_1 T_1(t) + \dots + C_n T_n(t) \\ &= \sum_{i=0}^n C_i T_i(t) \end{aligned} \quad (9.35)$$

where $T_i(t)$ is the *Chebyshev basis polynomial* of order i in t and C_i the Chebyshev coefficient. Equation (9.35) is known as Chebyshev interpolation polynomial. Chebyshev polynomial $T_i(t)$ is given by

$$T_0(t) = 1$$

$$T_1(t) = t$$

$$T_k(t) = 2t T_{k-1}(t) - T_{k-2}(t) \quad k = 2, \dots, n$$

C_i are computed as follows:

$$C_0 = \frac{1}{n+1} \sum_{k=0}^n f(x_k) T_0(t_k) = \frac{1}{n+1} \sum_{k=0}^n f(x_k)$$

$$C_j = \frac{2}{n+1} \sum_{k=0}^n f(x_k) T_j(t_k)$$

where

$$T_j(t_k) = \cos \left[j \frac{(2k+1)\pi}{2(n+1)} \right]$$

Therefore

$$C_j = \frac{2}{n+1} \sum_{k=0}^n f(x_k) \cos \left[j \frac{(2k+1)\pi}{2(n+1)} \right] \quad j = 1, 2, \dots, n$$

Evaluation of $p_n(x)$, given x :

$$t = \frac{x - (b+a)/2}{(b-a)/2}$$

$$p_n(x) = \sum_{i=0}^n C_i T_i(t)$$



SUMMARY

In this chapter, we discussed various methods for constructing interpolation polynomials for tables of well-defined functions. They include:

- Lagrange interpolation
- Newton's interpolation
- Newton-Gregory forward interpolation
- Spline interpolation

To facilitate the construction of interpolation functions, we presented different forms of polynomials that included

- power form
- shifted power form
- Newton form

We have also discussed how to build different types of difference tables and how to use them for estimating function values at any point. Finally, we considered how Chebyshev points and Chebyshev polynomials may be used to minimise the truncation error.

We have given computer programs and test results for the following methods:

- Lagrange interpolation
- Newton's interpolation
- Spline interpolation

Key Terms

<i>Approximating functions</i>	<i>Least-squares polynomials</i>
<i>Backward difference</i>	<i>Least-squares regression</i>
<i>Central cubic spline</i>	<i>Linear interpolation</i>
<i>Central difference</i>	<i>Natural cubic spline</i>
<i>Chebyshev basis polynomial</i>	<i>Newton form</i>
<i>Chebyshev interpolation</i>	<i>Newton Interpolation polynomial</i>
<i>Chebyshev points</i>	<i>Newton's interpolation</i>
<i>Chebyshev polynomial</i>	<i>Newton-Gregory formula</i>
<i>Cubic spline</i>	<i>Newton-Gregory interpolation</i>
<i>Curve fitting</i>	<i>Nodes</i>
<i>Divided difference table</i>	<i>Piecewise polynomial</i>
<i>Divided differences</i>	<i>Power form</i>
<i>Forward difference</i>	<i>Shifted power form</i>
<i>Interpolation</i>	<i>Simple difference</i>
<i>Interpolation function</i>	<i>Spline function</i>
<i>Interpolation polynomial</i>	<i>Spline interpolation</i>
<i>Knots</i>	<i>Spline interpolation polynomial</i>
<i>Lagrange basis polynomial</i>	<i>Splines</i>
<i>Lagrange interpolation</i>	<i>Taylor expansion</i>
<i>Lagrange interpolation polynomial</i>	<i>Tridiagonal system</i>

REVIEW QUESTIONS

1. What is curve fitting? What is the need for such an exercise?
2. What is interpolation? //
3. What are the methods available for interpolation?
4. Discuss the possible sources of errors in interpolation?
5. What is interpolation function?
6. List, with examples, different forms of polynomials that could be used for constructing interpolation functions.
7. Given two points (x_1, y_1) and (x_2, y_2) , state the linear interpolation formula in terms of these points. //
8. Given a set of $n + 1$ points, state the general form of n th degree Lagrange interpolation polynomial.
9. What is the computational effort required in using Lagrange polynomial?
10. What is the major pitfall of using Lagrange polynomial?
11. What are divided differences?
12. State the second order Newton's divided difference interpolation polynomial.
13. How is the Newton's interpolation formula better than Lagrange formula?
14. What is a divided difference table? How is it useful?
15. Construct a divided difference table for four data points.
16. Entries under a particular column in a divided difference table are constants. What does it indicate?
17. Distinguish between the simple difference and divided difference.
18. What is the difference between the forward difference table and backward difference table?
19. Look at Examples 9.8 and 9.9. Answers are different. Why?
20. What are piecewise polynomials?
21. What are spline functions?
22. What is spline interpolation?
23. What are cubic splines?
24. State the conditions for a spline to be cubic.
25. What are natural cubic splines?
26. What is tridiagonal system?
27. State the contribution of Russian mathematician Chebyshev in minimizing the truncation error in interpolation.

REVIEW EXERCISES

1. Construct the power form of the straight line $p(x)$ which takes on the values

$$p(200) = 1/3$$

318 Numerical Methods

$$p(202) = -2/3$$

using four-digit floating-point arithmetic.

2. Solve the problem in Exercise 1 using the shifted-power form and compare the results.
3. Find the linear interpolation polynomial for each of the following pairs of points:
 - (a) (0, 1) and (1, 3)
 - (b) (-2, 3) and (7, 12)
4. Find the quadratic interpolating polynomial for each of the following set of points:
 - (a) (-1, 1), (0, 1) and (1, 3)
 - (b) (0, -1), (1, 0) and (2, 9)
5. Table below gives values of square of integers:

x	1	2	3	4	5
x^2	1	4	9	16	25

Using the linear interpolation formula estimate the square of 3.25

- (a) using the points 3 and 4
- (b) using the points 2 and 4

Compare and comment on the results.

