NETAJI SUBHAS OPEN UNIVERSITY

STUDY MATERIAL

MATHEMATICS
POST GRADUATE

PG (MT) 04
GROUPS A & B

- Numerical Analysis
- Computer Programming and its application to Numerical Analysis

PREFACE

In the curricular structure introduced by this University for students of Post Graduate degree programme, the opportunity to pursue Post Graduate course in a subject is introduced by this University is equally available to all learners. Instead of being guided by any presumption about ability level, it would perhaps stand to reason if receptivity of a learner is judged in the course of the learning process. That would be entirely in keeping with the objectives of open education which does not believe in artificial differentiation.

Keeping this in view, study materials of the Post Graduate level in different subjects are being prepared on the basis of a well laid-out syllabus. The course structure combines the best elements in the approved syllabi of Central and State Universities in respective subjects. It has been so designed as to be upgradable with the addition of new information as well as results of fresh thinking and analysis.

The accepted methodology of distance education has been followed in the preparation of these study materials. Cooperation in every form of experienced scholars is indispensable for a work of this kind. We, therefore, owe an enormous debt of gratitude to everyone whose tireless efforts went into the writing, editing and devising of a proper lay-out of the materials. Practically speaking, their role amounts to an involvement in 'invisible teaching'. For, whoever makes use of these study materials would virtually derive the benefit of learning under their collective care without each being seen by the other.

The more a learner would seriously pursue these study materials the easier it will be for him or her to reach out to larger horizons of a subject. Care has also been taken to make the language lucid and presentation attractive so that they may be rated as quality self-learning materials. If anything remains still obscure or difficult to follow, arrangements are there to come to terms with them through the counselling sessions regularly available at the network of study centres set up by the University.

Needless to add, a great deal of these efforts is still experimental—in fact, pioneering in certain areas. Naturally, there is every possibility of some lapse or deficiency here and there. However, these do admit of rectification and further improvement in due course. On the whole, therefore, these study materials are expected to evoke wider appreciation the more they receive serious attention of all concerned.

Prof. (Dr.) Subha Sankar Sarkar
Vice-Chancellor

Sixth Reprint : October, 2017

Printed in accordance with the regulations of the Distance Education Bureau of the University Grants Commission.

Subject : Mathematics **Post Graduate**

Paper : PG (MT) 04 : Group A

Writer	**Editor**
Prof. Debasis Sarkar	Prof. Subhas Ch. Bose

Revised by Prof. Madhumangal Pal

Paper : PG (MT) 04 : Group B (Revised Syllabus)

Writer	**Editor**
Prof. Madhumangal Pal	Prof. Kajal De

NETAJI SUBHAS OPEN UNIVERSITY

PG (MT)–04
Numerical Analysis, Computer Programming and its application to Numerical Analysis

Group A

Numerical Analysis

Group B

Computer Programming and its application to Numerical Analysis

Unit 1 ❐ Introduction

On Errors :

The basic objective of numerical analysis is to provide an approximate solution of a mathematical problem where exact solution is not easily available. In the process of approximation or numerical computation, we usually express numbers in some representations, such as, binary, decimal, octal, etc., depending upon the methods or machines available for computations.

§ Objectives

After going though this unit you will be able to learn about—

- Errors in numerical computation
- Round-off errors and instability
- Control of round-off errors.

Let, x be a real number, positive or negative. Then, in decimal system x can be represented uniquely as,

$$|x| = a_i 10^i + a_{i-1}\ 10^{i-1} + \ldots + a_1 10^1 + a_0 10^1 + \ldots + a_{-j} 10^{-j} + \ldots \qquad \ldots\ldots(1)$$

and (1) may be concisely written as

$$x = \pm\ (a_i a_{i-1} \ldots\ a_1 a_0\ a_{-1}\ a_{-2} \ldots\ a_{-j} \ldots\ ) \qquad \ldots\ldots(1')$$

where $a_i \neq 0$ and all a_k's $\in \{0, 1, 2, \ldots, 9\}$, $k = i,\ i-1, \ldots, 1, 0, -1, \ldots, -j, \ldots$

It x is a rational number, (1) is either teminates or recurs; if x is irrational, (1) is non-recurring and non-terminating. The leftest digit a_i is the most significant digit of $|x|$ and the significance of a_k's decrease as it advances towards right.

Now, instead of using exact value of the number x, we use an approximate of x, say $\hat{x}$ and an error occurs. Then irrespective of the nature of the error, one can define an absolute and a relative error. The absolute error is defined by $\varepsilon = |x - \hat{x}|$ and the relative error is defined by $\dfrac{\varepsilon}{|x|} = \left|\dfrac{x - \hat{x}}{x}\right|$, provided $|x| \neq 0$.

Example 1. Suppose we use instead of two numbers $x = 8{\cdot}5250$, $y = 3{\cdot}1415$ their approximates $\hat{x} = 8{\cdot}5000$ and $\hat{y} = 3{\cdot}1400$.

Then absolute errors in two cases are,

$|x - \hat{x}| = 8.5250 - 8{\cdot}5000 = {\cdot}0250$ and

$|y - \hat{y}| = 3.1415 - 3{\cdot}1400{\cdot} = {\cdot}0015.$

While relative errors are,

$$\left|\frac{x - \hat{x}}{x}\right| = \frac{{\cdot}0250}{8{\cdot}5250} = {\cdot}0029326 \text{ (corrected to 7 decimal places) and}$$

$$\left|\frac{y - \hat{y}}{y}\right| = \frac{{\cdot}0015}{3{\cdot}1415} = {\cdot}0004775 \text{ (correct to 7 decimal places)}$$

And these two types of errors differ not too much. But if we consider a third case where exact number $z = 1{,}0000$ and approximate number $\hat{z} = 9999$, then,

absolute error $= 10000 - 9999 = 1$ while,

relative error $= \dfrac{1}{10000} = {\cdot}0001$, and they differ considerably.

Now to classify the errors in numerical computations it might be fruitful to study the sources of the errors and the growth of the individual errors. The sources of errors are usually static, while the growth takes place dynamically. Essentially the following three types of errors are of fundamental importance.

(i) Initial errors, (ii) truncation errors, (iii) round-off errors.

The initial errors are errors in initial data. An example of it is, when data are collected from a physical or chemical apparatus. Truncation errors arise when an infinite process is replaced by a finite one or when the process is finite but large enough and is replaced by smaller number of terms or steps. Well-known examples are computation of a definite integral through approximation with a sum or integration of an ordinary or partial differential equation by some difference method. Round-off errors depend on the fact that each number in a numerical computation must be rounded to a certain number of digits. We now briefly describe the rounding rules to approximate numbers.

Let us require a number x as represented in (1) to approximate upto m-th decimal point. Then the rules of rounding x are :

(i) If $a_{-(m+1)} \geq 5$ and at least one of $a_{-(m+2)}, a_{-(m+3)}$ ..... are not equal to zero then take,

$$\hat{x} = \pm [a_i\, 10^i + \ldots + a_1 10^1 + a_0 + a_{-1}10^{-1} + \ldots + (a_{-m} + 1).\, 10^{-m}] \quad \ldots\ldots (2)$$

(ii) If $a_{-(m+1)} = 5$ and all $a_{-(m+2)}, a_{-(m+3)}, \ldots$ are equal to zero, then take,

$$\hat{x} = \pm [a_i\, 10^i + \ldots + a_0 + a_{-1}10^{-1} + \ldots + (a_{-m} + 1).\, 10^{-m}] \quad \ldots\ldots (3)$$

when, a_{-m} is an odd digit and

$$\hat{x} = \pm [a_i\, 10^i + \ldots + a_0 + a_{-1}10^{-1} + \ldots + a_{-m}10^{-m}] \quad \ldots\ldots (4)$$

when a_{-m} is an even digit

(iii) If $a_{-(m+1)} < 5$, then take,

$$\hat{x} = \pm [a_i\, 10^i + \ldots + a_1 10^1 + a_0 + a_{-1}10^{-1} + \ldots + a_{-m}\, 10^{-m}] \quad \ldots\ldots (5)$$

Sometimes the rule (ii) designated as rule of an even digit.

Since, in place of x, we are considering a round approximate value $\hat{x}$ of x, an error occurs in the process of rounding. This error $x - \hat{x}$ is usually known as round-off error in approximating a number x.

While in truncation a number x to m-th decimal place, the digits beyond a_{-m} are simply dropped to get $\hat{x}$ i.e.,

$$\hat{x} = \pm [a_i 10^i + \ldots + a_1 10^1 + a_0 + a_{-1}10^{-1} + \ldots + a_{-m}10^{-m}] \quad \ldots\ldots (6)$$

and the corresponding error is known as truncation error.

Briefly, the relative merits of the two methods to size a number are that truncation is less expensive whereas rounding produces better accuracy.

It is clear from the rounding rule that, if $\varepsilon = x - \hat{x}$, then $|\varepsilon| \leq \frac{1}{2}.10^{-m}$.

Now, any rounded number can be written in the standard form, $\pm (\cdot b_{-1} b_{-2} \ldots\ldots b_{-k} \times 10^n)$, where n is an integer, either positive, or negative or zero and k is a positive integer and $b_{-1} \neq 0$. Then we call the rounded number is correct to k significant digits or figures and $b_{-1}, b_{-2}, \ldots, b_{-k}$ are respectively the 1st, 2nd, ..., k-th significant digit of the number. This form of a number is called the floating point representation of the number, the fraction $\cdot b_{-1} \ldots b_{-k}$ called its mantissa and the interger n as its exponent.

Example 2. Round-off the numbers, 13·7839, 1·5615, ·02342, 314·5 upto 3 decimal places and also upto 3 significant figures.

Ans. : Upto 3 decimal places rounding-off :

13·784, 1·562, ·023, 324·500.

While upto 3 significant figure rounding-off :

13·8, 1·56, ·0234, 314.

§ Round-off Errors and Instability : Inherent and Induced :

In the process of serious numerical computations we use machines, such as, calculators or computers. These machines or devices have limited storage capacity of numbers which varies from devices to devices. For example, a machine may have capacity that it can store numbers about then significant digits at most. So, if we require higher accuracy in significant digits, it cannot provide us accurately beyond its capacity. It readily causes an error in rounding a number and store the number with an error in its significant digits. This type of error may be referred as an inherent error. So we must enter the numbers in machines as inputs with rounding within the storage capacities of the devices. Now apart from the inherent round-off error, there are some other types of errors caused due to some arithmetic operations such as, addition, multiplication, etc., or, some further operations. These may be referred as induced or generated round-off errors.

[**Note** : *We omit some details regarding propagated errors due to arithmetic operations, some other related facts as they are the part of undergraduate courses, rather we concentrate here on some basic facts with illustrations which are also relevant in our courser.*]

Let us consider two numbers $x = 4429{\cdot}3$ and $y = {\cdot}0023495$, to store in a machine with capacity upto 4 significant digits. i.e., the device will store them as $\hat{x} = {\cdot}4429 \times 10^4$ and $\hat{y} = {\cdot}2350 \times 10^{-2}$. We want to calculate the sum, i.e. the value of $x + y$. Then, by machine, we have $\hat{x} + \hat{y} = {\cdot}4429 \times 10^4 = \hat{x}$, where as, $x + y = 4429{\cdot}3023495$. So the sum of a larger number (obviously, this largeness varies from device to device) with a relatively small number, is the larger number in this case. We may call such an addition as negligible addition. Again, if we multiply a large number with a relatively small number, or, divide by a small number, we may observe propagated round-off error, which is referred as error magnification.

One of the most important round-off error is commonly known as loss of significant

digits. It is generally caused in subtracting nearly two equal numbers. Consider, for example, $x = \cdot 342355$ and $y = \cdot 342346213$. Then, with 4 significant digit storing capacity machine, $\hat{x} = \cdot 3424$ and $\hat{y} = \cdot 3423$.

Therefore, $\hat{x} - \hat{y} = \cdot 0001 = \cdot 1000 \times 10^{-3}$, whereas, $x - y = \cdot 8787 \times 10^{-5}$.

It is sometimes known as catastrophic cancellation. Such loss can often be avoided by anticipating its occurrence. Consider the problem of finding the roots of the quadratic equation,

$$ax^2 + bx + c = 0 \qquad \ldots\ldots (7)$$

The roots are, $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ $\ldots\ldots (8)$

Let us assume that $b^2 - 4ac > 0$ and $b > 0$ and we wish to find the smaller in absolute value of the two roots by using (8);

$$\text{i.e., } x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad \ldots\ldots (9)$$

If $4ac$ is small compared with b^2, then, $\sqrt{b^2 - 4ac}$ will agree with b to several significant digits. Therefore, the calculated root by using (9), will be accurate to fewer places than were used during the calculation. More specifically, let us consider the example,

$$x^2 + 111 \cdot 11x + 1 \cdot 2121 = 0 \qquad \ldots\ldots (10)$$

Using (9) and five decimal digit floating point chopped arithmetic, we have,

$b^2 = 12{,}345$

$b^2 - 4ac = 12{,}340$

$\sqrt{b^2 - 4ac} = 111 \cdot 09$

then, $x_1 = -0 \cdot 01000$,

while infact, $x_1 = -0 \cdot 010910$ is correct root upto the number of digits shown. Here, we may avoid the loss of significance by using an alternative formula for (9),

$$\text{i.e., } x_1 = \frac{-2c}{b+\sqrt{b^2-4ac}} \qquad \ldots\ldots (11)$$

Using (11) we have with five decimal digit arithmetic,

$x_1 = -0{\cdot}010910$, which is accurate to five digits.

Another example of error avoidance is the evaluation of the function,

$f(x) = 1 - \cos x$, in six-decimal-digit arithmetic.

Since $\cos x$ is nearly equal to 1 if x is near to zero, there will be a loss of significant digits if we calculate $f(x)$ for x near zero by evaluating first $\cos x$ and then by subtracting from 1. But, if we use the formula,

$$f(x) = 1 - \cos x = \frac{1-\cos^2 x}{1+\cos x} = \frac{\sin^2 x}{1+\cos x}$$

or, by Taylor series expansion for $f(x)$, i.e.,

$$f(x) = \frac{x^2}{2} - \frac{x^4}{4!} + \ldots\ldots$$

The computed value of $f(x)$ for x near zero, agrees with $f(x)$ to at least six significant digits for a considerable number of x's near zero.

Now in dealing with errors we sometimes investigate two related concepts, such as, condition and instability. A problem is designated as ill-conditioned, if small changes in its initial data causes a large deviation in final result. There is a formal way to designate condition of a function f at x. It is usually measured by the maximum relative change in the function value $f(x)$ caused by a unit relative change in the argument x. In other words, conditions of f at x

$$= \max \left\{ \left| \frac{f(x)-f(\hat{x})}{f(x)} \right| \Big/ \left| \frac{x-\hat{x}}{x} \right| : |x-\hat{x}| \text{is small} \right\}$$

$$\approx \left| \frac{xf'(x)}{f(x)} \right|$$

where $\hat{x}$ is the approximate to the exact value x.

Larger the condition, more ill-conditioned the function is said to be. As for

example, if we want to calculate $f(x) = \sqrt{x}$, then, $\left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{x.\frac{1}{2\sqrt{x}}}{\sqrt{x}}\right| = \frac{1}{2}$, i.e., taking square root is a well-conditioned process. But, if we consider the function

$$f(x) = \frac{1}{1-x^3}, \text{ then, } f'(x) = \frac{3x^2}{(1-x^3)^2}.$$

$$\text{Therefore, } \left|\frac{xf'(x)}{f(x)}\right| = \frac{x.3x^2}{(1-x^3)^2}(1-x^3)\Bigg| = \left|\frac{3x^3}{1-x^3}\right| = \left|\frac{3x^2.x}{1-x^3}\right|$$

i.e., when x is near 1, the condition number is quite large. Hence the function may be called as ill-conditioned.

Now the concept of instability describes the sensitivity of numerical computation of a function $f(x)$ from x due to the errors that arise from the finite precision arithmetic. It is hard to measure the precise effect of these errors unless we compare the exact result with the result of calculations by using finite precision arithmetic. However it is possible to estimate these effects by considering the round-off errors one at a time. Consider, for example, the evaluation of $f(x) = \sqrt{x+1} - \sqrt{x}$, for large x, say, with order 10^4. The condition number of $f(x)$ at x is $\left|\frac{xf'(x)}{f(x)}\right| = \frac{1}{2}\left|\frac{\left(\frac{1}{\sqrt{x+1}} - \frac{1}{\sqrt{x}}\right)x}{\sqrt{x+1}-\sqrt{x}}\right|$

$= \frac{1}{2}\frac{|x|}{\sqrt{x(x+1)}} \approx \frac{1}{2}$, for large x, which is fairly good. In particular, if we calculate $f(5942)$ in six decimal arithmetic, then we have,

$$f(5942) = \sqrt{5943} - \sqrt{5942} = 77{\cdot}090855 - 77{\cdot}084309$$
$$= {\cdot}006486.$$

where as $f(5942) = {\cdot}006486$ accurate upto six decimal places. So, in this case error is not too much. Now, we analyze our computaional scheme :

Step 1 : Take the value 5942,

Step 2 : Compute 5942 + 1,

Step 3 : Calculate $\sqrt{5943}$.

Step 4 : Calculate $\sqrt{5942}$.

Step 5 : Calculate $\sqrt{5943} - \sqrt{5942}$.

Consider another function $\phi(y)$ which describes how we reach step 5 from step 4.

Let $\phi(y) = \sqrt{5943} - y$, then its condition at y is approximately,

$$\left|\frac{y\phi'(y)}{\phi(y)}\right| \approx \left|\frac{-y}{\sqrt{5943}-y}\right| = \frac{|y|}{|\sqrt{5943}-y|}.$$

This number is usually near 1 for large y except when y is near $\sqrt{5943}$. In our case, $y = \sqrt{5942} \approx 77{\cdot}08$, whereas $\sqrt{5943} - y \approx {\cdot}0064$ i.e., condition is ≈ 12043, which is a very large number. Hence, we conclude that the way we have calculated is as unstable way to evaluate $f(x)$. However, if we calculate $f(x)$ by the formula,

$f(x) = \dfrac{1}{\sqrt{x+1}+\sqrt{x}}$, then we may avoid such instability in computation.

Thus we are facing several types of problems in approximate comptations. There are several types of errors and sometimes it is really difficult to control all the errors. For example, one may think that inherent error may be controlled by increasing the accuracy level. But it cannot be possible if the initial data itself contains an inherent error. Sufficient precision in computation must be taken to avoid inherent error. Sometimes propagation of error is completely random in nature and it may grow like a rolling snow ball. Also, sources of errors are not always known to us and sometimes their nature is like parasites. Now we describe briefly some of the control measures for round-off errors.

Control of Round-off errors :

Rule 1 : To minimize inherent errors, we must input real data using the device which can store as many as significant digits.

Rule 2 : To minimize the overflow or underflow (i.e., the error caused due to the fact that the machines are unable to store the numbers), we must try to keep all our intermediate calculations which also keep the characteristic close to zero.

Rule 3 : To minimize propagated round-off error, we must try to keep mathematical expressions in a form which require fewest arithmetic operations. For example, in claculating values from a polynomial $a_0x^4 + a_1x^3 + \ldots + a_4$, we may calculate in the

nested form, i.e., $(((a_0x + a_1) . x + a_2).x + a_3).x + a_4$, etc., or use some algorithm which perhaps help us to check worse round-off errors.

Rule 4 : We must introduce such algorithm which tries to avoid any anticipated loss of significance that might occur from subtraction, division, etc.

For example, sometimes re-arrangement of functions avoid loss of significance. In calculation sum of a series, particular form of series representation of functions, such as, Maclaurin's form, etc., help us to avoid loss of significance.

In some cases statistical analysis of error propagation may help us to estimate round-off errors. There are several stochasitc models on propagation of round-off errors taking local errors as random variables. They are usually follow some distribution such as, uniform, normal, etc., from where the statistical analysis with the terms, such as, standard deviation, variance, moments, etc., help us to estimate accumulated round-of errors. These types of analyses have some advantages that we may aware of the facts about the error bounds and spreads and try to control the propagation of errors with suitable measures.

§ Summary :

In this unit, different type of errors and their effects are discussed with examples. Role of round-off errors and truncation errors are explained with suitable examples. The outlines to control round-off errors are also provided.

EXERCISES

1. For the following numbers x and $\hat{x}$, how many significant digits are there in $\hat{x}$ with respect to x?

   (i) $\hat{x} = 241{\cdot}023$, $x = 241{\cdot}01$

   (ii) $\hat{x} = -\ {\cdot}042223$, $x = -\ {\cdot}04228$

   (iii) $\hat{x} = 44{\cdot}3213$, $x = 44{\cdot}3604$

2. Find absolute error, relative error. Also determine the number of significant digits in the approximation.

   (i) $x = 2{\cdot}718281$, $\hat{x} = 2{\cdot}718$

   (ii) $x = 85450$, $\hat{x} = 85000$

   (iii) $x = {\cdot}000023$, $\hat{x} = {\cdot}00002$

3. Find round-off error and truncation error of the following numbers using 3 decimal point and 5 decimal point floating point representations.

(i) 47·318295, (ii) ·0222718, (iii) 300537·12059

4. Find a way to evaluate following functions to avoid loss of significance.

(i) $f(x) = ln\ (x + 1) - ln(x)$ for large x

(ii) $f(x) = \sqrt{\frac{1+\cos x}{2}}$ for $x = \pi$

(iii) $f(x) = x - \sqrt{x^2 - a}$ for $x >> a$

(iv) $f(x) = \sin x - x$. for x near 0

5. Find the condition number of

(i) $f(x) = \sqrt{x^2+1} - x$

(ii) $f(x) = e^x$

(iii) $f(x) = \frac{x}{1-x^2}$

(iv) $f(x) = x \sin x$, and comment on each case.

6. Use appropriate formula for computing roots of the following quadratic equations.

(i) $x^2 - 100{\cdot}001x + 1 = 0$

(ii) $x^2 - 10000{\cdot}0001x + 1 = 0$

7. Use Taylor series approximations to avoid loss of significance in the following computations.

(i) $f(x) = \frac{e^x - e^{-x}}{x}$

(ii) $f(x) = \frac{\log(1-x) + xe^{x/2}}{2x^3}$

Unit 2 □ Solving System of Linear Algebraic Equations in n Unknowns

A system of linear algebraic equations occurs in many areas of science of technology. Finding solution of a large system of such equations is a very complicated task. Several direct and iterative methods are available to solve a system of linear equations. Some of them are discussed in this unit.

§ Objectives

After going through this unit you will be able to learn about—

- LU decomposition method
- Gaussian elimination method
- Pivoting and scaling
- Least squares method to solve an over-determined linear system.
- Solution of tri-diagonal system of equations

Let us consider a system of n linear equations in n unknowns :

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

where $x_1, x_2, \ldots, x_n$ are n unknowns and a_{ij}, $i, j = 1(1)n$, b_i, $i = 1(1)n$ are known real constants. In vector-matrix notation, we can write (1) as :

$$AX = b, \qquad \ldots\ldots (2)$$

where $A = (a_{ij})$, $i, j = 1(1)n$, $X = (x_1, x_2, \ldots x_n)^T$, $b = (b_1, b_2 \ldots, b_n)^T$ (T indicates transpose).

The matrix A is called the co-efficient matrix and is assumed to be non-singular so that (2) admits a unique solution vector $X = (x_1, x_2, \ldots, x_n)^T$. The matrix A may combined with b to form the augmented matrix $[A, b] = \begin{pmatrix} a_{11}a_{12}\ldots a_{1n}b_1 \\ a_{21}a_{22}\ldots a_{2n}b_2 \\ \ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots \\ a_{n1}a_{n2}\ldots a_{nn}b_n \end{pmatrix}$

It is convenient to work directly with the augmented matrix when using eliminations.

There are two classes of methods for solving system of n linear equations in n unknowns possessing a unique solution. Direct methods find the solution in a finite number of steps by reducing or transforming the given problem to an equivalent one which can be more readily solved. Indirect or iterative methods starts with an arbitrary first approximation to the solution and then try to improve the estimate in an infinite but convergent sequence of steps. Direct methods have some other advantages over solving a system of linear equations, like finding values of determinates, matrix inversions, etc. We now briefly describe some of the direct methods for solving a system of linear equations of the type (1).

§ Gauss Elimination Method :

The first step eliminates the variable x_1 from the 2nd, 3rd, ..., n-th equations of (1). This is achieved by subtracting multiples $m_{i1} = \frac{a_{i1}}{a_{11}}$ of row 1 for $i = 2(1)n$, where, we have assumed $a_{11} \neq 0$ and call it as the first pivotal element.[We shall discuss later to find pivotal elements].

[Note : For notaional convenience elements of initial (or first) augmented matrix will be marked with superscript (1) and the elements of j-th (reduced) augumented matrix will be marked with superscript (j), $j = 2(1)n$. For example, we write $a_{ij}^{(1)}$ for a_{ij}, $j = 1(1)n$ and $b_i^{(1)}$ for b_i, $i = 1(1)n$,]

Thus we find all elements of the augumented matrix $[A, b]$ as follows :

writting, $a_{ij}^{(1)} = a_{ij}$, $i, j = 1(1)n$

$$b_i^{(1)} = b_i,\ i = 1(1)n$$

$$m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}},\ i = 2(1)n$$

we define,

$$a_{ij}^{(2)} = a_{ij}^{(1)} - m_{i1}a_{1j}^{(1)},\ i = 2(1)n,\ j = 1(1)n$$

$$b_i^{(2)} = b_i^{(1)} - m_{i1}b_1^{(1)},\ i = 2(1)n$$

Then, this process leaves first row of $[A, b]$ unchanged but reduces all elements of the first column below $a_{11}^{(1)}$ to zero.

Therefore, under the above operation, $AX = b$ changes to the equivalent system (An equivalent system is one which has the same solution as the original one) which looks like :

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & & a_{2n}^{(2)} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ 0 & a_{n2}^{(2)} & & a_{nn}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \cdots \\ \cdots \\ b_n^{(2)} \end{bmatrix} \quad \ldots\ldots (3)$$

Call it $A^{(2)}X = b^{(2)}$ denoting equation (2) as $A^{(1)}X = b^{(1)}$.

Next or second step eliminates x_2 from 3rd, 4th, ..., n-th equations by subtracting multiples $m_{i2} = \dfrac{a_{i2}^{(2)}}{a_{22}^{(2)}}$ of row 2 for $i = 3(1)n$, where we have assumed $a_{22}^{(2)} \neq 0$ and denote it as second pivotal element.

i.e., setting, $m_{i2} = \dfrac{a_{i2}^{(2)}}{a_{22}^{(2)}}$, $i = 3(1)n$

we define, $a_{ij}^{(3)} = a_{ij}^{(2)} - m_{i2} \cdot a_{2j}^{(2)}$, $i = 3(1)n$, $j = 2(1)n$

$$b_i^{(3)} = b_i^{(2)} - m_{i2} b_2^{(2)}, \; i = 3(1)n$$

which leaves first two rows and first column of system (3) unchanged and reduces all elements of second column below $a_{22}^{(2)}$ to zero.

Then we have an equivalent system of equations (say, $A^{(3)} X = b^{(3)}$) of the original one and it looks like :

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & a_{n3}^{(3)} & \cdots & a_{nn}^{(3)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ b_3^{(3)} \\ \cdots \\ \cdots \\ b_n^{(3)} \end{bmatrix} \quad \ldots\ldots (4)$$

We continue the process. A general step k is as follows. Step k eliminates x_k from $(k+1)$th, ..., n-th equations by subtracting multiples $m_{ik} = \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$ of k-th row for $i = k + 1(1)n$ where again we have assumed $a_{kk}^{(k)} \neq 0$ and denote it as k-th pivotal element.

i.e. setting $\quad m_{ik} = \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}},\ i = k + 1(1)n$

we define $\quad a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik}\, a_{kj}^{(k)},\ i = k + 1(1)n,\ j = k\,(1)n$

$$b_i^{(k+1)} = b_i^{(k)} - m_{ik}\, b_k^{(k)},\ i = k + 1(1)n$$

which leaves all the earlier rows (from first row to k-th row) and coloumns from first to $(k-1)$-th coloumns unchanged but reduces all elements of k-th column below $a_{kk}^{(k)}$ to zero. Therefore we have an equivalent system of equations, say $A^{(k+1)}X = b^{(k+1)}$ which looks like :

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{2n}^{(2)} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & a_{kk}^{(k)} & a_{kk+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ 0 & 0 & \cdots & 0 & 0 & a_{k+1k+1}^{(k+1)} & \cdots & a_{k+1n}^{(k+1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & 0 & a_{nk+1}^{(k+1)} & \cdots & a_{nn}^{(k+1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_k^{(k)} \\ b_{k+1}^{(k+1)} \\ \vdots \\ b_n^{(k+1)} \end{bmatrix}$$

..... (5)

Continuing in this way we ultimately get the equivalent system $A^{(n)}X = b^{(n)}$ which when written explicitly, is the system,

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n-1}^{(1)} & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n-1}^{(2)} & a_{2n}^{(2)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & a_{n-1n-1}^{(n-1)} & a_{n-1n}^{(n-1)} \\ 0 & 0 & \cdots & 0 & a_{nn}^{(n)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ \cdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \cdots \\ \cdots \\ b_{n-1}^{(n-1)} \\ b_n^{(n)} \end{bmatrix} \qquad \text{..... (6)}$$

The solution of this upper traingular system is found by solving the last equation for x_n, then the $(n-1)$ equation for x_{n-1}, and continuing in this way [i.e., by backward substitution] until x_1 is found.

Hence x_i's, $i = n(-1)1$ are calculated by,

$$x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}}$$

$$x_{n-1} = \frac{1}{a_{n-1\,n-1}^{(n-1)}} \left[b_{n-1}^{(n-1)} - a_{n-1\,n}^{(n-1)} x_n \right], \qquad \ldots\ldots (7)$$

.....

$$x_k = \frac{1}{a_{kk}^{(k)}} \left[b_k^{(k)} - \sum_{j=k+1}^{n} a_{kj}^{(k)} x_j \right], \; k = n-1(-1)\,1$$

which completes the process of Gaussian elimination. Now, if we form a unit lower triangular matrix with the identification, $L = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ m_{21} & 1 & 0 & \ldots & 0 \\ m_{31} & m_{32} & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ m_{n1} & m_{n2} & m_{n3} & \ldots & 1 \end{bmatrix}$

and $U = A^{(n)}$, then we have, $A = LU$, i.e., we have also reached a triangular factorization scheme through the process of Gaussian elimination under the assumptions made in the steps. Decomposing A into LU in which L is a unit lower traingular matrix is sometimes called Doolittle's method.

[Note : A triangular matrix with 1's on its diagonal, i.e., unit diagonal elements is called unit lower traingular if the matrix is lower traingular and called unit upper traingular if the matrix is upper traingular.]

Example 1. Solve the following system of equations using Gaussian elimination scheme.

$$2x_1 + 2x_2 + x_3 = 3$$
$$4x_1 + 3x_2 + 2x_3 = 5$$
$$3x_1 + 4x_2 + 2x_3 = 5$$

Solution : The augmented matrix $[A, b] = \begin{bmatrix} 2 & 2 & 1 & 3 \\ 4 & 3 & 2 & 5 \\ 3 & 4 & 2 & 5 \end{bmatrix} = [A^{(1)}, b^{(1)}]$ (say).

Then, $a_{11}^{(1)} = 2 \neq 0$. So, we have,

$$m_{21} = \frac{a_{21}^{(1)}}{a_{11}^{(1)}} = \frac{4}{2} = 2, \; m_{31} = \frac{a_{31}^{(1)}}{a_{11}^{(1)}} = \frac{3}{2} = 1{\cdot}5$$

Therefore, $a_{21}^{(2)} = a_{21}^{(1)} - m_{21}\, a_{11}^{(1)} = 4 - 2.2 = 0,$

$a_{22}^{(2)} = a_{22}^{(1)} - m_{21}\, a_{12}^{(1)} = 3 - 2.2 = -1,$

$a_{23}^{(2)} = a_{23}^{(1)} - m_{21}\, a_{13}^{(1)} = 2 - 2.1 = 0,$

$a_{31}^{(2)} = a_{31}^{(1)} - m_{31}\, a_{11}^{(1)} = 3 - 1{\cdot}5.2 = 0,$

$a_{32}^{(2)} = a_{32}^{(1)} - m_{31}\, a_{12}^{(1)} = 4 - 1{\cdot}5.2 = 1,$

$a_{33}^{(3)} = a_{33}^{(1)} - m_{31}\, a_{13}^{(1)} = 2 - 1{\cdot}5.1 = {\cdot}5,$

$b_2^{(2)} = b_2^{(1)} - m_{21}\, b_1^{(1)} = 5 - 2.3 = -1,$

$b_3^{(2)} = b_3^{(1)} - m_{31}\, b_1^{(1)} = 5 - 1{\cdot}5.3 = {\cdot}5.$

So, after first step we have the equivalent system, $A^{(2)} X = b^{(2)}$ which looks like,

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

Again $a_{22}^{(2)} = -1 \neq 0$; so we have,

$$m_{32} = \frac{a_{32}^{(2)}}{a_{22}^{(2)}} = \frac{1}{-1} = -1,$$

$$a_{32}^{(3)} = a_{32}^{(2)} - m_{32} a_{22}^{(2)} = 1 - (-1){\cdot}(-1) = 0$$

$$a_{33}^{(3)} = a_{33}^{(2)} - m_{32} a_{23}^{(2)} = {\cdot}5 - (-1){\cdot}0 = {\cdot}5,$$

and $b_3^{(3)} = b_3^{(2)} - m_{32} b_2^{(2)} = {\cdot}5 - (-1){\cdot}(-1) = {\cdot}5.$

Hence, the equivalent system $A^{(3)}X = b^{(3)}$ is :

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

Therefore, by backward substitution, we have the solution,

$$x_3 = \frac{-\cdot 5}{\cdot 5} = -1,$$

$$x_2 = \frac{1}{-1} [-1 - 0.(-1)] = 1,$$

and $$x_1 = \frac{1}{2} [3 - 2.1 - 1.(-1)] = 1.$$

And the corresponding decomposition of $A = LU$ is :

$$\begin{bmatrix} 2 & 2 & 1 \\ 4 & 3 & 2 \\ 3 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1\cdot 5 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & \cdot 5 \end{bmatrix}$$

Here one can check that the value of the determinant of the matrix A is equal to $a_{11}^{(1)} \cdot a_{22}^{(2)} \cdot a_{33}^{(3)} = 2.(-1).(\cdot 5) = -1$.

Example 2. Show that Gaussian elimination scheme may be used to find inverse of a matrix.

Solution : Consider the matrix $A = \begin{bmatrix} 2 & 2 & 1 \\ 4 & 3 & 2 \\ 3 & 4 & 2 \end{bmatrix}$ used in the above example and

form the augumented matrix $[A, I] = \left[\begin{array}{ccc:ccc} 2 & 2 & 1 & 1 & 0 & 0 \\ 4 & 3 & 2 & 0 & 1 & 0 \\ 3 & 4 & 2 & 0 & 0 & 1 \end{array}\right]$

Now we apply Gaussian elimination scheme for the matrix A and instead of taking a column matrix b, we consider here the identify matrix I.

From above example we have after step-1 $A \equiv A^{(1)}$ changes to $A^{(2)} = \begin{bmatrix} 2 & 2 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & \cdot 5 \end{bmatrix}$ and correspondingly I changes to

$$I' = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1{\cdot}5 & 0 & 1 \end{bmatrix}$$ [i.e., apply operations of step-1 for each column of I as in the above example we have applied for $b = (3\ 5\ 5)^T$]

Again, in step-2 we have $A^{(3)} = \begin{bmatrix} 2 & 2 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & {\cdot}5 \end{bmatrix}$ and correspondingly

I' changes to $I'' = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3{\cdot}5 & 1 & 1 \end{bmatrix}$.

Now we find solutions for each column of I'' by backward substitution. They are as follows :

For column 1, the solutions are, (we use the notation x_{11}, x_{21}, x_{31} for column 1, x_{12}, x_{22}, x_{32} for column 2, etc., as solutions)

$$x_{31} = \frac{-3{\cdot}5}{{\cdot}5} = -7$$

$$x_{21} = \frac{1}{-1} = [-2 - 0.(-7)] = 2$$

$$x_{11} = \frac{1}{2} = [1 - 2.2 - 1.(-7)] = 2$$

For column 2,

$$x_{32} = \frac{-1}{{\cdot}5} = 2$$

$$x_{22} = \frac{1}{-1} = [1 - 0.2] = -1$$

$$x_{12} = \frac{1}{2} = [0 - 2.(-1) - 1.(2)] = 0$$

For column 3, $x_{33} = \frac{1}{{\cdot}5} = 2$

$$x_{23} = \frac{1}{-1} = [0 - 0.2] = 0$$

$$x_{13} = \frac{1}{2} = [0 - 2.0 - 1.2] = -1$$

Now we construct the matrix $(x_{ij}) = \begin{bmatrix} 2 & 0 & 1 \\ 2 & -1 & 0 \\ -7 & 2 & 2 \end{bmatrix}$

It is easy to check, $\begin{bmatrix} 2 & 0 & -1 \\ 2 & -1 & 0 \\ -7 & 2 & 2 \end{bmatrix} \begin{bmatrix} 2 & 2 & 1 \\ 4 & 3 & 2 \\ 3 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Hence the inverse of $A = A^{-1} = (x_{ij}) = \begin{bmatrix} 2 & 0 & -1 \\ 2 & -1 & 0 \\ -7 & 2 & 2 \end{bmatrix}$

§ Crout's Reduction or Simply *LU* Factorizaion :

It is a modification of the elimination method in which the coefficient matrix A is transformed into LU, where L is lower triangular matrix and U is a unit upper traingular matrix. Here the basic scheme is as follows :

(i) First decompose the coefficient matrix A of the system of equations (1) as $A = LU$, where L is a lower triangular matrix and U is a unit upper triangular matrix. We assume that this decomposition is possible. This is possible if the leading sub matrices of the matrix A are non-singular, i.e., if $a_{11} \neq 0$, $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \neq 0$, ..., $|A| \neq 0$.

(ii) Then find the solution X as follows :

$AX = b \Rightarrow LUX = b$.

Let $UX = C$. Then $LC = b$, which gives C using forward substitution. Backward substitution is then used to calculate X from $UX = C$ as in the Gaussian elimination.

The explicit details of the scheme are as follows :

Step 1 : Let us first assume that $a_{11} \neq 0$, call it first pivotal element and denote it by l_{11}, i.e., $l_{11} = a_{11}$.

Then we divide each element of the first row of the augmented matrix $[A, b]$ by l_{11}.

Set. $u_{11} = 1$, $u_{12} = \dfrac{a_{12}}{a_{11}}$, $u_{13} = \dfrac{a_{13}}{a_{11}}$, ..., $u_{1n} = \dfrac{a_{1n}}{a_{11}}$ and $b'_1 = \dfrac{b_1}{a_{11}}$,

and $l_{21} = a_{21}$, $l_{31} = a_{31}$, ..., $l_{n1} = a_{n1}$.

Step 2 : Assume $l_{22} \neq 0$, where $l_{22} = a_{22} - l_{21} \cdot u_{12}$, call it 2nd pivotal element, then set $l_{32} = a_{32} - l_{31}.u_{12}$, ..., $l_{n2} = a_{n2} - l_{n1}.u_{12}$,

And $u_{22} = 1$, $u_{23} = \dfrac{1}{l_{22}} \{a_{23} - l_{21}.u_{13}\}$, ..., $u_{2n} = \dfrac{1}{l_{22}} \{a_{2n} - l_{21}.u_{1n}\}$,

$$b'_2 = \frac{1}{l_{22}} \{b_2 - l_{21}.b'_1\}.$$

We continue the process as above. The step k is as follows :

Step k : Assume $l_{kk} \neq 0$ where,

$l_{kk} = a_{kk} - [l_{k1}\, l_{k2} \ldots l_{kk-1}]\, [u_{1k} u_{2k} \ldots u_{k-1k}]^T$, call it k-th pivotal element. Then set,

$l_{k+1k} = a_{k+1k} - [l_{k+11}\, l_{k+12} \ldots l_{k+1\,k-1}]\, [u_{1k}\, u_{2k} \ldots u_{k-1k}]^T$,

.....

.....

$l_{nk} = a_{nk} - [l_{n1} l_{n2} \ldots l_{nk-1}]\, [u_{1k}\, u_{2k} \ldots u_{k-1k}]^T$,

and $u_{kk} = 1$,

$u_{kk+1} = \dfrac{1}{l_{kk}} \{a_{kk+1} - [l_{k1}\, l_{k2} \ldots l_{kk-1}]\, [u_{1k+1}\, u_{2k+1} \ldots u_{k-1\,k+1}]^T\}$

.....

.....

$u_{kn} = \dfrac{1}{l_{kk}} \{a_{kn} - [l_{k1} l_{k2} \ldots l_{kk-1}]\, [u_{1n} u_{2n} \ldots u_{k-1n}]^T\}$

$$b'_k = \frac{1}{l_{kk}} \{b_k - [l_{k1}\, l_{k2} \ldots l_{kk-1}]\, [b'_1 b'_2 \ldots b'_{k-1}]^T\}.$$

Proceeding as above we reach step n and find

$A = LU$ where L and U are given by,

$$L = \begin{bmatrix} l_{11} & 0 & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ l_{n1} & l_{n2} & l_{n3} & \dots & \dots & l_{nn} \end{bmatrix} \text{ and } U = \begin{bmatrix} 1 & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & 1 & u_{23} & \dots & u_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

and find $b' = [b'_1\ b'_2 \dots b'_n]^T$. It transforms the system $AX = b$ into equivalent system $UX = b'$.

Now we find the solution X by backward substraction,

i.e.. $x_n = b'_n$

.....

$$x_k = b'_k - \sum_{j=k+1}^{n} u_{kj}\ x_j,\ k = n - 1\ (-1)1.$$

Example 3. Find solution of the system of equations in example 1 by Crout's reduction.

Solution : We have, $A = \begin{bmatrix} 2 & 2 & 1 \\ 4 & 3 & 2 \\ 3 & 4 & 2 \end{bmatrix}$, $b = [3\ 5\ 5]^T$.

Step 1 : Here $a_{11} = 2 \neq 0$. Therefore, first pivotal element, $l_{11} = 2$.

Then, we have, $u_{11} = 1$, $u_{12} = \frac{a_{12}}{a_{11}} = \frac{2}{2} = 1$, $u_{13} = \frac{a_{13}}{a_{11}} = \frac{1}{2} = \cdot 5$, $b'_1 = \frac{b_1}{a_{11}} = \frac{3}{2}$ $= 1 \cdot 5$, $l_{21} = a_{21} = 4$, $l_{31} = a_{31} = 3$.

Step 2 : $l_{22} = a_{22} - l_{22}\ u_{12} = 3 - 4.1 = -1 \neq 0$, 2nd pivotal element.

So, $l_{32} = a_{32} - l_{31} u_{12} = 4 - 3.1 = 1$,

$$u_{22} = 1,\ u_{23} = \frac{1}{l_{22}}\ \{a_{23} - l_{21} u_{13}\}$$

$$= \frac{1}{-1}\ \{2 - 4 \times \cdot 5\} = 0$$

$$b'_2 = \frac{1}{l_{22}}\ (b_2 - l_{21}\ b'_1) = \frac{1}{-1}\{5 - 4.1 \cdot 5\} = 1.$$

Step 3 : $l_{33} = a_{33} - [l_{31}\ l_{32}]\ [u_{13} u_{23}]^T$

$= 2 - [3\ 1]\ [\cdot 5\ 0]^T = \cdot 5 \neq 0$, 3rd pivotal element. As the order of the matrix is 3, we are in the last step. Set $u_{33} = 1$, b'_3

$$= \frac{1}{l_{33}} \{b_3 - [l_{31}\ l_{32}]\ [b'_1\ b'_2]^T\}$$

$$= \frac{1}{\cdot 5} \{5 - [3\ 1]\ [1\cdot 5\ 1]^T\}$$

$$= -1$$

Hence, $A = LU$ where $L = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 4 & -1 & 0 \\ 3 & 1 & \cdot 5 \end{bmatrix}$

and $U = \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdot 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. $b' = [b'_1\ b'_2\ b'_3]^T = [1\cdot 5\ 1\ -1]^T$

Therefore the solution X is,

$x_3 = b'_3 = -1.$

$x_2 = b'_2 - u_{23}x_3$

$= 1 - 0.(-1) = 1$

$x_1 = b'_1 - \sum_{j=2}^{3} u_{1j}x_j$

$= 1\cdot 5 - u_{12}x_2 - u_{13}x_3$

$= 1\cdot 5 - 1.1 - \cdot 5.(-1)$

$= 1$

i.e., $X = [x_1 x_2 x_3]^T = [1\ 1\ -1]^T$.

§ General *LU*-Factorization Scheme :

The basic methodology is to decompose the matrix A into product of two parts L and U where L is a lower traingular matrix and U is a upper triangular matrix.

$$\text{Let, } A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & \cdots & \cdots & l_{nn} \end{bmatrix}$$

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & \cdots & u_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & u_{nn} \end{bmatrix}$$

Then the basic steps are as follows :

Step 1 : Set $l_{ij} = 0$, for $j > i$

and $u_{ij} = 0$, for $j < i$, $i, j = 1(1)n$.

Step 2 : For $k = 1$, find elements of column 1 of L by $l_{i1}u_{11} = a_{i1}$, $i = 1(1)n$, and elements of row 1 of u by, $l_{11}u_{1j} = a_{1j}$, $j = 1(1)n$.

Step 3 : For $2 \le k \le n$, find

elements of column k of L by,

$$(l_{i1}u_{1k} + \ldots + l_{ik-1}u_{k-1k}) + l_{ik}u_{kk} = a_{ik},\ i = k(1)n,$$

and elements of row k of U by,

$$(l_{k1}u_{1j} + \ldots + l_{kk-1}u_{k-1j}) + l_{kk}u_{kj} = a_{kj},\ j = k(1)n.$$

In short, $LU = A \Rightarrow$

$$a_{ij} = \sum_{k=1}^{n} l_{ik}u_{kj}$$

$$= \sum_{k=1}^{j} l_{ik}u_{kj},\ j \le i = 1(1)n,$$

and $$a_{ij} = \sum_{k=1}^{i} l_{ik}u_{kj},\ i < j = 2(1)n.$$

Clearly, if we set $u_{ii} = 1$, $i = 1(1)n$, then we have Crout's decomposition method. In this case, step 3 reduces to :

For elements of column k of L, $l_{ik} = a_{ik} - \sum_{j=1}^{k-1} l_{ij}u_{jk}$, $i = k(1)n$, and for elements of row k of U, $u_{kj} = \frac{1}{l_{kk}}\left[a_{kj} - \sum_{i=1}^{k-1} l_{ki}u_{ij}\right]$, $j = k(1)n$.

If we set $l_{ii} = 1$, $i = 1(1)n$ in the general scheme, we have the Doolittle's method for decomposing A and if we set $l_{ii} = u_{ii}$, $i = 1(1)n$, then we have Cholesky's decomposition. In Cholesky's decomposition, the diagonal elements of L and U are given by $l_{kk} = u_{kk} = \sqrt{a_{kk} - \sum_{i=1}^{k-1} l_{ki}u_{ik}}$, $k = 1(1)n$, and for other elements of L and U we have,

$$l_{ik} = u_{ki} = \frac{1}{l_{kk}}\left[a_{ik} - \sum_{j=1}^{k-1} l_{ij}u_{jk}\right], \; i > k = 1(1)n - 1.$$

Cholesky's method generally is used in the case when A is symmetric. Also for positive definite symmetric matrix, it is really an useful technique. It actually decompose A as : $A = LL^T$, where L is a lower triangular matrix.

From all the methods described above, as off shoot we find some other results of interest, such as, determinant of the matrix A, inverse of A, etc. (Actually we have assumed implicitly in all cases A is a non-singular matrix).

Result 1. $\det(A) = |A| = |L|.|U|$

$= a_{11}^{(1)}.a_{22}^{(2)} \ldots a_{nn}^{(n)}$ in Gaussian elimination, (Here $|L| = 1$)

$= l_{11}\, l_{22} \ldots l_{nn}$, in Crout's decomposition, (Here $|U| = 1$)

$= u_{11}\, u_{22} \ldots u_{nn}$, in Doolittle's decomposition, (Here $|L| = 1$)

Result 2. $A = LU \Rightarrow A^{-1} = U^{-1}L^{-1}$.

As U and L are respectively upper and lower traingular matrices, therefore U^{-1} and L^{-1} are aslo upper and lower traingular matrices.

$$\text{Let } U^{-1} = \begin{bmatrix} u'_{11} & u'_{12} & \ldots & u'_{1n} \\ 0 & u'_{22} & \ldots & u'_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & u'_{nn} \end{bmatrix} \quad L^{-1} = \begin{bmatrix} l'_{11} & 0 & \ldots & 0 \\ l'_{21} & l'_{22} & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ l'_{n1} & l'_{n2} & 0 & l'_{nn} \end{bmatrix}$$

Then by solving system of equations, with the identifications $LL^{-1} = I$, $UU^{-1} = I$, we find L^{-1} and U^{-1} by forward and backward substitutions respectively and then $A^{-1} = U^{-1} L^{-1}$.

Operations count for Gaussian Algorithm :

(i) Total number of multiplications and divisions for LU decomposition

$$= \frac{n(n^2-1)}{3} \approx \frac{n^3}{3}.$$

Total number of additions and substractions for LU decomposition

$$= \frac{n(n-1)(2n-1)}{6} \approx \frac{n^3}{3}.$$

(ii) Modification of b to b' :

Total number of multiplications and divisions

$$= (n-1) + (n-2) + \ldots 1 = \frac{n(n-1)}{2}.$$

Total number of additions and substractions $= \frac{n(n-1)}{2}$.

(iii) Solution $UX = b'$.

No. of multiplications and divisions $= \frac{n(n+1)}{2}$.

No. of additions and substractions $= \frac{n(n-1)}{2}$.

Hence for solutions of the system $AX = b$,

Total no. of multiplications and divisions $= \frac{n^3}{3} + n^2 - \frac{n}{3} \approx \frac{1}{3} n^3$

and total no. of additions and substractions $= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3} n^3$.

Therefore, for large n, total count $\approx \frac{2}{3} n^3$.

For inversion of A, if we count only multiplication's and divisions, we find it is approximately 4. $\frac{n^3}{3}$ i.e., takes more steps than to find solutions for $AX = b$ by Gaussian algorithm.

Pivoting and Scaling :

In all the processes discussed above, it is assumed that either $a_{kk}^{(k)} \neq 0$, or, $l_{kk} \neq 0$, or, $u_{kk} \neq 0$, as the case may be. These assumptions may be removed if we find at each step a non-zero key or pivotal element by interchanging all the remaining rows. If we fails to find a pivotal element at and intermediate step, then we terminate our process and conclude that the matrix we have considered, is singular. Obviously, if the matrix A of the system of equations $AX = b$, is singular, then there does not exist an unique solution for that system. Again, for a non-singular matrix, to find pivotal elements, it may be necessary to interchange rows. This is actually equivalent with the multiplication of A by a permutation matrix P. Let, $A' = PA$. We decompose, $A' = LU$ and find the solution of the system $AX = b$ by considering the equivalent system $A'X = Pb$.

Now we shall describe the necessity of pivoting and scalling. In all the processes described above, we have used arithmetic operations, which may not be carried out exactly, to complete the processes. So, there are always chances of propagating round-of errors in each stage. To control such propagation of errors, we generally consider the pivoting strategies. There are two types of pivoting : (i) partial pivoting and (ii) complete pivoting.

(i) **Partial pivoting :** Consider Gaussian scheme. At each stage k when $1 \leq k \leq n-1$, we find $c_k = \max\limits_{k \leq i \leq n} \left|a_{ik}^{(k)}\right|$. Let $i_0 \geq k$ be the smallest row index for which c_k is attained. If $i_0 > k$, then we interchange row k with row i_0 in the corresponding augmented matrix. It is then easy to check that all the multipliers satisfy the inequality, $|m_{ik}| \leq 1$, for $i = k + 1(1)n$. It prevents growth of elements in $A^{(k)}$ which ultimately reduces loss of significance.

(ii) **Complete pivoting :** At stage k of Gaussian algorithm, we define,

$$c_k = \max_{k \leq i, j \leq n} \left|a_{ij}^{(k)}\right|.$$

Then we switch rows of $A^{(k)}$ and $b^{(k)}$, and columns of $A^{(k)}$ to bring out the element c_k as pivotal element. In this change the order of unknown variable are interchanged but as far as solution of the system of equations $AX = b$ is concerned, there is no harm. Only at the time of back substitution process we must arrange them in order. This pivoting also slows down the propagation of error in Gaussian scheme.

Scaling : If the elements of the coefficient matrix A vary largely in size, then it is likely that large loss of significance will occur and it will cause worse propagation of round-off errors. To avoid such problems, we usually enter the process of scaling.

Let us consider scaling in the Gaussian scheme.

Let $s_i = \max_{1 \le j \le n} |a_{ij}|$, $i = 1(1)n$,

and $b_{ij} = \frac{a_{ij}}{s_i}$, $i, j = 1(1)n$,

so that, $\max_{1 \le j \le n} |b_{ij}| = 1$, $i = 1(1)n$.

Now we find a matrix $\tilde{A} = (b_{ij})$ and a diagonal matrix $D_1 = (d_{ii}) = (s_i)$. Then we have,

$$D_1 \tilde{A} = A$$

Therefore, $AX = b$.

$$\Rightarrow D_1 \tilde{A} X = b.$$

or, $\tilde{A} X = D_1^{-1} b$, where D_1^{-1} is also diagonal.

It may affect in the choice of pivotal element. Again, at stage k of Gaussian scheme, we further use some implicit scaling. Let us define,

$c_k = \max_{k \le i \le n} \frac{|a_{ik}^{(k)}|}{s_i^{(k)}}$ and then interchange rows correspondingly. It will ultimately cause less propagation of errors.

§ Error Analysis :

Let us denote by $\|\|$ as the norm function. Let us assume that our system of linear equations $AX = b$ is uniquely solvable. Now consider the solution of perturbed system,

$A\tilde{X} = b + r$, where $\tilde{X} = [\tilde{x}_1 \tilde{x}_2 \ldots \tilde{x}_n]^T$ be a column of unknowns and r is called a residual vector.

Set, $e = \tilde{X} - X$.

Then, $Ae = A\tilde{X} - AX = b + r - b = r$

$\Rightarrow e = A^{-1} r$ [As we have assumed A^{-1} exists].

Now taking norm on both sides of $Ae = r$ and $e = A^{-1} r$, we have,

$$\|r\| = \|Ae\| \leq \|A\|.\|e\|$$

and $$\|e\| = \|A^{-1}r\| \leq \|A^{-1}\|.\|r\|$$

Therefore, $$\frac{\|r\|}{\|A\|\|X\|} \leq \frac{\|e\|}{\|X\|} \leq \frac{\|A^{-1}\|\|r\|}{\|X\|}$$

Again, from $AX = b$ and $X = A^{-1}b$, we have,

$$\|b\| = \|AX\| \leq \|A\|\|X\|$$

and $$\|X\| = \|A^{-1}b\| \leq \|A^{-1}\|\|b\|.$$

Therefore, $$\frac{1}{\|A\|\|A^{-1}\|}\frac{\|r\|}{\|b\|} \leq \frac{\|r\|}{\|A\|\|X\|} \leq \frac{\|e\|}{\|X\|} \leq \frac{\|A^{-1}\|\|r\|}{\|X\|}$$

$$\leq \frac{\|A\|.\|A^{-1}\|.\|r\|}{\|b\|}.$$

Let us define the condition number of a matrix A by cond $(A) = \|A\|.\|A^{-1}\|$.

Then from the above inequality, we have,

$$\frac{1}{\text{cond}(A)}.\frac{\|r\|}{\|b\|} \leq \frac{\|e\|}{\|X\|} \leq \text{cond } (A) \cdot \frac{\|r\|}{\|b\|}.$$

or, $$\frac{1}{\text{cond}(A)} \leq \frac{\frac{\|e\|}{\|X\|}}{\frac{\|r\|}{\|b\|}} \leq \text{cond } (A),$$

Again, form the relation $A^{-1} A = I$, we have,

$$1 \leq \|I\| = \|A^{-1}A\| \leq \|A^{-1}\|\|A\| = \text{cond } (A),$$

Hence, if we find cond (A) of a matrix A is near to 1, then small perturbation in b will lead to small perturbation in the solution and if cond (A) is large, it may be sometimes catastrophic. Then we call the matrix is ill-conditioned or the system of linear equations is ill-conditioned as the solution X is unstable with the small perturbation in b.

For example, the Hilbert matrix $H_n = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n-1} \end{bmatrix}$

is an example of ill-conditioned matrix where the cond (H_n) increases as n increases.

Example 4. Find solution of the following system of equations.

$$3x_1 + x_2 + 6x_3 = 2$$
$$2x_1 + x_2 + 3x_3 = 7$$
$$x_1 + x_2 + x_3 = 4$$

Solution : Here the coefficient matrix $A = \begin{bmatrix} 3 & 1 & 6 \\ 2 & 1 & 3 \\ 1 & 1 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 2 \\ 7 \\ 4 \end{bmatrix}$ and $X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$.

Then Gaussian elimination with pivoting yields,

$$\begin{bmatrix} \textcircled{3} & 1 & 6 & 2 \\ 2 & 1 & 3 & 7 \\ 1 & 1 & 1 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{1}{3} & -1 & \frac{17}{3} \\ 0 & \textcircled{\frac{2}{3}} & -1 & \frac{10}{3} \end{bmatrix}, \; m_{21} = 2/3, \; m_{31} = 1/3$$

$$\rightarrow \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{2}{3} & -1 & \frac{10}{3} \\ 0 & 0 & \textcircled{-1/2} & 4 \end{bmatrix}, \; m_{32} = 1/2.$$ The pivotal elements are encircled. In step 2, 2nd and 3rd rows are interchanged for pivoting. Therefore the permutation matrix used is,

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \text{ So, } PA = \begin{bmatrix} 3 & 1 & 6 \\ 1 & 1 & 1 \\ 2 & 1 & 3 \end{bmatrix} \text{ and}$$

$PA = LU$ gives, $L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{2}{3} & \frac{1}{2} & 1 \end{bmatrix}$, $U = \begin{bmatrix} 3 & 1 & 6 \\ 0 & \frac{2}{3} & -1 \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}$

Hence the solution is given by, $x_3 = -8$, $x_2 = -7$, $x_1 = 19$.

Also, $\det(PA) = |PA| = 3.\frac{2}{3}.\left(-\frac{1}{2}\right) = -1.$

$\therefore \det(A) = \frac{-1}{\det(P)} = \frac{-1}{-1} = 1.$

Example 5. Applying Crout's method decompose the following matrix and find inverse, correct upto three decimal places.

$$A = \begin{bmatrix} 3{\cdot}3 & 1{\cdot}2 & 1{\cdot}4 & 0{\cdot}5 \\ 1{\cdot}5 & 4{\cdot}2 & 1{\cdot}3 & 2{\cdot}5 \\ 2{\cdot}2 & 3{\cdot}5 & 5{\cdot}1 & 3{\cdot}2 \\ 3{\cdot}2 & 2{\cdot}5 & 1{\cdot}25 & 7{\cdot}2 \end{bmatrix}$$

Solution : Check $L = \begin{bmatrix} 3{\cdot}3 & 0 & 0 & 0 \\ 1{\cdot}5 & 3{\cdot}654 & 0 & 0 \\ 2{\cdot}2 & 2{\cdot}700 & 3{\cdot}676 & 0 \\ 3{\cdot}2 & 1{\cdot}336 & -{\cdot}350 & 5{\cdot}997 \end{bmatrix}$

and $U = \begin{bmatrix} 1{\cdot}0 & {\cdot}364 & {\cdot}424 & {\cdot}152 \\ 0 & 1{\cdot}000 & {\cdot}182 & {\cdot}622 \\ 0 & 0 & 1{\cdot}000 & {\cdot}323 \\ 0 & 0 & 0 & 1{\cdot}000 \end{bmatrix}$

Therefore, by $A^{-1} = U^{-1} L^{-1}$, we have, $L^{-1} = \begin{bmatrix} {\cdot}303 & 0 & 0 & 0 \\ -{\cdot}124 & {\cdot}274 & 0 & 0 \\ -{\cdot}090 & -{\cdot}201 & {\cdot}272 & 0 \\ -{\cdot}139 & -{\cdot}073 & {\cdot}016 & {\cdot}167 \end{bmatrix}$

$\Rightarrow A^{-1} = \begin{bmatrix} {\cdot}354 & -{\cdot}041 & -{\cdot}094 & {\cdot}032 \\ -{\cdot}030 & {\cdot}351 & -{\cdot}058 & -{\cdot}094 \\ -{\cdot}045 & -{\cdot}177 & {\cdot}267 & -{\cdot}054 \\ -{\cdot}139 & -{\cdot}073 & {\cdot}016 & {\cdot}167 \end{bmatrix}$ correct to 3 decimal places.

Example 6. Consider the following system of equations,

$$10x_1 + 7x_2 + 8x_3 + 7x_4 = 32$$
$$7x_1 + 5x_2 + 6x_3 + 5x_4 = 23$$
$$8x_1 + 6x_2 + 10x_3 + 9x_4 = 33$$
$$7x_1 + 5x_2 + 9x_3 + 10x_4 = 31$$

Putting $x_1 = 6$, $x_2 = -7{\cdot}2$, $x_3 = 2{\cdot}9$ and $x_4 = -0{\cdot}1$, we found left hand sides are equal to 32·1, 22·9, 32·9 and 31·1. From the smallness of residues it appears that the assumed solution is near to the exact solution. However, if we set $x_1 = 1{\cdot}5$ $x_2 = {\cdot}18$, $x_3 = 1{\cdot}19$, $x_4 = {\cdot}89$ we obtain left-hand sides are equal to 32·01, 22·99, 32·99 and 31·01. The exact solution of the equations is $x_1 = x_2 = x_3 = x_4 = 1$. We conclude this system as an ill-conditioned system. Actually, $AX = b$ and $A\tilde{X} = b + r \Rightarrow A(\tilde{X} - X) = r$. If $X \approx \tilde{X}$, then the residue r is small, but if r is small then one cannot always assert that $X \approx \tilde{X}$. This is the situation what we earlier called ill-conditioned systems. Sometimes they are called inherently unstable systems. In this example we have faced that situation.

Example 7. Consider the system,

$$\begin{bmatrix} {\cdot}005 & 1{\cdot}000 \\ 1{\cdot}000 & 1{\cdot}000 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} {\cdot}500 \\ 1{\cdot}000 \end{bmatrix}$$

The exact solution is $x = {\cdot}503$, $y = {\cdot}497$. Now we use Gaussian elimination process. If we take the first pivotal element $a_{11} = {\cdot}005$ we find the equivalent system.

$$\begin{bmatrix} {\cdot}005 & 1{\cdot}000 \\ 0 & -200{\cdot}000 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} {\cdot}500 \\ -99{\cdot}000 \end{bmatrix}$$

So that, $x = 1{\cdot}000$ and $y = {\cdot}495$.

Again if we take the first pivotal element as $a_{21} = 1$, then we find the equivalent system,

$$\begin{bmatrix} 1{\cdot}0 & 1{\cdot}0 \\ 0 & 1{\cdot}0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1{\cdot}0 \\ {\cdot}5 \end{bmatrix}$$

$\Rightarrow y = {\cdot}5$, $x = {\cdot}5$ which is near to the exact solution.

So, we may conclude that pivoting is essential to avoid round-off errors.

§ Least Squares Solutions of Over-determined Linear System :

Given a system of N linear equations $AX = d$ in n unknowns, x_i, $i = 1(1)n < N$, it is impossible, in general, to find a vector $X = [x_1 x_2 \dots x_n]^T$ that will exactly satisfy all the equations. We, therefore first define the residual vector $r(X) = AX - d$ and then define the solution of $AX = d$ to be vector X^* that minimizes the l_2-norm of the resiual, that is, $\| r(X^*) \| = \min_X \| r(X) \| = \min_X \left[\sum_{i=1}^{N} |r_i(X)|^2 \right]^{1/2}$

X^* is then called the least-squares solution of the over-determined system, $AX = d$. We denote $AX = d + r$ as the error equations, ... (1)

where $A = [a_{ij}]_{N \times n}$, $X = [x_1 x_2 \dots x_n]^T$, $d = [d_1 d_2 \dots d_N]^T$ and

$r(X) = r = [r_1 r_2 \dots r_N]^T$.

Let us assume that A has maximal rank, i.e., n. So all the columns of A are linearly independent. We have to determine unknowns x_k, $k = 1(1)n$ by the least-square principle; i.e., sum of squares of residuals r_i is to be minimal. The method has some advantages : (i) It is easy to develop, (ii) statistically, it is a preferable method.

Now, $r^T r = (AX - d)^T (AX - d)$

$= (X^T A^T - d^T)(AX - d)$

$= X^T A^T AX - X^T A^T d - d^T AX + d^T d.$

$= F(X)$, say, a quadratic in n unknowns $x_1, x_2 \dots x_n$. ..... (2)

Let us denote $C = A^T A$ and $b = A^T d$ ..... (3)

Then C is a symmetric $n \times n$ matrix and as A has maximal rank, C is positive definite. It follows directly from the relation.

$Q(X)$, a quadratic $= X^T CX = X^T A^T AX = (AX)^T AX \geq 0$, $\forall X \in R^n$, and $Q(X) = 0 \Leftrightarrow AX = 0 \Leftrightarrow X = 0$.

So our problem reduces to,

Minimize $F(X) = r^T r = X^T CX - X^T b - b^T X + d^T d$ ..... (4)

A necessary condition to minimize $F(X)$ at a vector X is, gradient $\vec{\nabla} F(X)$ must vanish at the point that minimizes $F(X)$. Explicitly, the i-th component of $\vec{\nabla} F(X)$ is,

$$\frac{\partial F(X)}{\partial x_i} = 2\sum_{k=1}^{n} C_{ik}x_k - 2b_i,\ i = 1(1)n.$$

$$\text{or, } 0 = \sum_{k=1}^{n} C_{ik}x_k - b_i,\ i = 1(1)n. \qquad \ldots\ldots (5)$$

i.e., we have a system of linear equations,

$$CX = b \qquad \ldots\ldots (6)$$

for n unknowns $x_1, x_2 \ldots x_n$.

We call (6) as normal equations, corresponding to the error equation (1). As the matrix C is positive definite, there exists unique solution of (6). We find it by Cholesky's decomposition method on C, of the form LL^T where L stands for a lower triangular matrix.

The whole scheme is as follows :

(i) Find $C = A^TA$ and $b = A^Td$.

i.e., $c_{ij} = a_i^Ta_j$ and $b_i = a_i^Td$, $i, j = 1(1)n$,

where a_i is a column vector of matrix A.

(ii) Form the normal equations $CX = b$.

(iii) Then find Cholesky's decomposition of C as $C = LL^T$.

(iv) Consider $LY - b = 0$ and $L^TX - Y = 0$, and find solutions using forward and backward substitution.

(v) Lastly, compute residuals $r = AX - d$.

The method of finding normal equations may suffer when the condition number of the matrix C is quite large. To avoid this, some safer method or numerical procedure such as QR-decomposition, singular value decomposition, etc., may be used.

Example 8. Fit a quadratic function in t by the method of least squares for the following data,

t	0·04	0·32	0·51	0·73	1·03	1·42	1·60
x	2·63	1·18	1·16	1·54	2·65	5·41	7·67

Solution : Let $z(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2$ where $\alpha_0, \alpha_1, \alpha_2$ are arbitrary constants.

Then the i-th error equations is, $\alpha_0 + \alpha_1 t_i + \alpha_2 t_i^2 - z_i = r_i$, $i = 1(1)7$.

Therefore the system of error equations without residuals are :

α_0	$\alpha_1(t_i)$	$\alpha_2(t_i^2)$	$1(t_i)$
1	·04	·0016	2·63
1	·32	·1024	1·18
1	·51	·2601	1·16
1	·73	·5329	1·54
1	1·03	1·0609	2·65
1	1·42	2·0164	5·41
1	1·60	2·5600	7·67

First three columns then corresponds to the matrix A and fourth column corresponding to the vector d.

The normal equations are given by (upto six significant figures)

α_0	α_1	α_2	1
7·00000	5·65000	6·53430	22·2400
5·65000	6·53430	8·60652	24·8823
6·53430	8·60652	12·1071	34·6027

Cholesky's decomposition with forward and backward substitution yields,

$$L = \begin{bmatrix} 2·64575 & 0·00000 & 0·00000 \\ 2·13550 & 1·40497 & 0·00000 \\ 2·46973 & 2·37187 & ·617867 \end{bmatrix} \quad \alpha = \begin{bmatrix} 2·74928 \\ -5·95501 \\ 5·60745 \end{bmatrix}$$

which gives, $z(t) = 2·74928 - 5·95501t + 5·60745t^2$ with residual vector,

$r = [-·1099\ ·2379\ ·0107 - ·1497 - ·0854\ ·1901 - ·0936]^T$ correct to 4 decimal places.

§ Solution of Tri-diagonal System of Equations

If the system of equations is of the form

$$
\begin{aligned}
b_1x_1 + c_1x_2 &= d_1 \\
a_2x_1 + b_2x_2 + c_2x_3 &= d_2 \\
a_3x_2 + b_3x_3 + c_3x_4 &= d_3 \\
&\cdots\cdots\cdots\cdots \\
a_nx_{n-1} + b_nx_n &= d_n
\end{aligned}
\qquad \ldots\ldots (1)
$$

then the coefficient matrix is

$$
A = \begin{bmatrix}
b_1 & c_1 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 \\
a_2 & b_2 & c_2 & 0 & \ldots & 0 & 0 & 0 & 0 \\
0 & a_3 & b_3 & c_3 & \ldots & 0 & 0 & 0 & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & 0 & \ldots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\
0 & 0 & 0 & 0 & \ldots & 0 & 0 & a_n & b_n
\end{bmatrix}
\quad \text{and } d = \begin{bmatrix} d_1 \\ d_2 \\ \ldots \\ d_n \end{bmatrix}
\qquad \ldots\ldots (2)
$$

It may be noted that the main diagonal and the adjacent coefficients on either side of it consist of only non-zero elements and all other elements are zero. The matrix is called tri-diagonal matrix and the system of equations is called a **tri-diagonal system.** These type of matrices occur frequently in the solution of ordinary and partial differential equations by difference method.

A tri-diagonal system can be solved using **LU** decomposition method.

Let **A = LU** where

$$
L = \begin{bmatrix}
\gamma_1 & 0 & 0 & \ldots & 0 & 0 & 0 \\
\beta_2 & \gamma_2 & 0 & \ldots & 0 & 0 & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & \ldots & \beta_{n-1} & \gamma_{n-1} & 0 \\
0 & 0 & 0 & \ldots & 0 & \beta_n & \gamma_n
\end{bmatrix}
$$

$$
\text{and } U = \begin{bmatrix}
1 & \alpha_1 & 0 & \ldots & 0 & 0 & 0 \\
0 & 1 & \alpha_2 & \ldots & 0 & 0 & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & \ldots & 0 & 1 & \alpha_{n-1} \\
0 & 0 & 0 & \ldots & 0 & 0 & 1
\end{bmatrix}
$$

Then

$$LU = \begin{bmatrix} \gamma_1 & \gamma_1\alpha_1 & 0 & \dots & 0 & 0 & 0 \\ \beta_2 & \alpha_1\beta_2+\gamma_2 & \alpha_2\gamma_2 & \dots & 0 & 0 & 0 \\ 0 & \beta_3 & \alpha_2\beta_3+\gamma_3 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 & \beta_n & \beta_n\alpha_{n-1}+\gamma_n \end{bmatrix}$$

Now comparing matrix LU with A and obtain the non-zero elements of L and U as

$$\gamma_1 = b_1, \ \gamma_i\alpha_i = c_i \ \text{ or, } \ \alpha_i = \frac{c_i}{\gamma_i}, \ i = 1(1)n-1$$

$$\beta_i = a_i, \ i = 2(1)n$$

$$\gamma_i = b_i - \alpha_{i-1}\beta_i = b_i - a_i\frac{c_{i-1}}{\gamma_{i-1}}, \ i = 2(1)n$$

Thus the elements of L and U are given by the following relations.

$$\gamma_1 = b_1$$

$$\gamma_i = b_i - \frac{a_i c_{i-1}}{\gamma_{i-1}}, \ i = 2(1)n \qquad \dots(3)$$

$$\beta_i = a_i, \ i = 2(1)n \qquad \dots (4)$$

$$\alpha_i = \frac{c_i}{\gamma_i}, \ i = 1(1)n-1$$

The solution of the equation (1) i.e., $Ax = d$ where $d = (d_1, d_2, \dots, d_n)'$ can be obtained by solving $Lz = d$ using forward substitution and then solving $Ux = z$ using back substitutions. The solution of $Lz = d$ is given by

$$z_1 = \frac{d_1}{b_1}, \ z_i = \frac{d_i - a_i z_{i-1}}{\gamma_i}, \ i = 2(1)n \qquad \dots (5)$$

The solution of the equation $Ux = z$ is

$$x_n = z_n, \ x_i = z_i - \alpha_i x_{i+1} = z_i - \frac{c_i}{\gamma_i}x_{i+1}, \ i = (n-1)(-1)1. \qquad \dots(6)$$

Example. Solve the following tri-diagonal system of equations

$$x_1 + x_2 = 3. \quad -x_1 + 2x_2 + x_3 = 6. \quad 3x_2 + 2x_3 = 12.$$

Solution. Here $b_1 = c_1 = 1$, $a_2 = -1$, $b_2 = 2$, $c_2 = 1$, $a_3 = 3$, $b_3 = 2$, $d_1 = 3$, $d_2 = 6$, $d_3 = 12$.

Therefore,

$$\gamma_1 = b_1 = 1$$

$$\gamma_2 = b_2 - a_2 \frac{c_1}{\gamma_1} = 2 - (-1).1 = 3$$

$$\gamma_3 = b_3 - a_3 \frac{c_2}{\gamma_2} = 2 - 3\frac{1}{3} = 1$$

$$z_1 = \frac{d_1}{b_1} = 3, \quad z_2 = \frac{d_2 - a_2 z_1}{\gamma_2} = 3, \quad z_3 = \frac{d_3 - a_3 z_2}{\gamma_3} = 3$$

$$x_3 = z_3 = 3, \quad x_2 = z_2 - \frac{c_2}{\gamma_2} x_3 = 2, \quad x_1 = z_1 - \frac{c_1}{\gamma_1} x_2 = 1$$

Hence the required solution is $x_1 = 1$, $x_2 = 2$, $x_3 = 3$.

§ Summary :

The Gauss-elimination method is described to solve a system of linear equations. It is shown that the number of arithmetic operations used in this method is approximately $\frac{2}{3}n^3$, where n is the number of unknowns. Also it is explained and illustrated that this method is useful to find out the inverse of a square non-singular matrix. Due to the rounding/truncation error, this method fails for some matrices. The concept of pivoting is incorporated with the Gauss-elimination method, and this modified method is applicable to determined the inverse of any matrix.

Another method known as LU decomposition is described to solve a system of linear algebraic equations. It may sometimes happen that the exact solution of a system does not exist. In this cases, one can determine the approximate solution with least error. Such solution can be obtained by least square method, which is discussed in this unit.

An efficient method to solve a linear tri-diagonal system of equations is also discussed along with an example.

EXERCISES

(1) Solve by Gaussian elimination method the following systems. Check also $A = LU$ and find det (A), where A represents the coefficient matrix.

(i) $2x_1 + x_2 + 4x_3 = 12$

$8x_1 - 3x_2 + 2x_3 = 20$

$4x_1 + 11x_2 - x_3 = 33$ [Ans. : $x_1 = 3, x_2 = 2, x_3 = 1$]

(ii) $2x_1 + 3x_2 - 5x_3 + 10 = 0$

$4x_1 + 8x_2 - 3x_3 + 19 = 0$

$-6x_1 + x_2 + 4x_3 + 11 = 0$ [Ans. : $x_1 = 2, x_2 = -3, x_3 = 1$]

(iii) $x + y + z = 6$

$2x + 3y + z = 1$

$x - y + z = 3$ [Ans. : $x = -8, y = 1{\cdot}5, z = 12{\cdot}5$]

(2) Show by any method of LU decomposition A^{-1} is the inverse of the matrix A.

(i) $A = \begin{bmatrix} -3 & 5 & -4 \\ 2 & -6 & 12 \\ 1 & -2 & 2 \end{bmatrix}$ $\quad A^{-1} = \begin{bmatrix} -3 & 0{\cdot}5 & -9 \\ -2 & 0{\cdot}5 & -7 \\ -0{\cdot}5 & 0{\cdot}25 & -2 \end{bmatrix}$

(ii) $A = \begin{bmatrix} -1 & 1 & 4 \\ -1 & 0 & 1 \\ 1 & -1 & -5 \end{bmatrix}$ $\quad A^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ -4 & -9 & -5 \\ 1 & 2 & 1 \end{bmatrix}$

(iii) $A = \begin{bmatrix} 2 & 3 & 3 \\ 1 & 2 & 3 \\ 2 & 4 & 5 \end{bmatrix}$ $\quad A^{-1} = \begin{bmatrix} 2 & 3 & -3 \\ -1 & -4 & 3 \\ 0 & 2 & -1 \end{bmatrix}$

(3) Using pivoting solve the following system of equations.

(i) $x + y + z = 6$

$2x - y + z = 3$

$3x + 2y - z = 4$

(i) $x_1 + x_2 + 2x_3 = -1$

$2x_1 - x_2 + 2x_3 = -4$

$4x_1 + x_2 + 4x_3 = -2$

(4) Find the triangular factorization $A = LU$ for the following matrices with partial pivoting.

(i) $A = \begin{bmatrix} 1 & 1 & 2 \\ 2 & -1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$ (ii) $A = \begin{bmatrix} 1 & 2 & 6 \\ 4 & 8 & -1 \\ -2 & 3 & 5 \end{bmatrix}$

(5) Use Court's decomposition to solve the following system.

(i) $2x_1 - 6x_2 + 8x_3 = 0$

$5x_1 + 4x_2 - 3x_3 = 5{\cdot}5$

$3x_1 + x_2 + 2x_3 = 5$

(ii) $x_1 + x_2 = 5$

$2x_1 - x_2 + 5x_3 = -9$

$3x_2 - 4x_3 + 2x_4 = 19$

$2x_3 + 6x_4 = 2$

(6) Find the determinant of the following matrix by (i) Gaussian elimination process, (ii) Crout's decomposition method, correct to three significant figures.

$$\begin{bmatrix} \cdot 596 & \cdot 497 & \cdot 263 \\ 4{\cdot}07 & 3{\cdot}21 & 1{\cdot}39 \\ \cdot 297 & \cdot 402 & \cdot 516 \end{bmatrix}$$

(7) A matrix $A = (a_{ij})_{n \times n}$ is said to be diagonally dominant iff $|a_{ii}| > \sum_{\substack{k=1 \\ k \neq i}}^{n} |a_{ik}|$,

$= 1(1)n$.

Show that the choices of successive pivotal elements along the diagonal (i.e., diagonal strategy) is feasible for the solution of a system of linear equations whose coefficient matrix A is diagonally dominant.

(8) Show that a real symmetric matrix $A_{n \times n}$ is positive definite iff Gaussian elimination with diagonal strategy is feasible with n positive pivotal elements.

(9) Show that if a real matrix is non singular, then there exists a pivotal element in the k-th column for the k-th step of the process of inversion.

(10) Show that the following systems are ill-conditioned.

(i) $7x_1 + 10x_2 = 1$

$5x_1 + 7x_2 = \cdot 7$

(ii) $\cdot 24x_1 + \cdot 36x_2 + \cdot 12x_3 = \cdot 84$

$\cdot 12x_1 + \cdot 16x_2 + \cdot 24x_3 = \cdot 52$

$\cdot 15x_1 + \cdot 21x_2 + \cdot 25x_3 = \cdot 64$

(iii) $10x_1 + 7x_2 + 8x_3 + 7x_4 = 32$

$7x_1 + 5x_2 + 6x_3 + 5x_4 = 23$

$8x_1 + 6x_2 + 10x_3 + 9x_4 = 33$

$7x_1 + 5x_2 + 9x_3 + 10x_4 = 31$

(iv) $1 \cdot 00x_1 + \cdot 99x_2 = 1 \cdot 99$

$\cdot 99x_1 + \cdot 98x_2 = 1 \cdot 97.$

(11) Fit a quadratic function in t correct to four significant figures by the method least-squares for the following data.

t	0	·5	1·0	1·5	2·0	3·0	5·0	8·0	10·0
x	3·85	2·95	2·63	2·33	2·24	2·05	1·82	1·80	1·75

(12) Solve the following tri-diagonal system of equations :

(a) $x_1 + 2x_2 = 5$

$-x_1 + 3x_2 + x_3 = 8$

$2x_2 + x_3 = 7$

(b) $x_1 - x_2 = 1$

$x_1 + 2x_2 - x_3 = 3$

$5x_2 - 3x_3 = 2$

Unit 3 ❐ Eigen Values and Eigen Vectors of $n \times n$ Numerical Matrix

§ Objectives

After going through this unit you will be able to learn about—

- What is eigenvalues and eigenvectors?
- Power method to determine largest (in magnitude) eigen value
- Method to determine least (in magnitude) eigenvalue.

§ Preliminaries :

Let $A = (a_{ij})_{n \times n}$ be a matrix where the elements a_{ij}, $i, j = 1(1)n$, of A may be real or complex.

Then a scaler λ (real or complex) is said to be an eigenvalue of A if and only if there is a non-zero vector X such that $AX = \lambda X$ ..... (1)

The non-zero vector X is called the eigenvector corresponding to the eigen value λ of A. The pair (λ, X) is called an eigen-pair of A.

Since, $AX = \lambda X$

$\Rightarrow A(cX) = cAX$

$= c\lambda X = \lambda(cX)$, where c is a non-zero constant. Therefore, the eigenvalue corresponding to an eigenvector is unique but the reverse is not true. The eigenvector corresponding to an eigenvalue is arbitrary to the extent of a multiplicative constant.

Now equation (1) can be written in the form, $(A - \lambda I)X = 0$, where I is the $n \times n$ identity matrix, or, alternatively we may write.

$$(\lambda I - A)X = 0 \quad \text{..... (2)}$$

Since X is a non-zero vector,

Therefore, $(\lambda I - A)X = 0$

$\Rightarrow \det(\lambda I - A) = 0$ ..... (3)

The last equation when written in the form of a polynomical equation (of degree n) in λ, is called the characteristic equation of A and the corresponding polynomial

is known as characteristic polynomical of A. Alternatively, the roots of this characteristic equation is known as characteristic roots or eigen values of A. From linear algebra we have an important result for square matrices due to Cayley-Hamilton.

Theorem (Cayley-Hamilton) : Every square matrix satisfies its characteristic equation.

Another important related result is the following.

Theorem : Eigen-vectors, corresponding to distinct eigen values are linearly independent.

Clearly, if a $n \times n$ matrix has n linearly independent eigen-vectors, then they form a basis for n-dimensional vector space.

Now we discuss an iterative method for finding dominant eigen pair of a real matrix A, i.e., the eigen pair of A which contains the numerically largest eigenvalue of the matrix A.

Power Method :

Let us first assume that the eigen values λ_i, $i = 1(1)n$ of an $n \times n$ real matrix A are indexed in decreasing order, i.e., $|\lambda_1| \geq \lambda_2 | \geq \ldots \geq \lambda_n$ and let $x_1, x_2, \ldots, x_n$ be the corresponding eigenvectors. Now, the power method is a useful technique to compute the numerically largest eigen value of a real matrix. The scheme for power method is as follows :

Let us assume further that,

(i) $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \ldots \geq |\lambda_n|$, so that λ_1 real.

(ii) $x_1, x_2, \ldots\ldots\ldots, x_n$ form a basis for an n-dimensional vector space, i.e., x_i, $i = 1(1)n$ are linearly independent vectors.

Since, $x_1, x_2, \ldots, x_n$ form a basis, therefore, any n-vector x in space can be expressed as linear combination of $x_1, x_2, \ldots\ldots\ldots, x_n$, i.e. we may write,

$$x = \alpha_1 x_1 + \alpha_2 x_2 + \ldots \alpha_n x_n, \text{ where } \alpha_1, \alpha_2, \ldots, \alpha_n \text{ are scalars.} \quad \ldots\ldots (4)$$

We assume that x has a component in the direction of x_1, i.e.we assume $\alpha_1 \neq 0$.

Now multiplying both sides of (4) by A and considering the fact that $x_1, x_2, \ldots\ldots\ldots, x_n$ are eigenvectors corresponding to the eigenvalues $\lambda_1, \lambda_2, \ldots\ldots\ldots, \lambda_n$, we have,

$$\begin{aligned} Ax &= A(\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n) \\ &= \alpha_1 \lambda_1 x_1 + \alpha_2 \lambda_2 x_2 + \ldots \alpha_n \lambda_n x_n \end{aligned}$$

or, $$Ax = \lambda\left(\alpha_1 x_1 + \frac{\lambda_2}{\lambda_1}\alpha_2 x_2 + \ldots + \frac{\lambda_n}{\lambda_1}\alpha_n x_n\right) \quad \ldots\ldots (5)$$

Again multiplying both side by A we have,

$$A^2x = A(Ax) = \alpha_1\lambda_1^2 x_1 + \alpha_2\lambda_2^2 x_2 + \ldots \ \alpha_n\lambda_n^2 x_n$$

$$= \lambda_1^2\left(\alpha_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^2\alpha_2 x_2 + \ldots + \left(\frac{\lambda_n}{\lambda_1}\right)^2\alpha_n x_n\right)$$

Repeating the above process, after k-times operating A we have,

$$A^k x = \lambda_1^k\left(\alpha_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k\alpha_2 x_2 + \ldots + \left(\frac{\lambda_n}{\lambda_1}\right)^k\alpha_n x_n\right)$$

$$\text{or, } \left(\frac{1}{\lambda_1}A\right)^k x = \alpha_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k\alpha_2 x_2 + \ldots + \left(\frac{\lambda_n}{\lambda_1}\right)^k\alpha_n x_n$$

$$\text{Since, all } \left(\frac{\lambda_i}{\lambda_1}\right)^k \to 0 \text{ as } k \to \infty\left(\text{As}\frac{|\lambda_i|}{|\lambda_1|} < 1 \text{ for all } i = 2(1)n\right)$$

[since λ_1 is numerically largest eigenvalue, therefore, $\lambda_1 \neq 0$, otherwise all eigenvalues of A are zero, i.e. the matrix is similar to null matrix].

$$\text{Therefore, } \left(\frac{1}{\lambda_1}A\right)^k x = \alpha_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k\alpha_2 x_2 + \ldots + \left(\frac{\lambda_n}{\lambda_1}\right)^k\alpha_n x_n \to \alpha_1 x_1 \text{ as } k \to \infty.$$

which is also an eigenvector of A corresponding to the eigenvalue λ_1.

Hence, if we set a numerical procedure as follows :

Set v_0 as the initial choice vector for x as in (4), i.e., $v_0 = \alpha_1 x_1 + \ldots + \alpha_n x_n$

Then the sequence $\{v_k\}$ of vectors generated recursively by,

$$y_k = Av_k \text{ and } v_{k+1} = \frac{1}{a_{k+1}}y_k,\ k = 0, 1, 2 \ldots\ldots \quad \ldots\ldots (6)$$

where $a_{k+1} = \max_r |(y_k)_r|$, $(y_k)_r$ denotes the r-th component of the vector y_k, $r = 1(1)n$.

Converges to the dominant eignvector x_1 and correspondingly the sequence $\{a_k\}$ converges to the eigenvalue λ. Actually, from the relation

$$\left(\frac{1}{\lambda_1}A\right)^k x = \alpha_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k \alpha_2 x_2 + \ldots + \left(\frac{\lambda_n}{\lambda_1}\right)^k \alpha_n x_n$$

we have, $\frac{a_k a_{k-1} \ldots a_1}{\lambda_1^{\,k}} v_k = \alpha_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k \alpha_2 x_2 + \ldots + \left(\frac{\lambda_n}{\lambda_1}\right)^k \alpha_n x_n$ ..... (7)

[As, $Av_0 = y_0 = a_1 v_1$, $A^2 v_0 = Ay_0 = a_1 Av_1 = a_1 y_1 = a_2 a_1 v_2$, ... $A^k v_0 = a_k a_{k-1} \ldots a_1 v_k$]

So, from (7) we find, $\lim_{k\to\infty} v_k = \lim_{k\to\infty} \frac{\alpha_1 \lambda_1^{\,k}}{a_k a_{k-1} \ldots a_1} x_1$ ..... (8)

i.e., the limiting vector v_k as constructed above converges to the eignvector corresponding to the dominant eigenvalue λ_1. Now if we require both x_1 and v_k are scaled vector so that largest component is 1 then must have,

$$\lim_{k\to\infty} \frac{\alpha_1 \lambda_1^{\,k}}{a_k a_{k-1} \ldots a_1} = 1 \qquad \ldots\ldots (9)$$

So, scaling in each step, reduces to the fact that, $\lim_{k\to\infty} v_k = x_1$. ..... (10)

Now, replacing k by $k-1$ in (9) we have

$$\lim_{k\to\infty} \frac{\alpha_1 \lambda_1^{\,k-1}}{a_{k-1} a_{k-2} \ldots a_1} = 1 \qquad \ldots\ldots (11)$$

Therefore by (9) and (11), we have

$$\lim_{k\to\infty} \frac{\lambda_1}{a_k} = \lim_{k\to\infty} \frac{\left(\alpha_1 \lambda_1^{\,k}\right)/\left(a_k a_{k-1} \ldots a_1\right)}{\left(\alpha_1 \lambda_1^{\,k-1}\right)/\left(a_{k-1} \ldots a_1\right)} = \frac{1}{1} = 1$$

i.e., $\lim_{k\to\infty} a_k = \lambda_1$ ..... (12)

Hence, we find the sequence $\{a_k\}$ as constructed above also converges to the dominant eigenvalue λ_1 of A, i.e., the pair (a_k, v_k) converges to the dominant eigenpair (λ_1, x_1).

Clearly, the rate of convergence is determined by the quotient $\left|\frac{\lambda_2}{\lambda_1}\right|$ as it is

greater than all the other quotients $\left|\frac{\lambda_i}{\lambda_1}\right|$, $i = 3, 4, \ldots, n$. Smaller the valuse of $\frac{\lambda_2}{\lambda_1}$ larger the convergence rate.

In short, the numerical procedure for power method to find dominant eigen pair is as follows :

Choose an initial guess v_0 for x_1. Then find, $Av_0 = y_0 = a_1v_1$, where a_1 is the numerically largest component of y_0. Now apply again A on v_1 and find, $Av_1 = y_1 = a_2v_2$, where a_2 is the numerically largest component of y_2.

i.e., we have, $A^2v_0 = Ay_0 = a_1Av_1 = a_1y_1 = a_2a_1v_2$, where $a_2 = \max_r |(y_1)_r|$

Repeat the process. After k steps, find

$$y_{k-1} = Av_{k-1} = a_kv_k, \text{ where } a_k = \max_r |(y_{k-1})_r|$$

i.e., we have, $A^kv_0 = (a_k a_{k-1} \ldots a_1)v_k$.

The process stops if two consecutive values of a_k, say a_{k-1} and a_k do not differ much, i.e., difference is within our desired tolerance level and then we declare that a_k is the numerically largest eigen value with eigen vector v_k of the matrix A. This method is sometimes called the scaled power method. Now the choice of the initial vector v_0 may sometimes prove to be troublesome. If the vector v_0 lies in the orthogonal subspace corresponding to the eigenvector x_1, then $a_1 = 0$ and the directions of the sequence of vector $\{A^kx\}$ will converge to the direction of x_2 instead of x_1. Therefore, if in the process of iteration, the values a_i oscillate, then the choice of v_0 must be changed immediately. At least choose a vector v'_0 orthogonal to v_0. It may not always overcome the difficulty.

Example 1. Consider the matrix $A = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & 3 \\ 0 & 3 & 1 \end{bmatrix}$

True eigenvalues are, $\lambda_1 = 4{\cdot}70156$, $\lambda_2 = -1{\cdot}70156$, and $\lambda_3 = 1{\cdot}00000$, correct to six significant figures.