6. Using the data in Exercise 5, estimate the square of 3.25 using the second-order Lagrange formula. Compare the error with the errors obtained in Exercise 5.
7. When the value of x at which we wish to estimate the value of $f(x)$, lies outside the given range, we call it extrapolation. Use the Lagrange formula to find the quadratic equation that takes the following values:

x	1	2	3
$f(x)$	1	1	2

Find $f(x)$ at $x = 0$ and $x = 4$

8. Given the points below, obtain a cubic polynomial using the Lagrange formula:

x	0	1	2	3
$f(x)$	1	-1	-1	0

9. Find the Lagrange interpolation polynomial which agrees with the following data:

x	1.0	1.1	1.2
$\cos x$	0.5403	0.4536	0.3624

Use it to estimate $\cos 1.15$

10. Find the polynomial of degree three to fit the following points:

x	-1	0	1	3
$f(x)$	-6	-2	2	10

11. Show that when $n = 2$, Lagrangian interpolation formula reduces to the linear interpolation formula.
12. The Lagrange interpolation polynomial can be derived directly from Newton's interpolating polynomial. Prove this using the linear case.
13. Fit a second-order Newton's interpolating polynomial to estimate $\cos 1.15$ using the data from Exercise 9.
14. Fit a third-order Newton's interpolating polynomial to estimate $\cos 1.15$ using the data from Exercise 9 along with the additional point $\cos 1.3 = 0.2675$.
15. Given the data

x	1.2	1.3	1.4	1.5
$f(x)$	1.063	1.091	1.119	1.145

- (a) Calculate $f(1.35)$ using Newton's interpolating polynomial of order 1 through 3. Choose base points to attain good accuracy.
- (b) Comment on the accuracy of results on the order of polynomial.
16. Find the divided differences $f[x_0, x_1]$, $f[x_1, x_2]$ and $f[x_0, x_1, x_2]$ for the data given below.

i	0	1	2
x_i	1.0	1.5	2.5
$f(x_i)$	3.2	3.5	4.5

Also find the divided differences $f[x_0, x_2]$ and $f[x_0, x_1, x_2]$. Compare the results $f[x_0, x_1, x_2]$ and $f[x_0, x_2, x_1]$.

17. Estimate the value of $\ln(3.5)$ using Newton-Gregory forward difference formula given the following data:

x	1.0	2.0	3.0	4.0
\ln	0.0	0.6931	1.0986	1.3863

18. Repeat Exercise 17 using Newton's backward difference formula. Compare the accuracy of results.
19. Construct difference tables for the following data:

x	0.1	0.3	0.5	0.7	0.9	1.1	1.3
$f(x)$	0.003	0.067	0.148	0.248	0.370	0.518	0.697

Find $f(0.6)$ using a cube that fits at $x = 0.3, 0.5, 0.7$ and 0.9 .

20. What is the minimum degree of polynomial that will exactly fit all seven pairs of data in Exercise 19.
21. Construct a divided difference table for the data in Exercise 19. How do the values compare with those in the table obtained in Exercise 19.
22. State whether the following functions are splines or not.

$$(a) f(x) = \begin{cases} x & 0 \leq x \leq 1 \\ \frac{x^2 + 1}{2} & 1 \leq x \leq 3 \\ 5x - 8 & 3 \leq x \leq 4 \end{cases}$$

$$(b) f(x) = \begin{cases} x^2 - 3x + 1 & 0 \leq x \leq 1 \\ x^3 + x^2 - 3 & 1 \leq x \leq 2 \\ x^3 + 5x - 9 & 2 \leq x \leq 3 \end{cases}$$

$$(c) f(x) = \begin{cases} -x + 5.5 & 3.0 \leq x \leq 4.5 \\ 0.64x^2 - 6.76x + 18.46 & 4.5 \leq x \leq 7.0 \\ -1.6x^2 + 24.6x - 91.3 & 7.0 \leq x \leq 9.0 \end{cases}$$

23. Find the values of a and b such that the function

$$f(x) = \begin{cases} ax^2 - x + 1 & 1 \leq x \leq 2 \\ 3x - b & 2 \leq x \leq 3 \end{cases}$$

is a quadratic spline.

24. Fit quadratic splines to the data given below:

x	1	2	3
$f(x)$	1	1	2

Predict $f(2.5)$.

25. Develop cubic splines for the data given below and predict $f(1.5)$

x	0	1	2	3
$f(x)$	1	-1	-1	0

26. Given the data points

i	0	1	2	3
x_i	1.0	3.0	4.0	7.0
$f(x_i)$	1.5	4.5	9.0	25.5

Estimate the function value at $x = 1.5$ using cubic splines.

27. The velocity distribution of a fluid near a flat surface is given below:

x	0.1	0.3	0.5	0.7	0.9
v	0.72	1.81	2.73	3.47	3.98

x is the distance from the surface (cm) and v is the velocity (cm/sec). Using a suitable interpolation formula obtain the velocity at $x = 0.2, 0.4, 0.6$ and 0.8 .

28. The steady-state heat-flow equation $f(x,y)$ is solved numerically and temperature values obtained at the pivotal points of a grid in the domain of interest are tabulated below. (This type of problems are discussed in Chapter 15).

Table of $f(x, y)$

$x \backslash y$	0.5	1.0	1.5	2.0
0.5	15.0	21.0	25.0	31.0
1.0	20.0	20.0	20.0	20.0
1.5	25.5	19.0	15.0	9.0
2.0	30.0	20.0	10.0	0.0

Solution of heat-flow equations by numerical methods gives information only at the nodes and not at the intermediate points. We are interested in the temperature at the point $(1.25, 1.25)$. Estimate this value using the data available in the table.

PROGRAMMING PROJECTS

1. Write subprograms
 - (a) COSPLN to compute the coefficients cubic splines, and
 - (b) VSPLN to evaluate the spline function at the specified point.
2. Write an interactive main program that will read the given set of table points and the point of interest, estimate the interpolation at the specified point using the subprograms COSPLN and VSPLN developed in Project 1, and then print the results.
3. Write a program to evaluate forward differences and print a forward difference table for a set of n function values.
4. Following is a table that lists values of cube roots of numbers from 1.0 to 2.0 in steps of 0.1.

x	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
$\sqrt[3]{x}$	1.0	1.032	1.063	1.091	1.119	1.145	1.170	1.193	1.216	1.239	1.260

Write a program for linear interpolation of this table of data and produce another table of cube roots for numbers 1.25 to 1.75 in steps of 0.05 shown as follows:

x	cube root of x
1.25	
1.30	
1.35	
.	
.	
1.75	

5. Modify the program in Project 4 to produce the following table:

x	Interpolated cube root of x	True value of $\sqrt[3]{x}$	Error
1.25			
1.30			
.			
.			
1.75			

6. Using a table of cosines, accurate to four digits, write a program to implement the following tasks:
- Read the cosine of $0^\circ, 10^\circ, \dots, 90^\circ$
 - Compute the cosine of angle for any value between 0° and 90° using linear interpolation.
 - Compare the results of (b) with the output of intrinsic `cos` function.
7. Write a program to estimate a value $f(x, y)$ from a given table of values of x and y by interpolation.
 Test your program by solving the problem in Exercise 28.