Now we employ power method to find the largest eigenpair. Let us take as an initial guess $v_0 = [1, 1, 1]^T$.

Then after 15 iterations we have the following table :

No of iteration	$(Av_{i-1})^T = y^T_{i-1}$	v_i^T	a_i
1	$[0, 4, 4]^T$	$[0, 1, 1]^T$	4
2	$[-1, 5, 4]^T$	$[-\cdot2, 1, \cdot8]^T$	5
3	$[-1\cdot2, 4\cdot6, 3\cdot8]^T$	$[-\cdot26, 1, \cdot83]^T$	4·6
4	$[-1\cdot26, 4\cdot75, 3\cdot83]^T$	$[-\cdot265, 1, \cdot806]^T$	4·75
5	$[-1\cdot265, 4\cdot683, 3\cdot806]^T$	$[-\cdot270, 1, \cdot813]^T$	4·683
6	$[-1\cdot270, 4\cdot709, 3\cdot813]^T$	$[-\cdot2697, 1, \cdot8097]^T$	4·709
7	$[-1\cdot2697, 4\cdot6988, 3\cdot8097]^T$	$[-\cdot27022, 1, \cdot81078]^T$	4·6988
8	$[-1\cdot27022, 4\cdot70256, 3\cdot81078]^T$	$[-\cdot27011, 1, \cdot810781]^T$	4·70256
9	$[-1\cdot27011, 4\cdot70245, 3\cdot810781]^T$	$[-\cdot270095, 1, \cdot810382]^T$	4·70245
10	$[-1\cdot270095, 4\cdot701241, 3\cdot810382]^T$	$[-\cdot2701616, 1, \cdot8103056]^T$	4·701241
11	$[-1\cdot2701616, 4\cdot7016784, 3\cdot8105056]^T$	$[-\cdot2701507, 1, \cdot81045645]^T$	4·7016784
12	$[-1\cdot2701507, 4\cdot70152005, 3\cdot81045645]^T$	$[-\cdot27015746, 1, \cdot81047330]^T$	4·70152005
13	$[-1\cdot27015746, 4\cdot70157736, 3\cdot81047330]^T$	$[-\cdot27015560, 1, \cdot81046700]^T$	4·70157736
14	$[-1\cdot27015560, 4\cdot70155660, 3\cdot81046700]^T$	$[-\cdot27015634, 1, \cdot81046924]^T$	4·70155660
15	$[-1\cdot27015634, 4\cdot70156406, 3\cdot81046924]^T$	$[-\cdot27015613, 1, \cdot81046843]^T$	4·70156406

The numerically largest eigenvalue of the given matrix after 15 iterations is found to be 4·70156 (correct upto six significant figure) and the corresponding eigen vector is, $[-\cdot270156, 1\cdot00000, \cdot810468]^T$.

Example 2. Consider the matrix $A = \begin{bmatrix} 4 & 4 & 0 \\ 4 & 4 & 0 \\ 0 & 0 & 8 \end{bmatrix}$. True eigenvalues are $\lambda_1 = 8$, $\lambda_2 = 8$, $\lambda_3 = 0$. Now we employ power method with initial guess $v_0 = [0, 1, 2]^T$. We find the following results :

No of iteration	$(Av_{i-1})^T = y^T_{i-1}$	v_i^T	a_i
1	$[4, 4, 16]^T$	$[\cdot25, \cdot25\ 1]^T$	16
2	$[2, 2, 8]^T$	$[\cdot25, \cdot25\ 1]^T$	8
3	$[2, 2, 8]^T$	$[\cdot25, \cdot25\ 1]^T$	8

So, after two iterations we find the dominant eigenvalue. However if we start with initial guess $v_0 = [1, -1, 0]^T$, then after one iteration we find,

$(Av_0)^T = [0, 0, 0]^T$. So the process terminates. We are unable to progress further. Actually, the vector v_0 is orthogonal to the subspace generated by the eigenvectors corresponding to the dominant eigenvalue 8. Here it is also interesting to note that both the non zero eigenvalues are 8, i.e., dominant.

Least Eigenpair :

The power method is also useful to find least eigenpair, i.e., numerically smallest eigenvalue with eigenvector, of a non-singular matrix A. Clearly, for a singular matrix, zero is the numerically smallest eignvalue. Now, to show, how the power method works to find numerically least eigen value of a non-singular matrix, A, we state the following theorem from linear algebra.

Theorem : If (λ, v) is an eigenpair of a nonsingular matrix A, then $\left(\frac{1}{\lambda}, v\right)$ is an eigenpair of the matrix A^{-1} and vice-versa.

Now, consider the dominant eigenvalue of A^{-1}, say λ_1. Then, clearly, reciprocal of λ_1, i.e., $\frac{1}{\lambda_1}$ is the least eigenvalue of A.

Therefore, to find least eigenpair of a non singular matrix A, we have to find dominant (numerically) or greatest eigenpair of the matrix A^{-1} by the power method. So the scheme to find numerically lowest eigenvalue of a nonsingular matrix A is as follows :

(i) Check det $(A) \neq 0$.

(ii) Find A^{-1}.

(iii) Use power method to find numerically greatest eigenvalue of A^{-1}. Let it be λ.

(iv) Declare $\frac{1}{\lambda}$ is the numerically lowest eignvalue of A. This method is sometimes called as inverse power method.

Example 3. Consider the matrix $A = \begin{bmatrix} 3 & 0 & 1 \\ 0 & -3 & 0 \\ 1 & 0 & 3 \end{bmatrix}$

True eigen values are $\lambda_1 = 4$, $\lambda_2 = 2$, $\lambda_3 = -3$.

Now, $A^{-1} = \begin{bmatrix} \frac{3}{8} & 0 & -\frac{1}{8} \\ 0 & -\frac{1}{3} & 0 \\ -\frac{1}{8} & 0 & \frac{3}{8} \end{bmatrix}$

Using power method with initial guess $v_0 = [0, 1, 2]^T$, we find after 5 iterations the coefficients a_i for the matrix A^{-1} as follows :

$$a_1 = \frac{3}{4},\ a_2 = \frac{5}{12},\ a_3 = \frac{9}{20},\ a_4 = \frac{17}{36},\ a_5 = \frac{33}{68}.$$

Which converges towards $\frac{1}{2}$, the reciprocal of the least eigenvalue 2 of A.

Shifted Eigenvalues :

Let c be any scalar (real or complex) and (λ, v) be an eigenpair of a matrix A. Then,

$$Av = \lambda v \Leftrightarrow (A - cI)v = \lambda v - cv = (\lambda - c)v. \qquad \ldots\ldots (13)$$

i.e., if (λ, v) is an eigenpair of A, then $(\lambda - c, v)$ is an eigenpair of $A - cI$ and vice-versa.

Let λ_c be the eigenvalue of A nearest to c and let v_c be the corresponding eigenvector. Then $(\lambda_c - c, v_c)$ is the least eigenpair of $A - cI$ implies

$((\lambda_c - c)^{-1}, v_c)$ is the dominant eigenpair $(A - cI)^{-1}$. Therefore, we may find λ_c by the following computational scheme :

$$\lambda_c = \frac{1}{\text{dominant eigenvalue of}(A - cI)^{-1}} + c \qquad \ldots\ldots (14)$$

This procedure of finding eigenpair (λ_c, v_c) of a matrix A is known as Shifted-Power method.

Example 4. Consider the matrix $A = \begin{bmatrix} 3 & 0 & 1 \\ 0 & -3 & 0 \\ 1 & 0 & 3 \end{bmatrix}$ of the above example. With shift $c = 3$, we have, $A - 3I = \begin{bmatrix} 0 & 0 & 1 \\ 0 & -6 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

Therefore, $(A - 3I)^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & -\frac{1}{6} & 0 \\ 1 & 0 & 0 \end{bmatrix}$ True eigenvalues are $\lambda_1 = 1$, $\lambda_2 = -1$, $\lambda_3 = -\frac{1}{6}$, i.e., $|\lambda_1| = |\lambda_2| = 1$. However, considering the initial guess $v_0 = [1, 1, 1]^T$ we find by power method, the numerically dominant eigen value of $(A - 3I)^{-1}$ is 1 with eigenvector approaching to $[1, 0, 1]^T$.

Example 5. Consider another matrix $A = \begin{bmatrix} 1+2 & 2 & 3 \\ 2 & 3+2 & 4 \\ 3 & 4 & 5+2 \end{bmatrix}$ The resulting matrix with shift 2 is $A - 2I = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$ which is singular. Hence we are unable to find $(A - 2I)^{-1}$ and there is no scope of finding any eigenvalue (if exists) of A near 2 by shifted power method. However, the true eigenvalues of A are $\lambda_1 = 2$, $\lambda_2 = \frac{13 + \sqrt{105}}{2}$, $\lambda_3 = \frac{13 - \sqrt{105}}{2}$. Clearly, $A - 2I$ singular means it has an eigenvalue 0, so, A must have an eigenvalue 2.

§ Summary :

The eigenvalues of a square matrix can be determined from its characteristic equation. But, the solution of the characteristic equation is very difficult for large matrices. Some suitable numerical methods are used to determined eigenvalues of a matrix. In this unit, the power method is discussed to findout the numerically largest eigenvalue of any square matrix. Also, an outline is provided to determine the least (in magnitude) eigenvalue.

EXERCISES

(1) (i) Prove that if λ is an eigenvalue of a square matrix A, then $a\lambda + b$ is an eigenvalue of the matrix $aA + bI$.

(ii) Prove that if λ is an eigenvalue of a square matrix A, then for any polynomial $p(x)$, $p(\lambda)$ is an eigenvalue of the matrix $p(A)$.

(iii) Prove that if $\lambda(\neq 0)$ in an eigenvalue of a non-singular matrix A, then $\frac{1}{\lambda}$ is an eigenvalue of A^{-1}.

(2) Use power method to find dominant eigenpair of the following matrices.

(i) $A = \begin{bmatrix} 4 & -2 \\ -3 & 5 \end{bmatrix}$, start with $v_0 = [1, -1]^T$.

(ii) $A = \begin{bmatrix} 4 & -1 \\ -3 & 5 \end{bmatrix}$, start with $v_0 = [-1, 1]^T$.

(iii) $A = \begin{bmatrix} 9 & -10 & 8 \\ 10 & 5 & -1 \\ 8 & -1 & 3 \end{bmatrix}$, start with $v_0 = [1, 1, 1]^T$.

(iv) $A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$, start with $v_0 = [1, 2, 3]^T$, correct to 3 decimal places.

(v) $A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$, start with yor choice.

(vi) $A = \begin{bmatrix} 4 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 1 \end{bmatrix}$, start with your choice, correct to 3 decimal places.

(vii) $A = \begin{bmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{bmatrix}$, with initial vector $v_0 = [1, 1, 1, 1]^T$, correct to 5 decimal places.

(3) Use Power method to find numerically least eigenpair of the following matrices.

(i) All the matrices of exerciese 2 with your initial choice vector, correct to 3 decimal places.

(ii) $A = \begin{bmatrix} 4 & 5 & 6 \\ 5 & 6 & 7 \\ 7 & 8 & 9 \end{bmatrix}$, with your initial choice, correct to 2 decimal places.

(iii) $A = \begin{bmatrix} 7 & 6 & -3 \\ -12 & -20 & 24 \\ -6 & -12 & 16 \end{bmatrix}$, with $v_0 = [1, 1, 1]^T$ correct to 3 decimal places.

(4) (i) Use shifted power method with shift 3·01 to find eigenvalue of the matrix $A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$, correct to 4 decimal places. Start with initial vector $v_0 = [1, 1, 1]^T$.

(ii) Compute eigenvalue of the following matrix near to 4 using shift—power method.

$$A = \begin{bmatrix} 14 & 7 & 6 & 9 \\ 7 & 9 & 4 & 6 \\ 6 & 4 & 9 & 7 \\ 9 & 6 & 7 & 15 \end{bmatrix}$$ correct to 5 decimal places.

(iii) Use shift power method to find the eigenpairs of the following matrices.

(a) $A = \begin{bmatrix} 0 & 11 & -5 \\ -2 & 17 & -7 \\ -4 & 26 & -10 \end{bmatrix}$ with shift 4·2, correct to 5 decimal places.

(b) $A = \begin{bmatrix} -12 & -72 & -59 \\ 2 & 29 & 23 \\ -2 & -12 & -9 \end{bmatrix}$ with shift 3, correct to 5 decimal places.

Unit 4 ☐ Solutions of Non-linear Equations

§ Objectives

After going through this unit you will be able to learn about—

- Location of root of non-linear single equation
- Bisection method
- Regula-falsi method
- Fixed point interation method
- Newton-Raphson method
- Convergence of a method
- Roots of a polynomial equation by Bairstow's method
- Roots of a system of non-linear equations by Newton's method

§ Single Equation :

Here we shall restrict ourselves to the numerical solutions of equations of single unknown variable x (real or complex), of the form $f(x) = 0$.

We further assume that $f(x)$ is continuous in the domain of the required solution. Sometimes we may also assume the continuity or differentiability of higher order derivatives of $f(x)$ is required.

Now a point $x = x_0$ is said to be a zero of $f(x)$ or a root of the equation $f(x) = 0$, if and only if $f(x_0) = 0$; e.g., the equation $x^2 - 5x + 6 = 0$ has two real roots $x = 2$ and $x = 3$, equivalently, $x = 2$ and $x = 3$ are the zeros of the function $f(x) = x^2 - 5x + 6$. An equation may have also complex roots. The simplest example of an equation containing complex roots is $x^2 + 1 = 0$. It has two complex roots (conjugate to each other), $x = i$ and $x = -i$, where $i = \sqrt{-1}$. Further, a root $x = x_0$ of the equation $f(x) = 0$ is said to be of multiplicity r if and only if for all non-negative integers $k \leq r$, we have,

$$\lim_{x \to x_0} \frac{|f(x)|}{|x - x_0|^k} < \infty$$

Here we recall a theorem from calculus.

Theorem : A point $x = x_0$ is said to be a zero of at least multiplicity r, for some integer, of the function $f(x)$ which is r times continuously differentiable at $x = x_0$ if and only if $f(x_0) = f'(x_0) = \ldots\ldots = f^{(r-1)}(x_0) = 0$.

If the multiplicity is exactly r, then, of course, $f^{(r)}(x_0) \neq 0$. Also if $f(x)$ is $r+1$ times continuously differentiable at $x = x_0$ and if x_0 is a zero of multiplicity r of $f(x)$, then $f^{(r)}(x_0) \neq 0$.

Example 1. Let $f(x) = (x-a)^2 e^x$. Then $x = a$ is a root of the equation $f(x) = 0$ with multiplicity 2. The function $f(x)$ is differentiable continuously any times. We have, $f'(x) = 2(x-a)e^x + (x-a)^2 e^x$, $f''(x) = 2e^x + 4(x-a)e^x + (x-a)^2 e^x$, $f(a) = f'(a) = 0$, but $f''(a) \neq 0$.

§ Locating Roots : Isolation or Bracketing of a Root :

Now our task is to find a root of the equation $f(x) = 0$, if possible exactly, or otherwise, to some desired degree of accuracy. For that, we first need to locate roots of $f(x) = 0$ in separate small intervals. The process is known as isolation or separation or bracketing of roots. It helps us to find roots within our desired accuracy very quickly. A root $x = x_0 \in [a, b]$ of the equation $f(x) = 0$ is said to be separated if the equation $f(x) = 0$ has no other roots except x_0 in $[a, b]$. Now we first describe a method called graphical method.

Graphical Method :

Let $y = f(x)$ be a function of x as shown in Fig. 1. The figure shows, it has two

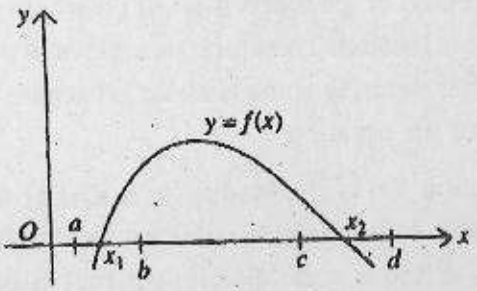


Fig. 1

roots at $x = x_1$ and $x = x_2$, one between $x = a$ and $x = b$, other between $x = c$ and $x = d$. There are no other roots of $f(x) = 0$ on the intervals $[a, b]$ and $[c, d]$. Now it may be the case that the curve $y = f(x)$ is not easy to construct graphically but the

equation $f(x) = 0$ may be written in the form $g(x) = h(x)$, where $g(x)$ and $h(x)$ are two functions which we may represent graphically.

We can locate roots of $f(x) = 0$, by representing $f(x) = 0$ first as $g(x) = h(x)$ and then by observing the points of intersection of $y = h(x)$ and $y = g(x)$, like, $x = x_0$ between $x = a$ and $x = b$, in Fig. 2.

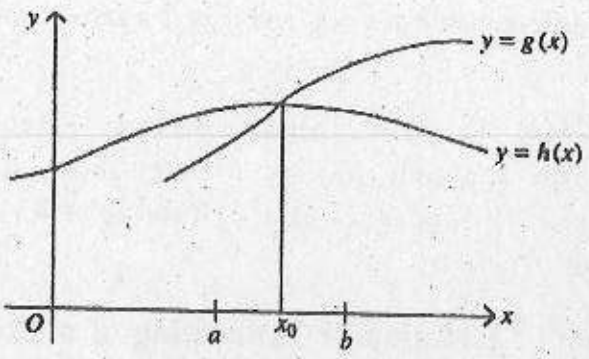


Fig. 2

For example, let us consider the function $y = f(x) = x - \ln x - 1$. It is in generally not easy to plot $f(x)$ graphically. But we may draw $y = x - 1$ and $y = \ln x$ in much better way than $y = f(x)$. In the interval $[1 - \varepsilon, 1 + \varepsilon]$ where $\varepsilon > 0$ is a small quantity, we find $x = 1$ is a point of intersection of both the functions $y = x - 1$ and $y = \ln x$, and hence a root of the equation $x - \ln x - 1 = 0$.

Here we should remember some properties of functions having roots of different multiplicities. For example, for roots with multiplicity 2, the graph will touch x-axis. For roots with multiplicity 3, it is a point of inflection. Now graphical method for locating roots is not a method of great precision. It only help us to determine roughly the interval where a root is located. Therefore, to achieve better result, we need some better methods. We further describe some analytic properties of functions which will help us to locate roots of an equation.

Theorem : If a function $f(x)$ is continuous for $x \in [a,b]$ and takes opposite signs at end points $x = a$ and $x = b$, i.e., $f(a)f(b) < 0$, then there exists at least one real root of $f(x) = 0$ between $x = a$ and $x = b$. [clearly, the root is of odd multiplicity].

Theorem : If a function $f(x)$ is continuous and monotonic on $[a, b]$ and takes opposite signs at $x = a$ and $x = b$, then there is exactly one root of the equation $f(x) = 0$, in the interval $[a, b]$.

[Clearly, the root is simple root].

Theorem : Let a function $f(x)$ be continuous on $[a, b]$ and takes opposite signs at $x = a$ and $x = b$. Further, $f'(x)$ exist and retain its sign on $[a, b]$ (i.e., either $f(x) > 0$, or $f'(x) < 0$, on $[a, b]$). Then, $f(x)$ has exactly one zero on $[a, b]$.

Now we recall some notions regarding some analytic properties of functions.

The set of all x for which a function $f(x)$ is defined, is called the domain of definition of $f(x)$. The function $f(x)$ is called monotonic increasing on $[a, b]$ if and only if for all $x_1 \geq x_2$ in $[a, b]$, $f(x_1) \geq f(x_2)$ and it is called monotonic decreasing if and only if for all $x_1 \geq x_2$ in $[a, b]$, $f(x_1) \leq f(x_2)$ holds. If the function $f(x)$ is continuous on $[a, b]$ and its derivative exist at all points on (a, b), then the necessary and sufficient condition for monotonicity of $f(x)$ on $[a, b]$ are, $f'(x) \geq 0$ for monotonic increasing and $f'(x) \leq 0$ for monotonic decreasing. Also if $f(x)$ is continuous on $[a,b]$, $f'(x)$ exist on (a,b) and has steady sign, then,

(i) $f(x)$ increases on $[a, b]$, if $f'(x) > 0$ on (a, b), and

(ii) $f(x)$ decreases on $[a, b]$, if $f'(x) < 0$ on (a, b).

If further, $f(a)$ and $f(b)$ are of opposite signs, then $f(x)$ must cut x-axis at a point, say, $x = x_0 \in (a,b)$.

Again, if a function $f(x)$ has second order derivative on (a, b) which retains its sign, then

(i) the graph of $f(x)$ is convex downwards on $[a, b]$, if $f''(x) > 0$, and

(ii) the graph of $f(x)$ is convex upwards on $[a, b]$, if $f''(x) < 0$.

The points where $f'(x) = 0$, or does not exist, are called critical points.

Therefore, for continuous functions an useful technique to locate root on an interval $[a, b]$ is to check whether $f(a).\ f(b) < 0$ or not.

Next we describe two further useful algorithms for bracketing roots with higher degree of accuracy.

Method of Bisection :

Have we first assume that $f(x)$ is continuous on $[a, b]$ and $f(a)f(b) < 0$. Then there is at least one real root of $f(x) = 0$ between $x = a$ and $x = b$. Let $c = \frac{a+b}{2}$. We divide $[a, b]$ into two parts $[a, c]$ and $[c, b]$ and compute $f(c)$. Then we check whether, (i) $f(a)f(c) < 0$ or, (ii) $f(c)f(b) < 0$ or, (iii) $f(c) = 0$. If (iii) occurs, then $x = c$ is the exact root and the process is terminated. Now, if $f(x)$ has only one root in $[a, b]$ then one

of three conditions (i)–(iii) must be satisfied. Suppose, either (i) or (ii) is satisfied.

Then we call the interval $[a, c]$ or $[c, b]$ as $[a_1, b_1]$ and divide $[a_1, b_1]$ into two parts $[a_1, c_1]$ and $[c_1, b_1]$, where $c_1 = \frac{a_1 + b_1}{2}$. As before we now compute $f(c_1)$ and study the same procedure as in (i)–(iii) for the interval $[a_1, b_1]$. We continue the process unit the value of $f(c_n)$, where $c_n = \frac{a_n + b_n}{2}$, known as n-th iterate of c's, reaches zero of $f(x)$, to our desired accuracy. So, if $x = x_0$, is the only zero of $f(x)$ on $[a, b]$ then, we must have the following relations between a_i, b_i, for $i = 1, 2, \ldots\ldots$, $a \le a_1 \le a_2 \le \ldots\ldots \le a_n \le a_{n+1} \ldots\ldots \le x_0 \le \ldots\ldots \le b_{n+1} \le b_n \le \ldots\ldots \le b_1 \le b$.

Now we discuss convergence of the above procedure. We first mention the following theorem.

Theorem : If $f(x)$ is continuous on $[a, b]$ and $f(a)f(b) < 0$, then there is a number $x_0 \in (a, b)$ such that $f(x_0) = 0$.

Therefore, if we consider $\{c_n\}$ as a sequence of numbers generated by the bisection method, then $|x_0 - c_n| \le \frac{b-a}{2^{n+1}}$, for $n = 0, 1, 2, \ldots\ldots$, where $c_0 = c = \frac{a+b}{2}$ and the sequence converges to $x = x_0$, i.e. $\lim_{n\to\infty} c_n = x_0$.

Proof. Since both the root x_0 and the number c_n lie in the interval $[a_n, b_n]$, therefore, $|x_0 - c_n|$ is less than or equal to the half of the length of the interval $[a_n, b_n]$, for any $n = 0, 1, 2, \ldots\ldots$ i.e.,

$$|x_0 - c_n| \le \frac{|b_n - a_n|}{2} \quad \forall n = 0, 1, 2, \ldots\ldots$$

Now, $|b_1 - a_1| = \frac{b-a}{2}$, $|b_2 - a_2| = \frac{|b_1 - a_1|}{2}, \ldots\ldots$

$$|b_n - a_n| = \frac{|b_{n-1} - a_{n-1}|}{2} = \frac{|b_{n-2} - a_{n-2}|}{2^2} = \ldots = \frac{(b-a)}{2^n}$$

Therefore, $|x_0 - c_n| \le \frac{|b_n - a_n|}{2} = \frac{(b-a)}{2^{n+1}}$, $\forall n = 0, 1, \ldots.$

Again, for any non-negative integers n, m, we have,

$$|c_n - c_m| = |x_0 - c_m - (x_0 - c_n)|$$

$$\leq |x_0 - c_m| + |x_0 - c_n|$$

$$\leq \frac{(b-a)}{2^{m+1}} + \frac{(b-a)}{2^{n+1}} = \frac{b-a}{2^{m+1}}\left(1 + \frac{1}{2^{n-m}}\right)$$

$< \varepsilon$, if we take n, m $(n > m)$ sufficiently large for given $\varepsilon > 0$.

i.e., $\{c_n\}$ is a Cauchy sequence in $[a, b]$ and hence it is convergent. The above relation clearly indicates that c_n converges to x_0 i.e. $\lim_{n\to\infty} c_n = x_0$.

Example 2. Find the root of the equation $f(x) \equiv \cos x \cosh x + 1 = 0$, by bisection method correct to eight decimal places where the root lies between 1·8 and 1·9.

Solution : Here it is given that the initial interval $[a_0, b_0]$ is [1·8, 1·9]. After 24 iterations we have the following table.

k	a_k	b_k	c_k	$f(c_k)$
0	1·8	1·9	1·85	+ve
1	1·85	1·9	1·875	+ve
2	1·875	1·9	1·8875	−ve
3	1·875	1·8875	1·88125	−ve
.....	.....	.....	.....	.....
.....	.....	.....	.....	.....
23	1·875104060	1·875104072	1·875104066	+ve
24	1·875104066	1·875104072	1·875104069	

The required root is 1·87510407, correct to eight decimal places.

Method of false position or Regula-falsi method :

Here again we assume that $f(x)$ is continuous on $[a, b]$ and $f(a).\ f(b) < 0$. Now instead of considering the first approximation to the root as the mid-point of $[a,b]$,

it may be sometimes better to take the first approximation to the root $x = x_0$ of $f(x) = 0$ as $x_1 = \dfrac{bf(a) - af(b)}{f(a) - f(b)}$ or, as $x_1 = a - \dfrac{f(a)(b-a)}{f(b) - f(a)}$, where $(x_1, 0)$ is the point where the chord joining the points $(a, f(a))$ and $(b, f(b))$ cuts the x-axis. Here also, as in the bisection method, three possibilities will arise :

(i) $f(x_1) = 0$, in which case x_1 is the root;

(ii) $f(a) f(x_1) < 0$, in which case the root lies between a and x_1, and

(iii) $f(x_1) f(b) < 0$, in which case the root lies between x_1 and b.

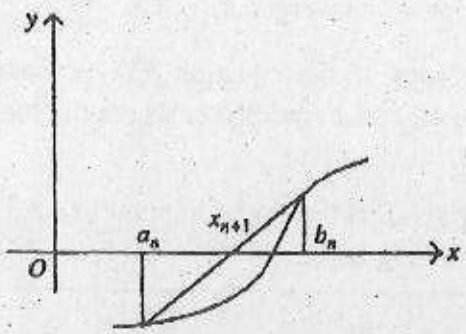


Fig. 3

If case (i) occurs, the process is terminated; if either case (ii) or case (iii) occurs, then the above process continues until the root is obtained to the desired accuracy.

Therefore, for second iteration x_2 of the root $x = x_0$, we call the new interval as $[a_1, b_1]$ where either the condition (ii) or (iii) occurs, and x_2 is given by the formula,

$$x_2 = a_1 - \frac{f(a_1)(b_1 - a_1)}{f(b_1) - f(a_1)},$$

Hence, the $(n + 1)$-th iterate is given by,

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

It is easy to prove that this method converges and the rate of convergence is faster than the bisection method. Here we must be careful about the differences $b_n - a_n$, it may not converge to zero. For example, if we consider $f(x) = x \sin x - 1$. Then it has a zero in the interval $[0, 2]$ but $b_n - a_n \neq 0$.

An estimate of error in regula-falsi method is given by,

$|\varepsilon_{n+1}| \leq \frac{1}{2}\frac{k}{m}(b_n - a_n)|\varepsilon_n|$, where ε_n is the error in the n-th stage, $k = \sup_x |f''(x)|$ and $m = \inf_x |f'(x)|$.

Example 3. Determine the root of the equation in Example 2 by regula-falsi method, correct to eight decimal places.

Solution : The following table gives the details of the computations by regula-falsi method.

k	a_k	b_k	x_k	$f(x_k)$
0	1·8	1·9	1·873697942	$5·8127 \times 10^{-3}$
1	1·873697942	1·9	1·875078665	$1·0512 \times 10^{-4}$
2	1·875078665	1·9	1·875103609	$1·9021 \times 10^{-6}$
3	1·875103609	1·9	1·875104061	$3·190 \times 10^{-8}$
4	1·875104061	1·9	1·875104068	$2·9 \times 10^{-9}$

Therefore the root is 1·87510407, correct to eight decimal places.

Now it should be noted here that if roots have multiplicity more than one, then the processes described above are not quicker enough. In the neighbourhoods of the roots, the processes slow down. In actual computation there are always some error occur due to round-off or instability in the computations. We usually call a root finding method is well-conditioned, if in the neighbourhood of the root $f(x) = 0$, is steep enough so that it is easy to obtain a solution with a great number of significant digits. Otherwise, we call the problem as ill conditioned. At the neighbourhoods of multiple roots it occurs.

Next we describe a general iterative scheme for finding roots of a equation $f(x) = 0$.

Iterative Method (Fixed point iteration) :

Consider an equation $f(x) = 0$ with one unknown variable x. Now to find a root in a certain interval of the equation, we first recast the equation in the following manner.

$$x = \phi(x) \qquad \ldots\ldots (1)$$

so that if α is a root of the equation $f(x) = 0$, then it satisfies the relation $\phi(\alpha) = \alpha$. Clearly, then α is a fixed point of the function ϕ. For this reason, the method is known as fixed point iteration process. Here we recall the famous Banach fixed point theorem.

Theorem : Let (X, d) be a complete metric space and $T : X \to X$ be a contraction map. Then, T has a unique fixed point in X, i.e., there exists a unique $x_0 \in X$ such that $T(x_0) = x_0$.

The scheme for fixed point iteration process is as follows :

Let x_0 be an initial guess for the root of the equation $f(x) = 0$, or, equivalently $x = \phi(x)$. Then we calculate successive approximates x_k as follows :

$$x_1 = \phi(x_0),$$

$$x_2 = \phi(x_1),$$

$$x_3 = \phi(x_2),$$

.....

.....

$$x_{k+1} = \phi(x_k) \qquad \text{..... (2)}$$

.....

.....

To examine the usefulness of the above algorithm, we first analyse the following questions.

(i) Does the sequence $x_0, x_1, \ldots\ldots, x_n, \ldots\ldots$converge to a point, say, α?

(ii) Is the limit α a fixed point of ϕ, or not?

To answer the above questions we make some assumptions about the nature of the function $\phi(x)$:

(1) ϕ must be a self map on an interval $[a, b]$ where we want to find the root of the equation, i.e., $\phi(x) \in [a, b]$, $\forall x \in [a, b]$.

(2) ϕ is continuous on $[a, b]$.

(3) ϕ is differentiable on (a,b) and $\exists$ a non-negative constant $K < 1$, such that $|\phi'(x)| \leq K$, $\forall x \in [a,b]$.

Clearly, assumption (3) implies assumption (2). Also, assumptions (1) and (2) imply, ϕ has a fixed point in $[a, b]$.

Proof. Let $h(x) = \phi(x) - x$. Then $h(x)$ is also continuous on $[a, b]$ by assumption (2). Now, $h(a) = \phi(a) - a$ and $h(b) = \phi(b) - b$. Since, by assumption (1), $a \leq \phi(a) \leq b$ and $a \leq \phi(b) \leq b$, therefore, $\phi(a) - a \geq 0$ and $\phi(b) - b \leq 0$.

If $\phi(a) = a$ and $\phi(b) = b$, then a, b are fixed points of ϕ, hence, both of them are roots of the equation $f(x) = 0$. Let $\phi(a) \neq a$ and $\phi(b) \neq b$. Then $\phi(a) - a > 0$ and $\phi(b) - b < 0$.

Therefore, $h(a)\, h(b) < 0$, i.e., h changes its sign on $[a, b]$.

So, $\exists$ a zero $x = \alpha$ of h on $[a, b]$, (by intermediate value theorem) such that $h(\alpha) = 0$, i.e. $\alpha = \phi(\alpha)$, is a fixed point.

Next, we shall show that the sequence $x_0, x_1, \ldots, x_n, \ldots$ converges to a unique fixed point α of ϕ on $[a, b]$.

Proof. Since α is a fixed point of ϕ on $[a,b]$,

therefore, $\phi(\alpha) = \alpha$.

Let, $y_n = \alpha - x_n$, $\forall n = 0, 1, 2, \ldots$

or $y_n = \phi(\alpha) - \phi(x_{n-1})$

$= (\alpha - x_{n-1})\, \phi'(\xi_n)$, by M.V.T. of differential calculus, where ξ_n is a point in between α and x_{n-1}.

Therefore, $|y_n| \leq K |\alpha - x_{n-1}|$, by assumption (3)

or, $|y_n| \leq K |y_{n-1}|$

Similarly, $|y_{n-1}| \leq K |y_{n-2}| \leq K^2 |y_{n-3}| \ldots$

So, $|y_n| \leq K |y_{n-1}| \leq K^2 |y_{n-2}| \leq \ldots \leq K^n |y_0| = K^n |\alpha - x_0|$

Since, $0 \leq K < 1$, $K^n \to 0$ as $n \to \infty$.

Therefore, $|y_n| \to 0$ as $n \to \infty$. Clearly, it does not depend on choice of x_0 in $[a, b]$.

Hence $x_n \to \alpha$ as $n \to \infty$.

Proof of uniqueness of α is left as an exercise to the reader.

Here we must take care of fact that if $|\phi'(x)| > 1$, $\forall x \in [a,b]$, then the scheme will not converge to α, the fixed point of ϕ on $[a, b]$.

The above theorem tells us about the error bound in this scheme.

Since x_n is the n-th approximate to α, therefore, by the above theorem we have,

$$|\alpha - x_n| \le K^n |\alpha - x_0|, \forall n \ge 1.$$

Also, $|\alpha - x_0| = |\alpha - x_1 + x_1 - x_0| \le |\alpha - x_1| + |x_1 - x_0| \le K|\alpha - x_0| + |x_1 - x_0|$

or, $$|\alpha - x_0| \le \frac{1}{1-K}|x_1 - x_0|$$

Therefore, $|\alpha - x_n| \le \frac{K^n}{1-K}|x_1 - x_0|, \forall n \ge 1$, which gives us a computable estimate of error in this scheme.

Let e_n = error in the n-th approximate.

$$= \alpha - x_n$$

Then, we have $|e_{n+1}| = |\alpha - x_{n+1}| = |\phi(\alpha) - \phi(x_n)|$

$$= |\alpha - x_n| \phi'(\xi_{n+1})$$

$$\approx |e_n| . \phi'(\alpha), \text{ for large } n.$$

Because, if the iteration process converges, then $\xi_{n+1} = \alpha$, for large, n, hence, we have,

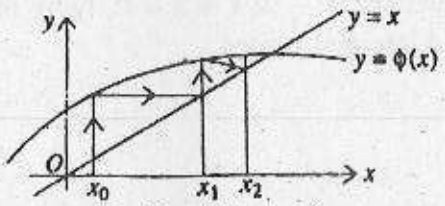
$$|e_{n+1}| \simeq |e_n| \phi'(\alpha).$$

The constant $\phi'(\alpha)$ is known as asymptotic convergence factor. Figures below now show the path of convergences or divergences of the iteration process for different values of $\phi'(\alpha)$.

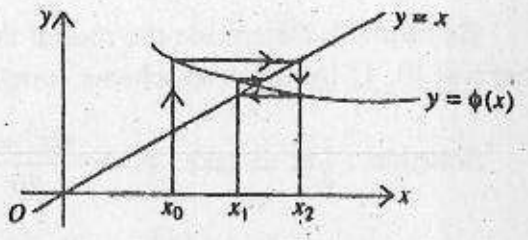
Now let us assume that,

$$\phi'(\alpha) = \phi''(\alpha) = \ldots\ldots = \phi^{(r-1)}(\alpha) = 0 \text{ but } \phi^{(r)}(\alpha) \neq 0.$$

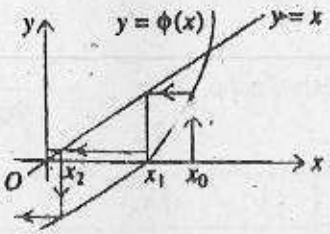
i.e.; $\phi(x)$ is an iterative function defining an iterative process of order r.



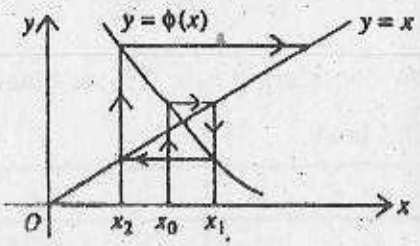
$0 < \phi'(\alpha) < 1$



$-1 < \phi'(\alpha) < 0$



$\phi'(\alpha) > 1$



$\phi'(\alpha) < -1$

Fig. 4

Then, $x_{n+1} - \alpha = \phi(x_n) - \phi(\alpha)$

$= \phi(\alpha + x_x - \alpha) - \phi(\alpha)$

$= \dfrac{(x_n - \alpha)^r}{r!}\phi^{(r)}(\xi_n)$, by Taylor's series expansion, where ξ_n lies between x_n and α.

or, $e_{n+1} = (-1)^r \dfrac{e_n^r}{r!}\phi^{(r)}(\xi_n)$ [As we have defined $e_n = \alpha - x_n$]

or, $|e_{n+1}| \le \dfrac{K_r}{r!}|e_n|^r$, where $K_r = \max\limits_{x\in[a,b]} |\phi^{(r)}(x)|$.

This gives an estimate of error in the process.

Again, if the iteration process converges, then the above relation we have,

$$\lim_{n\to\infty}\left|\frac{e_{n+1}}{e_n^r}\right| = \frac{|\phi^{(r)}(\alpha)|}{r!} \neq 0, \text{ as } \xi_n \text{ will also converges to } \alpha \text{ with } n \to \infty.$$

Thus, generally, we say that the order of convergence of the iteration process is r with asymptotic error constant $\dfrac{|\phi^{(r)}(\alpha)|}{r!}$. If $r = 1$, we call convergence is linear, if $r = 2$, we call convergence is quadratic, etc.

Example 4. Determine the root of the equation $5x^3 - 20x + 3 = 0$, lying in the interval [0, 1] by iterative scheme correct to 4 decimal places.

Solution : Let us take, $\phi(x) = \frac{5x^3 + 3}{20}$.

Then, $|\phi'(x)| = \frac{3x^2}{4} < 1$ on [0, 1].

Now considering $x_0 = \cdot 75$, we obtain by the iterative scheme $x_{n+1} = \frac{5x_n^3 + 3}{20}$, the following table.

n	x_n	$\phi(x_n)$
0	0·75	0·25547
1	0·2555	0·154144
2	0·1541	0·151413
3	0·1514	0·151361
4	0·15136	0·151361

Therefore, the root is 0·1514 Correct to 4 decimal places.

Method of Tangent / Newton-Rephson Method :

Here we first assume that $f(x)$, $f'(x)$, $f''(x)$ are continuous near a root α of the equation $f(x) = 0$. Then we have a fixed point iteration scheme which is faster than the usual bisection method, regula-falsi method for estimating the root α. Actually, the convergence is quadratic for simple roots in this method. Let x_n be the n-th iterate of the root α and $\varepsilon_n = \alpha - x_n$ be the corresponding error term. Then, by Taylor's expansion, $0 = f(\alpha) = f(x_n + \alpha - x_n) = f(x_n + \varepsilon_n)$

$= f(x_n) + \varepsilon_n f'(x_n) + \frac{1}{2}\varepsilon_n^2 f''(\xi_n)$, where ξ_n is a point lies between α and x_n.

Now, assuming x_n is close enough to α, we may neglect ε_n^2 as a very small quantity. Then, we have an approximation to ε_n, say, h_n such that,

$$h_n = \frac{-f(x_n)}{f'(x_n)}, \text{ provided } f'(x_n) \neq 0.$$

Let, $x_{n+1} = x_n + h_n = x_n - \frac{f(x_n)}{f'(x_n)}$, $n = 0, 1, 2, \ldots$, where x_0 is the initial choice for α.

Then, we call x_{n+1} as $(n+1)$-th interate to the root α and the procedure of finding successive iterates by this scheme is known as Newton-Raphson method.

Geometrically, consider the point $(x_n, f(x_n))$ on the curve $y = f(x)$. Then draw a tangent at this point. It cuts the x-axis, say, at the point $(x_{n+1}, 0)$. We call x_{n+1} as the next approximate to α. Clearly $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ and for this reason the method is also known as method of tangent.

It is easy to show that Newton-Raphson method is a fixed point iteration method with the iteration function ϕ, defined by, $\phi(x) = x - \frac{f(x)}{f'(x)}$. Since, α is the root of $f(x) = 0$, therefore, $\phi(\alpha) = \alpha - \frac{f(\alpha)}{f'(\alpha)} = \alpha - \frac{0}{f'(\alpha)} = \alpha$ i.e., α is a fixed point of ϕ.

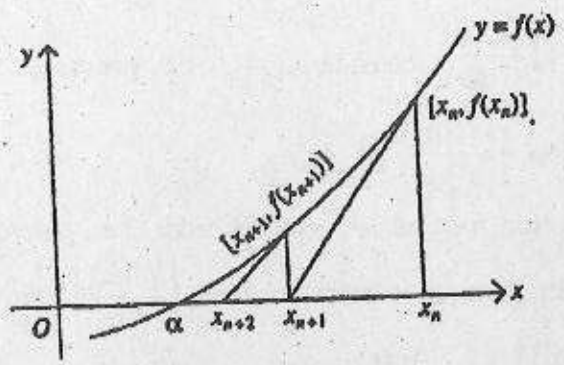


Fig. 5

Hence, the convergence criteria for Newton-Raphson scheme is, $|\phi'(x)| < 1$, where $x \in [a,b]$, which brackets the sole root α of $f(x) = 0$, i.e., $[a,b]$ is a small neighbourhood of α containing no other roots of $f(x) = 0$.

or, $\left|1 - \frac{[f'(x)]^2 - f(x)f''(x)}{[f'(x)]^2}\right| < 1$,

or, $\left|\frac{f(x)f''(x)}{[f'(x)]^2}\right| < 1$,

or, $|f(x)f''(x)| < [f'(x)]^2$ for $x \in [a,b]$.

Now, to estimate the error occurring in this method, we have,

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

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

or, $\varepsilon_{n+1} = \varepsilon_n + \frac{f(\alpha - \varepsilon_n)}{f'(\alpha - \varepsilon_n)},$

$$= \varepsilon_n + \frac{f(\alpha) - \varepsilon_n f'(\alpha) + \frac{1}{2}\varepsilon_n^2 f''(\alpha) + \ldots}{f'(\alpha) - \varepsilon_n f''(\alpha) + \frac{1}{2}\varepsilon_n^2 f'''(\alpha) + \ldots}, \text{ by Taylor's expansion,}$$

$$= \varepsilon_n + \frac{-\varepsilon_n f'(\alpha) + \frac{1}{2}\varepsilon_n^2 f''(\alpha) + \ldots.}{f'(\alpha) - \varepsilon_n f''(\alpha) + \frac{1}{2}\varepsilon_n^2 f'''(\alpha) + \ldots..}, \text{ as } f'(\alpha) = 0,$$

$$= -\frac{1}{2}\varepsilon_n^2 \frac{f''(\alpha)}{f'(\alpha)}, \text{ considering } \varepsilon_n << 1, \text{ provided } f'(\alpha) \neq 0.$$

i.e., $\left|\frac{\varepsilon_{n+1}}{\varepsilon_n^2}\right| \to \frac{1}{2}\left|\frac{f''(\alpha)}{f'(\alpha)}\right|$ as $n \to \infty$.

Hence, Newton-Raphson method is a second order (i.e., convergence is quadratic) iteration process with asymptotic error constant $\left|\frac{f''(\alpha)}{2f'(\alpha)}\right|$, provided $f'(\alpha) \neq 0$, i.e., the root is simple and $|\varepsilon_{n+1}| \leq \frac{1}{2}\frac{M}{m}\varepsilon_n^2$,

where $M = \max |f''(x)|$, $m = \min |f'(x)|$, $x \in [a,b]$

When α is a multiple root of order r, then

$$x_{n+1} - \alpha = \phi(x_n) - \phi(\alpha)$$

$$= \frac{(x_n - \alpha)^r}{r!}\phi^{(r)}(\xi_n), \text{ where } \xi_n \text{ is a point lies in between } \alpha \text{ and } x_n.$$

or, $|\varepsilon_{n+1}| = \frac{|\varepsilon_n|^r}{r!}|\phi^{(r)}(\xi_n)|$

$$\leq \frac{k_r}{r!}|\varepsilon_n|^r, \text{ where } k_r = \max_{x \in [a,b]} |\phi^{(r)}(x)|.$$

It is then found for Newton-Raphson method, if α is a multiple root of order $r > 1$, for large n,

$$|\varepsilon_{n+1}| \approx \frac{r-1}{r}|\varepsilon_n|, \text{ i.e., convergence is linear.}$$

In such cases, the usual Newton-Raphson method is modified in such a manner that the quadratic convergence is again restored. The method is known as Modified / Accelerated Newton-Raphson method.

Suppose, α is a root of the equation $f(x) = 0$ of order $r > 1$. Then the $(n + 1)$-th iterate x_{n+1} to α is given by $x_{n+1} = x_n - \frac{rf(x_n)}{f'(x_n)}$, which converges to α quadratically.

Some points to note :

Firstly, in any calculation, division by zero should be avoided carefully. Secondly, we should check, whether $f(x)$ has real roots or not. If it has complex roots, the iteration may not converge. Again, if initial approximation x_0 is not considered carefully, then the iteration process may lead to another root. So the initial choice should be taken with care and check the convergence criteria repeatedly. There may also be the case of cycling, i.e., iteration process is cycled near the root but not converges to the root. Here, also the initial choice is important and convergence criteria should be revisited.

Example 5. Consider the function $f(x) = \cos x \cosh x + 1 = 0$ of Example 2. We obtain the root by Newton-Raphson method correct to 8 significant figures as follows :

Here, $f'(x) = \cos x \sinh x - \sin x \cosh x$.

Table :

k	x_k	$f(x_k)$	$f'(x_k)$	$f(x_k)/f'(x_k)$
0	1·8	$2{\cdot}939755852 \times 10^{-1}$	−3·694673552	$7{\cdot}95674046 \times 10^{-2}$
1	1·879567405	$-1{\cdot}8530467 \times 10^{-2}$	−4·165295668	$-4{\cdot}44877590 \times 10^{-3}$
2	1·875118629	$-6{\cdot}0253 \times 10^{-5}$	−4·138222447	$-1{\cdot}4560116 \times 10^{-5}$
3	1·875104069	$-1{\cdot}0 \times 10^{-9}$	−4·138133987	$-2{\cdot}4165 \times 10^{-10}$
4	1·875104069			

Hence the root is 1·8751041 correct to 8 significant figures.

Example 6. Consider the equation $f(x) \equiv x^3 - 3x + 2 = 0$. It has a double root at $x = 1$. We use the Modified Newton-Raphson scheme,

$$x_{k+1} = x_k - \frac{2f(x_k)}{f'(x_k)} = \frac{x_k^3 + 3x_k - 4}{3x_k^2 - 3}.$$

Starting with $x_0 = 1{\cdot}2$, we have the following table.

k	x_k	$x_{k+1} - x_k$	$\varepsilon_k = 1 - x_k$
0	1·2	− 0·1)3939394	− 0·200000000
1	1·006060606	− 0·006054519	− 0·006060606
2	1·000006087	− 0·000006087	− 0·000006087
3	1·000000000	0·000000000	0·000000000

§ Roots of a Polynomial Equation with Real Numerical Co-efficients :

Let $P_n(x) \equiv a_0 x^n + a_1 x^{n-1} + \ldots + a_n = 0$, ... (1)

where $a_0 \neq 0$, and $a_i, i = 0, 1, \ldots, n$ are real,

be a polynomial equation of degree n. As a polynomial of degree n, $P_n(x)$ has exactly n roots, real or complex. Here we should keep in mind that if $p + \sqrt{q}$ is a root of the polynomial equation (1) then $p - \sqrt{q}$ is so also and if $\alpha + i\beta$ is a root, then $\alpha - i\beta$ is also a root. Therefore, an odd degree polynomial equation with real coefficients has at least one real root. Now, maximum how many real roots, a polynomial equation of the type (1) can have, can be conveniently guessed, using Descartes' rule of signs :

The number of positive roots (counting multiplicities) of the polynomial equation $P_n(x) = 0$ with real coefficients, is either equal to the number of sign changes in the sequence of coefficients $a_0, a_1, a_2, \ldots, a_n$ of the equation, or, is less than that number by an even integer. Similarly, the maximum number of negative real roots of the equation (1) is either equal to the number of sign changes in the sequence of coefficients of the equation, $P_n(-x) = 0$ or, is less than that number by an even integer. A more complete way to determining the number of real roots of a polynomial equation in a given interval is the Sturm's method. Now, before going to state the theorem, we first describe the evolution of Sturm's system. Let, in some way, we know that all

roots of $P_n(x) = 0$ lie in the interval $[a,b]$, $a < b$. Now, we calculate $P'_n(x)$ and divide $P_n(x)$ by $P'_n(x)$. Let $R_1(x)$ be the remainder with opposite sign. Then we divide $P'_n(x)$ by $R_1(x)$ and call the remainder with opposite sign as $R_2(x)$. Again we divide $R_1(x)$ by $R_2(x)$ and find $R_3(x)$ in a similar fashion. Since we are dealing with polynomials of finite degree, this process terminates whenever $R_k(x)$ is constant for some k. The sequence, $P_n(x)$, $P'_n(x)$, $R_1(x)$, $R_2(x)$, ..., $R_k(x)$ (= constant), is usually known as Strum's system. Now, we substitute a and b respectively in place of x and calculate the number of sign changes in the Sturm's sequence. Let, $W(a)$ and $W(b)$ be the respective numbers. Then we have the following theorem :

Theorem (Sturm's theorem) : If a, b $(a < b)$ are not roots of a polynomial equation $P_n(x) = 0$ and $P_n(x)$ does not have multiple roots, then $W(a) \geq W(b)$ and $W(a) - W(b)$ is the number of real roots of the polynomial equation in the interval (a,b).

In Netwon-Raphson method we observe that, it is necessary to evaluate the function and its derivative to find a root. Therefore, if we use that method for the polynomial function $P_n(x)$, we must evaluate values of $P_n(x)$ and its derivative for a prescribe value of x. There is algorithm developed for division with linear factor, which is as follows :

Let, for a given point p, we write,

$$P_n(x) = (x-p)P_{n-1}(x) + R \qquad \text{..... (2)}$$

Let, the quotient polynomial $P_{n-1}(x)$ be,

$$P_{n-1}(x) = b_0x^{n-1} + b_1x^{n-2} + ... + b_{n-1} \qquad \text{..... (3)}$$

Now, we take $b_n = R$ and by (1), (2) and (3) we have,

$$a_0x^n + a_1x^{n-1} + ... + a_n = (x-p)(b_0x^{n-1} + ... + b_{n-1}) + b_n$$

Compairing coefficients of both sides we have,

$$b_0 = a_0$$

$$b_j = a_j + pb_{j-1},\ j = 1,2,...,n \qquad \text{..... (4)}$$

and $R = b_n$

Therefore, the value of the polynomial $P_n(x)$ at $x = p$ is given by the division algorithm, i.e., $R = P_n(p)$ ..... (5)

Again, from (2) we have,

$$P_n'(x) = P_{n-1}(x) + (x-p)P_{n-1}'(x) \quad \text{..... (6)}$$

This implies, $P_n'(p) = P_{n-1}(p)$ ..... (7)

i.e., the value of the first derivative of $P_n(x)$ at $x = p$ is given by the quotient polynomial $P_{n-1}(x)$ at $x = p$.

Now, following (2) we can write,

$$P_{n-1}(x) = (x-p)P_{n-2}(x) + R_1, \quad \text{..... (8)}$$

where, $P_{n-2}(x) = c_0 x^{n-2} + c_1 x^{n-3} + \ldots + c_{n-2}$

and $R_1 = c_{n-1}$, say.

Then we have, $R_1 = P_{n-1}(p) = c_{n-1}$

$$c_0 = b_0, \quad \text{..... (9)}$$

$$c_j = b_j + pc_{j-1},\ j = 1, 2, \ldots, n-1$$

So, $P_{n-1}'(p) = P_{n-2}(p)$ and by (6) we have,

$$P_n''(x) = 2P_{n-1}'(x) + (x-p)P_{n-1}''(x) \quad \text{..... (10)}$$

which implies, $P_n''(p) = 2P_{n-1}'(p) = 2P_{n-2}(p)$ ..... (11)

Proceeding as above, we obtain

$$P_n^{(m)}(p) = m!\, P_{n-m}(p),\ m = 1, 2, \ldots, n \quad \text{..... (12)}$$

where $P_0 = a_0$.

The quantities generated as above may be found easily by Horner's scheme :

Coefficients of $P_n \equiv$	a_0	a_1	a_2	.....	a_{n-1}	a_n
		pb_0	pb_1	.....	pb_{n-2}	pb_{n-1}
Coefficients of $P_{n-1} \equiv$	b_0	b_1	b_2	.....	b_{n-1}	$b_n = P_n(p)$
		pc_0	pc_1	.....	pc_{n-2}	
Coefficients of $P_{n-2} \equiv$	c_0	c_1	c_2	.....	c_{n-2}	$c_{n-1} = P'_n(p)$

Now, if we know a zero p_1 of the polynomial $P_n(x)$, then dividing $P_n(x)$ by $(x - p_1)$ we remove a zero from $P_n(x)$ and get a quotient polynomial $P_{n-1}(x)$. Again, if we know another zero p_2, then we divide $P_{n-1}(x)$ by $(x - p_2)$ to remove this zero from $P_{n-1}(x)$. This process of successive deflation of zeros from $P_n(x)$ have some hazards as we actually approximates in every steps the roots and obtain the successive quotient polynomials by division algorithm, in both cases rounding errors inevitably occur. However, simple care may be taken for stable computation. In dividing off a root, the coefficients of deflated or quotient polynomial $P_{n-1}(x) = b_0x^{n-1} + \ldots\ldots + b_{n-1}$, may be computed either in the order $b_0, b_1\ldots\ldots, b_{n-1}$ or, in the reverse order. First one is known as forward deflation and later is backward deflation. The former is numerically stable if the roots are eliminated in increasing order of magnitudes.

Example 7. Let us compute a zero of the polynomial, $P_5(x) = x^5 - 5x^3 + 4x + 1$, for which the approximate value $x_0 = 2$ is known.

The Horner scheme gives,

P_5	=	1	0	− 5	0	4	1	
2 ×			2	4	− 2	− 4	0	
P_4	=	1	2	− 1	− 2	0	1	$= P_5(2)$
2 ×			2	8	14	24		
P_3	=	1	4	7	12	24	$= P'_5(2)$	

With these values we obtain first iterate x_1 to the root by Newton-Raphson method, which is,

$$x_1 = 2 - \frac{1}{24} = 1{\cdot}95833.$$

V repeat the process again, The Horner scheme for x_1 yields.

5	=	1	0	− 5	0	4	1
·95833 ×			1·95833	3·83506	− 2·28134	− 4·46762	− 0·915754
P_4	=	1	1·95833	− 1·16494	− 2·28134	− 0·46762	0·084246
1·95833 ×			1·95833	7·67011	12·7393	20·4802	
P_3	=	1	3·91666	6·50517	10·4580	20·0126	

Then we find second iterate x_2 as,

$$x_2 = 1{\cdot}95833 - {\cdot}00420965 \simeq 1{\cdot}95412.$$

Proceeding as above we obtain $x_3 = 1{\cdot}95408$, rounding upto six significant figures. At this stage, we find, $P_5(1{\cdot}95408) \simeq 0{\cdot}000008$.

§ Bairstow's Method :

In this section, we are going to describe a method of finding roots (even if complex) of polynomials with real coefficients by means of Newton's technique. The implicit idea is as follows : since complex roots appear in pair, therefore, if $\alpha \pm i\beta$ be a pair of roots, then $x^2 - 2\alpha x + \alpha^2 + \beta^2$ is a quadratic factor of the corresponding polynomial, say, $P_n(x)$, where suffix n (≥ 2) designates the degree of the polynomial. So, if we somehow, find a quadratic factor of a polynomial, then from this factor, separation of roots (whether real or complex) is almost an easy task for anyone. Bairstow's method is such a method which is used to find a quadratic factor of a polynomial. Therefore, if we are able to find a quadratic factor, then dividing the polynomial by this factor we obtain another polynomial of degree lower by 2 and repeat the process if the resulting factor has degree greater than 2. Since the method we are going to discuss is an approximate method in the sense that the quadratic factors we obtain are approximates to the actual factors, inherent rounding errors occur. Therefore, if roots are not well-separated, hazards will occur. Let us now describe the method systematically :

Let, $P_n(x) = a_0x^n + a_1x^{n-1} + \ldots\ldots + a_n$, where $a_0 \neq 0$ and $a_0, a_1, \ldots, a_n$ are real. Consider a quadratic expression $x^2 - px - q$ where p, q are real. We divide $P_n(x)$ by $x^2 - px - q$ and write.

$$P_n(x) = (x^2 - px - q)P_{n-2}(x) + b_{n-1}(x - p) + b_n, \qquad \ldots\ldots (13)$$

where $P_{n-2}(x) = b_0x^{n-2} + b_1x^{n-3} + \ldots + b_{n-2}$, say.

Then, comparing coefficients of x^j, $j = 0, 1, \ldots, n$, in (13) we have,

$$\begin{aligned} a_0 &= b_0, \\ a_1 &= b_1 - pb_0, \\ a_2 &= b_2 - pb_1 - qb_0, \qquad \ldots\ldots (14) \\ &\ldots\ldots\ldots\ldots \\ a_j &= b_j - pb_{j-1} - qb_{j-2},\ j = 2, 3, \ldots, n \end{aligned}$$

Therefore, we obtain the coefficients of $P_{n-2}(x)$ and remainder $R_1(x) \equiv b_{n-1}(x-p)+b_n$, by the following recursive scheme,

$$b_0 = a_0$$

$$b_1 = a_1 + pb_0$$

$$b_2 = a_2 + pb_1 + qb_0 \qquad \text{..... (15)}$$

.....

.....

$$b_j = a_j + pb_{j-1} + qb_{j-2},\ j = 2, 3, ..., n$$

Now, $x^2 - px - q$ will be a factor of $P_n(x)$ if and only if $R_1(x)$ is exactly zero, i.e., if and only if $b_{n-1} = b_n = 0$. Since, for given $P_n(x)$, coefficient of $P_{n-2}(x)$ and $R_1(x)$ are uniquely determined, therefore, we shall consider all b_j's as functions of two variables p and q, i.e., the problem of finding a quadratic divisor of $P_n(x)$ is reduced to finding unknowns p, q such that,

$$b_{n-1}(p,q) = 0 = b_n(p,q) \qquad \text{..... (16)}$$

This is equivalent to find roots of two simultaneous equations given by (16). We compute them by Newton's method. Now, for arbitrary p, q, the relation (16) is not satisfied in general. Therefore, we find corrections $p + \Delta p$, $q + \Delta q$ to p, q such that,

$$b_{n-1}(p+\Delta p, q+\Delta q) = 0 = b_n(p+\Delta p, q+\Delta q) \qquad \text{..... (17)}$$

Expanding (17) in Taylor's series and neglecting second and higher order terms in Δp, Δq, we have,

$$b_{n-1}(p,q) + \frac{\partial b_{n-1}}{\partial p}\Delta p + \frac{\partial b_{n-1}}{\partial q}\Delta q = 0$$

..... (18)

and $$b_n(p,q) + \frac{\partial b_n}{\partial p}\Delta p + \frac{\partial b_n}{\partial q}\Delta q = 0$$

which implies,

$$\Delta p = \frac{b_n \frac{\partial b_{n-1}}{\partial q} - b_{n-1} \frac{\partial b_n}{\partial q}}{\frac{\partial b_{n-1}}{\partial p}\frac{\partial b_n}{\partial q} - \frac{\partial b_{n-1}}{\partial q}\frac{\partial b_n}{\partial p}}$$

and $$\Delta p = \frac{b_{n-1} \frac{\partial b_n}{\partial p} - b_n \frac{\partial b_{n-1}}{\partial p}}{\frac{\partial b_{n-1}}{\partial p}\frac{\partial b_n}{\partial q} - \frac{\partial b_{n-1}}{\partial q}\frac{\partial b_n}{\partial p}}$$ ..... (19)

provided the Jacobian, $\frac{\partial b_{n-1}}{\partial p}\frac{\partial b_n}{\partial q} - \frac{\partial b_{n-1}}{\partial q}\frac{\partial b_n}{\partial p} \neq 0$.

Now, from the recursion relations (15), we have for $j = 2, 3, ..., n$,

$$\frac{\partial b_j}{\partial p} = b_{j-1} + p\frac{\partial b_{j-1}}{\partial p} + q\frac{\partial b_{j-2}}{\partial p}$$

$$\frac{\partial b_j}{\partial q} = b_{j-2} + p\frac{\partial b_{j-1}}{\partial q} + q\frac{\partial b_{j-2}}{\partial q}$$ ..... (20)

Also, we have, $\frac{\partial b_0}{\partial p} = 0 = \frac{\partial b_0}{\partial q}$ and $\frac{\partial b_1}{\partial p} = b_0$, $\frac{\partial b_1}{\partial q} = 0$, ... (21)

Let, $c_{j-1} = \frac{\partial b_j}{\partial p}$, $j = 0, 1, 2, ..., n$.

Then, from (20) and (21) we have,

$$c_{-1} = 0,\ c_0 = b_0, c_1 = b_1 + pc_0$$

and $$c_{j-1} = b_{j-1} + pc_{j-2} + qc_{j-3},\ j = 3, 4, ..., n.$$

Therefore, we can find c_j's by he recursive scheme,

$$c_0 = b_0$$

$$c_1 = b_1 + pc_0$$ ..... (22)

.....

$$c_j = b_j + pc_{j-1} + qc_{j-2},\ j = 2, 3, ..., n-1$$

Again, from (20) and (21) we have,

$$\frac{\partial b_j}{\partial q} = b_{j-2} + p\frac{\partial b_{j-1}}{\partial q} + q\frac{\partial b_{j-2}}{\partial q},\ j = 2,3,\ldots,n,$$

and $\dfrac{\partial b_0}{\partial q} = 0 = \dfrac{\partial b_1}{\partial q}$.

Therefore, by (22) we have,

$$\left.\begin{aligned}
&\frac{\partial b_2}{\partial q} = b_0 = c_0\\
&\frac{\partial b_3}{\partial q} = b_1 + pc_0 = c_1\\
&\frac{\partial b_4}{\partial q} = b_2 + pc_1 + qc_0 = c_2\\
&\cdots\cdots\cdots\cdots\cdots\cdots\\
&\frac{\partial b_j}{\partial q} = b_{j-2} + pc_{j-3} + qc_{j-4} = c_{j-2},\ j = 4,5,\ldots,n-2.\\
&\text{and } \frac{\partial b_{n-1}}{\partial p} = c_{n-2} = \frac{\partial b_n}{\partial q},\ \frac{\partial b_{n-1}}{\partial q} = c_{n-3}
\end{aligned}\right\} \quad \ldots\ldots (23)$$

Also, we have, $c_{n-1} = \dfrac{\partial b_n}{\partial p}$.

Hence, replacing partial derivatives on (19) by c_j's we have,

$$\left.\begin{aligned}
\Delta p &= \frac{b_n c_{n-3} - b_{n-1}c_{n-2}}{c_{n-2}^2 - c_{n-1}c_{n-3}}\\
\text{and } \Delta q &= \frac{b_{n-1}c_{n-1} - b_n c_{n-2}}{c_{n-2}^2 - c_{n-1}c_{n-3}}
\end{aligned}\right\} \quad \ldots\ldots (24)$$

Therefore, improved values of p and q are to be found by (24) provided the denominator which is equal to the Jacobian must not vanish. If it vanishes initially, we shift p, q randomly. This process of finding b's and c's are to be repeated until $\Delta p \to 0$ and $\Delta q \to 0$, or, in other words $b_{n-1} \to 0$, $b_n \to 0$, by considering new p, q as $p + \Delta p$, $q + \Delta p$ respectively.

There is an implicit restriction $p^2 + 4q \neq 0$ on the choice of p, q. However, for initial choice of p, q we use two special choices : (i) for large roots,

$$p_0 = -\frac{a_1}{a_0}, q_0 = -\frac{a_2}{a_1} \text{ and (ii) for small roots, if } a_{n-2} \neq 0,$$

$$p_0 = -\frac{a_{n-1}}{a_{n-2}}, \quad q_0 = -\frac{a_n}{a_{n-2}}. \text{ Otherwise, we may start with arbitrary initial}$$

choice of p_0 and q_0.

Hence, in short, the basic steps in finding a quadratic factor $x^2 - px - q$ of a polynomial $P_n(x)$ by Bairstow's method are as follows :

(1) Assuming approximate p, q, we compute b_j's and c_j's by recursive schemes.

(2) If Jacobian is non-zero, we find next iterates of p, q, namely, $p + \Delta p$, $q + \Delta q$ by the formula (24).

(3) The iteration is stopped if b_{n-1}, b_n or, Δp, Δq are sufficiently small in accordance with our tolerance level.

(4) Then we find zeros of quadratic $x^2 - px - q$.

(5) Next we divide $P_n(x)$ by $x^2 - px - q$, i.e., deflation by two zeros $P_n(x)$ is performed. The coefficients of the quotient polynomial are found by the formula (15).

It is to be noted here that Bairstow's method has order of convergence two as it uses the method of Newton-Raphson for system of two non-linear equations. The quantities b_j's and c_j's may be found by double rounded Horner's scheme.

For $n = 6$, we have,

a_0	a_1	a_2	a_3	a_4	a_5	a_6
		qb_0	qb_1	qb_2	qb_3	qb_4
	pb_0	pb_1	pb_2	pb_3	pb_4	pb_5
b_0	b_1	b_2	b_3	b_4	b_5	b_6
		qc_0	qc_1	qc_2	qc_3	
	pc_0	pc_1	pc_2	pc_3	pc_4	
c_0	c_1	c_2	c_3	c_4	c_5	

Example 8. Using Bairstow's method for the polynomial equation $x^4 - 6{\cdot}1x^3 + 9{\cdot}83x^2 + 13{\cdot}755x - 90{\cdot}405 = 0$, we find after five iterations, $\Delta p = -0{\cdot}00000$, $\Delta q = -0{\cdot}00000$. Then we find quadratic factors $(x^2 + 2{\cdot}1x + 8{\cdot}82)$ and $(x^2 + 4{\cdot}0x - 10{\cdot}25)$, as the degree of the polynomial is 4. The roots are $-2{\cdot}1000$, $4{\cdot}2000$, $2{\cdot}0000 \pm i\,1{\cdot}2500$, correct to four decimal places.

§ Nonlinear Systems :

Newton's Method :

Let us first illustrate Newton's method for two unknowns. The generalization is obvious.

$$\text{Let, } f(x, y) = 0 = g(x, y), \qquad \ldots\ldots (25)$$

be a system of two non-linear equations with two unknowns x, y. We assume that both f and g has first order continuous partial derivatives. Let (x_0, y_0) be an approximation to the actual solution (x, y) (an isolated root of (25)). Let (x_k, y_k) and ε_k, η_k be the values of the k-th approximation and error terms respectively.

$$\text{Then, } \left.\begin{aligned} x &= x_k + \varepsilon_k \\ y &= y_k + \eta_k \end{aligned}\right\} \qquad \ldots\ldots (26)$$

If (x_k, y_k) is a sufficiently good approximation to (x, y), then we may neglect squares and higher powers of ε_k, η_k as small quantities. Then, by Taylor series expansion, we have,

$$f(x, y) = f(x_k + \varepsilon_k, y_k + \eta_k) = 0$$

$$\Rightarrow f(x, y) = f(x_k, y_k) + \varepsilon_k \left.\frac{\partial f}{\partial x}\right|_{(x_k, y_k)} + \eta_k \left.\frac{\partial f}{\partial y}\right|_{(x_k, y_k)} = 0 \qquad \ldots\ldots (27)$$

$$\text{and similarly, } g(x, y) = g(x_k, y_k) + \varepsilon_k \left.\frac{\partial g}{\partial x}\right|_{(x_k, y_k)} + \eta_k \left.\frac{\partial g}{\partial y}\right|_{(x_k, y_k)} = 0 \qquad \ldots\ldots (28)$$

Thus we have a system of linear equations (27) and (28) in ε_k, η_k. In matrix form, we can write,

$$\begin{pmatrix} \varepsilon_k \\ \eta_k \end{pmatrix} = -\begin{pmatrix} \left.\frac{\partial f}{\partial x}\right|_{(x_k, y_k)} & \left.\frac{\partial f}{\partial y}\right|_{(x_k, y_k)} \\ \left.\frac{\partial g}{\partial x}\right|_{(x_k, y_k)} & \left.\frac{\partial g}{\partial y}\right|_{(x_k, y_k)} \end{pmatrix}^{-1} \begin{pmatrix} f_k \\ g_k \end{pmatrix}$$

$$= -\frac{1}{|J|_{\text{at}\,(x_k, y_k)}} \begin{pmatrix} fg_y - gf_y \\ gf_x - fg_x \end{pmatrix}_{\text{at}\,(x_k, y_k)} \qquad \ldots\ldots (29)$$

where $f_k = f(x_k, y_k)$, $g_k = g(x_k, y_k)$, provided the Jacobian $|J|_{\text{at}\,(x_k, y_k)} \neq 0$,

i.e. $(f_x g_y - f_y g_x)_{\text{at}\,(x_k, y_k)} \neq 0$,

Therefore, $(k + 1)$ iterates to the solution (x, y) can be found as follows :

$$\left.\begin{aligned} x_{k+1} &= x_k + \varepsilon_k = x_k + \left.\frac{gf_y - fg_y}{f_x g_y - f_y g_x}\right|_{\text{at}\,(x_k, y_k)} \\ \text{and } y_{k+1} &= y_k + \eta_k = y_k + \left.\frac{fg_x - gf_x}{f_x g_y - f_y g_x}\right|_{\text{at}\,(x_k, y_k)} \end{aligned}\right\} \qquad \ldots\ldots (30)$$

for $k = 0, 1, 2, \ldots\ldots$

The generalization of above process to several unknown is obvious. Now to study the convergence of the procedure, we mention the following theorem.

Theorem : Let $f(x, y)$, $g(x, y)$ be two three times continuously differentiable functions within a domain D that contains the solution of the system of equations $f(x, y) = 0 = g(x, y)$, in its interior and the Jacobian $|J|$ is non-vanishing. Them Newton's method has order of convergence at least two.

Now, Newton's method has some advantages as well as disadvantages as compared with the other methods. Main advantages are, it is simple, flexible and good order of convergence. But one of the disadvantages is that three are other methods which are less expensive as far as computational complexity is concerned. One such method is quasi-Newton method.

Quasi-Newton Method :

A simplified approach may be taken, if instead of calculating Jacobian $|J|$ at each stage, we use a constant Jacobian $|J|$ at (x_0, y_0) with good initial approximation

(x_0, y_0). Then $(k+1)$-th iterate (x_{k+1}, y_{k+1}) can be evaluated by the relation (30), just replacing the Jacobian at (x_k, y_k) by the Jacobian at (x_0, y_0). Clearly, in case of several unknowns, this will reduce computational complexity. The speed of convergence is however slower than the previous one but the actual computation time of this modified method is often much less. This procedure is usually known as quasi-Newton method.

Example 9. Consider the system of equations,

$$f(x,y) \equiv x^2 + y^2 + 0{\cdot}6y - 0{\cdot}16 = 0$$

and $g(x,y) \equiv x^2 - y^2 + x - 1{\cdot}6y - 0{\cdot}14 = 0$.

Then, $f_x = 2x$, $f_y = 2y + 0{\cdot}6$, $g_x = 2x + 1$, $g_y = -2y - 1{\cdot}6$.

Let the initial guess of the solution of the system be, $x_0 = 0{\cdot}6$, $y_0 = 0{\cdot}25$. Then by Newton's method we construct the following table.

k	x_k	y_k	ε_k	η_k	error
0	·6000000000	·2500000000	$-2{\cdot}54960 \times 10^{-1}$	$-9{\cdot}68623 \times 10^{-2}$	$3{\cdot}531 \times 10^{-1}$
1	·3450404858	·1531376518	$-6{\cdot}75094 \times 10^{-2}$	$-3{\cdot}06747 \times 10^{-2}$	$8{\cdot}050 \times 10^{-2}$
2	·2775310555	·1224629827	$-5{\cdot}64594 \times 10^{-3}$	$-2{\cdot}79860 \times 10^{-3}$	$6{\cdot}347 \times 10^{-3}$
3	·2718851108	·1196643843	$-4{\cdot}06023 \times 10^{-5}$	$-2{\cdot}10056 \times 10^{-5}$	$4{\cdot}572 \times 10^{-5}$
4	·2718445085	·1196433787	$-2{\cdot}1579 \times 10^{-9}$	$-1{\cdot}1043 \times 10^{-9}$	$2{\cdot}460 \times 10^{-9}$
5	·2718845063	·1196433776			

i.e., the solution is (·27188451, ·11964338) correct to eight significant figures.

Example 10. Solve the above problem by quasi-Newton Method.

Solution : We choose the better initial approximation $x_0 = \cdot 3$, $y_0 = \cdot 1$. Then the Jacobian at (·3, ·1) is $|J|_0 = \begin{vmatrix} \cdot 6 & \cdot 8 \\ 1{\cdot}6 & -1{\cdot}8 \end{vmatrix}$ and we obtain the following table by using quasi-Newton method.

k	x_k	y_k	ε_k	η_k	error
0	·3000000000	·1000000000	$-2{\cdot}71186 \times 10^{-2}$	$2{\cdot}03390 \times 10^{-2}$	$3{\cdot}433 \times 10^{-2}$
1	·2728813559	·1203389831	$-9{\cdot}85495 \times 10^{-4}$	$-6{\cdot}97248 \times 10^{-4}$	$1{\cdot}249 \times 10^{-3}$
2	·2718958608	·1196417355	$-4{\cdot}81441 \times 10^{-5}$	$2{\cdot}92648 \times 10^{-6}$	$5{\cdot}138 \times 10^{-5}$
3	·2718477167	·1196446620	$-3{\cdot}03256 \times 10^{-6}$	$-1{\cdot}25483 \times 10^{-6}$	$3{\cdot}458 \times 10^{-6}$
4	·2718446841	·1196434072	$-1{\cdot}67246 \times 10^{-7}$	$-2{\cdot}64407 \times 10^{-8}$	$1{\cdot}802 \times 10^{-7}$
5	·2718445169	·1196433808	$-9{\cdot}9322 \times 10^{-9}$	$-3{\cdot}0508 \times 10^{-9}$	$1{\cdot}107 \times 10^{-8}$
6	·2718445070	·1196433777	$-6{\cdot}1017 \times 10^{-10}$	$-4{\cdot}2373 \times 10^{-11}$	$7{\cdot}07 \times 10^{-10}$
7	·2718445064	·1196433777			

Clearly, the process is slower than the above method.

§ Summary :

In this unit, a non-linear equation of the form $f(x) = 0$ is considered. Different methods, viz., bisection, regula falsi, fixed point iteration and Newton-Raphson are used to solve such equation. The rate of convergence and condition of convergence of these methods are studied. These methods are generally used to determined a single root of the equation. To find all the roots of a polynomial equation, several methods are available. Here Bairstow's method is discussed to find all the roots of a polynomial equation. Also, Newton's method is discussed to solve a system of non-linear equations.

EXERCISES

1. Using bisection and regula-falsi methods find the roots of the following equations correct to five significant figures.

(i) $e^{2x} - \sin x - 2 = 0$ in the interval (0, 1)

(ii) $x + \ln x = 2$ in the interval (1, 2)

(iii) $x - \cdot 2 \sin x - \cdot 5 = 0$ in the interval (·5, 1).

2. Find smallest positive roots of the following equations correct to six significant figures by (i) fixed point iteration, (ii) Newton-Raphson and (iii) regula-falsi method.

(a) $2x - \sin x - \cdot 5 = 0$

(b) $\tan x - \tanh x = 0$

(c) $\tan x - x - 4x^3 = 0$

(d) $e^{2x} - e^x - 2 = 0$.

3. Show that modified Newton-Raphson method converges quadratically. Using this method find a double root close to zero of the equation $x^5 - 7x^4 + 10x^3 + 10x^2 - 7x + 1 = 0$.

4. Use Bairstow's method to find zeros of the polynomials correct to four significant figures.

(i) $x^5 - 2x^4 + 3x^3 - 12x^2 + 18x - 12$

(ii) $x^5 - 2x^4 - 13x^3 + 14x^2 + 24x - 1$

(iii) $x^4 - 9{\cdot}00x^3 + 29{\cdot}08x^2 - 39{\cdot}52x + 18{\cdot}82$

(iv) $x^5 - 5x^3 + 4x + 1$.

5. Solve the following system of equations by Newton's method correct to six significant figures.

$e^{xy} + x^2 + y - 1{\cdot}2 = 0$

$x^2 + y^2 + x - {\cdot}55 = 0$, near $(0{\cdot}6, 0{\cdot}5)$.

Then solve it by using quasi-Newton method with better initial choice $({\cdot}4, {\cdot}25)$.

6. Solve the following systems by Newton's or quasi-Newton's method.

(i) $4{\cdot}72 \sin 2x - 3{\cdot}14e^y - {\cdot}495 = 0$,

$3{\cdot}61 \cos 3x + \sin y - {\cdot}402 = 0$, near $(1{\cdot}5, -4{\cdot}6)$

(ii) $x - \sinh y = 0$

$2y - \cosh x = 0$ near $({\cdot}6, {\cdot}6)$.

7. Find an iterative scheme to compute $\sqrt{35}$ correct to six significant digit.

8. Show that the scheme $x_{n+1} = \dfrac{x_n(x_n^2 + 3a)}{3x_n^2 + a}$, is a third order method to compute $\sqrt{a}, a > 0$.

Unit 5 ❐ Polynomial Interpolation

§ Objectives

After finishing this unit you will be able to learn about—

- Interpolation
- Uniqueness of interpolation polynomial
- Central difference interpolation due to Gauss
- Piecewise polynomial interpolation
- Cubic splines interpolation
- Inverse interpolation

Before going to any specific method we first mention two theorems about polynomial interpolation.

Let $(x_i, y_i), i = 0, 1, 2, ..., n$ be $(n + 1)$ arbitrary points where each y_i is a value of a function $f(x)$ at x_i, i.e., $f(x_i) = y_i, \forall i = 0, 1, 2, ..., n$. Then we have the following theorem.

Theorem : Given $(n + 1)$ distinct points $x_0, x_1, ..., x_n$ and $(n + 1)$ ordinates $y_0, y_1, ..., y_n$ there is a unique polynomial $P(x)$ that interpolates y_i at x_i $\forall i = 0, 2, ..., n$ (i.e., $P(x_i) = y, \forall i = 0, 1, ..., n$) whose degree is at most n.

Proof. Existence : Such a polynomial exists that can be shown by using Lagrange's polynomials $L_i(x)$ which are defined by

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \forall i = 0, 1, 2, ..., n.$$

Clearly, $L_i(x_j) = \delta_{ij}$.

Now, $P_n(x) \equiv \sum_{i=0}^{n} y_i L_i(x)$ is a polynomial of degree at most n and $P_n(x_j) = \sum_{i=0}^{n} y_i L_i(x_j) = \sum_{i=0}^{n} y_i \delta_{ij} = y_j, \forall j = 0, 1, ..., n$. That completes the proof of existence.

Uniqueness : Suppose $q_n(x)$ is a second polynomial of degree $\leq n$, with the property that $q_n(x_j) = y_j, \forall j = 0, 1, ..., n$.

Now $\hat{P}_n = P_n - q_n$ is a polynomial of degree $\leq n$ and $\hat{P}_n(x_j) = 0, \forall j = 0, 1, 2, ..., n$. However, by the fundamental theorem of algebra, a polynomial of degree n has at most n roots unless it is identically zero, i.e., $\hat{P}_n(x) \equiv 0$.

Hence, $P_n = q_n$.

Next theorem is about the error estimate in polynomial interpolation.

Theorem : Let $f(x) \in C^{n+1}[a,b]$ be a real-valued function and $P_n(x)$ be the interpolation polynomial for $f(x)$ w.r.t, $(n + 1)$ distinct points $x_i \in [a,b], i = 0(1)n$. Then for each $x \in [a,b]$, there exists a point $\xi = \xi(x)$ in the open interval $\min_i(x_i, x) < \xi < \max_i(x_i, x)$ such that,

$$f(x) - P_n(x) \equiv R_{n+1}(x) = \omega_n(x) f^{(n+1)}(\xi) / (n+1)!,$$

where $\omega_n(x) \equiv (x - x_0)(x - x_1).....(x - x_n)$.

Proof : Consider the function $g(x)$ defined by

$$g(x) = f(x) - P_n(x) - \frac{\omega(x)}{\omega(t)}[f(t) - P_n(t)],\ x \in [a,b] \qquad \text{..... (1)}$$

where t is an arbitrarily fixed point of $[a, b]$ other than the interpolation points x_i, $i = 0(1)n$.

Clearly, $g(x) \in C^{n+1}[a,b]$ and $g(t) = 0 = g(x_i), i = 0(1)n$. i.e., $g(x) = 0$, for $x \in \{x_0, x_1, ..., x_n, t\}$. ..... (2)

By repeated application of Rolle's theorem we find that there is some point ξ in (a, b), i.e., in the open interval $I = (m, M) \subset (a,b)$ at which

$$g^{(n+1)}(\xi) = 0 \qquad \text{..... (3)}$$

where $m \equiv \min\{x_0, x_1, ..., x_n, t\}$, $M \equiv \max\{x_0, x_1, ..., x_n, t\}$.

Since $P_n(x)$ is polynomial of degree n and $\omega(x)$ is a monic polynomial (i.e. a polynomial in which the coefficient of the highest degree term is unity) of degree $n + 1$, the last equation, i.e., $g^{(n+1)}(\xi) = 0$ implies that,

$$0 = f^{(n+1)}(\xi) - 0 - \frac{(n+1)!}{\omega(t)}[f(t) - P_n(t)],$$

i.e., $f(t)-P_n(t)=\dfrac{\omega(t)f^{(n+1)}(\xi)}{(n+1)!}$.

Therefore, as $t\in[a,b]$ is arbitrary, changing to x we have,

$$R_{n+1}(x)=f(x)-P_n(x)=\frac{\omega(x)f^{(n+1)}(\xi)}{(n+1)!}, \qquad \ldots\ldots (4)$$

$\forall x\in[a,b]$ and $x\neq x_i$, $i=0(1)n$, where $\xi=\xi(x)\notin(a,b)$.

Note : Although it was assumed that $t\in\{x_0,x_1,\ldots,x_n\}$, (4) is valid for $\forall x\in[a,b]$, because $R_{n+1}(x_i)=0$ and ξ for these x_i's may be picked up arbitrarily.

Now we discuss some specific methods of finding interpolating polynomial.

§ Central Difference Interpolation Formulas – Gauss :

Suppose $y=f(x)$ is an well-defined given function and $y_i=f(x_i)$, $x_i=x_0+ih$, $i=0,\pm1,\pm2,\ldots,\pm n$, where h is a positive constant and is called the argument spacing or step-length.

We have to construct a polynomial $P(x)$ of degree at most $2n$ which interpolates the values y_i of $f(x)$ at the knots or nodal points x_i, i.e., subject to the conditions, $P(x_i)=y_i$, $\forall i=0,\pm1,\ldots,\pm n$. $\ldots\ldots (5)$

Therefore, $\Delta y_i\equiv y_{i+1}-y_i=P(x_{i+1})-P(x_i)=\Delta P(x_i)$.

Δ is called the forward difference operator and Δy_i is called the first order forward difference at x_i.

Similarly, $\Delta^2 y_i=\Delta(\Delta y_i)=\Delta(y_{i+1}-y_i)=\Delta y_{i+1}-\Delta y_i$

$=y_{i+2}-2y_{i+1}+y_i$,

is the second order forward difference at x_i, and so on. The k-th order forward difference at x_i is,

$\Delta^k y_i=\Delta^{k-1}y_{i+1}-\Delta^{k-1}y_i$, where k is an interger ≥ 1.

Now consider $P(x)$ as follows :

$P(x)=a_0+a_1(x-x_0)+a_2(x-x_0)(x-x_1)+a_3(x-x_{-1})(x-x_0)(x-x_1)+a_4(x-x_{-1})(x-x_0)(x-x_1)(x-x_2)+a_5(x-x_{-2})(x-x_{-1})(x-x_0)(x-x_1)(x-x_2)+\ldots\ldots+a_{2n}(x-x_{-(n-1)})(x-x_{-(n-2)})\ldots(x-x_0)\ldots(x-x_{n-1})(x-x_n)$ $\ldots\ldots (6)$

where a_i, $i=0,1,\ldots,2n$ are the coefficients we want to find under constraints (5).

Clearly, $P(x_0) = y_0 = a_0$,

$P(x_1) = y_1 = a_0 + a_1(x_1 - x_0) = a_0 + ha_1$,

$P(x_{-1}) = y_{-1} = a_0 + a_1(x_{-1} - x_0) + a_2(x_{-1} - x_0)(x_{-1} - x_1)$

$= a_0 - ha_1 + 2h^2 a_2$.

$P(x_2) = y_2 = a_0 + 2ha_1 + 2h^2 a_2 + 6h^3 a_3$.

.....

.....

Then, we have $a_0 = y_0$

$$a_1 = \frac{\Delta y_0}{h},$$

$$a_2 = \frac{\Delta^2 y_{-1}}{2!h^2},$$

$$a_3 = \frac{\Delta^3 y_{-1}}{3!h^3},$$

$$a_4 = \frac{\Delta^4 y_{-2}}{4!h^4}$$

.....

.....

$$a_{2n-1} = \frac{\Delta^{2n-1} y_{-(n-1)}}{(2n-1)!h^{2n-1}},$$

and $$a_{2n} = \frac{\Delta^{2n} y_{-n}}{(2n)!h^{2n}}$$

Defining, u by, $u = \frac{x - x_0}{h}$, i.e. by $x = x_0 + hu$, we have,

$$P(x) = y_0 + u\Delta y_o + \frac{u(u-1)}{2!}\Delta^2 y_{-1} + \frac{(u+1)u(u-1)}{3!}\Delta^3 y_{-1}$$

$$+ \frac{(u+1)u(u-1)(u-2)}{4!}\Delta^4 y_{-2}$$

$$+ ..... + \frac{(u+n-1)...(u-n+1)}{(2n-1)!}\Delta^{2n-1} y_{-(n-1)}$$

$$+\frac{(u+n-1)\ldots(u-n)}{2n!}\Delta^{2n}y_{-n}.$$

$$= y_0 + \binom{u}{1}\Delta y_0 + \binom{u}{2}\Delta^2 y_{-1} + \binom{u+1}{3}\Delta^3 y_{-1} + \binom{u+1}{4}\Delta^4 y_{-2} + \ldots\ldots +$$

$$\binom{u+n-1}{2n-1}\Delta^{2n-1}y_{-(n-1)} + \binom{u+n-1}{2n}\Delta^{2n}y_{-n}. \qquad \ldots\ldots (7)$$

The formula (7) is known as Gauss' forward interpolation formula without the remainder. The central differences $\Delta y_0, \Delta^2 y_{-1}$ ..., etc,, may be found by constructing a difference table to follows :

x	y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$	.....
x_{-n}	y_{-n}					
		Δy_{-n}				
$x_{-(n-1)}$	$y_{-(n-1)}$		$\Delta^2 y_{-n}$			
		$\Delta y_{-(n-1)}$		$\Delta^3 y_{-n}$		
$x_{-(n-2)}$	$y_{-(n-2)}$		$\Delta^2 y_{-(n-1)}$		$\Delta^4 y_{-n}$	.....
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
x_{-2}	y_{-2}					
		Δy_{-2}				
x_{-1}	y_{-1}		$\Delta^2 y_{-2}$			
		Δy_{-1}		$\Delta^3 y_{-2}$		
x_0	y_0		$\Delta^2 y_{-1}$		$\Delta^4 y_{-2}$	
		Δy_0		$\Delta^3 y_{-1}$	$\vdots$	
x_1	y_1	Δy_1	$\Delta^2 y_0$	$\vdots$	$\vdots$	
x_2	y_2	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\Delta^4 y_{n-1}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\Delta^3 y_{n-1}$		
x_{n-1}	y_{n-1}		$\Delta^2 y_{n-1}$			
		Δy_{n-1}				
x_n	y_n					

Again taking,

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_{-1}) + a_3(x - x_{-1})(x - x_0)(x - x_1)$$
$$+ \ldots\ldots + a_{2n-1}(x - x_{-(n-1)})\ldots\ldots(x - x_0)\ldots\ldots(x - x_{n-1})$$
$$+ a_{2n}(x - x_{-n})\ldots\ldots(x - x_0)\ldots\ldots(x - x_{n-1}) \quad \ldots\ldots (8)$$

and proceeding as above we get,

$$P(x) = y_0 + u\,\Delta y_{-1} + \binom{u+1}{2}\Delta^2 y_{-1} + \binom{u+1}{3}\Delta^3 y_{-2} + \binom{u+2}{4}\Delta^4 y_{-2}$$

$$+ \ldots\ldots + \binom{u+n-1}{2n-1}\Delta^{2n-1} y_{-n} + \binom{u+n}{2n}\Delta^{2n} y_{-n} \quad \ldots\ldots (9)$$

The formula (9) is known as Gauss' backward interpolation formula. Differences can also be found by constructing a difference table as shown above.

The remainder term of the two formulas is as follows :

$$R(x) = \binom{u+n}{2n+1} h^{2n+1} f^{2n+1}(\xi), \text{ where } a < \xi < b \quad \ldots\ldots (10)$$

$[a, b]$ is the domain of the function where we are interested to find the interpolating polynomial for $f(x)$, i.e., take $a = \min\{x, x_{-n}, x_n\}$ and $b = \max\{x, x_{-n}, x_n\}$.

Both the formulas are quivalent, if we consider the nodes $x_0, x_{\pm1}, x_{\pm2}, \ldots, x_{\pm n}$. But, if we have the nodes $x_0, x_{\pm1}, x_{\pm2}, \ldots, x_{\pm n}, x_{n+1}$, then forward formula is preferable and in case of the nodes $x_0, x_{\pm1}, x_{\pm2}, \ldots, x_{\pm n}, x_{-(n+1)}$, the backward formula is preferable. In the first case, better result is expected if we calculate value of $f(x)$ at x between x_0 and x_1 and for the later case, between x_{-1} and x_0. The formulas are not of frequent use but useful for deriving other formulas.

§ Piecewise Polynomial Interpolation :

Let $\Delta \equiv \{a = x_0 < x_1 < \ldots\ldots < x_n = b\}$ be a partition of the interval $[a, b]$ and y_i's, $= 0(1)n$, the values of a function $f(x)$, given at the nodes $x_0, x_1, \ldots\ldots, x_n$, i.e., $y_i = f(x_i)$, $i = 0(1)n$.

Then a real function $S(x)$ is said to be a piecewise polynomial of degree k on Δ if in each subinterval $[x_{i-1}, x_i]$ it is a k-th degree polynomial, say, $S_i(x)$, $i = 0(1)n$. The function $S(x)$ is called piecewise interpolating polynomial for $f(x)$ on $[a,b]$, if $S(x_i) = y_i$, $\forall i = 0, 1, \ldots\ldots n$, i.e., in piecewise polynomial interpolation, in each subinterval $[x_{i-1}, x_i]$ of $[a, b]$, we fit a polynomial $S_i(x)$ of certain degree and the interpolation

polynomial $S(x)$ consists of polynomials $S_i(x)$, $i = 0(1)n$, pieced together at the knots x_i, $i = 0(1)n$. Splines are examples of most useful piecewise interpolating polynomials. Here we consider only cubic splines.