CHAPTER 10

Curve Fitting: Regression

10.1 INTRODUCTION

In the previous chapter we discussed various methods of curve fitting for data points of well-defined functions. In this chapter, we will discuss methods of curve fitting for experimental data.

In many applications, it often becomes necessary to establish a mathematical relationship between experimental values. This relationship may be used for either testing existing mathematical models or establishing new ones. The mathematical equation can also be used to predict or forecast values of the dependent variable. For example, we would like to know the maintenance cost of an equipment (or a vehicle) as a function of age (or mileage) or the relationship between the literacy level and population growth. The process of establishing such relationships in the form of a mathematical equation is known as *regression analysis* or *curve fitting*.

Suppose the values of y for the different values of x are given. If we want to know the effect of x on y , then we may write a functional relationship

$$y = f(x)$$

The variable y is called the *dependent variable* and x the *independent variable*. The relationship may be either linear or nonlinear as shown in Fig. 10.1. The type of relationship to be used should be decided by the experiment based on the nature of scatteredness of data.

It is a standard practice to prepare a *scatter diagram* as shown in Fig. 10.2 and try to determine the functional relationship needed to fit the points. The line should best fit the plotted points. This means that the

average error introduced by the assumed line should be minimum. The parameters a and b of the various equations shown in Fig. 10.1 should be evaluated such that the equations best represent the data.

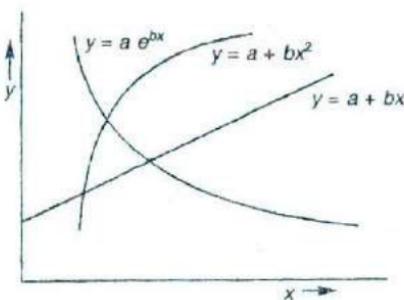


Fig. 10.1 Various relationships between x and y

We shall discuss in this chapter a technique known as *least-squares regression* to fit the data under the following situations:

1. Relationship is linear
2. Relationship is transcendental
3. Relationship is polynomial
4. Relationship involves two or more independent variables

10.2 FITTING LINEAR EQUATIONS

Fitting a straight line is the simplest approach of regression analysis. Let us consider the mathematical equation for a straight line

$$y = a + bx = f(x)$$

to describe the data. We know that a is the intercept of the line and b its slope. Consider a point (x_i, y_i) as shown in Fig. 10.2. The vertical distance of this point from the line $f(x) = a + bx$ is the error q_i . Then,

$$\begin{aligned} q_i &= y_i - f(x_i) \\ &= y_i - a - bx_i \end{aligned} \quad (10.1)$$

There are various approaches that could be tried for fitting a "best" line through the data. They include:

1. Minimise the sum of errors, i.e., minimise

$$\sum q_i = \sum (y_i - a - bx_i) \quad (10.2)$$

2. Minimise the sum of absolute values of errors

$$\sum |q_i| = \sum |(y_i - a - bx_i)| \quad (10.3)$$

3. Minimise the sum of squares of errors

$$\sum q_i^2 = \sum (y_i - a - bx_i)^2 \quad (10.4)$$

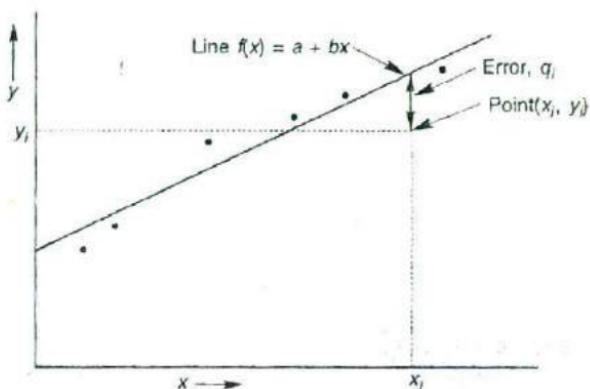


Fig. 10.2 Scatter diagram

It can be easily verified that the first two strategies do not yield a unique line for a given set of data. The third strategy overcomes this problem and guarantees a unique line. The technique of minimising the sum of squares of errors is known as *least squares regression*. In this section we consider the least-squares fit of a straight line.

Least Squares Regression

Let the sum of squares of individual errors be expressed as

$$\begin{aligned} Q &= \sum_{i=1}^n q_i^2 = \sum_{i=1}^n [y_i - f(x_i)]^2 \\ &= \sum_{i=1}^n (y_i - a - bx_i)^2 \end{aligned} \quad (10.5)$$

In the method of least squares, we choose a and b such that Q is minimum. Since Q depends on a and b , a necessary condition for Q to be minimum is

$$\frac{\partial Q}{\partial a} = 0 \quad \text{and} \quad \frac{\partial Q}{\partial b} = 0$$

Then

$$\begin{aligned} \frac{\partial Q}{\partial a} &= -2 \sum_{i=1}^n (y_i - a - bx_i) = 0 \\ \frac{\partial Q}{\partial b} &= -2 \sum_{i=1}^n x_i (y_i - a - bx_i) = 0 \end{aligned} \quad (10.6)$$

Thus

$$\begin{aligned} \sum y_i &= na + b \sum x_i \\ \sum x_i y_i &= a \sum x_i + b \sum x_i^2 \end{aligned} \quad (10.7)$$

These are called *normal equations*. Solving for a and b , we get

$$\boxed{b = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}}$$

$$\boxed{a = \frac{\sum y_i}{n} - b \frac{\sum x_i}{n} = \bar{y} - b \bar{x}} \quad (10.8)$$

where \bar{x} and \bar{y} are the averages of x values and y values, respectively.

Example 10.1

Fit a straight line to the following set of data

x	1	2	3	4	5
y	3	4	5	6	8

The various summations are given as follows:

x_i	y_i	x_i^2	$x_i y_i$
1	3	1	3
2	4	4	8
3	5	9	15
4	6	16	24
5	8	25	40
Σ	15	55	90

Using Eq. (10.8),

$$b = \frac{5 \times 90 - 15 \times 26}{5 \times 55 - 15^2} = 1.20$$

$$a = \frac{26}{5} - 1.20 \times \frac{15}{5} = 1.60$$

Therefore, the linear equation is

$$y = 1.6 + 1.2x$$

The regression line along with the data is shown in Fig. 10.3.

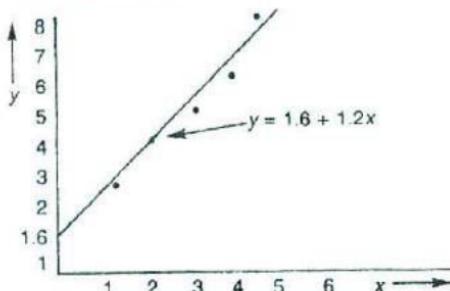


Fig. 10.3 Plot of the data and regression line of example 10.1

Algorithm

It is relatively simple to implement the linear regression on a computer. The coefficients a and b can be evaluated using Algorithm 10.1

Linear Regression

1. Read data values
2. Compute sum of powers and products
 $\Sigma x_i, \Sigma y_i, \Sigma x_i^2, \Sigma x_i y_i$
3. Check whether the denominator of the equation for b is zero.
4. Compute b and a .
5. Print out the equation.
6. Interpolate data, if required.

Algorithm 10.1

Program LINREG

Program LINREG implements Algorithm 10.1. The program reads a table of data points and decides a straight line equation to fit the data using the method of least squares regression.

```

*-----*
*      PROGRAM LINREG
*-----*
* Main program
*   This program fits a line Y = A + BX to a given
*   set of data points by the method of least squares
*-----*
* Functions invoked
*   ABS
*-----*
* 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
*
* -----
INTEGER MAX,N
REAL X,Y,SUMX,SUMY,SUMXX,SUMXY,XMEAN,YMEAN,DENOM,A,B
INTRINSIC ABS
PARAMETER( MAX = 10 )
DIMENSION X(MAX),Y(MAX)

WRITE(*,*)
WRITE(*,*)      'LINEAR REGRESSION'
WRITE(*,*)

* Reading data values
WRITE(*,*) 'Input number of data points N'
READ(*,*) N
WRITE(*,*) 'Input X and Y values,',
+           'one set on each line'
DO 10 I = 1, N
  READ(*,*) X(I), Y(I)
10 CONTINUE

* Computing constants A and B
SUMX = 0.0
SUMY = 0.0
SUMXY = 0.0
SUMXX = 0.0
DO 20 I = 1, N
  SUMX = SUMX + X(I)
  SUMY = SUMY + Y(I)
  SUMXX = SUMXX + X(I) * X(I)
  SUMXY = SUMXY + X(I) * Y(I)
20 CONTINUE

XMEAN = SUMX/N
YMEAN = SUMY/N
DENOM = N * SUMXX - SUMX * SUMX
IF (ABS (DENOM) .GT. 0.00001) THEN
  B = (N * SUMXY - SUMX * SUMY)/DENOM
  A = YMEAN - B * XMEAN
ELSE
  WRITE(*,*)
  WRITE(*,*) 'NO SOLUTION'
  STOP
ENDIF

* Printing results
WRITE(*,*)

```

```

WRITE(*,*) 'LINEAR REGRESSION LINE Y = A + BX'
WRITE(*,*) ,
WRITE(*,*) 'THE COEFFICIENTS ARE:'
WRITE(*,*) '    A = ', A
WRITE(*,*) '    B = ', B
WRITE(*,*) ,
STOP
END

```

* ----- End of main LINREG ----- *

Test Run Results Shown below is the interactive data input and the linear regression line parameters computed by the program LINREG.

```

LINEAR REGRESSION
Input number of data points N
5
Input X and Y values, one set on each line
1 3
2 5
3 7
4 9
5 11

LINEAR REGRESSION LINE Y = A + BX
THE COEFFICIENTS ARE:
A = 1.0000000
B = 2.0000000
Stop - Program terminated.

```

10.3 FITTING TRANSCENDENTAL EQUATIONS

The relationship between the dependent and independent variables is not always linear. Look at Fig. 10.4. The nonlinear relationship between

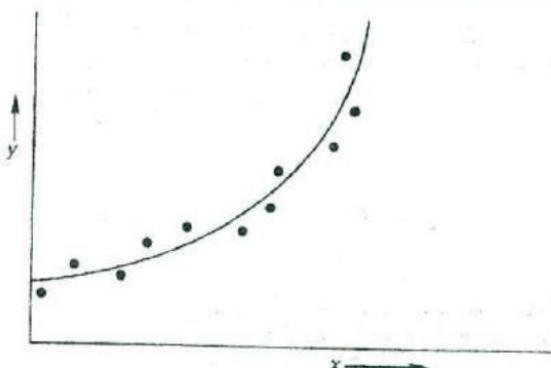


Fig.10.4 Data would fit a nonlinear curve better than a linear one

them may exist in the form of transcendental equations (or higher order polynomials). For example, the familiar equation for population growth is given by

$$P = P_0 e^{kt} \quad (10.9)$$

where P_0 is the initial population, k is the rate of growth and t is time. Another example of nonlinear model is the gas law relating to the pressure and volume, as given by

$$p = a v^b \quad (10.10)$$

Let us consider Eq. (10.10) first. If we observe values of p for various values of v , we can then determine the parameters a and b . Using the method of least squares, the sum of the squares of all errors can be written as

$$Q = \sum_{i=1}^n [p_i - av_i^b]^2$$

To minimise Q , we have

$$\frac{\partial Q}{\partial a} = 0 \quad \text{and} \quad \frac{\partial Q}{\partial b} = 0$$

We can prove that

$$\begin{aligned} \sum p_i v_i^b &= a \sum (v_i^b)^2 \\ \sum p_i v_i^b \ln v_i &= a \sum (v_i^b)^2 \ln v_i \end{aligned}$$

These equations can be solved for a and b . But since b appears under the summation sign, an iterative technique must be employed to solve for a and b .