§ Cubic Splines :

Definition : A Cubic spline $S(x)$ on Δ is a real-valued function from $[a, b] \to R$ with the properties :

(i) $S \in C^2[a,b]$, i.e., S is twice continuously differentiable function on $[a, b]$,

(ii) $S(x)$ coincides with a polynomial of degree 3 on each subinterval $[x_{i-1}, x_i]$, $i = 0(1)n$.

And, S is called a cubic spline interpolant to f if it also satisfies $S(x_i) = y_i = f(x_i)$, $i = 0(1)n$.

More specifically, $S(x)$ is an interpolating cubic spline for $f(x)$ on $[a, b]$ if and only if

(i) each $S_i(x)$ is a cubic polynomial in x on the subinterval $[x_{i-1}, x_i]$ of $[a, b]$, for $i = 0(1)n$.

(ii) $S(x_i) = y_i$, for $i = 0(1)n$.

(iii) $S_i(x_i) = S_{i+1}(x_i)$, for $i = 1(1)n - 1$, i.e. continuous at each nodal points x_1, x_2, ..., x_{n-1}.

(iv) $S_i'(x_i) = S_{i+1}'(x_i)$, for $i = 1(1)n - 1$, i.e., splines are smooth functions.

(v) $S_i''(x_i) = S_{i+1}''(x_i)$, for $i = 1(1)n - 1$, i.e., second derivatives at each x_1, ... x_{n-1} are continuous.

An interpolating cubic spline $S(x)$ as defined involves $4n$ parameters, 4 for each of n cubic polynomials $S_i(x)$ and so $S(x)$ cannot be uniquely determined by the $4n - 2$ conditions mentioned in (i) – (v). We need two more additional conditions. One of the following conditions known as end conditions/side conditions/boundary conditions, is most commonly used :

$$\left.\begin{array}{ll} \text{(I)} & S'(a) = y_0',\ S'(b) = y_n', \text{ for given number } y_0', y_n'. \\ \text{(II)} & S''(a) = S''(b) = 0. \\ \text{(III)} & S^{(k)}(a) = S^k(b) \text{ with } y_n = y_0,\ k = 0,1,2. \end{array}\right\} \quad \ldots \text{(vi)}$$

Conditions (i)-(v) together with one of the three end conditions (vi) determine S uniquely on $[a, b]$.

The Cubic spline S determined under side condition

(I) is called clamped cubic spline,

(II) is called natural cubic spline,

(III) is called periodic spline.

Construction of Cubic spline function :

Let us consider,

$$S_i(x) = a_i(x - x_{i-1})^3 + b_i(x - x_{i-1})^2 + c_i(x - x_{i-1}) + d_i \text{ for } i = 1, 2, \ldots, n, \quad \ldots\ldots (1)$$

(as $S_i(x)$, $i = 1, 2, \ldots, n$ are cubic polynomials), where a_i, h_i, c_i, d_i, $i = 1, 2, \ldots, n$ are $4n$ constants that we have to find under the above mentioned conditions (i)-(vi).

Let us denote, $M_i = S''(x_i), i = 0(1)n$ and $h_i = x_i - x_{i-1}, i = 1(1)n$ ..... (2)

$M_i's$ are usually referred to as the moments of $S(x)$.

Now, conditions (ii), i.e., $S(x_i) = y_i$, $i = 0(1)n$, implies,

$$\left.\begin{aligned} & S_i(x_{i-1}) = y_{i-1} = d_i, i = 1(1)n, \\ \text{and} \quad & S_n(x_n) = a_n h_n^3 + b_n h_n^2 + c_n h_n + d_n = y_n \end{aligned}\right\} \quad \ldots\ldots (3)$$

Condition (iii) i.e., $S_{i+1}(x_i) = S_i(x_i), i = 1(1)n-1$, implies,

$$d_{i+1} = a_i h_i^3 + b_i h_i^2 + c_i h_i + d_i = y_i, i = 1(1)n-1 \quad \ldots\ldots (4)$$

Again, by (1), $S_i'(x) = 3a_i(x - x_{i-1})^2 + 2b_i(x - x_{i-1}) + c_i, i = 1(1)n$.

Therefore, by condition (iv), i.e., $S_i'(x_i) = S_{i+1}'(x_i), i = 1(1)n-1$,

We have, $3a_i h_i^2 + 2b_i h_i + c_i = c_{i+1}, i = 1(1)n-1$, ... (5)

Next $S_i''(x) = 6a_i(x - x_{i-1}) + 2b_i, i = 1(1)n$ and condition (v) implies,

$$S_i''(x_i) = 6a_i h_i + 2b_i = S_{i+1}''(x_i) = 2b_{i+1}, i = 1(1)n-1.$$

$$\text{i.e., } a_i = \frac{2(b_{i+1} - b_i)}{6h_i}, i = 1(1)n-1$$

Now, by (2), $M_i = S''_{i+1}(x_i), i = 0(1)n-1$

and $M_n = S''_n(x_n)$. ..... (7)

so that, $M_i = 2b_{i+1}, i = 0(1)n-1$,

and $M_n = 6a_n h_n + 2b_n$, i.e., $a_n = \dfrac{M_n - 2b_n}{6h_n}$ ..... (8)

Hence, by (3), (4), (6) and (8), we have,

$$\left.\begin{aligned} a_i &= \frac{M_i - M_{i-1}}{6h_i}, i = 1(1)n, \\ b_i &= \frac{1}{2}M_{i-1}, i = 1(1)n, \\ c_i &= \frac{1}{h_i}(y_i - y_{i-1}) - \frac{1}{6}h_i(M_i + 2M_{i-1}), i = 1(1)n, \\ d_i &= y_{i-1}, i = 1(1)n \end{aligned}\right\} \quad \text{..... (9)}$$

Thus S has been characterized by its moments M. Therefore using (9), we have from (5), a set of $n-1$ linear equations with $n+1$ unknowns, $M_0, M_1, \ldots, M_n$, which are as follows :

$$\frac{1}{h_{i+1}}(y_{i+1} - y_i) - \frac{1}{6}h_{i+1}(M_{i+1} + 2M_i) = \frac{1}{h_i}(y_i - y_{i-1}) - \frac{1}{6}h_i(M_i + 2M_{i-1})$$

$$+ 3h_i^2\frac{(M_i - M_{i-1})}{6h_i} + 2h_i \cdot \frac{1}{2}M_{i-1}, \text{ for } i = 1(1)n-1,$$

or $$\frac{h_i}{6}M_{i-1} + \frac{2(h_i + h_{i+1})}{6}M_i + \frac{h_{i+1}}{6}M_{i+1} = \frac{1}{h_{i+1}}(y_{i+1} - y_i) - \frac{1}{h_i}(y_i - y_{i-1}),$$

or, $$h_i M_{i-1} + 2(h_i + h_{i+1})M_i + h_{i+1}M_{i+1}$$

$$= \frac{6}{h_{i+1}}(y_{i+1} - y_i) - \frac{6}{h_i}(y_i - y_{i-1}), i = 1(1)n-1 \quad \text{..... (10)}$$

Two additional equations can be obtained separately from each of the end conditions (I), (II) and (III) of equation (vi).

Case I. Clamped cubic spline :

Here we specify the values $S'(a) = y'_0$ and $S'(b) = y'_n$.

Therefore,

$$y'_0 = c_1 = \frac{1}{h_1}(y_1 - y_0) - \frac{1}{6}h_1(M_1 + 2M_0)$$

or $\quad 2h_1M_0 + h_1M_1 = \dfrac{6}{h_1}(y_1 - y_0) - 6y_0'$ ..... (11)

and $\quad y_n' = S'(b) = S_n'(x_n) = 3a_nh_n^2 + 2b_nh_n + c_n$

$$= \frac{1}{h_n}(y_n - y_{n-1}) + \frac{h_n}{6}(2M_n + M_{n-1})$$

or, $\quad h_nM_{n-1} + 2h_nM_n = 6y_n' - \dfrac{6}{h_n}(y_n - y_{n-1})$ ..... (12)

Case II. Natural cubic spline : Here we have, $S''(a) = 0 = S''(b)$,

i.e., $M_0 = 0 = M_n$ ..... (13)

Similarly for Case III of (vi), i.e., periodic spline we get two other relations except (10).

Now, for clamped cubic splines we have a system of $(n + 1)$ linear equations with $(n + 1)$ unknowns, M_i, $i = 0(1)n$. In matrix form, we may write (10) together with (11) and (12), as $AM = D$, ... (14)

where, $A = \begin{bmatrix} 2h_1 & h_1 & 0 & 0 & 0 & \dots & \dots & \dots & 0 \\ h_1 & 2(h_1+h_2) & h_2 & 0 & 0 & \dots & \dots & \dots & 0 \\ 0 & h_2 & 2(h_2+h_3) & h_3 & 0 & \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 & \dots & h_{n-1} & 2(h_{n-1}+h_n) & h_n \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & h_n & 2h_n \end{bmatrix}$ .....(15)

and $\quad M = [M_0M_1M_2 \dots M_n]^T$

$$D = \left[\frac{6}{h_1}(y_1 - y_0) - 6y_0', \frac{6}{h_2}(y_2 - y_1) - \frac{6}{h_1}(y_1 - y_0), \dots \frac{6}{h_n}(y_n - y_{n-1}) \right.$$

$$\left. \frac{6}{h_{n-1}}(y_{n-1} - y_{n-2}), 6y_n' - \frac{6}{h_n}(y_n - y_{n-1}) \right]^T,$$

[T represents matrix transposition].

The symmetric matrix A is diagonally dominant and hence non-singluar. Therefore, the system $AM = D$ is uniquely solvable.

Note that A is a tridiagonal matrix and hance $AM = D$ can be solved efficiently.

Example 1. Find the clamped cubic spline that passes through (0, 0), (0·5, 0·5), (1·0, 1·5) and (1·5, 2·0) with $S'(0) = 0·5$ and $S'(1·5) = -1$.

Solution : Here, $h_1 = h_2 = h_3 = 0·5$, Therefore, matrix

$$A = \begin{bmatrix} 1 & ·5 & 0 & 0 \\ ·5 & 2 & ·5 & 0 \\ 0 & ·5 & 2 & ·5 \\ 0 & 0 & ·5 & 1 \end{bmatrix}, M = [M_0 M_1 M_2 M_3]^T$$

and $$D = \left[\frac{6}{·5}(·5-0) - 6.0·5, \frac{6}{·5}(1·5-·5) - \frac{6}{·5}(·5-0), \frac{6}{·5}(2-1·5)\right.$$

$$\left. -\frac{6}{·5}(1·5-·5), 6(-1) - \frac{6}{·5}(2-1·5)\right]^T$$

$$= [3 \;\; 6 \;\; -6 \;\; -12]^T$$

For the system $AM = D$ we use Gaussin algorithm which yields, $M_0 = 1·6$, $M_1 = 2·8$, $M_2 = -0·8$, $M_3 = -11·6$. Hence, the requird cubic spline is :

$$S(x) = \begin{cases} S_1(x) = 0·4x^3 + 0·8x^2 + 0·5x, 0 \le x \le 0·5, \\ S_2(x) = -1·2(x-·5)^3 + 1·4(x-·5)^2 + 1·9(x-·5) + 0·5, 0·5 \le x \le 1, \\ S_3(x) = -3·6(x-1)^3 - 0·4(x-1)^2 - 0·1(x-1) + 1·5, 1 \le x \le 1·5 \end{cases}$$

Example 2. Find a natural cubic spline for the above example, i.e., given, $S''(0) = 0 = S''(1·5)$.

Solution : By equation (10), for $i = 1, 2$, we have,

$$·5M_0 + 2M_1 + ·5M_2 = 6,$$

and $$·5M_1 + 2M_2 + ·5M_3 = -6,$$

Also, by $S''(0) = S''(1·5) = 0$, we have $M_0 = 0 = M_3$.

Therefore, $2M_1 + ·5M_2 = 6$.

and $·5M_1 + 2M_2 = -6,$

$\Rightarrow$ $M_1 = 4, M_2 = -4.$

Hence, the natural cubic spline is :

$$S(x) = \begin{cases} S_1(x) = \frac{4}{3}x^3 + 0.x^2 + \frac{2}{3}x, \ 0 \le x \le .5, \\ S_2(x) = -\frac{8}{3}(x-.5)^3 + 2(x-.5)^2 + \frac{5}{3}(x-.5) + .5, \ .5 \le x \le 1, \\ S_3(x) = \frac{4}{3}(x-1)^3 - 2(x-1)^2 + \frac{5}{3}(x-1) + 1.5, \ 1 \le x \le 1.5 \end{cases}$$

§ Inverse Interpolation

In interpolation, for a given set of values of x and y, the value of y is determined for a given value of x. But the inverse interpolation is the process which finds the value of x for a given y. Commonly used inverse interpolation formulae are based on successive iteration.

In the following the inverse interpolation based on Lagrange's formula is discussed.

Inverse interpolation based on Lagrange's formula

The Lagrange's interpolation formula of y on x is

$$y = \sum_{i=0}^{n} \frac{w(x)y_i}{(x - x_i)w'(x_i)}$$

When x and y are interchanged then the above relation changes to

$$x = \sum_{i=0}^{n} \frac{w(y)x_i}{(y - y_i)w'(y_i)} = \sum_{i=0}^{n} L_i(y)x_i$$

where

$$L_i(y) = \frac{w(y)}{(y - y_i)w'(y_i)} = \frac{(y-y_0)(y-y_1)\ldots(y-y_{i-1})(y-y_{i+1})\ldots(y-y_n)}{(y_i-y_0)(y_i-y_1)\ldots(y_i-y_{i-1})(y_i-y_{i+1})\ldots(y_i-y_n)}$$

This formula gives the value of x for given value of y and the formula is known as **Lagrange's inverse interpolation** formula.

Example 1. Given $y = f(x)$ in the following table.

x	:	10	15	17
y	:	3	7	11

Find the values of x when $y = 10$ and $y = 5$.

Solution. Here, inverse Lagrange's interpolation formula is used in the following form

$$\phi(y) = \sum_{i=0}^{n} L_i(y) x_i$$

The Lagrangian functions

$$L_0(y) = \frac{(y-y_1)(y-y_2)}{(y_0-y_1)(y_0-y_2)} = \frac{(y-7)(y-11)}{(3-7)(3-11)} = \frac{y^2-18y+77}{32}$$

$$L_1(y) = \frac{(y-y_0)(y-y_2)}{(y_1-y_0)(y_1-y_2)} = \frac{(y-3)(y-11)}{(7-3)(7-11)} = \frac{y^2-14y+33}{-16}$$

and $$L_2(y) = \frac{(y-y_0)(y-y_1)}{(y_2-y_0)(y_2-y_1)} = \frac{(y-3)(y-7)}{(11-3)(11-7)} = \frac{y^2-10y+21}{32}$$

Then $$\phi(y) = \frac{y^2-18y+77}{32} \times 10 - \frac{y^2-14y+33}{16} \times 15 + \frac{y^2-10y+21}{32} \times 17$$

$$= \frac{1}{32}\left(137 + 70y - 3y^2\right)$$

Hence, $$x(10) = \phi(10) = \frac{1}{32}(137 + 700 - 300) = 16.78125$$

and $$x(5) = \phi(5) = \frac{1}{32}(137 + 350 - 75) = 12.87500$$

§ Summary :

The concept of polynomial interpolation and its existence and uniqueness are introduced here. One central difference polynomial interpolation formula due to Gauss is described. It is well known that the degree of the interpolating polynomial is one less than the number of given points, and hence it is difficult to evaluate the polynomial for a given value of x. This difficulty has been removed in cubic splines interpolation.

The degree of this polynomial is almost three. In interpolation conventionally, the value of y is determined for a given value of x. Some times it is required to determine the value of x when y is given. This can be done by inverse interpolation, discussed in this unit.

EXERCISES

1. Prove that the polynomial of degree $\leq n$ which interpolates a polynomial $f(x)$ of degree $\leq n$, at $n + 1$ distinct point is $f(x)$ itself.

2. Calculate the value $f(1{\cdot}0)$ using the following data by (i) Gauss's forward formula, (ii) Gauss' backward formula.

x	0·5	0·7	·9	1·1	1·3	1·5	1·7
$f(x)$	1·47943	1·64422	1·78333	1·89121	1·96356	1·99749	1·99166

3. Calcutate the values $f(1{\cdot}41)$ and $f(1{\cdot}49)$ by Gauss' forward and backward formula respectively, using the following data.

x	1·1	1·2	1·3	1·4	1·5	1·6	1·7	1·8
$f(x)$	·69121	·73204	·76356	·78545	·79749	·79957	·79166	·77385

4. Show that given x_0, x_1, x_2, such that $x_0 < x_1 < x_2$ and given a_0, a_1, a_2 and b_0, b_1, b_2, there exists a unique polynomial $P(x)$ of degree four or less such that $P(x_0) = a_0$, $P(x_1) = a_1$, $P(x_2) = a_2$,, $P'(x_0) = b_0$, $P'(x_1) = b_1$, $P'(x_2) = b_2$. Find, $P(x_2)$ in the case $x_2 - x_1 = x_1 - x_0 = h$.

5. Find the unique polynomal $P(x)$ of degree four or less such that $P(0) = 1$, $P(1) = -2$, $P(2) = 0$, $P'(0) = 2$, $P'(1) = 3$ and $P'(2) = 2$.

6. Find a polynomial $P(x)$ of degree ≤ 2 that satisfies $P(x_0) = y_0$, $P'(x_0) = y'_0$, $P'(x_1) = y'_1$.

7. Determine the natural cubic spline function for the function $f(x) = \dfrac{1}{1+x^2}$ in the interval $[-5, 5]$ using

   (i) six equidistant nodes $x_k = -5 + 2k$, $k = 0, 1, \ldots, 5$,

   (ii) 11 equidistant nodes $x_k = -5 + k$, $k = 0, 1, \ldots, 10$.

Also find the error bounds for the two cases.

8. Find cubic spline function for the following data.

(i) $(-2, -8), (0, 0), (1, 1), (2, 8), S''(-2) = -12, S''(2) = 12.$

(ii) $(0, 0), (1, 1), (4, 2), (9, 3), (16, 4), S'(0) = 20, S'(16) = \frac{1}{8}.$

9. Show that the following function is not a cubic spline.

$$S(x) = \begin{cases} 11 - 24(x-1) + 18(x-1)^2 - 4(x-1)^3, & 0 \le x \le 1 \\ -160 + 144x - 42x^2 + 4x^3, & 1 \le x \le 2. \end{cases}$$

10. Use inverse interpolation to find the value of x when $y = 37$ from the given table

x	:	2	5	8	11	14
y	:	35	85	101	130	151

Unit 6 ❐ Approximation

§ Objectives

After going through this unit you will be able to learn the followings—

- Least square approximation of discrete data
- Approximation of function using Chebyshev polynomials
- Economization of power series evaluation

In this unit we shall consider approximations of functions by some suitable approximating polynomials. Before going to any specific methods, we first mention two important theorems which are essential for our discussion.

(i) **Weierstrass theorem :** Let $f(x)$ be a continuous function defined on $[a, b]$. Then for each $\varepsilon > 0$, there exists an integer $n = n(\varepsilon)$ and a polynomial $p_n(x)$ of degree at most n such that

$$|f(x) - p_n(x)| < \varepsilon, \ \forall x \in [a, b].$$

(ii) **Taylor's theorem :** Let $f(x)$ be a $(n + 1)$ times continuously differentiable function defined on $[a, b]$ and let $x, x_0 \in [a, b]$. Then, it is possible to express,

$$f(x) = p_n(x) + R_{n+1}(x),$$

where, $p_n(x) = f(x_0) + \frac{(x - x_0)}{1!} f'(x_0) + ... + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0)$,

and $R_{n+1}(x) = \frac{(x - x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi)$, for some ξ lies between x and x_0.

§ Least Squares Approximation to Discrete Data :

Let $S = \{y_0, y_1, ..., y_n\}$ be a set of given function values of function $f(x)$ in the interval $[a, b]$ at the points $x_0, x_1, ..., x_n$, i.e., $y_i = f(x_i)$, $i = 0(1)n$. ..... (1)

Then our task is to fit a polynomial of degree n, say, $P_n(x) = \sum_{i=0}^{n} a_i x^i$, such that the sum,

$$I = \sum_{i=0}^{n} [P_n(x_i) - f(x_i)]^2 \qquad \text{..... (2)}$$

$$= \sum_{i=0}^{n} [a_0 + a_1 x_i + a_2 x_i^2 + ... + a_n x_i^n - y_i]^2 \text{ is minimum.}$$

i.e., we have to find $a_0, a_1, ..., a_n$ such that the sum I is minimum. For that, we set $\frac{\partial I}{\partial a_i} = 0, i = 0(1)n$ and get normal equations that are satisfied by the optimum values of $a_0, a_1, ..., a_n$, say, $a_0^*, a_1^*, ..., a_n^*$. Now, $P_n^*(x) = \sum_{j=0}^{n} a_j^* x^j$, minimizes the sum I, of the squares of the residuals at the nodes x_i follows from the fact that I can be made arbitrarily large by selecting a_i's suitably. Clearly, the normal equations are,

$$na_0^* + a_1^* \sum_{i=0}^{n} x_i + a_2^* \sum_{i=0}^{n} x_i^2 + ... + a_n^* \sum_{i=0}^{n} x_i^n = \sum_{i=0}^{n} y_i ,$$

$$a_0^* \sum_{i=0}^{n} x_i + a_1^* \sum_{i=0}^{n} x_i^2 + ... + a_n^* \sum_{i=0}^{n} x_i^{n+1} = \sum_{i=0}^{n} x_i y_i , \qquad ..... (3)$$

.....

.....

$$a_0^* \sum_{i=0}^{n} x_i^n + a_1^* \sum_{i=0}^{n} x_i^{n+1} + ... + a_n^* \sum_{i=0}^{n} x_i^{2n} = \sum_{i=0}^{n} x_i^n y_i .$$

These are a set of $n + 1$ linear equations with $n + 1$ unknowns $a_0^*, a_1^*, ..., a_n^*$. Existence and uniqueness of solution of the methods, for finding a solution, have already been discussed while treating linear systems. Now instead of taking an approximating polynomial $P_n(x)$, we may also approximate the function $f(x)$ by an expression $\sum_{i=0}^{n} a_i \phi_i(x)$, where $\phi_i(x), i = 0(1)n$, are $n + 1$ appropriately chosen linearly independent functions and our $P_n(x)$ is then a particular case, if we take $\phi_0(x) = 1$, $\phi_1(x) = x, ..., \phi_n(x) = x^n$. Then our task is to minimize.

$$I = \sum_{i=0}^{n} \left[f(x_i) - \sum_{j=0}^{n} a_j \phi_j(x_i) \right]^2 \qquad ..... (4)$$

Again, if we have more nodal points than the degree of the polynomials or number of functions ϕ_i, say, we have $N + 1$ points, where $n \leq N$, then we usually minimize an weighted squared sum,

$$I = \sum_{i=0}^{N} \omega(x_i) \left[f(x_i) - \sum_{j=0}^{n} a_j \phi_j(x_i) \right]^2 \quad \ldots\ldots (5)$$

The corresponding normal equations are known as weighted normal equations. We then also calculate the residual function, $R(x) = f(x) - \sum_{i=0}^{n} a_i \phi_i(x)$, for determining the error.

Example 1. Consider the following data.

x	20·5	32·7	51·0	73·2	95·7
y	765	826	873	942	1032

Then, $\sum x_i = 273{\cdot}1$, $\sum x_i^2 = 18607{\cdot}27$, $\sum y_i = 4438$, $\sum x_i y_i = 254932{\cdot}5$. The least square straight line that fits the above data is,

$$y = 3{\cdot}395x + 702{\cdot}2.$$

§ Chebyshev Polynomials of the 1st Kind :

Now we shall describe a method of approximating functions by a system of orthogonal polynomials, viz., Chebyshev polynomials. Let $f \in C^{k+1}[a,b]$ be a given function. Then is it possible to choose nodes x_i's $\in [a, b]$, $i = 0(1)n$, for which $\max_{x \in [a,b]} |(x - x_0)(x - x_1) \ldots (x - x_n)|$ is minimum so that the error estimate corresponding to Lagrange interpolation polynomial would be also a minimum? Let us first restrict ourselves to the interval $[-1, 1]$, then by a suitable transformation we would be able to generalize the following discussion to any interval $[a, b]$. The Chebyshev's polynomials $T_n(x), n \geq 0$ on $[-1, 1]$ are defined by the formula.

$$T_n(x) = \cos(n \arccos x), \quad \ldots\ldots (6)$$

or recurisvely by,

$$T_0(x) = 1,$$

$$T_1(x) = x,$$

.....

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x), n = 1, 2, \ldots \quad \ldots\ldots (7)$$

The last relation follows from the trigonometric identity

$\cos(n+1)\phi = 2\cos\phi\cos n\phi - \cos(n-1)\phi$ by setting $\phi = \arccos x$. $T_n(x)$ may also be found as solutions of the following differntial equation,

$$(1-x^2)y'' - xy' + n^2 y = 0.$$

Some of the Chebyshev polynomials are as follows :

$T_0(x) = 1,$

$T_1(x) = x,$

$T_2(x) = 2x^2 - 1,$

$T_3(x) = 4x^3 - 3x,$

$T_4(x) = 8x^4 - 8x^2 + 1,$

$T_5(x) = 16x^5 - 20x^3 + 5x,$

.....

.....

Some properties of Chebyshev polynomials.

Property 1. The definition of Chebyshev polynomails clearly indicates that,

$$|T_n(x)| \le 1, \text{ for } x \in [-1,1], n = 0, 1, ... \quad (8)$$

Property 2. It is evident from definition and the recurrence relation that for an even (odd) n, the polynomial $T_n(x)$ contains only even (odd) powers of x, i.e., it is an even (odd) function, so that,

$$T_{2m}(-x) = T_{2m}(x) \text{ and } T_{2m+1}(-x) = -T_{2m+1}(x),$$

$$\forall m = 0, 1, 2, ... \quad (9)$$

Property 3. Leading coefficient of $T_n(x)$, i.e., of x^n is 2^{n-1}, for $n \ge 1$.

Proof. For $n = 1$, $T_1(x) = 2^{1-1}x = x$

For $n = 2$, $T_2(x) = 2xT_1(x) - T_0(x) = 2x^2 - 1 = 2^{2-1}x^2 - 1$.

Let us assume that coefficient of x^m in $T_m(x)$ is 2^{m-1} for all m upto $m = k$, say.

Then by recurrence relation,

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x)$$

we have, leading coefficient of $T_{k+1}(x)$, i.e. x^{k+1} is $2.2^{k-1} = 2^k = 2^{k+1-1}$, as leading coefficient of $T_k(x)$, i.e., of x^k is 2^{k-1} and that of x^{k-1} in $T_{k-1}(x)$ is 2^{k-2}.

Hence by induction, $T_n(x)$ has leading coefficient 2^{n-1}.

Property 4. $T_n(x)$ has n real roots on the interval $[-1, 1]$ and we may express them by the formula.

$$x_i = \cos\frac{(2i+1)\pi}{2n}, i = 0(1)n-1. \qquad \ldots\ldots (10)$$

They are usually known as Chebyshev abscissas or nodes.

Clearly, $T_n(x_i) = \cos(n \arccos x_i) = \cos\frac{(2i+1)\pi}{2} = 0, i = 0(1)n-1.$

Property 5. $\max_{x\in[-1,1]} |T_n(x)| = 1$ $\qquad \ldots\ldots (11)$

Proof : Since $T_n(x_m) = (-1)^m$ for $x_m = \cos\frac{m\pi}{n}, m = 0(1)n$

and as $|T_n(x)| \le 1, \forall x \in [-1, 1]$

therefore, $\max_{x\in[-1,1]} |T_n(x)| = 1$

Property 6. Among all n-th degree polynomials with leading coefficients unity, i.e., of all monic polynomials of degree n, the polynomials $\overline{T_n(x)} = 2^{1-n}T_n(x)$, $n \ge 1$ has the least maximum modulus value on the interval $[-1, 1]$, i.e., there is no polynomial $P_n(x)$ of degree n with leading coefficient 1 such that,

$$\max_{x\in[-1,1]} |P_n(x)| < \max_{x\in[-1,1]} |\overline{T_n(x)}| = 2^{1-n} \qquad \ldots\ldots (12)$$

Proof : Suppose $P_n(x) = a_0 + a_1x + \ldots\ldots + a_{n-1}x^{n-1} + x^n$ be a monic polynomial that satisfies the inquality (12). Then, $\overline{T_n(x)} - P_n(x)$ is a polynomial of degree at most $n-1$ and $P_n(x)$ satisfies strict inequality (12) therefore, $\overline{T_n(x)} - P_n(x)$ is not identically zero. So, for al the points $x_m = \cos\left(\frac{m\pi}{n}\right), m = 0(1)n, \overline{T_n(x)} - P_n(x)$ are non-zero with alternating signs. i.e. $\overline{T_n(x)} - P_n(x)$ must vanish at least at n points. But as it is a polynomial of degree at most $n-1$, it leads to a contradiction. Hence the proof.

This property of Chebyshev polynomials $T_n(x)$ are called polynomials least deviating from zero.

Now, consider a function $f(x)$ which we want to approximate by the Lagrange interpolation polynomial $L_n(x)$ on $[-1, 1]$, using Chebyshev nodes $x_i = \cos\left(\frac{2i+1}{2n+2}\pi\right)$, $i = 0(1)n$, which are roots of Chebyshev polynomial $T_{n+1}(x)$.

Then we have, $\max_{x\in[-1,1]} |f(x) - L_n(x)| \le \frac{M_{n+1}}{(n+1)!2^n}$, ... (13)

where, $M_{n+1} = \max_{x\in[-1,1]} |f^{(n+1)}(x)|$

i.e., the interpolation points are optimal for the estimate of error in $[-1, 1]$. We cannot improve it further.

If instead of the interval $[-1, 1]$, we consider the interval $[a, b]$, then a linear transformation of independent variable given by

$$\left.\begin{aligned} x &= \frac{b-a}{2}t + \frac{a+b}{2} \\ \text{or,}\quad t &= \frac{2(x-a)}{b-a} - 1, a \le x \le b,\ -1 \le t \le 1 \end{aligned}\right\} \quad \text{..... (14)}$$

will map $[a, b]$ onto $[-1, 1]$.

Since the roots of $T_{n+1}(t)$ in $[-1, 1]$ are,

$t_i = \cos\left(\frac{2i+1}{2n+2}\right)\pi$, $i = 0(1)n$, therefore, the corresponding roots in $[a, b]$ are,

$$x_i = \frac{1}{2}\left\{(b-a)\cos\left(\frac{2i+1}{2n+2}\pi\right) + (b+a)\right\}, i = 0(1)n. \quad \text{..... (15)}$$

In this case, $\max_{x\in[a,b]} |f(x) - L_n(x)| \le \frac{M_{n+1}}{(n+1)!}\frac{(b-a)^{n+1}}{2^{2n+1}}$ ..... (16)

where $M_{n+1} = \max_{x\in[a,b]} |f^{(n+1)}(x)|$

Property 7. Chebyshev polynomials form a sequence of orthogonal polynomials on $[-1, 1]$ with respect to the weight function $\omega(x) = \frac{1}{\sqrt{1-x^2}}$,

$$\text{i.e.}\quad \int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}dx = 0, \text{ if } i \ne j$$

$$= \frac{\pi}{2}, \text{ if } i = j \ne 0 \quad \text{..... (17)}$$

$$= \pi \text{ if } i = j = 0$$

The discrete version of orthogonality property is as follows.

If $x_i = \cos\dfrac{2i+1}{2n+2}\pi,\ i = 0(1)n$, then,

$$\left.\begin{aligned}\sum_{k=0}^{n} T_i(x_k)\,T_j(x_k) &= 0, \text{ for } i \neq j \\ &= \frac{n+1}{2}, \text{ if } i = j \neq 0 \\ \text{and} \quad \sum_{k=0}^{n} T_0(x_k)\,T_0(x_k) &= n+1.\end{aligned}\right\} \quad \ldots\ldots (18)$$

Chebyshev approximation : The Chebyshev approximating polynomial $P_n(x)$ of degree $\leq n$ for $f(x)$ in $[-1, 1]$ may be written as,

$$P_n(x) = \sum_{i=1}^{n} c_i T_i(x) \quad \ldots\ldots (19)$$

where c_i's are the constants given by,

$$\left.\begin{aligned} c_0 &= \frac{1}{n+1}\sum_{i=0}^{n} f(x_i)\,T_0(x_i) = \frac{1}{n+1}\sum_{i=0}^{n} f(x_i) \\ \text{and} \quad c_j &= \frac{2}{n+1}\sum_{i=0}^{n} f(x_i) T_j(x_i) \\ &= \frac{2}{n+1}\sum_{i=0}^{n} f(x_i)\cos\left(\frac{j(2i+1)}{2n+1}\pi\right),\ j = 1(1)n \end{aligned}\right\} \quad \ldots\ldots (20)$$

Example 2. The Chebyshev polynomial $P_3(x)$ which approximates the function $f(x) = e^x$ on $[-1, 1]$ is given by,

$P_3(x)$ = 1·26607 $T_0(x)$ + 1·13032 $T_1(x)$ +0·27145 $T_2(x)$ + 0·04379 $T_3(x)$

= 0·99462 + 0·99893x + 0·54290x^2 + 0·17518x^3, where calculations are made correct to five decimal places amd the nodes $x_k = \cos\dfrac{2k+1}{8}\pi,\ k = 0, 1, 2, 3$ are given by,

$$c_0 = \frac{1}{4}\sum_{k=0}^{3} e^{x_k} T_0(x_k) = 1{\cdot}26607,$$

$$c_1 = \frac{1}{2}\sum_{k=0}^{3} e^{x_k} T_1(x_k) = 1{\cdot}13032,$$

$$c_2 = \frac{1}{2}\sum_{k=0}^{3} e^{x_k} T_2(x_k) = 0{\cdot}27145,$$

and $$c_3 = \frac{1}{2}\sum_{k=0}^{3} e^{x_k} T_3(x_k) = 0{\cdot}04379.$$

§. Economized Power Series Approximation of Functions :

Suppose, a polynomial approximation to $f(x)$ is already available. We want a more efficient approximation to $f(x)$.

$$\text{Let, } f(x) = \sum_{i=0}^{n} a_i x^i + E_n(x), \qquad \ldots\ldots (21)$$

where it is known that, $|E_n(x)| < \varepsilon'$ and $x \in [-1, 1]$ ..... (22)

ε' is a small quantity which is smaller than our prescribed tolerance level, say, ε, but $|a_n| + \varepsilon'$ is not a tolerable error so that the approximation $\sum_{i=0}^{n} a_i x^i$ is not a safe one.

Now, expanding $\sum_{i=0}^{n} a_i x^i$ in terms of Chebyshev polynomials,

i.e., let,

$$\sum_{i=0}^{n} a_i x^i = \sum_{i=0}^{n} C_i T_i(x), \qquad \ldots\ldots (23)$$

or, $a_0 + a_1 x + \ldots a_n x^n = C_n T_n(x) + C_{n-1} T_{n-1}(x) + \ldots + C_0 T_0(x)$.

Then using the relations,

$$T_i(x) = 2^{i-1}\left(x^i - \frac{i}{4}x^{i-2} + \ldots\right), i \geq 1 \qquad \ldots\ldots (24)$$

and $T_0(x) = 1$,

We have,

$$\left.\begin{aligned} C_n\, 2^{n-1} &= a_n, \\ C_{n-1}\, 2^{n-2} &= a_{n-1}, \\ C_{n-2}\, 2^{n-3} &= a_{n-2} + \frac{n}{4} a_n, \\ &\ldots\ldots\ldots\ldots \end{aligned}\right\} \qquad \ldots\ldots (25)$$

which implies

$$\left.\begin{aligned} C_n &= 2^{1-n}a_n, \\ C_{n-1} &= 2^{2-n}a_{n-1}, \\ C_{n-2} &= 2^{3-n}\left(a_{n-2}+\frac{n}{4}a_n\right), \\ &\cdots\cdots\cdots\cdots\cdots\cdots \\ &\cdots\cdots\cdots\cdots\cdots\cdots \end{aligned}\right\} \quad \cdots\cdots (26)$$

Clearly, for large n, coefficient of $T_n(x), \ldots, T_{n-m+1}(x)$ on (23), i.e., C_i's are relatively small as compared with the coefficients of $x^n, x^{n-1}, \ldots, x^{n-m+1}$ in (21) for some m and therefore, we may able to make $\left(\sum_{i=n-m+1}^{n} |C_i| + \varepsilon'\right) < \varepsilon$, our desired approximation to $f(x)$. Again, as $|T_i(x)| \le 1, \forall i$ and $x \in [-1, 1]$, therefore we may neglect last m terms from the approximation $\sum_{i=0}^{n} C_i T_i(x)$, i.e., we may consider $\sum_{i=0}^{n-m} C_i T_i(x)$ as approximate to $f(x)$ so that our original approximation $\sum_{i=0}^{n} a_i x^i$ is reduced to the smallest possible number of terms that will supply our desired accuracy within the prescribed tolerance level. This process of approximation is known as economization of power series by Lanczos, which minimizes the number of numerical calculations. The transformation of $[-1, 1]$ to any interval $[a, b]$ includes approximation of functions defined on any interval $[a, b]$. We now list the successive inversion of elements $1, x, x^2, \ldots$, to $T_0(x), T_1(x), \ldots$ etc., for some cases as follows :

$T_0(x) = 1,$	$1 = T_0$
$T_1(x) = x,$	$x = T_1(x)$
$T_2(x) = 2x^2 - 1,$	$x^2 = \frac{1}{2}(T_0 + T_2)$
$T_3(x) = 4x^3 - 3x,$	$x^3 = \frac{1}{4}(3T_1 + T_3)$
$T_4(x) = 8x^4 - 8x^2 + 1,$	$x^4 = \frac{1}{8}(3T_0 + 4T_2 + T_4)$
$T_5(x) = 16x^5 - 20x^3 + 5x,$	$x^5 = \frac{1}{16}(10T_1 + 5T_3 + T_5)$

Example 3. Let the function e^x be approximated by the 5-th degree Maclaurin's polynomial, i.e., $e^x \approx p_5(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!}$

Now we want to find a fourth degree economized polynomial for e^x on $\left[-\frac{1}{2}, \frac{1}{2}\right]$.

Since, $[a,b] = \left[-\frac{1}{2}, \frac{1}{2}\right]$, therefore, the transformation, $x = a + \frac{1}{2}(b-a)(t+1) = \frac{1}{2}t$ is required for $-1 \le t \le 1$.

Let $q_5(t) = p_5\left(\frac{t}{2}\right)$

$$= 1 + \frac{t}{2} + \frac{t^2}{8} + \frac{t^3}{48} + \frac{t^4}{384} + \frac{t^5}{3840}, \quad -1 \le t \le 1.$$

Then, $q_5(t) \approx 1{\cdot}063477\,T_0 + 0{\cdot}515788\,T_1 + 0{\cdot}063802\,T_2 + 0{\cdot}005290\,T_3$

$$+ 0{\cdot}000326\,T_4 + 0{\cdot}000052\,T_5.$$

Therefore, dropping T_5 we get economized,

$$q_5(t) = 1{\cdot}063477\,T_0 + 0{\cdot}515788\,T_1 + 0{\cdot}063802\,T_2 + 0{\cdot}005290\,T_3$$

$$+ 0{\cdot}000326\,T_4.$$

$$= 1 + 0{\cdot}4999186t + {\cdot}125000t^2 + {\cdot}0211589t^3 + 0{\cdot}00260417t^4$$

Again, transferring $t = 2x$, we have,

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

$$= 1 + {\cdot}999837x + {\cdot}500000x^2 + {\cdot}169271x^3 + {\cdot}041667x^4.$$

For accuracy we have,

$$|p_5(x) - \text{economized } p(x)| \le {\cdot}000052, \text{ so that for } -\frac{1}{2} \le x \le \frac{1}{2},$$

$$|e^x - \text{economized } p(x)| \le {\cdot}88 \times 10^{-4}.$$

§ Summary :

This unit is devoted to approximation of discrete data and approximation of function using some standard polynomials. Here, least square method is discussed to

approxamite discrete data. The chebyshev polynomials are used to approximate a function and economized the power series evaluation. These are discussed carefully with appropriate examples.

EXERCISES

1. Find the polynomial of degree two or less which gives an approximate to e^x in [0, 1] by the method of least squares.

2. Using the following data find least square polynomials of degree one and two.

(i)

x	0	0·4	0·8	1·2	1·6	2·0
$f(x)$	0·11	1·15	2·21	2·60	2·55	3·10

(ii)

x	−3	−1	1	3
$f(x)$	10	0	−4	0

(i)

x	−2	−1	0	1	2
$f(x)$	9	0	−1	1	8

3. Find the least square polynomial of degree one to approximate the function $f(x) = \ln x$ on [1, 2].

4. Find the Chebyshev's approximating polynomial for the following functions :

(i) $f(x) = \sqrt{x}$ on [0, 1] for $n = 2, 3, 4$.

(ii) $f(x) = e^{-x}$ on [0, 1] for $n = 2, 3$.

(iii) $f(x) = \cos x$ on $\left[0, \frac{\pi}{2}\right]$ for $n = 3$, k, where k is a positive integer.

(iv) $f(x) = \ln(x + 2)$ on [−1, 1] for $n = 2, 3, 4$.

(v) $f(x) = \sin\left(\frac{\pi}{2}x\right)$ on [−1, 1] for $n = 3, 5, 7$.

5. Find maximum and minimum values of Chebyshev polynomial $T_n(x)$ for $n = 2, 3, 4$.

6. Find error bounds in all cases of exercise 4.

7. Convert $1 + x + x^2 + x^3 + x^4$ by Cheyshev polynomials.

8. Use Chebyshev economization to approximate the following approximate polynomials on $[-1, 1]$.

(i) $e^{-x} \approx p_4(x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!}$.

(ii) $\cos x \approx p_5(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!}$.

(iii) $\sin x \approx p_3(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!}$.

(iv) $\ln(1+x) \approx p_4(x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4}$.

9. Discuss the accuracy level in all the cases of Excerise 8.

Unit 7 ❐ Numerical Integration

In this unit we shall consider the methods for computation of non-singular definite integrals approximately. The methods involve the approximation of the integrand $f(x)$, defined on a closed interval $[a, b]$ by suitable interpolating polynomials.

§ Objectives

After going through this unit you will be able to learn the folowings—

- Newton-Cotes quadrature formulae
- Romberg's integration formula
- Gaussian quadrature

§ Newton-Cotes Quadrature Formulas :

Let us consider the integral,

$$I = \int_a^b f(x)dx \qquad \ldots\ldots (1)$$

where $f(x)$ is a continuous, real-valued function defined on $[a,b]$, a finite interval. Let, $x_0, x_1, \ldots, x_n$ be a given set of distinct values of x. Without loss of generality, let us assume that,

$$a \le x_0 < x_1 < \ldots\ldots < x_n \le b$$

Now, corresponding to the support points x_k, $k = 0(1)n$, and the ordinates $f_k \equiv f(x_k)$, $k = 0(1)n$, there exists a unique interpolation polynomial $P_n(x)$ of degree at most n. Therefore, using Lagrange's interpolation polynomals $L_k(x)$, it has the form,

$$P_n(x) = \sum_{k=0}^{n} f(x_k)L_k(x) \qquad \ldots\ldots (2)$$

Hence the approximation of I is given by,

$$I \approx Q_n = \int_a^b P_n(x)dx = \sum_{k=0}^{n} f_k \int_a^b L_k(x)dx$$

$$\equiv (b-a)\sum_{k=0}^{n} f_k\omega_k , \qquad \ldots\ldots (3)$$

where, $\omega_k = \frac{1}{b-a}\int_a^b L_k(x)dx, k = 0(1)n$, $\qquad \ldots\ldots (4)$

which depends on $x_0, x_1, \ldots, x_n$ and $(b-a)$. ω_k's are called integration weights of the quadrature formula (3) corresponding to the nodes x_k. Any quadrature formula of the form (3) is called an interpolatory quadrature formula. It is clear that the value of Q_n is exact if I is evaluated for an integrand $f(x)$ which is a polynomial of degree of at most n. Otherwise, Q_n is not exact and the error occurs, say,

$$E_n(f) = I - Q_n = \int_a^b f(x)\,dx - (b-a)\sum_{k=0}^{n} \omega_k f_k \qquad \ldots\ldots (5)$$

Now to discuss about the accuracy of a quadrature formula we introduce some basic notions.

Definition : A quadrature formula has the degree of precision $m \in N$, if it integrates all polynomials of degrees $\leq m$ exactly and if m is the largest integer with that property.

Therefore, as error function $E_n(f)$ is a linear function, the quadrature formula (3) has the degree of precision m if and only if we have $E(x^j) = 0$, for $j = 0(1)m$ and $E(m^{m+1}) \neq 0$. We then have the following theorem.

Theorem. Let $a \leq x_0 < x_1 \ldots < x_n \leq b$ be arbitrarily given $n+1$ pairwise different nodes x_k. Then here is a uniquely determined interpolatory quadrature formula.

$$Q_n = (b-a)\sum_{k=0}^{n} \omega_k f(x_k) \quad \text{with} \quad \omega_k = \frac{1}{b-a}\int_a^b L_k(x)dx,\ k = 0(1)n\,, \quad \text{whose}$$

degree of precision is at least equal to n.

Next we present a class of quadrature formulas with equidistant nodes in the interval $[a,b]$ so that we have $x_i = a + ih$, $i = 0(1)n$, i.e., $h = \dfrac{b-a}{n}$. Since the nodes include the end points, they are known as Newton-Cotes closed type quadrature formulae and corresponding ω_k's are called Cotes coefficients.