However, this problem can be solved by using the algorithm given in the previous section in the following way: let us rewrite the equation using the conventional variables x and y as

$$y = ax^b$$

If we take logarithm on both the sides, we get

$$\ln y = \ln a + b \ln x \quad (10.11)$$

This equation is similar in form to the linear equation and, therefore, using the same procedure we can evaluate the parameters a and b .

$$b = \frac{n \sum \ln x_i \ln y_i - \sum \ln x_i \sum \ln y_i}{n \sum (\ln x_i)^2 - (\sum \ln x_i)^2} \quad (10.12)$$

$$\begin{aligned} \ln a = R &= \frac{1}{n} (\sum \ln y_i - b \sum \ln x_i) \\ a &= e^R \end{aligned} \quad (10.13)$$

Similarly, we can linearise the exponential model shown in Eq. (10.9) by taking logarithm on both the sides. This would yield

$$\ln P = \ln P_0 + kt \ln e$$

Since, $\ln e = 1$,
we have $\ln P = \ln P_0 + kt$ (10.14)

This is similar to the linear equation

$$y = a + bx$$

where $y = \ln P$, $a = \ln P_0$, $b = k$, and $x = t$. We can now easily determine a and b and then P_0 and k .

There is a third form of nonlinear model known as *saturation-growth-rate* equation, as shown below:

$$p = \frac{k_1 t}{k_2 + t} \quad (10.15)$$

This can be linearised by taking inversion of the terms. That is

$$\frac{1}{p} = \left(\frac{k_2}{k_1} \right) \frac{1}{t} + \frac{1}{k_1} \quad (10.16)$$

This is again similar to the linear equation

$$y = a + bx$$

where

$$y = \frac{1}{p}, \quad x = \frac{1}{t}$$

$$a = \frac{1}{k_1}, \quad b = \frac{k_2}{k_1}$$

Once we obtain a and b , they could be transformed back into the original form for the purpose of analysis.

Example 10.2

Given the data table

x	1	2	3	4	5
y	0.5	2	4.5	8	12.5

fit a power-function model of the form

$$y = ax^b$$

Various quantities required in equation (10.12) are tabulated below:

x_i	y_i	$\ln x_i$	$\ln y_i$	$(\ln x_i)^2$	$(\ln x_i)(\ln y_i)$
1	0.5	0	-0.6931	0	0
2	2	0.6931	0.6931	0.4805	0.4804
3	4.5	1.0986	1.5041	1.2069	1.6524
4	8	1.3863	2.0794	1.9218	2.8827
5	12.5	1.6094	2.5257	2.5903	4.0649
Sum		4.7874	6.1092	6.1995	9.0804

Using Eq. (10.12),

$$b = \frac{(5)(9.0804) - (4.7874)(6.1092)}{(5)(6.1995) - (4.7874)^2}$$

$$= \frac{45.402 - 29.2472}{30.9975 - 22.9192}$$

$$= 1.9998$$

$$\ln a = \frac{6.1092 - (1.9998)(4.7847)}{5}$$

$$= -0.6929$$

$$a = 0.5001$$

Thus, we obtain the power-function equation as

$$y = 0.5001 x^{1.9998}$$

Note that the data have been derived from the equation

$$y = \frac{x^2}{2}$$

The discrepancy in the computed coefficients is due to roundoff errors.

Example 10.3

The temperature of a metal strip was measured at various time intervals during heating and the values are given in the table below:

time, t (min)	1	2	3	4
temp, T ($^{\circ}$ C)	70	83	100	124

If the relationship between the temperature T and time t is of the form

$$T = be^{t/4} + a$$

estimate the temperature at $t = 6$ min.

We can write the temperature equation in the form

$$y = b f(x) + a$$

This is similar to the linear equation except that the variable x is replaced by the function $f(x)$. Therefore, we can solve for the parameters a and b using Eq. (10.8) by replacing

$$x_i \text{ by } f(x_i)$$

$$\sum x_i \text{ by } \sum f(x_i)$$

$$\sum x_i^2 \text{ by } \sum f(x_i)^2$$

Thus,

$$b = \frac{n(\sum f(x_i)y_i) - \sum f(x_i)\sum y_i}{n \sum [f(x_i)]^2 - [\sum f(x_i)]^2}$$

$$a = \frac{\sum y_i - b \sum f(x_i)}{n}$$

We can set up the following table to obtain the various terms. Note that $f(x) = e^{0.25x}$.

x	y	$f(x)$	$y \cdot f(x)$	$[f(x)]^2$
1	70	1.28	89.89	1.65
2	83	1.65	136.84	2.72
3	100	2.12	211.70	4.48
4	124	2.72	337.07	7.39
Sum	377	7.77	775.5	16.24

Now,

$$b = \frac{(4)(775.5) - (7.77)(377)}{(4)(16.24) - (7.77)^2}$$

$$= 37.62$$

$$a = \frac{377 - (37.62)(7.77)}{4}$$

$$= 21.16$$

The equation is

$$T = 37.62 e^{0.25t} + 21.16$$

The temperature, when $t = 6$, is

$$\begin{aligned} T &= 37.62 e^{0.25(6)} + 21.16 \\ &= 189.76^\circ\text{C} \end{aligned}$$

10.4

FITTING A POLYNOMIAL FUNCTION

When a given set of data does not appear to satisfy a linear equation, we can try a suitable polynomial as a regression curve to fit the data. The least squares technique can be readily used to fit the data to a polynomial.

Consider a polynomial of degree $m - 1$

$$\begin{aligned} y &= a_1 + a_2 x + a_3 x^2 + \dots + a_m x^{m-1} \\ &= f(x) \end{aligned} \tag{10.17}$$

334 Numerical Methods

If the data contains n sets of x and y values, then the sum of squares of the errors is given by

$$Q = \sum_{i=1}^n [y_i - f(x_i)]^2 \quad (10.18)$$

Since $f(x)$ is a polynomial and contains coefficients a_1, a_2, a_3 , etc., we have to estimate all the m coefficients. As before, we have the following m equations that can be solved for these coefficients.

$$\frac{\partial Q}{\partial a_1} = 0$$

$$\frac{\partial Q}{\partial a_2} = 0$$

...

...

...

$$\frac{\partial Q}{\partial a_m} = 0$$

Consider a general term,

$$\frac{\partial Q}{\partial a_j} = -2 \sum_{i=1}^n [y_i - f(x_i)] \frac{\partial f(x_i)}{\partial a_j} = 0$$

$$\frac{\partial f(x_i)}{\partial a_j} = x_i^{j-1}$$

Thus, we have

$$\sum_{i=1}^n [y_i - f(x_i)] x_i^{j-1} = 0 \quad j = 1, 2, \dots, m$$

$$\sum [y_i x_i^{j-1} - x_i^{j-1} f(x_i)] = 0$$

Substituting for $f(x_i)$

$$\sum_{i=1}^n x_i^{j-1} (a_1 + a_2 x_i + a_3 x_i^2 + \dots + a_m x_i^{m-1}) = \sum_{i=1}^n y_i x_i^{j-1}$$

These are m equations ($j = 1, 2, \dots, m$) and each summation is for $i = 1$ to n .

$$\begin{aligned} a_1 n + a_2 \sum x_i + a_3 \sum x_i^2 + \dots &+ a_m \sum x_i^{m-1} = \sum y_i \\ a_1 \sum x_i + a_2 \sum x_i^2 + a_3 \sum x_i^3 + \dots &+ a_m \sum x_i^m = \sum y_i x_i \\ \vdots &\quad \vdots \quad \vdots \quad \vdots \\ a_1 \sum x_i^{m-1} + a_2 \sum x_i^m + a_3 \sum x_i^{m+1} + \dots &+ a_m \sum x_i^{2m-2} = \sum y_i x_i^{m-1} \end{aligned} \quad (10.19)$$

The set of m equations can be represented in matrix notation as follows:

$$\mathbf{CA} = \mathbf{B}$$

where

$$\mathbf{C} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 & \dots & \sum x_i^{m-1} \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \dots & \sum x_i^m \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \sum x_i^{m-1} & \sum x_i^m & \dots & \dots & \sum x_i^{2m-2} \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_m \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} \sum y_i \\ \sum y_i x_i \\ \sum y_i x_2 \\ \vdots \\ \sum y_i x_i^{m-1} \end{bmatrix}$$

The element of matrix \mathbf{C} is

$$C(j, k) = \sum_{i=1}^n x_i^{j+k-2} \quad j = 1, 2, \dots, m \quad \text{and} \quad k = 1, 2, \dots, m$$

Similarly,

$$\mathbf{B}(j) = \sum_{i=1}^n y_i x_i^{j-1} \quad j = 1, 2, \dots, m$$

Implementation

Fit a second order polynomial to the data in the table below:

x	1.0	2.0	3.0	4.0
y	6.0	11.0	18.0	27.0

The order of polynomial is 2 and therefore we will have 3 simultaneous equations as shown below:

$$a_1 n + a_2 \sum x_i + a_3 \sum x_i^2 = \sum y_i$$

$$a_1 \sum x_i + a_2 \sum x_i^2 + a_3 \sum x_i^3 = \sum y_i x_i$$

$$a_1 \sum x_i^2 + a_2 \sum x_i^3 + a_3 \sum x_i^4 = \sum y_i x_i^2$$

The sums of powers and products can be evaluated in a tabular form as shown below:

x	y	x^2	x^3	x^4	yx	yx^2
1	6	1	1	1	6	6
2	11	4	8	16	22	44
3	18	9	27	81	54	162
4	27	16	64	256	108	432
Σ	10	62	30	100	354	644

Substituting these values, we get

$$4a_1 + 10a_2 + 30a_3 = 62$$

$$10a_1 + 30a_2 + 100a_3 = 190$$

$$30a_1 + 100a_2 + 354a_3 = 644$$

Solving these equations gives

$$a_1 = 3$$

$$a_2 = 2$$

$$a_3 = 1$$

Therefore, the least squares quadratic polynomial is

$$y = 3 + 2x + x^2 \quad (\text{verify using table data})$$

Algorithm for Polynomial Fit

The set of m equations given by Eq. (10.19) can be solved by using an elimination method discussed in Chapter 7. Algorithm 10.2 lists the steps involved in computing the coefficients of the regression polynomial.

Polynomial Regression

1. Read number of data points n and order of polynomial mp
2. Read data values
3. If $n \leq mp$,
 print out 'regression is not possible' and stop;
 else
 continue
4. Set $m = mp + 1$
5. Compute coefficients of **C** matrix
6. Compute coefficients of **B** matrix
7. Solve for the coefficients a_1, a_2, \dots, a_m
8. Write the coefficients
9. Estimate the function value at the given value of independent variable
10. Stop

Algorithm 10.2

Program POLREG

This program fits a polynomial curve to a given set of data points by the method of least squares. POLREG uses a subprogram NORMAL to compute the coefficients of normal equations and another subprogram GAUSS to solve the normal equations obtained. Finally, the program prints the polynomial coefficients, $a(1)$ to $a(m)$.

```

*-----*
      PROGRAM 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
*-----*
      REAL X,Y,C,A,B
      INTEGER N,MP,M,MAX
      PARAMETER (MAX = 10)
      DIMENSION X(MAX),Y(MAX),C(MAX,MAX),A(MAX),B(MAX)
      WRITE(*,*) ''
      WRITE(*,*)           'POLYNOMIAL REGRESSION'
      WRITE(*,*) ''

* Reading data values
      WRITE(*,*) 'Input number of data points(N)'
      READ(*,*) N
      WRITE(*,*) 'Input order of polynomial(MP) required'
      READ(*,*) MP
      WRITE(*,*) 'Input data values X and Y,'
      +           'one set on each line'