Since, $\omega_k = \dfrac{1}{b-a}\int_a^b \prod_{\substack{i=0 \\ i\neq k}}^{n} \dfrac{x - x_i}{x_k - x_i}\,dx$

$$= \int_0^1 \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{nt - i}{k - i}\,dt\,, \qquad \ldots\ldots (6)$$

if we set $x = a + (b-a)t$, i.e., $dx = (b-a)dt$, then we have some properties for ω_k's :

(i) $\omega_k = \omega_{n-k}$,

because $\omega_k = \int_0^1 \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{nt-i}{k-i} dt$

$$= \int_0^1 \prod_{\substack{i=0 \\ i\neq k}}^{n} \left(\frac{nq-\overline{n-i}}{k-n+i} \right) dq, \text{ where, } q = 1-t.$$

$$= \omega_{n-k}$$

(ii) $\sum_{k=0}^{n} \omega_k = 1$,

For $n = 1$, $\omega_0 = \int_0^1 \frac{t-1}{-1} dt = \frac{1}{2}$

$$\omega_1 = \int_0^1 t\, dt = \frac{1}{2}.$$

Therefore, $Q_1 = \frac{b-a}{2}(f_0 + f_1) = \frac{h}{2}(f_0 + f_1)$, where $f_i = f(x_i)$, ... (7)

is known as trapezoidal rule.

Now, $E_1(x) = \int_a^b x\, dx - \frac{b-a}{2}(a+b) = 0$

$$E_1(x^2) = \int_a^b x^2\, dx - \frac{b-a}{2}(a^2+b^2) = \frac{b^3-a^3}{3} - \frac{(b-a)(b^2+a^2)}{2} \neq 0.$$

So, degree of precision of the trapezoidal formula is 1.

For $n = 2$, we have,

$$\omega_0 = \int_0^1 \frac{(2t-1)(2t-2)}{(-1)(-2)} dt = \frac{1}{6},$$

$$\omega_1 = \int_0^1 \frac{2t(2t-2)}{(-1)} dt = \frac{2}{3},$$

$$\omega_2 = \int_0^1 \frac{2t(2t-1)}{2.1} dt = \frac{1}{6}.$$

Therefore, $Q_2 = \frac{b-a}{6}(f_0 + 4f_1 + f_2) = \frac{h}{3}(f_0 + 4f_1 + f_2)$, ... (8)

known as Simpson's $\frac{1}{3}$ rule.

Here, $E_2(x^3) = \int_a^b x^3 dx - \frac{b-a}{6}\left(a^3 + 4\left(\frac{a+b}{3}\right)^3 + b^3\right) = 0$,

but $E_2(x^4) \neq 0$.

So, the degree of precision is 3.

For $n = 3$, we find,

$$Q_3 = \frac{3h}{8}(f_0 + 3f_1 + f_3),\ h = \frac{b-a}{3}, \qquad \ldots\ldots (9)$$

known as Simpson's $\frac{3}{8}$ rule.

It is also easy to check that $E_3(x^4) \neq 0$ but $E_3(x^3) = 0$, i.e., degree of precision is 3.

For $n = 4$, $Q_4 = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4)$, $h = \frac{b-a}{4}$ ..... (10)

known as Boole's rule.

For $n = 5$, $Q_5 = \frac{5h}{288}(19f_0 + 75f_1 + 50f_2 + 50f_3 + 75f_4 + 19f_5)$. ..... (11)

Both the formulas Q_4 and Q_5 have the degree of precision 5.

Now it can be shown that Newton-Cotes quadrature formulas have degree of precision $m = n + 1$ for n even and $m = n$ for n odd. For this reason, Newton-Cotes formula is more advantageous for even n, i.e., with odd number of nodes. Also, it may be taken into account that for $n \geq 10$ some ω_k become negative and also the interpolation polynomials of high degree oscillate a lot near the end points of the interval. It is advisable that Newton-Cotes quadrature formula may be used for $n < 8$.

Error Analysis :

Let $f(x)$ be $n + 1$ times continously differentiable function defined on $[a, b]$. Then Newton-Cotes formula for $n + 1$ nodes has the error,

$$E_n(f) = \frac{1}{(n+1)!}\int_a^b f^{(n+1)}(\xi)\phi_n(x)\,dx, \qquad \ldots\ldots (12)$$

where $\phi_n(x) = \prod_{i=0}^{n}(x - x_i)$ and ξ depends on x.

Theorem : If n is even and $f(x)$ is $n+2$ times continuously differnetiable on $[a,b]$, then the error of closed type Newton-Cotes formula is given by,

$$E_n(f)=\frac{k_n}{(n+2)!}f^{(n+2)}(\zeta),\ a<\zeta<b,\ k_n=\int_a^b x\phi_n(x)dx \qquad \ldots\ldots (13)$$

Theorem : if n is odd and $f(x)$ is $n+1$ times continuously differentiable function defined on $[a, b]$, then the error of closed type Newton-Cotes formula is given by,

$$E_n(f)=\frac{k_n}{(n+1)!}f^{(n+1)}(\zeta),\ a<\zeta<b,\ k_n=\int_a^b \phi_n(x)dx \qquad \ldots\ldots (14)$$

For $n=1$, $k_1=\int_a^b (x-x_0)(x-x_1)dx=\int_a^b (x-a)(x-b)dx$

$$=-\frac{(b-a)^3}{6}=-\frac{h^3}{6}.$$

Therefore, $E_1=-\frac{h^3}{12}f''(\zeta)$.

For $n=2$ $k_2=\int_a^b (x-x_0)(x-x_1)(x-x_2)dx$

$$=-\frac{4}{15}h^5.$$

Therefore, $E_2=-\frac{1}{90}h^5 f^{(4)}(\zeta)$:

For $n=3$, $k_3=-\frac{9}{10}h^5$, which implies,

$$E_3=-\frac{3}{80}h^5 f^{(4)}(\zeta).$$

For $n=4$, we find, $E_4=\frac{-8h^7}{945}f^{(6)}(\zeta)$,

and for $n=5$, $E_5=\frac{275}{12096}h^7 f^{(6)}(\zeta)$, ...:

Sometimes to get better approximate, we partitioned the interval $[a, b]$ into some smaller subintervals and then in each of these subintervals we use Newton-Cotes formula. Adding these estimates we get the estimate of the intergral for the whole interval. This leads to the composite Newton-Cotes formula and the corresponding error term is the sum of the error terms for the separate subintervals.

§ Open type Newton-Cotes Quadrature Formulas :

This type of formulas use nodes that are equidistantly distributed in the interior of the interval $[a\ b]$. If only one point, say, the mid-point is chosen, call $x_1 = \frac{a+b}{2}$, then the interpolation polynomial $P_0(x)$ is taken as constant and equals to $f(x_1)$. In this way we obtain the mid-point rule,

$$Q_0^{\,0} = (b-a)f(x_1),\ x_1 = \frac{a+b}{2} \qquad \ldots\ldots (15)$$

The degree of precision is obviously 1 and error term is given by,

$$E_0^{\,0}(f) = \frac{1}{24}(b-a)^3 f''(\zeta),\ a<\zeta<b, \qquad \ldots\ldots (16)$$

If we subdivide $[a, b]$ into 3 subintervals of equal length $h = \frac{b-a}{3}$, then we have the quadrature rule,

$$Q_1^{\,0} = \frac{3h}{2}(f_1 + f_2),\ \text{where}\ f_1 = f(x_1) = f(a+h)$$

$$f_2 = f(x_2) = f(a+2h) \qquad \ldots\ldots (17)$$

Again, it has only the degree of precision $m = 1$ and the error term is given by,

$$E_1^{\,0}(f) = \frac{1}{108}(b-a)^3 f''(\zeta),\ a<\zeta<b \qquad \ldots\ldots (18)$$

Similarly, for 4 subintervals,

$$Q_2^{\,0} = \frac{4h}{3}(2f_1 - f_2 + 2f_3),\ h = \frac{b-a}{4},\ x_i = a+ih. \qquad \ldots\ldots (19)$$

It has degree of precision $m = 3$ and error is given by,

$$E_2^{\,0}(f) = \frac{7}{23040}(b-a)^5 f^{(4)}(\zeta),\ a<\zeta<b \qquad \ldots\ldots (20)$$

.....

.....

Since the upper limit and lower limit does not involved in the formulae, these formulae are used to evaluate improper integrals.

Example 1. For the integral $I = \int_0^1 \frac{4}{1+x^2}dx$, we have by simple Newton-Cotes closed type formula,

$$Q_2 \approx 3{\cdot}1333, \qquad E_2 \approx {\cdot}00826,\ h = \frac{1}{2},$$

$Q_3 \approx 3{\cdot}13846$, $E_3 \approx {\cdot}003131$, $h = \frac{1}{3}$,

$Q_4 \approx 3{\cdot}142118$, $E_4 \approx -{\cdot}000525$, $h = \frac{1}{4}$,

and $Q_5 \approx 3{\cdot}141878$, $E_5 \approx -{\cdot}000286$, $h = \frac{1}{5}$.

But, if we use compostie Newton-Cotes formula, then we have, for composite Simpson's rule, i.e., using Q_2 and also for composite Q_4, for n number of subintervals, the following table; which yeilds better result.

n	Composite Q_2	Error	Composite Q_4	Error
2	3·1415686275	·0000240261	3·1415940941	−·0000014405
3	3·1415917809	·0000008727	3·1415926976	−·0000000440
4	3·1415925025	·0000001511	3·1415926611	−·0000000076
6	3·1415926403	·0000000133	3·1415926543	−·0000000007

§ Romberg's Integration Formula :

Suppose the error term in approximating the integral $I = \int_a^b f(x)dx$ can be expressed in the form, $E(f) = ch^r f^{(r)}(\xi)$, where c is independent of h, r is a positive integer and $\xi \in (a,b)$. Let us calculate the value of I for two different step length h_1 and h_2. Then the errors (truncation) in approximations are,

$$\left.\begin{aligned} E_1 &= I - I_1 = ch_1^r f^{(r)}(\xi_1) \\ E_2 &= I - I_2 = ch_2^r f^{(r)}(\xi_2) \end{aligned}\right\} \quad \ldots\ldots (21)$$

where I_1, I_2 are the computed values of I and $\xi_1, \xi_2 \in (a,b)$. Now, if we consider $f^{(r)}(\xi_1)$ and $f^{(r)}(\xi_2)$ are nearly equal, then we have an extrapolation formula,

$$I \approx \frac{h_1^r I_2 - h_2^r I_1}{h_1^r - h_2^r} = I_2 + \frac{I_2 - I_1}{\left(\frac{h_1}{h_2}\right)^r - 1} \quad \ldots\ldots (22)$$

In particular, if we take $\frac{h_1}{h_2} = 2$, then we have,

$$I \approx I_2 + \frac{I_2 - I_1}{2^r - 1}$$

This process is usually known as Richardson extrapolation and it is very effective if $f^{(r)}$ does not fluctuate very rapidly in $[a,b]$. If we use trapezoidal rule then $r = 2$. Now to make it more useful we generalize this procedure. For that we first describe the error associated with trapezoidal rule. We have the following theorem.

Theorem (Euler-Maclaurin formula) :

Let $f(x)$ be a $2m + 2$ times continuously differentiable function defined on $[a,b]$ for some $m \geq 0$ and suppose $h = \frac{b-a}{n}$, $x_j = a + jh$, $j = 0(1)n$, $n \geq 1$. Then the error term for trapezoidal rule is,

$$E_n(f) = \int_a^b f(x)dx - h\sum_{j=1}^{n-1} f(x_j) - \frac{h}{2}\{f(x_0) + f(x_n)\}$$

$$= -\sum_{i=1}^{m} \frac{B_{2i}}{(2i)!} h^{2i}[f^{(2i-1)}(b) - f^{(2i-1)}(a)] - \frac{h^{2m+2}}{(2m+2)!}(b-a)B_{2m+2}f^{(2m+2)}(\xi),$$

for some $\xi(h), a < \xi < b$, ... (24)

where B_i, $i = 1, 2, \ldots$ are Bernoulli's numbers defined by the expansion,

$$\frac{t}{e^t - 1} = \sum_{j=0}^{\infty} B_j \frac{t^j}{j!} \qquad \ldots\ldots (25)$$

Some B_i's are, $B_0 = 1, B_1 = -\frac{1}{2}, B_2 = \frac{1}{6}, B_4 = -\frac{1}{30}, B_6 = \frac{1}{42}$, etc.

For all odd integers, $j \geq 3$, $B_j = 0$.

In short, we can write (24) as :

$$E_n(f) = c_1h^2 + c_2h^4 + \ldots\ldots + c_mh^{2m} - (b-a)\frac{h^{2m+2}}{(2m+2)!}B_{2m+2}f^{(2m+2)}(\xi) \qquad \ldots\ldots (26)$$

where c_i's are independent of h.

Let us denote $I_{k,0}$ as the trapezoidal approximation to I with spacing $h_k = \frac{b-a}{2^k}$. Then, the first Richardson extrapolate, based on $I_{k,0}$ and $I_{k+1,0}$, say $I_{k+1,1}$ is given by, $I_{k+1,1} = I_{k+1,0} + \frac{I_{k+1,0} - I_{k,0}}{2^2 - 1} = \frac{2^2 I_{k+1,0} - I_{k,0}}{2^2 - 1}$ ..... (27)

$I_{k+1,1}$ defined by equation (27) is also called the first Romberg extrapolate.

Now form (27) we have,

$$I - I_{k+1,1} = c_1' h_k^4 + c_2' h_k^6 + \ldots\ldots \qquad \ldots\ldots (28)$$

where $c'_1, c'_2, \ldots\ldots$, are the coefficients in the error formula and are independent of h_k.

Clearly, using Richardson extrapolation, i.e., from $I_{k,0}$ to $I_{k+1,1}$ and improvement in accuracy level is observed. Earlier, i.e., in (26) it started with h_k^2, but in (28) we find error term starts with h_k^4.

Again, using $I_{k,1}$ and $I_{k+1,1}$ we get the next Richardson extrapolate (i.e., second Romberg extrapolate)

$I_{k+1,2}$ defined by,

$$I_{k+1,2} = \frac{2^4 I_{k+1,1} - I_{k,1}}{2^4 - 1} \qquad \ldots\ldots (29)$$

$I_{k+1,2}$ is a sixth order approximation to I.

Proceeding as above, we have,

$$I_{k+1,m} = \frac{4^m I_{k+1,m-1} - I_{k,m-1}}{4^m - 1}, m = 1,2,\ldots \qquad \ldots\ldots (30)$$

and the corresponding error term starts with h_k^{2m+2}.

Hence, by this process of recursion we have a triangular array of approximation to the integral I.

$$\begin{array}{cccc} I_{0,0} & & & \\ I_{1,0} & I_{1,1} & & \\ I_{2,0} & I_{2,1} & I_{2,2} & \\ I_{3,0} & I_{3,1} & I_{3,2} & I_{3,3} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

The successive elements in the first column are approximated by trapezoidal rule

with $h_k = \frac{b-a}{2^k}, k = 0, 1, \ldots\ldots$ The above table is constructed row by row. A new trapezoidal rule estimate $I_{k,0}$ is affixed to the lower end of the first column and the previously computed values on the $(k-1)$-th row are used to extend the k-th row by calculating $I_{k,1}, I_{k,2}, \ldots I_{k,k}$ in turn. Now it can be easily verified (left as an exercise) that second column is nothing but the formula for Simpson's $\frac{1}{3}$ rule, third column Simpson's $\frac{3}{8}$-th rule, fourth column Boole's rule, etc. The above process of approximationg an integral I through the triangular array, i.e., through $I_{k,m}$ is known as Romberg's integration procedure. Sometimes the table for Romberg's integration procedure written as follows :

$$
\begin{array}{llll}
I_{1,0} & & & \\
I_{2,0} & I_{2,1} & & \\
I_{4,0} & I_{4,1} & I_{4,2} & \\
I_{8,0} & I_{8,1} & I_{8,2} & I_{8,3} \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots
\end{array}
$$

where, $I_{n,k} = \frac{4^k I_{n,k-1} - I_{n/2,n-1}}{4^k - 1}, n \geq 2^k$ ..... (31)

The step lenghts corresponding to $I_{n,k}$ and $I_{n/2,k}$ are just half in $I_{n/2,k}$ from that of $I_{n,k}$. And the terms $I_{1,0}, I_{2,1}, I_{4,2}$ .... are known as Romberg's successive approximation to I. Assuming the function $f(x)$, i.e., the integrand is sufficiently smooth, or, in other words infinitely differentiable, each column in the Romberg's procedure converges more rapidly than the previous one and each column itself converges to the integral I. Therefore, the diagonal approximation with the desired accuracy level is the most preferred one.

Example 2. For the integral $I = \int_1^2 \frac{dx}{x} = \log 2 = .69314718$, we have by the Romberg's integration procedure the following table, correct to eight decimal places.

h_k	$I_{k,0}$	$I_{k,1}$	$I_{k,2}$	$I_{k,3}$	$I_{k,4}$
1.0000	·75000000				
·5000	·70833333	·69444444			
·2500	·69702381	·69325397	·69317460		
·1250	·69412185	·69315453	·69314790	·69314748	
·0625	·69339120	·69314765	·69314718	·69314718	·69314718

§ Gaussian Quadrature Rule :

We now describe an interpolatory quadrature formula in which the nodes x_k and the wights ω_k are so chosen that it attains maximal degree of precision.

Let us first formulate our quadrature formula, say, Q_n in the interval $[-1, 1]$. The generalization to arbitrary interval $[a, b]$ is obvious.

Suppose, $\int_{-1}^{1} f(x)\,dx = \sum_{k=0}^{n} \omega_k f(x_k) + E_n(f)$

$$= Q_n + E_n(f),\ x_k \in [-1,1],\ k = 0(1)n, \qquad \ldots\ldots (32)$$

are $n + 1$ nodes and ω_k, $k = 0(1)n$ are $n + 1$ weights. Then we have the following theorem.

Theorem : The degree of precision of a quadrature formula of the type (32) cannot exceed $2n + 1$.

Proof : Consider the function,

$$f(x) = \prod_{k=0}^{n} (x - x_k)^2,\ \forall x \in [-1, 1].$$

Clearly, $f(x) \geq 0,\ \forall x \in [-1,1]$ and consequently

$$I = \int_{-1}^{1} f(x)dx > 0.$$

But, $Q_n = \sum_{k=0}^{n} \omega_k f(x_k) = 0$, as $f(x_k) = 0, \forall k = 0, 1, \ldots, n$.

Hence, $E_n(f) \neq 0$, which implies that the degree of precision of (32) must be less than $2n + 2$.

Next we have the following theorem.

Theorem : There exists a quadrature formula $Q = \sum_{k=0}^{n} \omega_k f(x_k)$ with $n + 1$ nodes $x_k \in [-1, 1]$ and has maximal degree of precision $2n + 1$. The nodes x_k are the zeros of the $(n + 1)$-th Legendre polynomial $P_{n+1}(x)$ and weights are defined by,

$$\omega_k = \int_{-1}^{1} \prod_{\substack{j=0 \\ j \neq k}}^{n} \left(\frac{x - x_j}{x_k - x_j} \right)^2 dx > 0, k = 0(1)n \qquad \ldots\ldots (33)$$

Proof : We first prove the existence of the desired quadrature formula and then find weight formulas for it and prove the uniqueness of the quadrature formula.

Existence :

Our aim is to show a quadrature formula with degree of precision $2n + 1$. For this purpose, we use the fact that in the interior of $[-1, 1]$, the Legendre polynomial $P_{n+1}(x)$ has $n + 1$ simple zeros, say, $x_0, x_1, \ldots, x_n$. Considering these as n nodes, we have always an interpolatory quadrature formula with degree of precision at least n.

Now, let us consider an arbitrary polynomal $p(x)$ whose degree is at most equal to $2n + 1$. Since $P_{n+1}(x)$ is polynomial of degree $n + 1$, therefore, by division algorithm, we can write $p(x)$ as :

$p(x) = q(x)P_{n+1}(x) + r(x)$, where degree of $q(x) \leq n$ and degree of $r(x) \leq n$.

Again, Legendre polynomials form an orthogonal systems on $[-1, 1]$ with weight function $\omega(x) = 1$.

Therefore, $P_{n+1}(x)$ is orthogonal to each of the Legendre polynomials of lower degree, viz., $P_0(x), P_1(x), \ldots\ldots, P_n(x)$ and hence it is orthogonal to $q(x)$, as $q(x)$ has degree at most n and it is always possible to write $q(x)$ as a linear combination of $P_0(x), P_1(x), \ldots\ldots, P_n(x)$. So, using the above fact, we have,

$$\int_{-1}^{1} p(x)\, dx = \int_{-1}^{1} [q(x)P_{n+1}(x) + r(x)]\, d_k$$

$$= 0 + \int_{-1}^{1} r(x)\, dx = \int_{-1}^{1} r(x) dx$$

Now, taking nodes $x_k, k = 0(1)n$, we have,

$$\sum_{k=0}^{n} \omega_k p(x_k) = \sum_{k=0}^{n} \omega_k [q(x_k) P_{n+1}(x_k) + r(x_k)]$$

$$= 0 + \sum_{k=0}^{n} \omega_k r(x_k) \text{ as } x_k\text{'s, for } k = 0(1)n \text{ are zeros of } P_{n+1}(x),$$

i.e., $P_{n+1}(x_k) = 0, \ \forall k = 0(1)n$.

Since the degree of precision of an interpolatory quadrature formula is at least n and degree of $r(x) \leq n$, therefore,

$$\sum_{k=0}^{n} \omega_k r(x_k) = \int_{-1}^{1} r(x)\, dx$$

$$= \int_{-1}^{1} p(x)\, dx.$$

Hence, we have, $\sum_{k=0}^{n} \omega_k p(x_k) = \int_{-1}^{1} p(x)\, dx$.

As, $p(x)$ is an arbitrary polynomial of degree $\leq 2n + 1$, therefore, the quadrature formula $Q = \sum_{k=0}^{n} \omega_k f(x_k)$, where x_k, is a zero of $P_{n+1}(x)$ for each $k = 0(1)n$, is of maximal degree of precision, i.e., $2n + 1$.

Now, we have to find the formula for calculating ω_k.

Let, $L_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \left(\frac{x - x_j}{x_k - x_j} \right), k = 0(1)n$, be the Lagrange polynomials of degree n with nodes $x_0, x_1, \ldots\ldots, x_n$.

Then, it is easy to check that $L_k(x_j) = \delta_{kj}$, for $k, j = 0(1)n$,

and $L_k^{\,2}(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \left(\frac{x - x_j}{x_k - x_j} \right)^2$ has degree $2n$.

Since the quadrature formula has degree of precision $2n + 1$, therefore $L_k^{\,2}(x)$ yields exact value for each $k = 0, 1, \ldots\ldots, n$. Thus we have,

$$0 < \int_{-1}^{1} L_k^{\,2}(x)\,dx = \int_{-1}^{1} \prod_{\substack{j=0 \\ j\neq k}}^{n} \left(\frac{x-x_j}{x_k-x_j}\right)^2 dx$$

$$= \sum_{\mu=0}^{n} \omega_\mu L_k^{\,2}(x_\mu) = \omega_k,\ k = 0(1)n \quad \text{as } L_k(x_\mu) = \delta_{k\mu}.$$

Hence, $\omega_k = \int_{-1}^{1} \prod_{\substack{j=0 \\ j\neq k}}^{n} \left(\frac{x-x_j}{x_k-x_j}\right)^2 dx > 0,\ \forall k = 0(1)n.$

Next, we shall show that our quadrature formula is unique. For this purpose, let us assume that there is another quadrature formula,

$Q'_n = \sum_{k=0}^{n} \omega'_k f(x'_k),\ x'_k \neq x'_j\ \forall k \neq j$, which has degree of precision $2n+1$ and all the weights $\omega'_k > 0$, for $k = 0(1)n$. Again, let us consider,

$$L'_k(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \left(\frac{x-x'_j}{x'_k-x'_j}\right) \text{ and } h(x) = L'_k(x)P_{n+1}(x).$$

Then, $h(x)$ is a polynomial of degree $2n+1$. So, by assumption about degree of precision of Q'_n, we have, for each $k = 0, 1, \ldots, n$,

$\int_{-1}^{1} L'_k(x)P_{n+1}(x) = 0$, [as $L'_k(x)$ has degree n and $P_{n+1}(x)$ has degree $n+1$, therefore, they are orthogonal]

i.e., $\int_{-1}^{1} h(x)dx = 0$

or, $0 = \sum_{\mu=0}^{n} h(x'_\mu)\omega'_\mu = \sum_{\mu=0}^{n} \omega'_\mu L'_k(x'_\mu)P_{n+1}(x'_\mu)$

$$= \sum_{\mu=0}^{n} \omega'_\mu \delta_{k\mu} P_{n+1}(x'_\mu) = \omega'_k P_{n+1}(x'_k)$$

Since, $\omega'_k > 0$ for each $k = 0(1)n$, therefore the above relation implies $P_{n+1}(x'_k) = 0$, for each $k = 0(1)n$.

i.e., x_k' are the zeros of $P_{n+1}(x)$. Hence apart from a permutation of $x_0, x_1, \ldots, x_n$, the nodes x'_k, $k = 0(1)n$, coincide with the nodes x_k, $k = 0(1)n$. So weights are uniquely determined for the quadrature formula.

The quadrature formula (based on zeros of Legendre polynomials) is known as Gaussian (or, Gauss-Legendre) quadrature formula. It is the formula with maximal degree of precision. The nodes x_k are pairwise symmetric with respect to the origin and the formula of ω_k implies that they are equal for every symmetric pair of nodes. We now discuss a computational procedure to find nodes x_k and hence ω_k. For the purpose, we have the following theorem.

Theorem. The $n+1$-th Legendre polynomial $P_{n+1}(x)$ of order $n+1$, $n \geq 0$, is equal to the determinant,

$$P_{n+1}(x) = \begin{vmatrix} a_0x & b_0 & 0 & 0 & \ldots & \ldots & \ldots & \ldots & \ldots & 0 \\ b_0 & a_1x & b_1 & 0 & \ldots & \ldots & \ldots & \ldots & \ldots & 0 \\ 0 & b_1 & a_2x & b_2 & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & \ldots & 0 & b_{n-2} & a_{n-1}x & b_{n-1} \\ 0 & \ldots & \ldots & \ldots & \ldots & \ldots & 0 & 0 & b_{n-1} & a_nx \end{vmatrix} \quad \ldots\ldots (34)$$

where $a_k = \dfrac{2k+1}{k+1}$, $b_k = \sqrt{\dfrac{k+1}{k+2}}$, $k = 0(1)n$..

With the use of the above result, it can be easily verified that the nodes x_k are the eigenvalues of the symmetric, tridiagonal matrix,

$$J_{n+1} = \begin{bmatrix} 0 & \beta_0 & 0 & 0 & \ldots & \ldots & \ldots & 0 \\ \beta_0 & 0 & \beta_1 & 0 & \ldots & \ldots & \ldots & 0 \\ 0 & \beta_1 & 0 & \beta_2 & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & \beta_{n-2} & 0 & \beta_{n-1} \\ 0 & \ldots & \ldots & \ldots & \ldots & 0 & \beta_{n-1} & 0 \end{bmatrix} \quad \ldots\ldots (35)$$

where $\beta_k = \dfrac{b_k}{\sqrt{a_k a_{k+1}}} = \dfrac{k+1}{\sqrt{4(k+1)^2-1}}$, $k = 0(1)n-1$.

An efficient algorithm can be useful to find x_k. We now construct a table for nodes and weights of Gaussian quadrature formula for some values of n.

$n = 1$	$-x_0 = x_1 = \dfrac{\sqrt{3}}{3} \simeq \cdot 5773502692$	$\omega_0 = \omega_1 = 1$
$n = 2$	$-x_0 = x_2 = \sqrt{\dfrac{3}{5}} = \cdot 7745966692,$ $x_1 = 0$	$\omega_0 = \omega_2 = \dfrac{5}{9} = \cdot 5555555556$ $\omega_1 = \dfrac{8}{9}$
$n = 3$	$-x_0 = x_3 = \cdot 8611363116$ $-x_1 = x_2 = \cdot 3399810436$	$\omega_0 = \omega_3 = \cdot 3478548451$ $\omega_1 = \omega_2 = \cdot 6521451549$
$n = 4$	$-x_0 = x_4 = \cdot 9061798459$ $-x_1 = x_3 = \cdot 5384693101$ $x_2 = 0$	$\omega_0 = \omega_4 = \cdot 2369268851$ $\omega_1 = \omega_3 = \cdot 4786286705$ $\omega_2 = \cdot 5688888889$

For the integral of the type $\int_a^b f(t)\,dt$, if we want to use Gaussian quadrature formula, then we change the variable as follows :

$t = \dfrac{b-a}{2}x + \dfrac{a+b}{2}$, so that we have,

$$I = \int_a^b f(t)\,dt = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right)dx \qquad \ldots\ldots (36)$$

The quadrature formula is then of the form,

$$Q_n = \frac{b-a}{2}\sum_{k=0}^{n}\omega_k f\left(\frac{b-a}{2}x_k + \frac{a+b}{2}\right) = \frac{b-a}{2}\sum_{k=0}^{n}\omega_k f(t_k),$$

where $t_k = \dfrac{b-a}{2}x_k + \dfrac{a+b}{2}$. $\qquad \ldots\ldots (37)$

The error term for the Gaussian quadrature formula is given by,

$$E_n(f) = \frac{(b-a)^{2n+3}((n+1)!)^4}{((2n+2)!)^3(2n+3)} f^{(2n+2)}(\xi) \qquad \ldots\ldots (38)$$

where $a < \xi < b$.

In particular, $E_1(f) = \dfrac{1}{135}\left(\dfrac{b-a}{2}\right)^5 f^{(4)}(\xi)$.

$$E_2(f) = \frac{1}{15750}\left(\frac{b-a}{2}\right)^7 f^{(6)}(\xi),$$

$$E_3(f) = \frac{1}{3472875}\left(\frac{b-a}{2}\right)^9 f^{(8)}(\xi).$$

It may be noted that Gaussian quadrature formula is also useful for improper integrals.

Example 3. Compute the integrals by Gaussian quadrature formula.

(i) $I_1 = \int_0^1 \frac{4}{1+x^2}dx$, (ii) $I_2 = \int_0^{\frac{\pi}{2}} x\cos x\, dx$, for different values of n.

Ans. Clearly exact values of $I_1 = \pi$ and $I_2 = \frac{\pi}{2} - 1$. The following table is the calculated values of I_1, I_2 by Gaussian quadrature formula.

n	I_1		I_2	
	Q_n	E_n	Q_n	E_n
1	3·1475409836	– 0·0059483300	0·563562244208	0·007234082587
2	3·1410681400	0·0005245136	0·570851127976	– 0·000054801181
3	3·1416119052	– 0·0000192517	0·570796127158	0·000000199637
4	3·1415926399	0·0000000138	0·570796327221	– 0·000000000426
5	3·1415926112	0·0000000424	0·570796326794	0·000000000001
6	3·1415926563	– 0·0000000027	0·570796326796	– 0·000000000001

§ Summary :

In this unit, Newton-Cotes quadrature formulae for closed type and open type are described. From closed type formula, the trapezoidal, Simphson 1/3 rule, etc. are derived. Romber's integration is discussed here. The Gaussian quadrature, particularly Gauss-Legendre is extensively studied. The merit of this method is illustrated by example.

EXERCISES

1. Find the coefficients that make the quadrature formula

$\int_0^3 f(x)dx \approx c_1 f(-1) + c_2 f(0) + c_3 f(2) + c_4 f(4)$, such that it is exact for cubic polynomials.

2. Veriry whether Simpson's $\frac{1}{3}$ rule is exact for $f(x)=x^3$ on [0, 2] or not.

3. Use Newton-Cotes quadrature formulae for $n = 1, 2, 3, 4, 5$ for the following integrals correct to that decimal places permitted by the corresponding formulae :

(i) $\int_{0\cdot1}^{1} x^{\cdot21}\sqrt{ln\left(\frac{1}{x}\right)}\,dx$, (ii) $\int_{1}^{2}\frac{\sin x}{x}dx$, (iii) $\int_{2}^{3}\frac{dx}{(1+x^2)^{1/2}}$

(iv) $\int_{0}^{1}\frac{dx}{x^2+e^{4x}}$, (v) $\int_{1}^{2}\frac{dx}{2+3x}$, (vi) $\int_{1}^{3}\frac{x}{1+x^2}dx$.

4. Use Romberg procedure to find the values of the following integrals correct six significant figures.

(i) $\int_{0}^{0\cdot95}\frac{dx}{1-x}$, (ii) $\frac{1}{\pi}\int_{0}^{\pi}\cos(x\sin\phi)\,d\phi$, for $x = 1, 3$ and 5.

(iii) $\int_{0}^{1}(1+\sin 10\pi x)\,dx$, (iv) $\int_{0}^{1}\sqrt{x}\,dx$, (v) $\int_{0}^{1}\frac{dx}{\sqrt{e^x+x-1}}$.

5. Compute the nodes x_k and the weights ω_k of the Gaussian quadrature formulae for $n = 4, 5, 7, 9$ by means of a suitable matrix algorithm.

6. Use Gauss quadrature rule to approximate the following integral.

(i) $\int_{0}^{3}e^{-4x^2}dx$, for $n = 3$, (ii) $\int_{-1}^{1}e^{-x^2}\sin 2x\,dx$, for $n = 4$.

(iii) $\int_{0}^{1}\frac{x^4-1}{x+1}dx$, for $n = 2$, (iv) $\int_{0\cdot1}^{1}x^{-0\cdot7}e^{-0\cdot4x}\cos x\,dx$, for $n = 3$,

(v) $\int_{\cdot2}^{1}\frac{(1-e^{-x})^{1/2}}{x}dx$, for $n = 4$, (vi) $\int_{1}^{2}\log\left(2\sin\frac{t}{2}\right)dt$, for $n = 5$.

7. Determine the abscissas and weights in the Gauss formula.

$$\int_{0}^{1}f(x)\log\frac{1}{x}dx = a_1f(x_1)+a_2f(x_2).$$

8. Compare the results obtained in the problems of exercise 3 by using n-point Gauss quadrature formula for $n = 2, 3, 4, 5$.

Unit 8 ❐ Numerical Solution of Ordinary Differential Equations : Initial Value Problems

§ Objectives

After going through this unit you will be able to learn about—

- Taylor series method to solve ODE
- Euler's and modified Euler's methods
- Runge-Kutta methods
- Adams-Bashforth-Moulton method
- Milne's method
- Analysis of stability

Let us first consider the initial value problem for a first order ordinary differential equation.

$$\frac{dy}{dx} = f(x, y) \text{ with initial condition } y(x_0) = y_0, \qquad \ldots\ldots (1)$$

where the function $f(x, y)$ is continuous in a domain D of xy plane and (x_0, y_0) is a point on that domain. Now, we say that $y(x)$ is a solution of the differential equation (1) on $[a, b]$, if for all $x \in [a, b]$,

(i) $(x, y(x)) \in D$,

(ii) $y(x_0) = y_0$,

and (iii) $y'(x)$ exist and $y'(x) = f(x, y(x))$, hold.

To ensure the solution to the problem (1) be unique, we consider the stability of the solution near the point (x_0, y_0). There is a powerful condition which ensures the stability of the solution. We present it in the manner of a theorem.

Theorem. Let $f(x, y)$ be a continuous function of x, y in domain D and let (x_0, y_0) be an interior point of D. Let us assume that $f(x, y)$ satisfies Lipschitz condition i.e.,

$|f(x, y_1) - f(x, y_2)| \leq k|y_1 - y_2|$ for all $(x, y_1), (x, y_2) \in D$ and for some constant $k \geq 0$ (known as Lipschitz constant). Then for a suitably chosen interval $[x_0 - \delta, x_0 + \delta]$, there is a unique solution $y(x)$ to the problem (1).

Before going into any specific numerical technique to solve the initial value problem (1), i.e., to find some approximate solution within our desired accuracy level, we first classify the techniques. Usually, we compute explicit or implicit solutions $y_j \approx y(x_j)$ at some equi-spaced discrete points $x_j = x_0 + jh$, $j = 0, 1, 2, \ldots$ in the domain D, where h is a suitable step-length. The points x_j are known as grid or mesh points and y_j is used to denote the estimate of $y(x_j)$. Now, if we compute y_{j+1} using a specific technique where we need only the previous approximate y_j, we call this specific technique or method, a single step method. If instead, we need multiple approximates, say $y_j, y_{j-1}, \ldots$ etc. , to compute y_{j+1}, then the method is called a multi-step method. Both the above methods are explicit in the sense that they do not require any approximate beyond y_{j+1}, including y_{j+1}, in those methods. If, we require further the approximates $y_{j+1}, y_{j+2}, \ldots$ etc., to compute y_{j+1}, then the methods are called implicit. In all the techniques, we must aware of the fact that apart from the truncation error, the accumulated round-off errors arose from each step.

§ Single-Step Methods :

(I) Taylor series approximation :-

A general single-step method for estimating $y(x_1)$, $y(x_2)$, ..., etc., in the small neighbourhood of the initial point (x_0, y_0), is the method of the Taylor series with remainder term R_{p+1},

$$y(x) = y(x_0) + \frac{(x-x_0)}{1!}\, y'(x_0) + \frac{(x-x_0)^2}{2!}\, y''(x_0) + \ldots$$

$$+ \frac{(x-x_0)^p}{p!}\, y^{(p)}(x_0) + R_{p+1}, \qquad \ldots\ldots (2)$$

where $R_{p+1} = \dfrac{(x-x_0)^{p+1}}{(p+1)!}\, y^{(p+1)}(x_0 + \theta h)$, $0 < \theta < 1$,

and, h = the step length

$= x_{k+1} - x_k$, $\forall\ k = 0, 1, \ldots$

Replacing, x_0 by x_k and x by x_{k+1}, the $(k+1)$-th approximate y_{k+1}, is given by,

$$y_{k+1} = y_k + \frac{h}{1!}\, y'_k + \frac{h^2}{2!} y''_k + \ldots + \frac{h^p}{p!} y^{(p)}{}_k \qquad \ldots\ldots (3)$$

$\forall\ k = 0, 1, \ldots$

assuming that $y(x)$ has sufficient smoothness (i.e., has continuous higher order derivatives) and R_{p+1} is negligible with respect to our desired accuracy level. The computations of y', y'', ... etc., can be made by the following procedure (which can be at times messy),

$$y' = f(x, y),$$

$$y'' = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\, y' = \frac{\partial f}{\partial x} + f\frac{\partial f}{\partial y}.$$

$$y''' = \frac{\partial^2 f}{\partial x^2} + 2f\,\frac{\partial^2 f}{\partial x \partial y} + f^2\frac{\partial^2 f}{\partial y^2} + \frac{\partial f}{\partial x}\cdot\frac{\partial f}{\partial y} + f\left(\frac{df}{dy}\right)^2,$$

.....

.....

There is another useful way to compute y-values by Taylor series method. Let us write the solution to the problem (1) in the form,

$$y(x) = y(x_0) + c_1(x - x_0) + c_2(x - x_0)^2 + \ldots\ldots, \qquad \ldots\ldots (4)$$

where c's are constants which we compute recursively. We illustrate this procedure through an example.

Example 1. Consider the initial value problem,

$$\frac{dy}{dx} = -2xy^2,\ y(0) = 1.$$

Expanding $y(x) = y(x_k + x - x_k)$ about x_k we get,

$$y(x) = y_k + c_1(x - x_k) + c_2(x - x_k)^2 + c_3(x - x_k)^3 + \ldots\ldots\ldots\ldots$$

where $c_i = \dfrac{y^{(i)}(x_k)}{i!}$.

We substitute this series in the differential equation, $\dfrac{dy}{dx} = -2xy^2$ and compute the coefficients of the like powers of $h = x - x_k$ on both sides, i.e.,

$$c_1 + 2c_2h + 3c_3h^2 + 4c_4h^3 + \ldots\ldots\ldots\ldots$$
$$= -2(x_k + h)(y_k + c_1h + c_2h^2 + c_3h^3 + \ldots\ldots)^2.$$

Comparing after some simple manipulations, we have,

$$c_1 = -2x_k y_k^2,$$

$$c_2 = -(y_k + 2c_1 x_k) y_k,$$

$$c_3 = -(4c_1 y_k + 2x_k (c_1^2 + 2c_2 y_k)) / 3,$$

$$c_4 = -\left(\frac{1}{2}c_1^2 + c_2 y_k + x_k (c_1 c_2 + c_3 y_k)\right),$$

.....

Clearly, c_i's as described above, are computed recursively. Now, we find approximate y_{k+1} at $x_{k+1} = x_k + h$, through the expression,

$$y_{k+1} = y_k + c_1 h + c_2 h^2 + c_3 h^3 .....$$

The results with $h = 0{\cdot}1$ and the corresponding error terms $e_k = y(x_k) - y_k$ for $k = 1, 2, 3, 4, 5$ are given in the following table where we have considered derivatives upto 4-th order.

x_k	y_k	c_1	c_2	c_3	c_4	e_k
0·0	1·0000000	0·0000000	–1·0000000	0·0000000	1·0000000	–
0·1	0·9901000	–0·1960596	–0·9414743	0·3805494	0·8567973	–0·0000010
0·2	0·9615455	–0·3698279	–0·7823272	0·6565037	0·4997400	–0·0000071
0·3	0·9174459	–0·5050242	–0·5637076	0·7736352	0·0913102	–0·0000147
0·4	0·8620892	–0·5945582	–0·3331480	0·7423249	–0·2247569	–0·0000202
0·5	0·8000218	–0·6400348	0·1279930	0·6144390	–0·3891673	–0·0000218

Taylor series method in some cases gives excellent approximation but it has the disadvantage that the set of recursion formulas are different for different differential equations. So we have to find out in each time a new set of recursion formulas for a new initial value problem.

(II) Euler's Method :–

Let us consider again the problem (1), i.e.,

given, $\frac{dy}{dx} = f(x, y)$ with $y(x_0) = y_0$, we have to find out approximates y_j at $x_j = x_0 + jh$ for suitable step length h. At $x = x_0$, the slope of the tangent to the desired

curve $y = y(x)$ that satisfies the differential equation, is $y'(x_0) = f(x_0, y_0)$. The simplest method of finding an approximation to the solution $y(x)$ is by its tangents on different points, i.e., through linearization. For, $x_k = x_0 + kh$, $k = 0, 1, 2, \ldots$, we find y_k, the estimates of the exact solution $y(x_k)$ by the following formula,

$$y_{k+1} = y_k + hf(x_k, y_k), \; k = 0, 1, 2, \ldots \qquad \ldots\ldots (5)$$

which is obtained either by truncating the R.H.S. of (3) after second term or by replacing $y'(x_k)$ by its forward difference approximate $\frac{y_{k+1} - y_k}{h}$ in the formula $y'(x_k) = f(x_k, y(x_k))$. This method is known as Euler's method. It is useful for obtaining the starting solutions for other methods. From the geometrical point of view it is also known as polygonal method. We illustrate the use of the method by estimating the values of $y' = -2xy^2$, $y(0) = 1$ at mesh points, x_k for $h = 0{\cdot}1$ and $0{\cdot}01$. In Euler's method unless the step lengths are very small, truncation error will be large and the estimates y_k's are inaccurate.

		$h = 0{\cdot}1$		$h = 0{\cdot}01$	
x_k	$y(x_k)$	y_k	e_k	y_k	e_k
0·0	1·00000	1·00000	–	1·00000	–
0·1	0·99010	1·00000	–0·00990	0·99107	–0·00097
0·2	0·96154	0·98000	–0·01846	0·96330	–0·00176
0·3	0·91743	0·94158	–0·02415	0·91969	–0·00226
0·4	0·86207	0·88839	–0·02632	0·86448	–0·00242
0·5	0·80000	0·82525	–0·02525	0·80229	–0·00229

Therefore, with the increase of step-length, the error grows up.

(III) Modified Euler Method :–

Another useful method for starting solutions is the modified Euler method. The name indicates, it is obtained by some improvement or modification of the Euler's method. Let y_k be the k-th estimate of $y(x)$ at $x = x_k$ with step length h and y_{2k} be $2k$-th estimate of $y(x)$ at $x = x_k$ with step-length $\frac{h}{2}$, by the Euler's method. Then

we have, approximately,

$$y_k \approx y(x_k) + c_1 h + 0(h^2)$$

and $$y_{2k} \approx y(x_k) + c_1 \frac{h}{2} + 0(h^2),$$

where c_1 is a constant.

Now, using the Richardson extrapolation, we find the extrapolate $\tilde{y}$ by the formula,

$$\tilde{y}(x_k) = 2y_{2k} - y_k \approx y(x_k) + 0(h^2),$$

which is relative to $y(x)$, is of second order.

Therefore, if we write,

$$y_{k+1}^{(1)} = y_k + hf(x_k, y_k),$$

then with the step-length $\frac{h}{2}$ we have,

$$y_{k+1/2}^{(2)} = y_k + \frac{h}{2} f(x_k, y_k),$$

and $$y_{k+1}^{(2)} = y_{k+1/2}^{(2)} + \frac{h}{2} f\left(x_k + \frac{h}{2}, y_{k+1/2}^{(2)}\right),$$

so that by Richardson extrapolation for $y_{k+1}^{(2)}$ and $y_{k+1}^{(1)}$ we have the estimate y_{k+1} by the formula,

$$\begin{aligned} y_{k+1} &= 2y_{k+1}^{(2)} - y_{k+1}^{(1)} \\ &= 2y_k + hf(x_k, y_k) + hf\left(x_k + \frac{h}{2}, y_{k+1/2}^{(2)}\right) - y_k - hf(x_k, y_k) \\ &= y_k + hf\left(x_k + \frac{h}{2}, y_k + \frac{h}{2} f(x_k, y_k)\right). \end{aligned}$$

In a more compact form, we can write,

$$\left.\begin{aligned} k_1 &= f(x_k, y_k) \\ k_2 &= f\left(x_k + \frac{h}{2}, y_k + \frac{1}{2} hk_1\right) \\ \text{and } y_{k+1} &= y_k + hk_2 \end{aligned}\right\} \quad \ldots\ldots (6)$$

This procedure is known as improved or modified Euler method. Here in a single-step we require two evaluation of the function $f(x, y)$.

Now we shall discuss about the errors occurred in the methods and corresponding order of convergence's.