```

338 Numerical Methods

```
DO 10 I = 1, N
      READ(*,*) X(I), Y(I)
10 CONTINUE
* Testing the order
      IF(N.LE.MP) THEN
          WRITE(*,*) 'REGRESSION IS NOT POSSIBLE'
          GO TO 20
      ENDIF
* Number of polynomial coefficients
      M = MP+1
* Computation of elements of C and B
      CALL NORMAL(X,Y,C,B,N,M,MAX)
* Computation of coefficients a(1) to a(m)
      CALL GAUSS(M,C,B,A)
* Output of coefficients a(1) to a(m)
      WRITE(*,*) 'POLYNOMIAL COEFFICIENTS'
      WRITE(*,*) (A(I), I=1,M)
      WRITE(*,*)
20 STOP
END
* ----- End of main program POLREG -----
*
* -----
* SUBROUTINE NORMAL(X,Y,C,B,N,M,MAX)
* -----
* Subroutine
* This subroutine computes the coefficients
* of normal equations
* -----
* Arguments
* Input
*   N - Number of data points
*   X,Y - Arrays of data values
*   M - Number of coefficients of the polynomial
*   MAX - Maximum size of arrays
* Output
*   C - Coefficient matrix of normal equations
*   B - Right side vector of normal equations
* -----
* Local Variables
*   NIL
* -----
```

```

* Functions invoked
* NIL
* -----
* Subroutines called
* NIL
*
REAL X,Y,C,B
INTEGER N,M,MAX
DIMENSION X(MAX),Y(MAX),C(MAX,MAX),B(MAX)
DO 30 J=1,M
  DO 20 K=1,M
    C(J,K) = 0.0
    L1 = K+J-2
    DO 10 I=1,N
      C(J,K) = C(J,K) + X(I) ** L1
10   CONTINUE
20   CONTINUE
30   CONTINUE
DO 50 J= 1,M
  B(J) = 0.0
  L2 = J-1
  DO 40 I = 1,N
    B(J) = B(J) + Y(I) * X(I) ** L2
40   CONTINUE
50   CONTINUE
RETURN
END
* ----- End of subroutine NORMAL -----
* -----
SUBROUTINE GAUSS(N,A,B,X)
* -----
* Subroutine
* This subroutine solves a set of n linear
* equations by Gauss elimination method
* -----
* Arguments
* Input
*   N - Number of equations
*   A - Matrix of coefficients
*   B - Right side vector
* Output
*   X - Solution vector
* -----
* Local Variables
*   PIVOT, FACTOR, SUM
* -----

```

```

* Functions invoked *
* NIL *
* -----
* Subroutines called *
* NIL *
* -----
* ----- Elimination begins -----
* INTEGER N
REAL A,B,X, PIVOT,FACTOR,SUM
DIMENSION A(10,10), B(10), X(10)
* ----- Elimination begins -----
DO 33 K = 1, N-1
    PIVOT = A(K,K)
    DO 22 I = K+1, N
        FACTOR = A(I,K)/PIVOT
        DO 11 J = K+1, N
            A(I,J) = A(I,J) - FACTOR * A(K,J)
11    CONTINUE
        B(I) = B(I) - FACTOR * B(K)
22    CONTINUE
33    CONTINUE
* ----- Back substitution begins -----
X(N) = B(N)/A(N,N)
DO 55 K = N-1,1,-1
    SUM = 0
    DO 44 J = K+1,N
        SUM = SUM + A(K,J) * X(J)
44    CONTINUE
    X(K) = (B(K) - SUM)/A(K,K)
55    CONTINUE
    RETURN
END
* ----- End of subroutine GAUSS -----

```

Test Run Results The program was used to fit a polynomial curve to the following data points:

x_i	1.0	2.1	3.2	4.0
y_i	2.0	2.5	3.0	4.0

The results are given below:

POLYNOMIAL REGRESSION

Input number of data points(N)

4

Input order of polynomial(MP) required

2

Input data values X and Y, one set on each line

1.0 2.0

2.1 2.5
3.2 3.0
4.0 4.0

POLYNOMIAL COEFFICIENTS

2.0740160 -2.053067E-001 1.680441E-001

Stop - Program terminated.



MULTIPLE LINEAR REGRESSION

There are a number of situations where the dependent variable is a function of two or more variables. For example, the salary of a salesperson may be expressed as

$$y = 500 + 5x_1 + 8x_2$$

where x_1 and x_2 are the number of units sold of products 1 and 2, respectively. We shall discuss here an approach to fit the experimental data where the variable under consideration is a linear function of two independent variables.

Let us consider a two-variable linear function as follows:

$$y = a_1 + a_2x + a_3z \quad (10.20)$$

The sum of the squares of errors is given by

$$Q = \sum_{i=1}^n (y_i - a_1 - a_2x_i - a_3z_i)^2$$

Differentiating with respect to a_1 , a_2 and a_3 , we get,

$$\frac{\partial Q}{\partial a_1} = -2 \sum (y_i - a_1 - a_2x_i - a_3z_i)$$

$$\frac{\partial Q}{\partial a_2} = -2 \sum (y_i - a_1 - a_2x_i - a_3z_i) x_i$$

$$\frac{\partial Q}{\partial a_3} = -2 \sum (y_i - a_1 - a_2x_i - a_3z_i) z_i$$

Setting these partial derivatives equal to zero results in

$$na_1 + (\sum x_i)a_2 + (\sum z_i)a_3 = \sum y_i$$

$$(\sum x_i)a_1 + (\sum x_i^2)a_2 + (\sum x_i z_i)a_3 = \sum y_i x_i$$

$$(\sum z_i)a_1 + (\sum x_i z_i)a_2 + (\sum z_i^2)a_3 = \sum y_i z_i$$

These are three simultaneous equations with three unknowns and, therefore, can be expressed in matrix form as

$$\begin{bmatrix} n & \sum x_i & \sum z_i \\ \sum x_i & \sum x_i^2 & \sum x_i z_i \\ \sum z_i & \sum x_i z_i & \sum z_i^2 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum y_i x_i \\ \sum y_i z_i \end{bmatrix} \quad (10.21)$$

This equation can be solved using any standard method. This is a two-dimensional case and, therefore, we obtain a regression "plane" rather than "line".

We can easily extend Eq. (10.21) to the more general case

$$y = a_1 + a_2 x_1 + a_3 x_2 + \dots + a_{m+1} x_m$$

Example 10.5

Given the table of data

x	1	2	3	4
z	0	1	2	3
y	12	18	24	30

Obtain a regression plane to fit the data.

The various sums of powers and products required for evaluation of coefficients are tabulated below:

x	z	y	x^2	z^2	xz	yx	yz
1	0	12	1	0	0	12	0
2	1	18	4	1	2	36	18
3	2	24	9	4	6	72	48
4	3	30	16	9	12	120	90
Σ	10	84	30	14	20	240	156

On substitution of these values in Eq. (10.21) we get

$$\begin{aligned} 4a_1 + 10a_2 + 5a_3 &= 84 \\ 10a_1 + 30a_2 + 20a_3 &= 240 \\ 6a_1 + 20a_2 + 14a_3 &= 156 \end{aligned}$$

Solution of these equations results in

$$a_1 = 10$$

$$a_2 = 2$$

$$a_3 = 4$$

Thus, the regression plane is

$$y = 10 + 2x + 4z$$

10.6

ILL-CONDITIONING IN LEAST SQUARES

METHODS

The problem of ill-conditioning can arise in implementing the least squares regression methods. As a consequence, the computed solution might differ substantially from its exact solution. This problem becomes more severe when the degree of approximating polynomial is large.

Ill-conditioning arises basically due to very large differences in the coefficients of the normal equations. Recall that the coefficients are sums of powers and products of data values. Techniques such as pivoting and iterative refinement can be incorporated to overcome the problem of ill-conditioning. The problem of ill-conditioning can also be tackled by increasing the precision of arithmetic operations.

Another way of overcoming the least-squares ill-conditioning problem is to use orthogonal polynomials. This would enable us to obtain the coefficients a_i in closed form, thus avoiding numerical solution of simultaneous equations. Further discussions on this approach is beyond the scope of this book.

10.7 SUMMARY

We often use experimental data for establishing a relationship between two variables. This relationship may be used for testing some existing mathematical models or establishing new ones or even estimating the values of dependent variables at some point. In this chapter, we have used a technique known as least squares regression to establish the following types of relationship between the variables of a table of experimental data.

- Linear relationship
- Transcendental relationship
- Polynomial relationship
- Multivariable relationship

Also presented are FORTRAN programs and test results for obtaining linear and polynomial equations for experimental data.

Key Terms

*Curve fitting
Dependent variable
Independent variable
Least squares regression
Normal equations*

*Regression analysis
Regression line
Regression plane
Saturation growth rate
Scatter diagram*

REVIEW QUESTIONS

1. What is regression analysis?
2. What is a scatter diagram?
3. What is the principle of least squares regression?
4. Show that the linear regression line of y on x passes through the point that represents the mean of x and y values.

6. Derive normal equations for evaluating the parameters a and b to fit data to
 (a) power-function model of the form

$$y = ax^b$$

 (b) population growth model of the form

$$y = a e^{bx}$$
- using the principle of least squares.
6. Draw a flow chart to illustrate the steps involved in developing a program for multiple regression.

REVIEW EXERCISES

1. Use least squares regression to fit a straight line to the data.

x	1	3	4	6	8	9	11
y	1	2	4	4	5	7	8

Along with the slope and intercept, also compute the standard error of the estimate.

2. In an organisation, systematic efforts were introduced to reduce the employee absenteeism and results for the first 10 months are shown below:

Months	1	2	3	4	5	6	7	8	9	10
Absentees (per cent)	10	9	9	8.5	9	8	8.5	7	8	7.5

Fit a linear least squares line to the data and from this equation estimate the average weekly reduction in absenteeism.

3. The following table shows heights (h) and weights (w) of 8 persons.

$h(\text{cm})$	175	165	160	180	150	170	155	185
$w(\text{kg})$	68	58	59	71	51	62	53	68

Assuming a linear relationship between the height and weight, find the regression line and estimate the weights of the persons with the following heights.

(a) 140 cm

(b) 163 cm

(c) 172.5

4. Fit a geometric curve

$$y = ax^b$$

to the following data:

x	-2	-1	0	1	2	3	4
y	38	6	0	-5	-41	130	300

5. Given the table of points

x	0	2	4	6	8	12	16	20
y	10	12	18	22	20	30	26	30

use least squares regression to fit

- (a) straight line, and
- (b) parabola

to the data. Compute and compare the errors.

6. Fit the saturation growth rate model

$$y = \frac{ax}{b+x}$$

to the data given below.

x	2	4	6	8
y	1.4	2.0	2.4	2.6

7. Fit the power equation

$$y = ax^b$$

to the data given in Exercise 6.

8. Fit a quadratic polynomial to the data given in Exercise 6.

9. Use the power equation to the data

x	7.5	10	12.5	15	17.5	20
y	2.4	1.6	1.2	0.8	0.6	0.6

10. Use the exponential model

$$y = a e^{bx}$$

to fit the data

x	0.4	0.8	1.2	1.6	2.0	2.4
y	75	100	140	200	270	375

11. Fit a parabola to the data given in Exercise 10.

12. Find the least squares line $y = ax + b$ that fits the following data, assuming that there are no errors in x values.

x	1	2	3	4	5	6
y	4.05	7.12	9.65	12.20	15.20	19.00

13. In Exercise 12, treat x as dependent variable on y and find the least squares line $x = ay + b$, assuming that there are no errors in y but x values contain errors. Observe that this is not the same line obtained in Exercise 12.

14. Use multiple linear regression to fit

x_1	1	2	3	4	5
x_2	4	3	2	1	0
y	18	16	16	12	10

Compute coefficients and the error of estimate.

15. Given the data points

x_1	5	4	3	2	1
x_2	3	-2	-1	4	0
y	15	-8	-1	26	8

obtain a regression plane to fit the data.

PROGRAMMING PROJECTS

1. Modify the program LINREG to calculate the sum of squares of the errors for the linear fit and print the error output.
2. A set of data, when plotted resembles an exponential curve

$$y = a(b^x)$$

- Write a program to evaluate the parameters a and b of this regression curve using the principle of least squares.
3. In fitting a polynomial, its degree should be chosen such that the error is minimum. Given a set of data, it would be difficult to decide the degree that would represent the data best. A good rule of thumb is to begin with the first degree and continue fitting higher order polynomials until

$$\frac{Q_i}{n-i-1} > \frac{Q_{i-1}}{n-i}$$

or until a polynomial of degree $n-1$ is obtained. Q_i is the sum of squares of errors of polynomial of degree i .

- (a) Prepare a flow chart to fit a polynomial that satisfies this condition.
- (b) Modify the program POLREG to incorporate these changes.
4. Develop a user-friendly program for multiple regression.
5. Develop a user-friendly, menu-driven program that allows us an option to select and use one of the following models to fit a given set of data.
 - (a) Straight line model
 - (b) Exponential model
 - (c) Power equation
 - (d) Saturation-growth rate model