For any single-step method, we can write the general scheme as follows :

$$y_{k+1} = y_k + h\phi\,(x_k, y_k, h),\ k = 0, 1, 2, \ldots, \qquad \ldots\ldots (7)$$

where the function $\phi(x_k, y_k, h)$ describes how the new estimate y_{k+1} can be computed from the information (x_k, y_k) with step-length h. For example, in Euler's method.

$$\phi(x_k, y_k, h) = f(x_k, y_k),$$

which does not depend on h. Again for Taylor series algorithm we find,

$$\phi(x_k, y_k, h) = c_1 + c_2 h + c_3 h^2 \ldots\ldots,$$

where coefficients c_i depend on the differential equation concerned, i.e., on $f(x, y)$ and the point (x_k, y_k). Next, we call a single-step method as consistent by the following definition.

Definition. A single-step method is called consistent with the differential equation $\frac{dy}{dx} = f(x, y)$, if and only if $\phi\,(x, y, 0) = f(x, y)$.

Clearly, Euler's method is consistent.

Now, if we neglect the rounding errors, then we can define the following,

Definition. The local discretization error d_{k+1} at x_{k+1} is defined by the expression,

$$d_{k+1} = y(x_{k+1}) - y(x_k) - h\phi(x_k, y(x_k), h) \qquad \ldots\ldots (8)$$

Local discretization error d_{k+1} indicates how well the exact solution fulfilled by the formula (7). In case of Euler's method and also in Taylor series method, d_{k+1} is just the difference between the exact value $y(x_{k+1})$ and the computed approximate y_{k+1} where we assumed that at x_k we have the exact value $y(x_k)$. Since in almost all step we use the approximates to the exact value, therefore we require the total error between computed value and the exact value.

Definition. The global discertization error g_k at x_k is defined by,

$$g_k = y(x_k) - y_k \qquad \ldots\ldots (9)$$

To estimate g_k, the function $\phi(x_k, y_k, h)$ should satisfy the Lipschitz condition with respect to the variable y in a suitably chosen domain B, i.e.,

$$|\phi(x, y, h) - \phi(x, y^*, h)| \le K|y - y^*|, \qquad \ldots\ldots (10)$$

for $(x, y, h), (x, y^*, h) \in B$ and $0 < K < \infty$.

For Euler's method this is just the condition that $f(x, y)$ is Lipschitz continuous, i.e., the condition for existence and uniqueness of the solution to the problem.

Now, from (8) and (9) we have,

$$g_{k+1} = g_k + h[\phi(x_k, y(x_k), h) - \phi(x_k, y_k, h)] + d_{k+1}.$$

Hence, by (10),

$$\begin{aligned} |g_{k+1}| &\le |g_k| + h\,|\phi(x_k, y(x_k), h) - \phi(x_k, y_k, h)| + |d_{k+1}| \\ &\le |g_k| + hK\,|y(x_k) - y_k| + |d_{k+1}| \\ &= (1 + hk)|g_k| + |d_{k+1}| \end{aligned}$$

Therefore, if we assume,

$\max\limits_k |d_k| \le \Delta$, then we have,

$$|g_{k+1}| \le (1 + hK)|g_k| + \Delta, \ \forall\ k = 0, 1, 2, \ldots, \qquad \ldots\ldots (11)$$

and the following theorem,

Theorem. The global discretization error g_n at the point $x_n = x_0 + nh$ is bounded by,

$$|g_n| \le \frac{\Delta}{hK}\left(e^{nhk} - 1\right) \le \frac{\Delta}{hK} e^{nhk} \qquad \ldots\ldots (12)$$

For Euler's method,

$$\text{if } \Delta = \max_{k=0,1,\ldots,n-1} |d_{k+1}| \le \frac{1}{2} h^2 \max_{x_0 \le \xi \le x_n} |y''(\xi)| = \frac{1}{2} h^2 M,$$

$$\text{then, } |g_n| \le \frac{hM}{2K} e^{K(x_n - x_0)}, \text{ so that as } h = \frac{x_n - x_0}{n} \to 0,\ y_n \to y(x_n) \text{ at } x_n.$$

We say that the convergence is linear and Euler's method is of order one.

Definition. A single-step method is said to be of order p, if its local discretization error d_k satisfies the relation,

$\max\limits_{1 \le k \le n} |d_k| \le \Delta = \text{const. } h^{p+1} = 0(h^{p+1})$, so that the global discretization error g_n is bounded by,

$$|g_n| \leq \frac{\text{constant}}{K} e^{nhK} . hP = 0(h^p). \qquad \ldots\ldots (13)$$

For Taylor series method we have,

$$d_{k+1} = \frac{h^{p+1}}{(p+1)!} y^{(p+1)} (x_k + \theta h),\ 0 < \theta < 1, \qquad \ldots\ldots (14)$$

and for modified Euler's method,

$$d_{k+1} = y(x_{k+1}) - y(x_k) - hf(x_k + h/2,\ y(x_k) + \frac{h}{2} f(x_k,\ y(x_k)))$$

$$= \frac{1}{6} \left(\frac{1}{4} G + Ff_y\right) h^3 + 0(h^4),$$

where $G = f_{xx} + 2ff_{xy} + f^2 f_{yy}$ and $F = f_x + ff_y$.

Thus the modified Euler method is of second order.

§ Runge-Kutta method for Differential Equations of Higher Orders and Systems :

In this section we shall describe the generalization of Runge-Kutta method for differential equations of higher orders and for systems with first order simultaneous equations. We restrict ourselves to second order differential equations and systems consist of two simltaneous equations. The generalization is straight-forward.

Consider initial value problem,

$y'' = f(x,\ y,\ y')$ with initial conditions,

$y(x_0) = y_0,\ y'(x_0) = y'_0$ in a domain D. $\qquad \ldots\ldots (15)$

If we put $y' = z$, then we have the following system of two first order equations,

$y' = z$

$z' = f(x,\ y,\ z)$

with initial conditions, $y(x_0) = y_0,\ z(x_0) = y'_0 = z_0$ (say).

This is a special case of system of two first order equations,

$$\frac{dy}{dx} = y' = F(x,\ y,\ z)$$

$$\frac{dz}{dx} = z' = G(x, y, z) \qquad \ldots\ldots (17)$$

with initial conditions $y(x_0) = y_0$, $z(x_0) = z_0$.

The generalization of the 4-th order Runge-Kutta method for the initial value problem (17) is as follows :

Let $x_j = x_0 + jh$, $j = 0, 1, 2, \ldots\ldots$ Then find,

$$k_1 = hF(x_j, y_j, z_j), \qquad l_1 = h\,G(x_j, y_j, z_j),$$

$$k_2 = hF\left(x_j + \frac{h}{2}, y_j + \frac{k_1}{2}, z_j + \frac{l_1}{2}\right), \qquad l_2 = h\,G\left(x_j + \frac{h}{2}, y_j + \frac{k_1}{2}, z_j + \frac{l_1}{2}\right),$$

$$k_3 = hF\left(x_j + \frac{h}{2}, y_j + \frac{k_2}{2}, z_j + \frac{l_2}{2}\right), \qquad l_3 = h\,G\left(x_j + \frac{h}{2}, y_j + \frac{k_2}{2}, z_j + \frac{l_2}{2}\right),$$

$$k_4 = hF\,(x_j + h, y_j + k_3, z_j + l_3), \qquad l_4 = h\,G(x_j + h, y_j + k_3, z_j + l_3),$$

$$k = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \qquad l = \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4) \qquad \ldots\ldots (18)$$

The new values at x_{j+1} are then,

$$y_{j+1} = y_j + k,\ z_{j+1} = z_j + l \text{ at } x_{j+1} = x_j + h \qquad \ldots\ldots (19)$$

(18) and (19) are the scheme for numerical solutions of the initial value problem (17), hence also for (15) by 4-th order Runge-Kutta method. Clearly, the generalizations to the higher order and systems consist of more than two equations are straightforward.

The scheme has the local discretization error,

$$\left\| \vec{d}_{j+1} \right\| = O(h^5) \text{ where } \|.\| \text{ is the Euclidean norm.}$$

Example 2. Solve $y'' = 5x - 3xy' - 4x^2y$ with initial conditions

$y(0) = 1$, $y'(0) = 1$. Use step-length $h = 0{\cdot}1$.

Solution : Put $y' = z \equiv F(x, y, z)$, therefore,

$$z' = 5x - 3xz - 4x^2y \equiv G(x, y, z),$$

and $y(0) = 1$, $z(0) = 1$.

Using the following computational table,

x	y	z	$G(x, y, z)$	$k_i = hF$	$l_i = hG$
x_j	y_j	z_j	$G(-, -, -)$	k_1	l_1
$x_j + \frac{h}{2}$	$y_j + \frac{k_1}{2}$	$z_j + \frac{l_1}{2}$	$G(-, -, -)$	k_2	l_2
$x_j + \frac{h}{2}$	$y_j + \frac{k_2}{2}$	$z_j + \frac{l_2}{2}$	$G(-, -, -)$	k_3	l_3
$x_j + h$	$y_j + k_3$	$z_j + l_3$	$G(-, -, -)$	k_4	l_4
x_{j+1}	$y_{j+1} = y_j + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$, $z_{j+1} = z_j + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4)$				

We find the vlaues of y and z at five different values of x :

x	y	z
0·00	1·00000	1·00000
0·10	1·10030	1·00850
0·20	1·20203	1·02683
0·30	1·30557	1·04220
0·40	1·40990	1·04043
0·50	1·51262	1·00766

§ Multi-Step Methods :

Now we consider a more general approach for solving numerically an initial value problem of the type (1). We call a method is a linear m-step method, if we include m previous approximates $y_k, y_{k-1}, \cdots y_{k-m+1}$ to determine y_{k+1} at x_{k+1}. For the equidistant support abscissa $x_{k-j} = x_k - jh$, $j = 0, 1, ..., k = 0, 1, ..., a$ general linear multi-step method is defined through the formula,

$$\sum_{j=0}^{m} a_j y_{s+j} = h \sum_{j=0}^{m} b_j f(x_{s+j}, y_{s+j}),\ m \geq 2 \qquad \text{..... (20)}$$

where $s = k - m + 1$, and we must have $a_m \neq 0$. Without loss of generality we set $a_m = 1$. When the coefficients a_0, b_0 are not allowed to vanish simultaneously, we

define the method (20) as a real m-step method. If $b_m = 0$, the method is an explicit method, otherwise it is an implicit m-step method. To determine the parameters of (20) we first define the following.

Definition. The linear multi-step method (20) is called of order p if in the expansion of the local discretization error d_{k+1} into a power series in h at an arbitary abscissa x, the following holds,

$$d_{k+1} = \sum_{j=0}^{m} [a_j y(x_{s+j}) - h b_j f(x_{s+j}, y(x_{s+j}))]$$

$$= c_0 y(x) + c_1 h y'(x) + \ldots\ldots + c_p h^p y^{(p)}(x) + c_{p+1} h^{p+1} y^{(p+1)}(x) + \ldots\ldots\ldots\ldots,$$

with $c_0 = c_1 = \ldots\ldots c_p = 0$ and $c_{p+1} \neq 0$ ..... (21)

In other words, the local discretization error is of the order $0(h^{p+1})$. The definition (21) does not specify x precisely. It can be proved that p does not depend on x from which the power series is formed and the coefficient c_{p+1} is independent of x. With a suitable choice of x we may represent the coefficients c_l as simple functions of the parameters a_j, b_j. If we choose $x = x_s$, then we have by using Taylor series expansion for $y(x_{s+j})$ and $y'(x_{s+j})$ with implicit assumption that $y(x)$ is sufficiently smooth, i.e., has sufficient number of higher order continuous derivatives,

$$\left.\begin{aligned} y(x_{s+j}) = y(x_s + jh) &= \sum_{l=0}^{q} \frac{(jh)^l}{l!} y^{(l)}(x_s) + R_{q+1}, \\ y'(x_{s+j}) = y'(x_s + jh) &= \sum_{l=0}^{q-1} \frac{(jh)^l}{l!} y^{(l+1)}(x_s) + \bar{R}_q. \end{aligned}\right\} \quad \ldots\ldots (22)$$

Comparing (21) and (22) we have,

$$\left.\begin{aligned} c_0 &= a_0 + a_1 + \ldots + a_m \\ c_1 &= a_1 + 2a_2 + \ldots + m a_m - (b_0 + b_1 + \ldots + b_m) \\ c_2 &= \frac{1}{2!}(a_1 + 2^2 a_2 + \ldots + m^2 a_m) - \frac{1}{1!}(b_1 + 2b_2 + \ldots + m b_m) \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ c_l &= \frac{1}{l!}(a_1 + 2^l a_2 + \ldots + m^l a_m) - \frac{1}{(l-1)!}(b_1 + 2^{l-1} b_2 + \ldots \\ &\qquad + m^{l-1} b_m) \text{ for } l = 2, 3, \ldots, q. \end{aligned}\right\} \quad \ldots\ldots (23)$$

For a given m, we seek the parameters a_j, b_j in such a way that the order is maximal. It varies with the structure of multi-step methods. The investigation of multistep methods are further associated with the following two characteristic polynomials,

$$\rho(z) = \sum_{j=0}^{m} a_j z^j, \quad \sigma(z) = \sum_{j=0}^{m} b_j z^j \qquad \ldots\ldots (24)$$

We call a multi-step method (20) is consistent if its order p is at least one. The method is said to be convergent if $\lim_{h \to 0} y_{k-m+1} = y(x_{k-m+1})$. A necessary condition for convergence is that the zeros z_i of $\rho(z)$ are such that $|z_i| \leq 1$ and all zeros on the unit circle are simple. This condition is usually known as stability condition. If this condition is not fulfilled, we call the method as strongly unstable. Now we shall describe three methods based on intergration.

Adams-Bashforth Method.

If we intergrate the different equation, $\frac{dy}{dx} = f(x, y)$ between x_k and x_{k+1}, then we have an integral equation,

$$y(x_{k+1}) = y(x_k) + \int_{x_k}^{x_{k+1}} f(x, y(x))dx \qquad \ldots\ldots (25)$$

with $(n + 1)$ equidistant points $x_{k-n}, x_{k-n+1}, \ldots, x_k$, let us interpolate $f(x, y(x))$ by the formula,

$$f(x, y(x)) = L_n(x) + R_{n+1}(x) \qquad \ldots\ldots (26)$$

where $L_n(x)$ is the interoplation polynomial and $R_{n+1}(x)$ is the remainder.

Using Newton's backward formula for $L_n(x)$, i.e.,

$$L_n(x) = \sum_{j=0}^{n} \binom{u+j-1}{j} \nabla^j f_k \qquad \ldots\ldots (27)$$

and setting $\alpha_j = \int_0^1 \binom{u+j-1}{j} du$, $j = 0, 1, \ldots, n+1$, where $u = \frac{x - x_k}{h}$, we have,

$$y_{k+1} = y_k + h \sum_{j=0}^{n} \alpha_j \nabla^j f_k + d_{k+1}, \qquad \ldots\ldots (28)$$

$$d_{k+1} = \alpha_{n+1} h^{n+2} y^{(n+2)}(\xi), \; x_{k-n} < \xi < x_{k+1}, \qquad \ldots\ldots (29)$$

which is obtained by intergrating $R_{n+1}(x)$ from x_k to x_{k+1}, using divided difference form of $R_{n+1}(x)$, i.e.,

$$d_{k+1} = \int_{x_k}^{x_{k+1}} (x - x_{k-n}) \ldots (x - x_k) y'(x, x_{k-n}, \ldots, x_k) dx$$

$$= h^{n+2} y^{(n+2)}(\xi) \int_0^1 \binom{u+n}{n+1} du \qquad \ldots\ldots (30)$$

Thus we have the $(k+1)$-th estimate of $y(x_{k+1})$ by the following formula,

$$y_{k+1} = y_k + h \sum_{j=0}^{n} \alpha_j \nabla^j f_k \,, \quad k \geq n, \qquad \ldots\ldots (31)$$

which is known as $(n+1)$ step Adams-Bashforth formula. Clearly, it is an explicit formula of order $(n+1)$ where the local truncation error is given by (30). For $n = 0$, the formula leads to Euler's method.

Some computed values of α_j's are given by the following table :

Table for α_j :

j	0	1	2	3	4	5	.....
α_j	1	$\frac{1}{2}$	$\frac{5}{12}$	$\frac{3}{8}$	$\frac{251}{720}$	$\frac{95}{288}$	.....

So, we can write (31) as follows :

$$y_{k+1} = y_k + h\left[f_k + \frac{1}{2}\nabla f_k + \frac{5}{12}\nabla^2 f_k + \frac{3}{8}\nabla^3 f_k + \ldots\right] \qquad \ldots\ldots (32)$$

If we rewrite explicitly the four-step Adams-Bashforth formula by the ordinates, then we have,

$$y_{k+1} = y_k + \frac{1}{24} h\,(55 f_k - 59 f_{k-1} + 37 f_{k-2} - 9 f_{k-3})$$

and $$d_{k+1} = \frac{251}{720} h^5 y^{(5)} + 0(h^6) \qquad \ldots\ldots (33)$$

For starting values we may use any single-step method described earlier.

Example 3. Given the differential equation,

$$\frac{dy}{dy} = \frac{xy + 1{\cdot}88}{x^2 + y^2 + 4}$$ with initial condition $y(0) = 0$, we find by Taylor series method using step-length $h = 0{\cdot}1$, the first three values at $x = 0{\cdot}1, 0{\cdot}2, 0{\cdot}3$ as follows.

$y_1 = 0{\cdot}046959$, $y_2 = 0{\cdot}093671$, $y_3 = 0{\cdot}139890$.

The following is the difference table to compute approxiamaes y_j by the 4-step Adams-Bashforth method.

n	x_n	y_n	f_n	Δf_n	$\Delta^2 f_n$	$\Delta^3 f_n$
0	0·00	0·000000	0·470060			
				–0·000259		
1	0·10	0·046959	0·469741		–0·000517	
				–0·000776		–0·000028
2	0·20	0·093671	0·468965		–0·000545	
				–0·001321		0·000137
3	0·30	0·139890	0·467644		–0·000408	
				–0·001729		0·000006
4	0·40	0·185664	0·465915		–0·000402	
				–0·002131		0·000054
5	0·50	0·232158	0·463784		–0·000348	
				–0·002479		0·000058
6	0·60	0·278413	0·461305		–0·000290	
				–0·002769		
7	0·70	0·324407	0·458536			
8	0·80	0·370112				

Adams-Moulton Method.

Here we consider the integral equation (25) again, but we take $(n + 1)$ equidistant points x_{k-n+1}, x_{k-n+2}, ..., x_k, x_{k+1} as the interpolating points, i.e., we have included the estimate y_{k+1} in the interpolating polynomial. Thus, if we write,

$f(x, y(x)) = L_n(x) + R_{n+1}(x)$ and intergrating between x_k and x_{k+1}, we have,

$$y_{k+1} = y_k + \int_{x_k}^{x_{k+1}} L_n(x)dx + d_{k+1}, \qquad \text{..... (34)}$$

where, d_{k+1} is given by,

$$d_{k+1} = \int_{x_k}^{x_{k+1}} R_{n+1}(x)dx$$

$$= \int_{x_k}^{x_{k+1}} (x - x_{k-n+1}) \ \ldots \ (x - x_{k+1})y'(x, x_{k-n+1}, \ldots, x_{k+1})dx,$$

$$= y'(\xi', x_{k-n+1}, \ldots, x_{k+1}) \int_{x_k}^{x_{k+1}} (x - x_{k-n+1}) \ldots$$

$$(x - x_{k+1}) dx, \; x_k < \xi' < x_{k+1}.$$

Since $(x - x_{k-n+1}) \ldots (x - x_{k+1})$ does not change sign in $[x_k, x_{k+1}]$, therefore, setting $u = \dfrac{x - x_{k+1}}{h}$, we have,

$$d_{k+1} = h^{n+2} y^{(n+2)}(\xi'') \int_0^1 \binom{v+n-1}{n+1} dv \qquad \ldots\ldots (35)$$

where $x_{k-n+1} < \xi'' < x_{k+1}$, $v - 1 = u$.

Now using backward interpolation formula,

$$L_n(x) = \sum_{j=0}^{n} \binom{u+j-1}{j} \nabla^j f_{k+1}, \text{ we have,}$$

$$y_{k+1} = y_k + h \sum_{j=0}^{n} \beta_j \nabla^j f_{k+1} + d_{k+1} \qquad \ldots\ldots (36)$$

where $\beta_j = \int_0^1 \binom{v+j-2}{j} dv, \; j = 0, 1, \ldots, n+1$

and $d_{k+1} = \beta_{n+1} h^{n+2} y^{(n+2)}(\xi'')$ $\qquad \ldots\ldots (37)$

Therefore, we have an n-step implicit formula for estimate y_{k+1}, i.e.,

$$y_{k+1} = y_k + h \sum_{j=0}^{n} \beta_j \nabla^j f_{k+1}, \qquad \ldots\ldots (38)$$

which is known as Adams–Moulton formula of order $(n+1)$ with local discretization error d_{k+1} given by (37).

Table for β_j :

j	0	1	2	3	4	5	.....
β_j	1	$-\frac{1}{2}$	$-\frac{1}{12}$	$-\frac{1}{24}$	$-\frac{19}{720}$	$-\frac{3}{160}$	.....

Explicitly we can write (38) as :

$$y_{k+1} = y_k + h\left[f_{k+1} - \frac{1}{2}\nabla f_{k+1} - \frac{1}{12}\nabla^2 f_{k+1} - \frac{1}{24}\nabla^3 f_{k+1} \cdots\right] \qquad \ldots\ldots (39)$$

For 3-step Adams-Moulton method, if we transfer (39) into ordinate form, then we have,

$$y_{k+1} = y_k + \frac{1}{24} h\,[9f_{k+1} + 19f_k - 5f_{k-1} + f_{k-2}] \quad \ldots\ldots (40)$$

Since, the formula (40) is an implicit formula, therefore for computation we use a predictor-corrector formula with Adams-Bashforth formula (33) as predictor and (40) as corrector. i.e., we use a predictor-corrector scheme,

$$\left.\begin{aligned} y^p_{k+1} &= y_k + \frac{1}{24} h\,[55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3}], \\ y^c_{k+1} &= y_k + \frac{1}{24} h\,[9f(x_{k+1}, y^p_{k+1}) + 19f_k - 5f_{k-1} + f_{k-2}] \end{aligned}\right\} \ldots\ldots (41)$$

The error estimate for 4-th order Adams-Moulton predictor-corrector scheme is given by,

$$e^c_{k+1} \approx -\frac{19}{270}(y^c_{k+1} - y^p_{k+1}) \quad \ldots\ldots (42)$$

Example 4. Consider the problem of Example 3. Here again we take $h = 0{\cdot}1$ and first three approximates y_1, y_2, y_3 by Taylor series method. The computational table for Adams-Moulton predictor-corrector method (41) is as follows :

n	x_n	y_n	f_n	Δf_n	$\Delta^2 f_n$	$\Delta^3 f_n$
0	0·00	0·000000	0·470000			
				–0·000259		
1	0·10	0·046959	0·469741		–0·000517	
				–0·000776		–0·000028
2	0·20	0·093671	0·468965		–0·000545	
				–0·001321		0·000137
3	0·30	0·139890	0·467644		–0·000408	
				–0·001729		0·000186
4	0·40	0·185664	0·465915		–0·000359	
				–0·001680		
		0·186571	0·465964			–0·000080
		0·186573	0·465964		–0·000439	
				–0·002119		–0·000197

5	0·50	0·233077	0·463845		–0·000556	
				–0·002236		–0·000081
		0·233067	0·463728		–0·000440	
				–0·002120		
		0·233063	0·463844		–0·000349	0·000091
				–0·002469		0·000092
6	0·60	0·279320	0·461375		–0·000348	
				–0·002468		0·000057
		0·279326	0·461376		–0·000291	
				–0·002759		0·000059
7	0·70	0·325329	0·458617		–0·000232	
		0·325328	0·458617			
				–0·002991		
8	0·80	0·371042	0·455626			
		0·371042	0·455626			

Milne's Method :

Here the integral equation is formed by integrating the differential equation $\frac{dy}{dx} = f(x, y)$ over the interval $[x_{k-1}, x_{k+1}]$, i.e.,

$$y_{k+1} = y_{k-1} + \int_{x_{k-1}}^{x_{k+1}} f(x, y(x))dx \qquad \ldots\ldots (43)$$

We evaluate this integral by Simpson's $\frac{1}{3}$ rule with interpolating points x_{k-1}, x_k, x_{k+1}, i.e.,

$$y_{k+1} = y_{k-1} + \frac{h}{3}(f_{k-1} + 4f_k + f_{k+1}) + d_{k+1}, \qquad \ldots\ldots (44)$$

where $d_{k+1} = -\frac{h^5}{90} y^{(5)}(\xi)$, $x_{k-1} < \xi < x_{k+1}$.

If $y^{(5)}(x)$ does not vary strongly on $[x_{k-1}, x_{k+1}]$, then we approximate d_{k+1} by the following formula,

$$d_{k+1} \approx -\frac{h}{90}(f_{k-2} - 4f_{k-1} + 6f_k - 4f_{k+1} + f_{k+2}) \quad \ldots\ldots (45)$$

Since, the formula (44) is an implicit formula, we use a predictor-corrector scheme for practical computation. For the predictor formula, we intergrate the differential equation $\frac{dy}{dx} = f(x, y)$ over $[x_{k-3}, x_{k+1}]$ and find $(k+1)$ th estimate y_{k+1} of $y(x_{k+1})$ as follows :

$$y_{k+1} = y_{k-3} + \int_{x_{k-3}}^{x_{k+1}} f(x, y(x))dx \quad \ldots\ldots (46)$$

Using 3-point open type Newton-Cotes quadrature formula with interpolating points x_{k-2}, x_{k-1}, x_k we have,

$$y_{k+1} = y_{k-3} + \frac{4h}{3}(2f_{k-2} - f_{k-1} + 2f_k) + d_{k+1} \quad \ldots\ldots (47)$$

where, $d_{k+1} = \frac{14}{15}h^5 y^{(5)}(\xi),\ x_{k-3} < \xi < x_{k+1}$ $\ldots\ldots (48)$

Thus, we have the Milne's predictor-corrector scheme,

$$\left.\begin{aligned} y^p_{k+1} &= y_{k-3} + \frac{4h}{3}(2f_{k-2} - f_{k-1} + 2f_k) \\ \text{and} \quad y^c_{k+1} &= y_{k-1} + \frac{h}{3}(f_{k-1} + 4f_k + f(x_{k+1}, y^p_{k+1})) \end{aligned}\right\} \quad \ldots\ldots (49)$$

The error estimate is given by,

$$e^c_{k+1} \approx -\frac{1}{29}\left(y^c_{k+1} - y^p_{k+1}\right) \quad \ldots\ldots (50)$$

Example 5. Consider again the problem of Example 3. The estimates for $x = 0{\cdot}4$, $0{\cdot}5$, $0{\cdot}6$, $0{\cdot}7$, $0{\cdot}8$ are found by Milne's method as follows!

x_n	y_n
0·40	0·187460
	0·187461
0·50	0·233957
	0·233957

0·60	0·280229
	0·280228
0·70	0·326238
	0·326238
0·80	0·371961
	0·371962

§ Analysis of Stability :

Stability analysis is an important part in the study of the numerical methods to solve differential equations. Most of the methods used to solve differential equation are based on difference equation. To study the stability, the model differential and difference equations are defined in the following.

Model differential problem

For convenience and feasibility of analytical treatment and without loss of generality, stability analysis will be performed on the **model initial value differential equation**

$$y' = \lambda y, \quad y(0) = y_0 \qquad \ldots\ (51)$$

where λ is a constant and it may be a real or a complex number. The solution of this problems is

$$y = e^{\lambda t} y_0 \qquad \ldots\ (52)$$

In our treatment, let consider $\lambda = \lambda_R + i\lambda_I$, where λ_R and λ_I represent respectively the real and imaginary part of λ and $\lambda_R \leq 0$

Model difference problem

Similar to the differential problem, let us consider the single first order linear model initial value difference problem.

$$y_{n+1} = \sigma y_n, \quad n = 0, 1, 2, \ldots \qquad \ldots\ (53)$$

where y_0 is given and σ is, in general, a complex number. The solution of this problem is

$$y_n = \sigma^n y_0 \qquad \text{.... (54)}$$

It may be noted that the solution remains bounded only if $|\sigma| \le 1$

The connection between the exact solution and the difference solution is evident if we evaluate the exact solution at $t_n = nh$, for $n = 0,1,$ .... where $h > 0$ and

$$y_n = e^{\lambda t_n} y_0 = e^{\lambda nh} y_0 = \sigma^n y_0 \qquad \text{.... (55)}$$

where $\sigma = e^{\lambda h}$

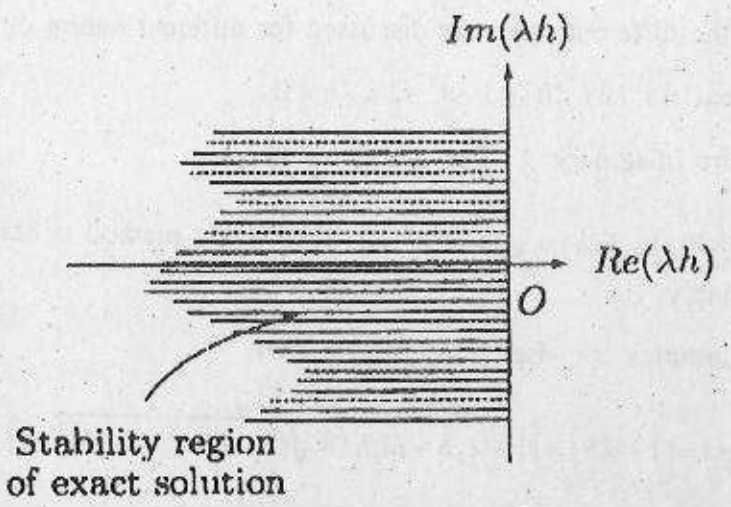


Fig. 1. Stability region of exact solution

If the exact solution is bounded then $|\sigma| = |e^{\lambda h}| \le 1$. This is possible if $Re(\lambda h) = \lambda_R h \le 0$.

That is, in the $Re(\lambda h) - Im(\lambda h)$ plane, the region of stability of the exact solution is that left half-plane as shown in Figure 1.

The single-step method is called **absolutely stable** if $|\sigma| \le 1$ and **relatively stable** if $|\sigma| \le e^{\lambda h}$. If λ is pure imaginary and $|\sigma| = 1$, then the absolute stability is called the **peroidic stability (P-stability).**

Stability of Euler's method

The solution scheme of (51) by Euler's method is

$$y_{n+1} = y_n + f(x_n, y_n) = y_n + \lambda h y_n = (1 + \lambda h) y_n. \quad \text{.... (56)}$$

The solution of this difference equation is

$$y_n = (1 + \lambda h)^n y_0 = \sigma^n y_0 \quad \text{.... (57)}$$

where $\sigma = 1 + \lambda h$

The numerical method is stable if $|\sigma| \leq 1$.

Now, the different cases are discussed for different nature of λ.

(i) Real λ: $|1 + \lambda h| < 1$ or $-2 < \lambda h < 0$.

(ii) Pure imaginary λ: Let $\lambda = iw$, w is real.

Then $|1 + iwh| = \sqrt{1 + w^2 h^2} > 1$ That is, the method is not stable when λ is pure imaginary.

(iii) Complex λ: Let $\lambda = \lambda_R + i\lambda_I$. Then

$$|\sigma| = |1 + \lambda h| = |1 + \lambda_R h + i\lambda_I h| = \sqrt{(1 + \lambda_R h)^2 + (\lambda_I h)^2} \leq 1.$$

It means λh lies inside the unit circle.

That is, only a small portion of the left half-plane is the region of stability for the Euler's method. This region is inside the circle $(1 + \lambda_R h)^2 + (\lambda_I h)^2 = 1$, which is shown in Figure 2.

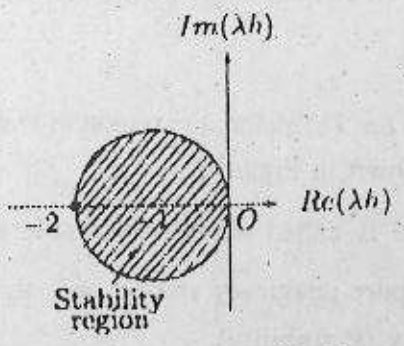


Fig. 2. Stability region of Euler's method

For any value of λh in the left half-plane and outside this circle, the numerical solution blows-up while the exact solution decays. Thus the numerical method is **conditionally** stable.

To get a stable numerical solution, the step size h must be reduced so that λh falls within the circle. If λ is real and negative then the maximum step size for stability is $0 \le h \le 2/|\lambda|$. The circle is only tangent to the imaginary axis. If λ is real and the numerical solution is unstable, then $|1 + \lambda h| > 1$, which means that $(1 + \lambda h)$ is negative with magnitude greater than 1. Since $y_n = (1 + \lambda h)^n y_0$, the numerical solutions exhibits oscillations with changes of sign at every step. This behavior of the numerical solutions is a good indicator of instability.

The numerical stability does not imply accuracy. A method can be stable even if it gives inaccurate result. From the stability point of view, our objective is to use the maximum step size h to reach the final destination at $x = x_n$. If h is large then number of function evaluations is low and needs low computational cost. This may not be the optimum h for acceptable accuracy, but, it is optimum for stability.

Stability of Runge-Kutta methods

Let us consider the second-order Runge-Kutta method for the model equation $y' = \lambda y$. Then,

$$k_1 = hf(x_n, y_n) = \lambda h y_n$$

$$k_2 = hf(x_n + h, y_n + k_1) = \lambda h(y_n + k_1) = \lambda h(y_n + \lambda h y_n)$$

$$= \lambda h(1 + \lambda h) y_n$$

and

$$y_{n+1} = y_n + \frac{1}{2}(k_1 + k_2) = y_n + \left(\lambda h + \frac{(\lambda h)^2}{2}\right) y_n$$

$$= \left(1 + \lambda h + \frac{\lambda^2 h^2}{2}\right) y_n \qquad \ldots. (58)$$

This expression confirms that the method is of second order accuracy. For stability $|\sigma| \le 1$ where

$$\sigma = 1 + \lambda h + \frac{\lambda^2 h^2}{2} \qquad (59)$$

Now, the stability is discussed for different cases of λ.

(i) Real λ : $1+\lambda h+\frac{\lambda^2 h^2}{2} \le 1$ or $-2 \le \lambda h \le 0$.

(ii) Pure imaginary λ: Let $\lambda = iw$. Then $|\sigma| = \sqrt{1+\frac{1}{4}w^4h^4} > 1$. That is, the method is unstable.

(iii) Complex λ: Let $1+\lambda h+\frac{\lambda^2 h^2}{2} = e^{i\theta}$ and find the complex roots, λh, of the polynomial for different values of θ. Note that $|\sigma| = 1$ for all values of θ.

The resulting stability region is shown in Figure 3.

When fourth-order Runge-Kutta method is applied to the model equation $y' = \lambda y$ then, $k_1 = \lambda h y_n$

$$k_2 = \lambda h\left(y_n + \frac{k_1}{2}\right) = \lambda h\left(1+\frac{\lambda h}{2}\right)y_n$$

$$k_3 = \lambda h\left(y_n + \frac{k_2}{2}\right) = \lambda h\left(1+\frac{1}{2}\lambda h+\frac{1}{4}\lambda^2 h^2\right)y_n$$

$$k_4 = \lambda h(y_n + k_3) = \lambda h\left(1+\lambda h+\frac{1}{2}\lambda^2 h^2+\frac{1}{4}\lambda^3 h^3\right)y_n$$

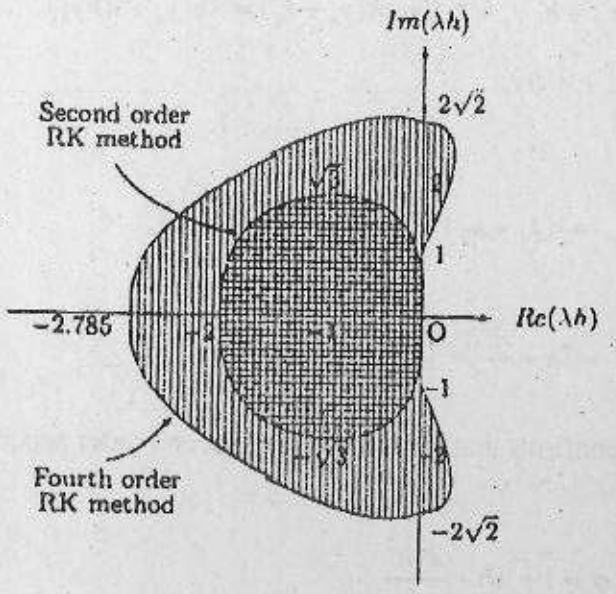


Fig. 3. Stability region of Runge-Kutta methods

Therefore

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$= \left\{1 + \lambda h + \frac{1}{2!}(\lambda h)^2 + \frac{1}{3!}(\lambda h)^3 + \frac{1}{4!}(\lambda h)^4\right\} y_n, \quad \ldots (60)$$

which confirms the fourth-order accuracy of the method.

For different λ, the stability of fourth-order Runge-Kutta method is discussed in the following :

(i) Real λ : $-2.785 \leq \lambda h \leq 0$.

(ii) Pure imaginary λ : $0 \leq |\lambda h| \leq 2\sqrt{2}$,

(iii) Complex λ : In this case, the stability region is obtained by finding the roots of the fourth-order polynomial complex coefficients :

$$1 + \lambda h + \frac{1}{2!}(\lambda h)^2 + \frac{1}{3!}(\lambda h)^3 + \frac{1}{4!}(\lambda h)^4 = e^{i\theta}$$

The region of stability is shown in Figure 3. It shows a significant improvement over the second-order Runge-Kutta scheme. In particular, it has a large stability region on the imaginary axis.

§ Summary :

In this unit, the numerical methods to solve first order differential equations are discussed. These methods are divided into two categories, viz., single-step and multi step methods. The Taylor's series approximation method and three single-step methods—Euler's, modified Euler's and Runge-Kutta are discribed to solve first-order ODE. Two multi-step methods—Adams-Bashforth-Moulton and Milne are discussed here. The stability analysis of Euler's and Runge-Kutta methods are also incorporated.

EXERCISES

1. Determine the exact solution of the initial value problem $y' = \frac{2x}{y^2} + 1$, $y(0) = 1$ and compute the approximate solutions in [0, 1],

(i) by Euler's method with step size $h = 0{\cdot}1, 0{\cdot}01$.

(ii) by Taylor series method with $h = 0{\cdot}1$ and $h = 0{\cdot}05$.

(iii) by Modified Euler's method with $h = 0{\cdot}2, 0{\cdot}1, 0{\cdot}05$.

Verify the orders of convergences in each case.

2. Solve the following systems of differential equations of Runge-Kutta method.

(i) $\frac{dx}{dt} = 1{\cdot}2x - x^2 - \frac{xy}{x + 0{\cdot}2}$,

$\frac{dy}{dt} = \frac{1{\cdot}5xy}{x + 0{\cdot}2} - y$, with initial conditions $x(0) = 1$, $y(0) = 0{\cdot}75$ and the step size $h = 0{\cdot}1$ in the interval [0, 1].

(ii) $x\frac{dy}{dx} + \frac{1}{2}y + z = 0$

$x\frac{dz}{dx} - y + \frac{1}{2}z = 0$, with initial conditions,

$y(0) = 0$, $z(0) = 1$ and step size $h = 0{\cdot}1$ in the interval [0, 1].

3. Solve the following second order equations by Runge-Kutta method.

(i) $y'' + 1{\cdot}2y' + 2y = 0$ with initial conditions $y(0) = 1$, $y'(0) = 0$ and step size $h = 0{\cdot}1$ in the interval [0, 1].

(ii) $y'' = xy + \frac{1}{1 + y^2}$, with initial conditions $y(0) = 0$, $y'(0) = 1$, and step size $h = 0{\cdot}1$ in the interval [0, 1].

4. Solve the following initial value problems by Adamas-Bashforth, Adams-Moulton and Milne's method and compare the results obtained. For starting values use any suitable single step method.

(i) $y' = \dfrac{1}{1+4x^2} - 8y^2$, $y(0) = 0$ in the interval [0, 1] with step size $h = 0{\cdot}1, 0{\cdot}2$.

(ii) $y' = \dfrac{1-x^2-y^2}{1+x^2+xy}$, $y(0) = 0$, in [0, 1] with step size $h = 0{\cdot}1$.

(iii) $y' = \dfrac{a}{\cos x}\, y \tan x$, $y(0) = 1$, in [0, 1] with $a = \dfrac{1}{0{\cdot}2+0{\cdot}01k}$, $k = 0, 1, 2, 3, 4, 5$ and step size $h = 0{\cdot}1$.

5. Find an explicit three-step method of maximal order, $a_0y_{k-2} + a_1y_{k-1} + a_2y_k + a_3y_{k+1} = h(b_0f_{k-2} + b_1f_{k-1} + b_2f_k)$, such that $a_0 = a_2 = 0$ and $a_3 = 1$.

6. Consider the approximate method for an initial value problem $y' = f(x, y)$, $y(x_0) = y_0$,

$$y_{k+1} = 4y_k - 3y_{k-1} - hf_{k-1}, \quad k \geq 1.$$

Determine its order.

7. Derive an implicit four-step method other than the described methods.

8. Prepare a report on stability analysis by consulting different reference books.

Unit 9 ❐ Two-Point Boundary Value Problems of Ordinary Differential Equaions

§ Objectives

After going through this unit you will be able to learn about—

- Finite difference scheme
- Solution of second order boundary value problem by finite difference method

Given a second order ordinary differential equation,

$$F(x, y, y', y'') = 0, \qquad \ldots\ldots (1)$$

by a two-point boundary value problem we mean :

Find a function $y \equiv y(x)$ inside the interval $[a, b]$, which satisfies the equation (1) and the boundary conditions,

$$\left.\begin{aligned} \phi_1(y(a), y'(a)) &= 0 \\ \phi_2(y(b), y'(b)) &= 0 \end{aligned}\right\} \qquad \ldots\ldots (2)$$

If both the equations (1) and (2) are linear, we call such a problem as linear boundary value problem. Such a problem can be written as :

$$y''(x) + f(x)\, y'(x) + g(x)\, y(x) = q(x) \qquad \ldots\ldots (3)$$

with boundary conditions,

$$\left.\begin{aligned} \alpha_0 y(a) + \alpha_1 y'(a) &= \alpha \\ \beta_0 y(b) + \beta_1 y'(b) &= \beta \end{aligned}\right\} \qquad \ldots\ldots (4)$$

where $f(x)$, $g(x)$, $q(x)$ are known functions of x on the interval $[a, b]$, α_0, α_1, β_0, β_1, α, β are constants such that $|\alpha_0| + |\alpha_1| \neq 0$ and $|\beta_0| + |\beta_1| \neq 0$.

When $\alpha = \beta = 0$, the boundary conditions (4) are called uniform. Now to get the approximate solutions of the problem (3), (4), we consider the finite difference methods.

§ Finite Difference Methods :

Let $x_0 = a$, $x_n = b$ and $x_k = x_0 + kh$, $k = 1(1)n - 1$, be a system of equally spaced points with spacing $h = \dfrac{b-a}{n}$. The points, x_k, for $k = 1\ (1)\ n - 1$ are called interior

mesh or grid points and if we deal with some points outside the interval $[a, b]$, then we call such points as exterior mesh points.

Now to solve the boundary value problem (3), (4) by the method of finite differences, we replace every derivatives appearing in the equation (3) and in the boundary conditions (4), by appropriate difference approximations. We usually prefer central differences, because they lead to greater accuracy. Due to non-availability of exterior mesh points, in boundary conditions, sometimes we approximate derivatives by forward and backward differences. We shall first discuss such a method.

Consider the finite difference approxiamates of y'' and y' in (3) by central differences as follows :

$$y'(x_k) \approx \frac{y_{k+1} - y_{k-1}}{2h},\ y''(x_k) \approx \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2},$$

$$\forall k = 1(1)n - 1, \ldots\ldots (5)$$

where y_j is the approxiamte value of y at x_j, $j = 0(1)n$. Then by (3), we have the following difference equation,

$$\frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + f(x_k)\ \frac{y_{k+1} - y_{k-1}}{2h} + g(x_k)y_k = q(x_k),$$

$$\forall k = 1(1)n - 1, \ldots\ldots (6)$$

Now multiplying both side of (6) by h^2 and writting $f(x_k) = f_k$, $g(x_k) = g_k$, and $q(x_k) = q_k$, we get,

$$y_{k-1} - 2y_k + y_{k+1} + \frac{h}{2}(y_{k+1} - y_{k-1})f_k + h^2 g_k y_k = h^2 q_k,\ \forall k = 1(1)n - 1$$

$$\text{or, } \left(1 - \frac{h}{2} f_k\right) y_{k-1} + (-2 + h^2 g_k) y_k + \left(1 + \frac{h}{2} f_k\right) y_{k+1} = h^2 q_k,$$

$$\forall k = 1(1)n - 1 \ldots\ldots (7)$$

For the boundary conditions (4), we use forward difference $\frac{y_1 - y_0}{h}$ in place of $y'(x_0)$ and backward difference $\frac{y_n - y_{n-1}}{h}$ in place of $y'(x_n)$. Then the corresponding relations for (4) are as follows :

$$\alpha_0 y_0 + \alpha_1 \frac{y_1 - y_0}{h} = \alpha \text{ and } \beta_0 y_n + \beta_1 \frac{y_n - y_{n-1}}{h} = \beta,$$

or, $(\alpha_0 h - \alpha_1)y_0 + \alpha_1 y_1 = \alpha h$

$$-\beta_1 y_{n-1} + (\beta_0 h + \beta_1)y_n = \beta h. \quad \ldots\ldots (8)$$

Therefore, (7) and (8) constitutes a system of $n + 1$ linear equations with $n + 1$ unknowns $y_0, y_1, \ldots, y_n$. In matrix notation, we can write,

$$AY = B \quad \ldots\ldots (9)$$

where $Y = [y_0, y_1, \ldots, y_n]^T$, $B = [\alpha h, h^2 q_1, \ldots, h^2 q_{n-1}, \beta h]^T$,

and $A = \begin{bmatrix} \alpha_0 h - \alpha_1 & \alpha_1 & 0 & 0 & \ldots & \ldots & \ldots & 0 \\ 1 - \frac{h}{2} f_1 & -2 + h^2 g_1 & 1 + \frac{h}{2} f_1 & 0 & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & \ldots & \ldots & 1 - \frac{h}{2} f_{n-1} & -2 + h^2 g_{n-1} & 1 + \frac{h}{2} f_{n-1} \\ 0 & 0 & \ldots & \ldots & \ldots & 0 & -\beta_1 & \beta_0 h + \beta_1 \end{bmatrix}$

$\ldots\ldots (10)$

This system of linear equations can be solved by any suitable LU decomposition method. The accuracy of the solution is of the order of h. To get the accuracy of the order h^2 everywhere, we must replace $y'(x_0)$ and $y'(x_n)$ by central differences $\frac{y_1 - y_{-1}}{2h}$ and $\frac{y_{n+1} - y_{n-1}}{2h}$, where we also use the approximates at the exterior mesh points x_{-1} and x_{n+1} respectively. One possible way to eliminate y_{-1} and y_{n+1} is to extend system of equations (7) to the cases $k = 0$ and $k = n$.

Now we shall describe a slight modified method to the earlier one, known as "passage method". In this method we replace the second boundary condition of (4) by the central finite difference apporximate. Then the finite difference scheme for the problem (3), (4) by the passage method is as follows :

$$\frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + f_k \frac{y_{k+1} - y_{k-1}}{2h} + g_k y_k = q_k,\ k = 1(1)n - 1,$$

$$\alpha_0 y_0 + \alpha_1 \frac{y_1 - y_0}{h} = \alpha,$$

and $$\beta_0 y_n + \beta_1 \frac{y_{n+1} - y_{n-1}}{2h} = \beta. \quad \ldots\ldots (11)$$

The first $n - 1$ equations of (11) can be written in the form,

$$y_{k+1} + c_k y_k + d_k y_{k-1} = \psi_k \quad \ldots\ldots (12)$$

where $c_k = \dfrac{2h^2 g_k - 4}{2 + hf_k}$, $d_k = \dfrac{2 - hf_k}{2 + hf_k}$ and $\psi_k = \dfrac{2h^2 q_k}{2 + hf_k}$ $\forall k = 1(1)n - 1$

..... (13)

Now using $\alpha_0 y_0 + \alpha_1 \dfrac{y_1 - y_0}{h} = \alpha$

or, $y_0 = \dfrac{\alpha_1 y_1 - \alpha h}{\alpha_1 - \alpha_0 h}$,

we get for $k = 1$ on (12),

$$y_1 = L_1(M_1 - y_2),$$

where $L_1 = \dfrac{\alpha_1 - \alpha_0 h}{c_1(\alpha_1 - \alpha_0 h) + \alpha_1 d_1}$ and $M_1 = \psi_1 + d_1 \cdot \dfrac{\alpha h}{\alpha_1 - \alpha_0 h}$

$$= \frac{2h^2 q_1}{2 + hf_1} + d_1 \frac{\alpha h}{\alpha_1 - \alpha_0 h}$$ ..... (14)

Again, by (14) and putting $k = 2$ on (12) we get,

$$y_2 = L_2 (M_2 - y_3),$$

where, $L_2 = \dfrac{1}{c_2 - L_1 d_2}$ and $M_2 = \psi_2 - d_2 L_1 M_1$

$$= \frac{2h^2 q_2}{2 + hf_2} - d_2 L_1 M_1$$ ..... (15)

Proceeding as above we get a recursion relation,

$$y_k = L_k (M_k - y_{k+1}),\ k = 1, 2, \ldots, n - 1$$ ..... (16)

where L_1, M_1 is given by (14) and L_k, M_k for $k = 2, \ldots, n - 1$ is given by the following recursion relation,

$$L_k = \frac{1}{c_k - d_k L_{k-1}},\ M_k = \psi_k - d_k L_{k-1} M_{k-1}$$

$$= \frac{2h^2 qk}{2 + hf_k} - d_k L_{k-1} M_{k-1}$$ ..... (17)

Then we find the values of $y_0, y_1, \ldots, y_n$ by the following two stages.

Stage 1. Find c_k, d_k and ψ_k by the formula (13) for $k = 1(1)n-1$. Then compute L_1, M_1 by (14) and successively L_k, M_k, $k = 2(1)n-1$ by (17).

Stage 2. From second boundary condition, i.e., from

$$\beta_0 y_n + \beta_1 \frac{y_{n+1} - y_{n-1}}{2h} = \beta,$$

using the last equation of (16), i.e., $y_{n-1} = L_{n-1}(M_{n-1} - y_n)$ and extending the relation (16) for $k = n$, i.e., $y_n = L_n (M_n - y_{n+1})$ we get,

$$y_n = \frac{2\beta h - \beta_1 (M_n - L_{n-1} M_{n-1})}{2\beta_0 h + \beta_1 \left(L_{n-1} - \frac{1}{L_n} \right)} \qquad \ldots\ldots (18)$$

i.e., we find y_n using the numbers L_n, M_n, L_{n-1}, M_{n-1}.

Then we find y_k, for $k = n-1, n-2, \ldots, 1$ by the recurrence formula (16). Lastly we find y_0, by the relation

$y_0 = \dfrac{\alpha_1 y_1 - \alpha h}{\alpha_1 - a_0 h}$, which we have obtained from first boundary condition.

Example 1. Solve $y'' - xy' + 2y = x + 1$ with boundary conditions $y(0{\cdot}9) - 0{\cdot}5y'(0{\cdot}9) = 2$ and $y(1{\cdot}2) = 1$ by finite difference method with an accuracy 0·001 and step length $h = 0{\cdot}1$.

Answer : We divide the interval [·9, 1·2] with the step length $h = 0{\cdot}1$. The mesh points are $x_0 = 0{\cdot}9$, $x_1 = 1{\cdot}0$, $x_2 = 1{\cdot}1$, and $x_3 = 1{\cdot}2$. Now using finite difference scheme (7) we have,

$$\left(1 + \frac{h}{2} x_k\right) y_{k-1} + (-2 + 2h^2) y_k + \left(1 - \frac{h}{2} x_k\right) y_{k+1} = (x_k + 1)h^2, k = 1, 2, \text{ as here,}$$

$q_k = x_k + 1$, $g_k = 2.0$, $f_k = -x_k$, $k = 1, 2$.

Also, from boundary conditions (8) we have

$$(h + {\cdot}5) y_0 - {\cdot}5 y_1 = 2h,$$

and $hy_3 = h$, as $\alpha_0 = 1, \alpha_1 = -{\cdot}5, \alpha = 2, \beta_0 = 1, \beta_1 = 0, \beta = 1$.

Therefore, we have the following system of equations,

$$\begin{aligned} 1{\cdot}2y_0 - y_1 &= {\cdot}4 \\ 2{\cdot}1y_0 - 3{\cdot}96y_1 + 1{\cdot}9y_2 &= {\cdot}04 \\ 1{\cdot}11y_1 - 3{\cdot}96y_2 + 1{\cdot}89y_3 &= {\cdot}42 \\ y_3 &= 1{\cdot}000 \end{aligned}$$

Solving we have, $y_0 = 1{\cdot}406$, $y_1 = 1{\cdot}287$, $y_2 = 1{\cdot}149$, $y_3 = 1{\cdot}000$.

Find desired accuracy check the results with step length $h = 0{\cdot}05$ and compare.

Example 2. Using passage method, find the solution of the equation $y'' + 2xy' + 2y = 4x$ correct to two decimal point, with boundary conditions $y(0) = 1$ and $y(\cdot5) = 1{\cdot}279$ and step length $h = \cdot1$.

Answer : Here $f(x) = 2x$, $g(x) = 2$, $q(x) = 4x$, $\alpha_0 = 1$, $\alpha_1 = 0$, $\alpha = 1$, $\beta_0 = 1$, $\beta_1 = 0$, $\beta = 1{\cdot}279$.

By passage method we have the following table,

k	x_k	c_k	d_k	Ψ_k	L_k	M_k	y_k
1	0·1	–1·960	0·980	0·004	– 0·510	– 0·976	1·089
2	0·2	–1·941	0·961	0·008	– 0·689	– 0·470	1·160
3	0·3	–1·992	0·942	0·012	– 0·786	– 0·293	1·214
4	0·4	–1·904	0·923	0·015	– 0·848	– 0·197	1·252
5	0·5						1·279

Hence, the values of y_1, y_2, y_3, y_4 correct to two decimal point are $y_1 = 1{\cdot}09$, $y_2 = 1{\cdot}16$, $y_3 = 1{\cdot}21$ and $y_4 = 1{\cdot}25$.

§ Summary :

In this unit, a finite difference scheme is device for second order ordinary derivatives and this scheme is used to solve second order boundary value problem.

EXERCISES

1. Solve the following boundary value problem correct to three decimal places by a finite difference scheme.

(i) $y'' + y = 0$, $y(0) = 0$, $y(1) = 1$, for $x = 0{\cdot}0(0{\cdot}1)\ 1{\cdot}0$.

(ii) $xy'' + 3y' = 4$, $y(1) = 3 = y(2)$, for $h = 0{\cdot}1$ and $h = 0{\cdot}2$.

(iii) $y'' = xy' + 1$, $y(0) = 1{\cdot}5$, $y(1) = {\cdot}5$, for $h = 0{\cdot}1$.

(iv) $y'' - xy' + x^2y = 1$, $y(-{\cdot}5) = 1$, $y(2) = 2$, for $h = 0{\cdot}1$.

(v) $y'' + 2y' + 2y = e^{-x}$, $y(0) = 0$, $y(1) = 1$, for $h = 0{\cdot}05$ and $h = {\cdot}1$.

2. Solve the following boundary value problems by passage method with step lengths $h = {\cdot}1$ and $h = {\cdot}05$.

(i) $y'' + x\sin x.y' + y = \dfrac{1}{1+\sin^2 x}$, $y(0) = 0$, $y(1) = 1$.

(ii) $y'' + 2x^2y' + (1-x)y = \dfrac{x}{2+x^2}$, $y(0) + y'(0) = 1$, $y(1) = 0$.

(iii) $y'' + xy' + \cos ax\, y = x^2 + 4x$, $y(0) = 0{\cdot}1$,

$y(1) = 1{\cdot}0$, for $a = 1{\cdot}1$, $1{\cdot}2$, $1{\cdot}3$, $1{\cdot}4$ and $1{\cdot}5$.

(iv) $y'' + e^{-x}y = x^2 + 1$, $y(0) = 0$, $y(1) = 1{\cdot}0$.

Unit 10 ❐ Elements of Finite Difference Method of Numerical Solution of Partial Differential Equations

§ Objectives

After going through this unit you will be able to learn about—

- Finite difference scheme for partial derivatives
- Solution of Poisson equation by finite difference method
- Solution of parabolic equation by finite difference method
- Solution of parabolic equation by Crank-Nicolson method
- Solution of hyperbolic equation

The subject matter of this unit is to find a function of two variables $u(x, y)$ in a given region $G \subset R^2$ that satisfies a second-order linear partial differential equation,

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu = H. \quad \ldots\ldots (1)$$

The given coefficients A, B, C, D, E, F, H, can be piecewise continuous functions of x and y. The classification of this equation in the region G is made in the following manner :

Let us assume $A^2 + B^2 + C^2 \neq 0$, then, (1) is called,

(a) elliptic, if $AC - B^2 > 0$ for all $(x, y) \in G$,

(b) hyperbolic, if $AC - B^2 < 0$ for all $(x, y) \in G$,

and (c) parabolic, if $AC - B^2 = 0$ for all $(x, y) \in G$.

Now, we further assume that the region G has a boundary Γ consists of three disjoint parts $\Gamma_1, \Gamma_2, \Gamma_3$ such that $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$. Γ may admit all the three parts or it may have lesser parts. Therefore, we may classify the boundary conditions on (1) as follows :

(i) $u = \phi$ on Γ_1 (Dirichlet condition),

(ii) $\dfrac{\partial u}{\partial n} = \lambda$ on Γ_2 (Neumann condition),

(iii) $\frac{\partial u}{\partial n} + \alpha u = \beta$ on Γ_3 (Cauchy condition),

where $\phi, \lambda, \alpha, \beta$ are given functions on the corresponding boundary parts and $\frac{\partial}{\partial n}$ denotes the partial derivative along the normal.

§ Poisson Equation on a Rectangular Region :

In this section we shall discuss numerical solution of an elliptic type equation, viz., Poisson equation, by finite difference scheme. The canonical form of a Poisson equation in two dimension is,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{dy^2} = f(x, y) \qquad \ldots\ldots (2)$$

where $(x, y) \in G$, a given region in R^2, enclosed by a boundary Γ. We seek the solution of (2) inside G. Let us restrict ourselves to the case of rectangular region.

Let $G \equiv \{(x, y) : a < x < b, c < y < d\}$ and

$$\Gamma \equiv \{(x, y) : x = a, c \le y \le d; x = b, c \le y \le d; a \le x \le b, y = c; a \le x \le b, y = d\}$$

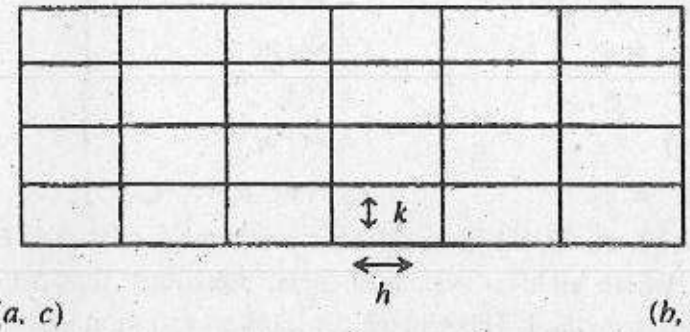
Let $u(x, y)$ satisfies the following boundary conditions,

$$\begin{aligned} u(x, y) &= \phi(y), x = a, c \le y \le d, \\ &= \psi(y), x = b, c \le y \le d, \\ &= g(x), a \le x \le b, y = c, \\ &= h(x), a \le x \le b, y = d, \end{aligned} \qquad \ldots\ldots (3)$$

with $\phi(c) = g(a)$, $\phi(d) = h(a)$, $\psi(c) = g(b)$, $\psi(d) = h(b)$

i.e., satisfies Dirichlet type boundary conditions.

Now, for the finite difference scheme, we first divide the whole region G with boundary Γ into rectangular nets of size hk square unit, where h is the step-length in x-direction and k is the step-length in y-direction.



We want to find the approximate values of u at grid points. Let us first use some notational convention. Let $x_i = a + ih$, $h = \frac{b-a}{n}$, $i = 0, 1, ..., n$ and $y_j = c + jk$, $k = \frac{d-c}{n}$, $j = 0, 1, ..., n$. We may consider different number of subintervals in x-direction and y-direction. Let us denote the approximate value of u at (x_i, y_j) by $u_{i,j}$. For the finite difference scheme, we then approximate $u_{xx} = \frac{\partial^2 u}{\partial x^2}$ and $u_{yy} = \frac{\partial^2 u}{\partial y^2}$ by the following discretization procedure,

$$\left.\begin{aligned} u_{xx}(x_i, y_j) &\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \\ u_{yy}(x_i, y_j) &\approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}, \ \forall\ i, j = 1(1)n-1 \end{aligned}\right\} \quad \ldots. (4)$$

Therefore, inserting the above approximates on equation (2), we have the following difference equation for u,

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = f(x_i, y_j)$$

$$= f_{ij} \text{ (say)}, \ \forall i, j = 1(1)n-1 \ldots.. (5)$$

or, $$u_{i+1,j} + u_{i-1,j} + \frac{h^2}{k^2} u_{i,j+1} + \frac{h^2}{k^2} u_{i,j-1} - 2\left(1 + \frac{h^2}{k^2}\right) u_{i,j} = h^2 f_{ij},$$

or, $$u_{i+1,j} + u_{i-1,j} + r^2 u_{i,j+1} + r^2 u_{i,j-1} - 2(1 + r^2) u_{i,j} = h^2 f_{ij},$$

$$\forall i, j = 1(1)n-1 \text{ where } r = \frac{h}{k} \ldots.. (6)$$

And from boundary conditions we have,

$$\left.\begin{aligned} u_{0,j} &= \phi(y_j) = \phi_j \text{ (say)}, & j &= 0(1)n, \\ u_{n,j} &= \psi(y_j) = \psi_j, & j &= 0(1)n, \\ u_{i,0} &= g(x_i) = g_i, & i &= 0(1)n, \\ u_{i,n} &= h(x_i) = h_i, & i &= 0(1)n, \\ \text{with} \quad \phi_0 &= g_0,\ \phi_n = h_0, & \psi_0 &= g_n,\ \psi_n = h_n. \end{aligned}\right\} \quad \ldots\ldots (7)$$

Equations (6) and (7) are called finite difference scheme for Poisson equation in a rectangular region, which yields a system of linear equations, that can be solved using any suitable matrix method. The scheme for Laplace equation is the same, just replace $f(x, y) = 0$ on (2). Therefore, the finite difference equation for Laplace equation in the rectangular region is,

$$u_{i+1,j} + u_{i-1,j} + r^2 u_{i,j+1} + r^2 u_{i,j-1} = 2(1 + r^2) u_{i,j}$$

$$\forall i, j = 1(1)n - 1 \ldots\ldots (8)$$

The local discretization error of the above five-point difference scheme is $0(h^2)$ for $r = 1$, i.e., $h = k$. We should take care of the fact that the error of the approximate solution obtained by the difference scheme occur due to the following three facts,

(i) due to the replacement of partial derivatives by finite differences,

(ii) approximating the boundary conditions,

and (iii) solving the system of equations by any approximate method.

Therefore, error in the above scheme changes if the boundary conditions varies.

Example 1. Solve the Laplace's equation,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0,\ 0 \le x \le 1,\ 0 \le y \le 1,$$

with initial and boundary conditions,

$$u(x, 0) = 0 = u(1, y),$$

$$u(x, 0) = 30y,\ u(x, 1) = 30(1 - x),\ 0 \le x \le 1,\ 0 \le y \le 1,$$

by finite difference scheme with spacing $h = k = \frac{1}{3}$.

Solution : Here $r = \frac{h}{k} = 1$, $h = \frac{1}{3}$. Therefore $n = 3$.

The finite difference equation inside the domain is,

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} = 4u_{i,j} \quad \forall i, j = 1(1)2.$$

Explicitly,

$$u_{2,1} + u_{0,1} + u_{1,2} + u_{1,0} = 4u_{1,1}$$

$$u_{3,1} + u_{1,1} + u_{2,2} + u_{2,0} = 4u_{2,1}$$

$$u_{2,2} + u_{0,2} + u_{1,3} + u_{1,1} = 4u_{1,2}$$

$$u_{3,2} + u_{1,2} + u_{2,3} + u_{2,1} = 4u_{2,2}$$

From initial and boundary conditions, we have,

$$u_{0,0} = u_{1,0} = u_{2,0} = u_{3,0} = 0 = u_{3,1} = u_{3,2} = u_{3,3}$$

$$u_{0,1} = 10,\ u_{0,2} = 20,\ u_{0,3} = 30,$$

$$u_{1,3} = 20,\ u_{2,3} = 10,\ u_{3,3} = 0.$$

Therefore, we have to solve a system of linear equations,

$$\begin{pmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{pmatrix} \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{2,1} \\ u_{2,2} \end{pmatrix} = \begin{pmatrix} 10 \\ 40 \\ 0 \\ 10 \end{pmatrix} =$$

By Gaussian elimination scheme we obtain the solutions,

$$u_{1,1} = \frac{20}{3} \qquad u_{1,2} = \frac{40}{3}$$

$$u_{2,1} = \frac{10}{3} \qquad u_{2,2} = \frac{20}{3}.$$

§ Parabolic Equation in One-Space Dimension :

Consider the parabolic partial differential equation,

$$\frac{\partial u}{\partial t} = c^2 \frac{\partial^2 u}{\partial x^2} + g(x, t), \qquad \ldots\ldots (9)$$

in a domain $D = \{(x, t) : a < x < b,\ 0 < t < T\}$ with initial condition.

$$u(x, 0) = f(x),\ a \le x \le b, \qquad \ldots\ldots (10)$$

and the boundary conditions,

$$u(a, t) = \phi(t),\ u(b, t) = \psi(t),\ 0 \le t \le T \qquad \ldots\ldots (11)$$

Sometimes it is called as the heat conduction equation in a rectangle $\overline{D} = \{(x, t) : a \le x \le b,\ 0 \le t \le T\}$ .It is a mixed, i.e. Cauchy type boundary value problem where first order partial derivatives with respect to x may also be present at the boundary conditions (11).

For the finite difference scheme the grid points on $\overline{D}$ are given by :

$$\left.\begin{aligned} & x_i = a + ih,\ \ h = \frac{b-a}{m},\ i = 0, 1, \ldots, m \\ \text{and} \quad & t_j = jk, \qquad k = \frac{T}{n},\ j = 0, 1, \ldots, n \end{aligned}\right\} \qquad \ldots\ldots (12)$$

So that $x_0 = a,\ x_m = b,\ T = nk$.

Again we denote the approximate of $u(x_i, t_j)$ by $u_{i,j}$. Then the problem is to find $u_{i,j}$ in the grid points. In the domain D, we replace the partial derivative with respect to t by the forward difference,

$$u_t(x_i, t_j) \approx \frac{u_{i,j+1} - u_{i,j}}{k}, \qquad \ldots\ldots (13)$$

and the second derivative, with respect to x by the second difference,

$$u_{xx}(x_i, t_j) \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \qquad \ldots\ldots (14)$$

(13) and (14) yields the following difference equation for (9),

$$\frac{u_{i,j+1} - u_{i,j}}{k} = c^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + g_{ij}, \qquad \ldots\ldots (15)$$

where $g_{ij} = g(x_i, t_j),\ i = 1, 2, \ldots, m-1,\ j = 1, 2, n-1$.

Rewriting the above expression we have,

$$u_{i,j+1} = ru_{i-1,j} + (1 - 2r)u_{i,j} + ru_{i+1,j} + kg_{ij} \quad \text{where } r^2 = \frac{k}{h^2}\ c^2,$$

$$\forall i = 1(1)m - 1,\ j = 1(1)n - 1 \qquad \ldots\ldots (16)$$

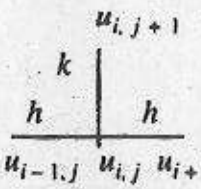
The initial condition (10) yields,

$$u_{i,0} = f_i = f(x_i),\ i = 0, 1, \ldots, m \qquad \ldots\ldots (17)$$

and the boundary conditions (11) yields,

$$\left.\begin{aligned} u_{0,j} &= \phi(t_j) \equiv \phi_j, \\ u_{m,j} &= \psi(t_j) \equiv \psi_j, \quad \forall j = 0, 1, \ldots, n \end{aligned}\right\} \quad \ldots\ldots (18)$$

Since, the initial values of $u_{i,0}$ are known for $i = 0, 1, \ldots, m$, therefore, the formulae (16) and (18) allow us to compute the approximations, $u_{i,j+1}$, $i = 1, 2, \ldots, m$, for fixed index j from the values $u_{i,j}$ in an explicit manner. Therefore, it is possible to find successively the approximate solutions with increasing j. This finite difference scheme applied to the parabolic differential equation is known as Richardson's explicit method. Pictorially we have the following explicit diagram.



For $c = 1$, expanding with respect to the solution we obtain the local discretization error from (16) as follows,

$$d_{i,j+1} = \frac{1}{2} k^2 u_{tt}(x_i, t_j) - \frac{1}{12} kh^2 u_{xxxx}(x_i, t_j) + \ldots\ldots$$

$$= 0(k^2) + 0(kh^2) \quad \ldots\ldots (19)$$

The global discretization error of this method is of the order, $0(k) + 0(h^2)$ i.e., loses a factor k in comparison to the local error $d_{i,j+1}$. To study the absolute stability, we rewrite the system (16) in matrix notation,

$$\vec{u}_{j+1} = A\vec{u}_j + \vec{b}_j, \; j = 0, 1, 2 \quad \ldots\ldots (20)$$

where, $\vec{u}_j = [u_{1,j}, u_{2,j}, \ldots\ldots\ldots, u_{m-1,j}]^T$ (T stands for matrix transpose)

$\vec{b}_j$ = a column matrix formed by the elements g_{ij} and the constants appearing from (17) and (18), i.e., known constants = $[b_1, b_2, \ldots\ldots\ldots, b_{m-1}]^T$ (say),

and $A = \begin{pmatrix} 1-2r & r & 0 & \ldots & \ldots & \ldots & 0 \\ r & 1-2r & r & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & r & 1-2r & r \\ 0 & \ldots & \ldots & \ldots & 0 & r & 1-2r \end{pmatrix}$ ..... (21)

The matrix A is tridiagonal and depends only on r. The absolute stability holds if and only if the eigen values of the matrix A have moduli smaller than one. For $c = 1$, we find the condition of the absolute stability as,

$$r \leq \frac{1}{2} \quad \text{or} \quad k \leq \frac{1}{2}h^2.$$

Example 2. Using the above method to the problem,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \; 0 < x < 1, \; t > 0,$$

$$u(x, 0) = x^2 e^{1-x}, \; 0 \leq x \leq 1,$$

$$u(0, t) = 0, \; u(1, t) = 1, \; t \geq 0.$$

We find the following table for $u_{i,j}$ with $h = 0{\cdot}1$ and $k = 0{\cdot}002$. Note that here $g(x, t) = 0$, $c = 1$, $f(x) = x^2 e^{1-x}$, $\phi(t) = 0$, $\psi(t) = 1$.

Table for $u_{i,j}$

x_i / t_j	0·000	0·002	0·004	0·006	0·008	0·010
0·0	0·0000	0·0000	0·0000	0·0000	0·0000	0·0000
0·1	0·0246	0·0326	0·0385	0·0431	0·0470	0·0503
0·2	0·0890	0·0946	0·1002	0·1055	0·1104	0·1149
0·3	0·1812	0·1849	0·1886	0·1923	0·1961	0·1998
0·4	0·2915	0·2936	0·2957	0·2979	0·3002	0·3025
0·5	0·4122	0·4130	0·4139	0·4149	0·4159	0·4170
0·6	0·5371	0·5370	0·5369	0·5369	0·5369	0·5370
0·7	0·6614	0·6606	0·6598	0·6591	0·6584	0·6578
0·8	0·7817	0·7803	0·7790	0·7778	0·7767	0·7757
0·9	0·8952	0·8935	0·8921	0·8911	0·8902	0·8895
1·0	1·0000	1·0000	1·0000	1·0000	1·0000	1·0000

§ Parabolic Equations : Implicit Method.

Crank-Nicolson Method.

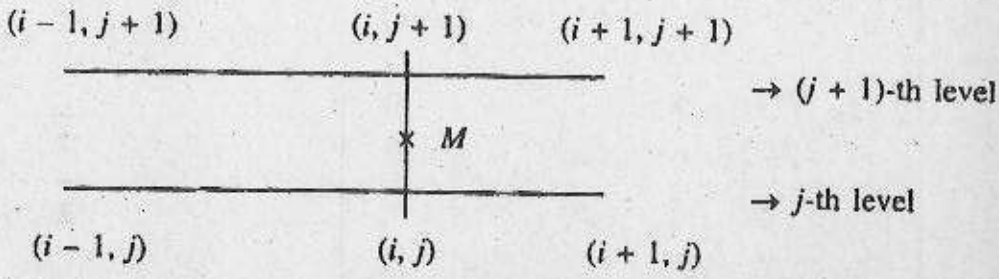
In this finite difference scheme we again replace the partial derivative u_t of (9) by the forward difference,

$$u_t(x_i, t_j) \approx \frac{u_{i,j+1} - u_{i,j}}{k}, \qquad \ldots\ldots (22)$$

but the partial derivative u_{xx} replaced by a six-point difference formula,

$$u_{xx}(x_i, t_j) \approx \frac{1}{2h^2}[u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1} + u_{i+1,j} - 2u_{i,j} + u_{i-1,j}] \ldots\ldots (23)$$

i.e., u_{xx} is approximated at the mid-point M of the pictorial representation,



The relations (22) and (23) after substituting on (9) yields the following difference equation,

$$\frac{u_{i,j+1} - u_{i,j}}{k} = c^2 \cdot \frac{1}{2h^2}[u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1} + u_{i+1,j} - 2u_{ij} + u_{i-1,j}] + g_{ij}$$

$$\forall i = 1, 2, \ldots, m-1, j = 1, 2, \ldots, n-1 \ldots\ldots (24)$$

Rewritting (24) we have,

$$-ru_{i-1,j+1} + (2 + 2r)u_{i,j+1} - ru_{i+1,j+1} = ru_{i-1,j} + (2 - 2r)u_{i,j} + ru_{i+1,j} + 2kg_{ij},$$

$$i = 1, 2, \ldots, m-1, j = 1, 2, \ldots, n-1 \ldots\ldots (25)$$

In this equation, the L.H.S. contains three unknown pivotal values of u at $(j+1)$-th level, whereas R.H.S. contains values of u at j-th level which are supposed to be known. Now at each j-th level, $j = 0, 1, \ldots, n$, we have a total of $(m+1)$ grid

points of which $m - 1$ are internal grid points. Now if we start with $j = 0$, all the elements of R.H.S. are known by initial and boundary conditions (17) and (18). And then we have a system of $(m - 1)$ linear equations with $(m - 1)$ unknowns,

$$u_{1,1},\ u_{2,1},\ \ldots,\ u_{m-1,1}.$$

In matrix notation, we have,

$$A_1 \vec{u}_{j+1} = A_2 \vec{u}_j + B' \text{ where, } \vec{u}_j = [u_{1,j}, \ldots u_{m-1,j}]^T$$

$$A_1 = \begin{pmatrix} 2+2r & -r & 0 & \ldots & \ldots & \ldots & 0 \\ -r & 2+2r & -r & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & -r & 2+2r & -r \\ 0 & \ldots & \ldots & \ldots & 0 & -r & 2+2r \end{pmatrix},$$

$$A_2 = \begin{pmatrix} 2-2r & r & 0 & \ldots & \ldots & \ldots & 0 \\ r & 2-2r & r & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & r & 2-2r & r \\ 0 & \ldots & \ldots & \ldots & 0 & r & 2-2r \end{pmatrix} \quad \ldots (26)$$

and B' is a column matrix formed by the elements g_{ij} and the constants appearing due to initial and boundary conditions (17) and (18), i.e., all are known constants, say, $[b'_1, b'_2, \ldots\ldots\ldots, b'_{m-1}]^T$.

Since at each step we have to solve this system of equations, therefore the Crank-Nicolson method is implicit. The coefficient matrix found is tridiagonal. The matrix A_1 is diagonally dominant since $r > 0$ and hence nonsingular. It is then easy to calculate that all eigen values have moduli smaller than one. Hence the Crank-Nicolson scheme is absolutely stable as r is not subject to any restrictions concering stability. The local discretization error $d_{i,j+1}$ of this method is of the order $0(k^3) + 0(kh^2)$ and the global discretization error is of the order $0(k^2) + 0(h^2)$.

Example 3. Consider the initial boundary value problem,

$u_t = u_{xx}$, $\quad 0 < x < 1,\ t > 0,$

$u(x, 0) = 0,$ $\quad 0 \le x \le 1,$

$u(0, t) = \sin(\pi t),$ $\quad u_x(1, t) = 0,\ t \ge 0.$

By using implicit Crank-Nicolson method with $h = 0{\cdot}1$, $k = 0{\cdot}1$, $r = 10$. we have the following table for $u_{i,j}$.

t	j	$u_{0,j}$	$u_{1,j}$	$u_{2,j}$	$u_{3,j}$	$u_{4,j}$	.....	$u_{10,j}$
0	0	0	0	0	0	0	.....	0
0·1	1	0·3090	0·1983	0·1274	0·0818	0·0527	.....	0·0073
0·2	2	0·5878	0·4641	0·3540	0·2637	0·1934	.....	0·0459
0·3	3	0·8090	0·6632	0·5436	0·4422	0·3565	.....	0·1344
0·4	4	0·9511	0·8246	0·7040	0·5975	0·5064	.....	0·2594
0·5	5	1·0000	0·8969	0·8022	0·7132	0·6320	.....	0·3946
:	:	...	...	...	...	...	...	...
:	:	...	...	...	...	...	...	...
:	:	...	...	...	...	...	...	...
:	:	...	...	...	...	...	...	...
1·0	10	0	0·1498	0·2701	0·3672	0·4440	.....	0·6179

§ Hyperbolic Equation in One-Space Dimension :

Consider the hyperbolic partial differential equaion,

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \qquad \text{..... (27)}$$

in a domain $D = \{(x, t) : 0 < x < a,\ 0 < t < T\}$ with initial conditions,

$$u(x, 0) = f(x),\ u_t(x, 0) = g(x),\ 0 \le x \le a, \qquad \text{..... (28)}$$

and the boundary conditions,

$$u(0, t) = \phi(t),\ u(a, t) = \psi(t),\ 0 \le t \le T \qquad \ldots\ldots (29)$$

This is actually known as wave equation in a region $\overline{D} = \{(x, t) : 0 \le x \le a,\ 0 \le t \le T\}$. It is also a mixed boundary value problem, i.e., Cauchy type.

For the finite difference scheme, the grid points on $\overline{D}$ are given by :

$$\left.\begin{aligned} & x_i = ih,\ h = \frac{a}{m},\ i = 0, 1, \ldots, m \\ & t_j = jk, k = \frac{T}{n},\ j = 0, 1, \ldots, n \\ & \text{so that } x_0 = 0,\ x_m = a,\ T = nk. \end{aligned}\right\} \qquad \ldots\ldots (30)$$

Now to find the approximates $u_{i,j}$ in the grid points, we replace both the partial derivatives u_{tt}, u_{xx} by the central difference formulae,

$$\left.\begin{aligned} & u_{tt}(x_i, t_j) \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}, \\ & u_{xx}(x_i, t_j) \approx \frac{u_{i+1,j} - 2u_{j,j} + u_{i-1,j}}{k^2}. \end{aligned}\right\} \qquad \ldots\ldots (31)$$

Substituing (31) on (27) we have the following difference equation,

$$\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = c^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2},$$

or, $$u_{i,j+1} = (2 - 2\alpha^2)\, u_{i,j} + \alpha^2 (u_{i+1,j} + u_{i-1,j}) - u_{i,j-1} \qquad \ldots\ldots (32)$$

where $\alpha^2 = \dfrac{c^2 k^2}{h^2}$, for $i = 1, 2, \ldots, m - 1,\ j = 1, 2, \ldots, n - 1$

The system (32) is known stable for $\alpha \le 1$ with $c = 1$. To find an approximate at the $(j + 1)$-th level, we require values at earlier two levels. Pictorially, we may represent it,

	$(i, j + 1)$		$\to (j + 1)$-th level
$(i - 1, j)$	(i, j)	$(i + 1, j)$	$\to j$-th level
	$(i, j - 1)$		$\to (j - 1)$-th level.

The local discretization error $d_{i,j+1}$ is of the order $0(k^2) + 0(h^2)$.

The initial and boundary conditions (28), (29), except the second initial conditiion yields,

$$\left.\begin{aligned} u_{i,0} &= f(x_i) \equiv f_i, i = 0, 1, ..., m \\ u_{0,j} &= \phi_j, \ j = 0, 1, ..., n \\ u_{m,j} &= \psi_j, \ j = 0, 1, ..., n. \end{aligned}\right\} \quad \text{..... (33)}$$

Now, depending upon the finite difference approximate to the second initial condition, we have three different methods to solve the system (32).

First Method. The second initial condition is replaced by the following difference scheme,

$$u_{i,1} - u_{i,0} = k.g\ (x_i) = kg_i.$$

Then by (33), $u_{i,1} = f_i + kg_i$, $i = 1, ..., m - 1$. ..... (34)

The error estimate for the values of $u_{i,1}$ is of the order 0(h).

Second Method. We replace the second initial condition with the central difference formula, i.e., by,

$$\frac{u_{i,1} - u_{i,-1}}{2k} = g_i, i = 0, 1, ..., m, \quad \text{..... (35)}$$

where we require the values of u at $j = -1$ level. Now, considering the difference equation is also valid at the level $j = 0$, we have,

$$u_{i,1} = (2 - 2\alpha^2)u_{i,0} + \alpha^2(u_{i+1,0} + u_{i-1,0}) - u_{i,-1}.$$

Rewritting the above relation, using the first equation of (33), we have,

$$u_{i,-1} = (2 - 2\alpha^2)f_i + \alpha^2(f_{i+1} + f_{i-1}) - u_{i,1}. \quad \text{..... (36)}$$

Replacing it on (35) we have,

$$u_{i,1} = \frac{1}{2}[2kg_i + (2 - 2\alpha^2)f_i + \alpha^2(f_{i+1} + f_{i-1})], \quad \text{for i = 1, ..., } m - 1.$$

..... (37)

The error estimate for the values $u_{i,1}$ is of the order $0(h^2)$.

Third Method. Here we assume that the functiion $f(x)$ has finite second order derivative, then we find the values of $u_{i,1}$ by the Taylor's formula,

$$u_{i,1} \approx u_{i,0} + ku_t(x_i, 0) + \frac{k^2}{2} u_{tt}(x_i, 0) \quad \ldots\ldots (38)$$

Using equation (27) and the initial conditions (28), we have,

$$u_{i,0} = f_i, u_t(x_i, 0) = g_i, u_{tt}(x_i, 0) = c^2 u_{xx}(x_i, 0) = c^2 f_i''. \quad \ldots\ldots (39)$$

Substituting (39) on (38) we have,

$$u_{i,1} \approx f_i + kg_i + \frac{k^2}{2} \cdot c^2 f_i''$$

Now replacing f_i'' by finite difference approximate,

$\frac{f_{i+1} - 2f_i + f_{i-1}}{h^2}$, we have,

$$u_{i,1} \approx f_i + kg_i + \frac{\alpha^2}{2}(f_{i+1} - 2f_i + f_{i-1}), \text{ for } i = 1, \ldots\ldots, m-1. \quad \ldots (40)$$

The error estimate for the values $u_{i,1}$ is of the order $0(k^3)$.

Example 4. Consider the following problem,

$$\frac{\partial^2 u}{\partial t^2} = 4\frac{\partial^2 u}{\partial x^2}, \ 0 < x < 1, \ t > 0,$$

$$u(x,0) = (x^3 + 1)\sin \pi x, \ 0 \le x \le 1,$$

$$u_t(x, 0) = 0$$

$$u(0, t) = 0, u(1, t) = 0, t \ge 0$$

We obtain the finite difference approximates $u_{i,j}$ with $h = 0.1$ and $t = 05$ as follows :

x_i / t_j	0·00	0·05	0·10	0·15	0·20	0·25
0·0	0	0	0	0	0	0
0·1	0·1801	0·1752	0·1619	0·1434	0·1232	0·1039
0·2	0·3213	0·3167	0·3029	0·2802	0·2498	0·2143
0·3	0·4257	0·4213	0·4083	0·3864	0·3556	0·3160
0·4	0·4954	0·4911	0·4783	0·4569	0·4263	0·3860

0·5	0·5312	0·5268	0·5135	0·4909	0·4584	0·4153
0·6	.....	.....	.....	.....	.....	.....
0·7	.....	.....	.....	.....	.....	.....
0·8	.....	.....	.....	.....	.....	.....
0·9	.....	.....	.....	.....	.....	.....
1·0	0	0	0	0	0	0

§ Summary :

In this unit, the second order partial derivatives are discretized by finite difference method. Three standard partial differential equations—parabolic, hyperbolic and elliptic are solved by finite difference method.

EXERCISES

1. Solve the following Laplace's equations by finite difference method.

(i) $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 0, 0 \le x, y \le 1,$

$u(0, y) = \cos(\pi y), u(1, y) = e^{\pi} \cos(\pi y),$

$u(x, 0) = e^{\pi x}, u(x, 1) = e^{-\pi x},$ with step size $h = k = \dfrac{1}{4}.$

(ii) $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 0, 0 \le x, y \le 1$

$u(0, y) = y,\ u(1, y) = \cos \dfrac{\pi y}{2},$

$u(x, 0) = 0,\ u(x, 1) = \cos \dfrac{\pi x}{2},$ with step size $h = k = \dfrac{1}{4}.$

2. Solve the following Poisson's equation by finite difference method.

$u_{xx} + u_{yy} = -1,\ 0 \le x, y \le 1,$

$u(x, 0) = 1,\ \left.\dfrac{\partial u}{\partial y}\right|_{(1,y)} = 0$

$$u(x, 1) = 0, \frac{\partial u}{\partial y}(0, y) + 2u(0, y) = 1,$$

with step size h = k = 1.

3. Solve the following parabolic equations by explicit finite difference scheme.

(i) $u_t = u_{xx} + 1, 0 < x < 1, t > 0$

$u(x, 0) = 0, \quad 0 \le x \le 1,$

$$u_x(0, t) - \frac{1}{2}u(0, t) = 0, u_x(1, t) + 0.2u(1, t) = 0, \text{ for } t \ge 0,$$

with step size h = 0·1 and h = 0·05

(ii) $u_t = u_{xx}, 0 < x < 1, 0 < t < 0{\cdot}02,$

$u(x, 0) = (x^2 + 1) \sin \pi x, 0 \le x \le 1,$

$$u(0, t) = 0, u(1, t) = e^{-1} \sin \frac{\pi t}{3}, 0 \le t \le 0{\cdot}02,$$

with step size $h = 0{\cdot}1$.

4. Solve along with the exercises of 3, the following parabolic equation by implicit Crank-Nicolson scheme.

$u_t = u_{xx} + 2x + t, 0 < x < 1, t > 0$

$u(x,0) = 2x^2 \sin(\pi x), 0 \le x \le 1.$

$u(0,t) = 0 = u(1,t), t \ge 0$, with step size $k = 0{\cdot}002$ and $h = 0{\cdot}1$.

5. Solve the following hyperbolic equations by finite difference method.

(i) $$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{dx^2}, 0 < x < 1, t > 0,$$

$$u(x,0) = 2x(1-x)\sin \pi x, u_t(x,0) = 0, 0 \le x \le 1,$$

$u(0,t) = u(1,t) = 0, t \ge 0$, with step size $h = k = 0{\cdot}1$.

(ii) $$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + x + t, 0 < x < 1, t > 0,$$

with spacing $h = k = 0{\cdot}1$ and initial boundary conditons,

$u(x,0) = (3x^2 + 1) \sin \pi x, u_t(x,0) = 0{\cdot}5x, 0 \le x \le 1, u(0, t) = u(1, t) = 0.$

REFERENCES

1. F. B. Hildebrand, Introduction to Numerical Analysis, Tata Mc-Graw Hill Publ., Co.. 1982, NY.
2. D. M. Young and R. T. Gregory, A survey of Numerical Mathematics, Dover Publ., Inc., New York, 1973.
3. S. D. Conte and C. de Boor, Elementary Numerical Analysis, Mc Graw-Hill Inc., 1981.
4. A. Gupta and S. C. Bose, Introduction to Numerical Analysis, Academic Publ., 1989.
5. C. E. Froberg, Introduction to Numerical Analysis, Addison-Wesley Publ. Co., Reading, 1979.
6. K. E. Atkinson, An introduction to Numerical Analysis, John Wiley and Sons, New York, 1989.
7. A Ralston and P. Rabinowitz, A First Course in Numerical Analysis, Mc. Graw-Hill, New York, 1978.
8. J. H. Mathews, Numerical Methods for Mathematics, Science and Engineering, Prentice-Hall India, 2001.
9. M. J. Maron, Numerical Analysis, Macmillan Publ. Co., Inc., New York, 1982.
10. H. R. Schwarz, Numerical Analysis, John Wiley and Sons, 1989.
11. J. Stoer and R. Bulirsch, Introduction to Numerical Analysis, Springer-Verlag, New York, 1993.

UNIT 1 ❐ ALGORITHMS AND FLOWCHARTS

1.1 Algorithms

The central concept in Computer Science is algorithm. The word 'algorithm' originates from the Persian word 'algorism'. The meaning of algorithm is 'any special method of solving a certain kind of problem'.

An **algorithm** is a finite sequence of steps/instructions to solve a problem. The possible steps are :

(a) Input,

(b) Computation and decision

(i) assignment, (ii) decision, and (iii) iteration or repetition

(c) Output

1.2 Objectives

After going through this unit will be able to learn about:

(i) What is algorithm?

(ii) How one can design an algorithm?

(iii) Analysis of algorithm

(iv) What is flowchart?

(v) Symbols used to draw a flowchart

1.3 Definition and Examples of Algorithms

An algorithm should follow the following criteria:

(i) *Input* : An algorithm has precise inputs which are externally supplied.

(ii) *Definiteness* : Each step must be clear, complete and unambiguous.

(iii) *Finiteness* : An algorithm should terminate after a finite number of steps.

(iv) *Effectiveness* : All the steps specified in an algorithm should be executed in a finite amount of time.

(v) *Generality* : An algorithm should be designed in such a way that it can solve any problem of a particular type for which it is designed.

(vi) *Output* : It has at least one output.

An algorithm can be described in many ways. A natural language such as English may be used to design an algorithm, but we must be careful that it is concise and satisfies all the conditions of the properties of algorithm and this code is known as **pseudo code**. There are no definite rules for describing algorithm but the main objective is that it should be easily understandable and the logic put in it should be able to solve the problem. The algorithm is independent of any programming language.

In the following we have considered two simple problems and designed their algorithms.

Example 1.3.1 Write an algorithm to find the area of a triangle whose three sides a, b, c are given.

Step 1: Read the sides as a, b, c.

Step 2: Compute $s = (a + b + c)/2$.

Step 3: Compute $area = \sqrt{s(s-a)(s-b)(s-c)}$

Step 4: Print $area$.

Step 5: Stop.

Example 1.3.2 Write an algorithm to find the maximum among three numbers a, b, c.

Step 1: Read the numbers as a, b, c.

Step 2: Set $Big = a$.

Step 3: If $Big < b$ then set $Big = b$.

Step 4: If $Big < c$ then set $Big = c$.

Step 5: Print the maximum number Big.

Step 6: Stop.

1.4 Flowchart

The pictorial representation of an algorithm is called a **flowchart**. In a flowchart each processing step is placed in a box and arrows are used to indicate the next step. Boxes of different shapes are used to indicate different operations.

There are many symbols used in drawing a flowchart, but many of them are not useful. Here we illustrate some of them, which are sufficient to draw the flowchart of any algorithm.

1. **Start/Stop box :** This is flat-oval or elliptic in shape and is used to indicate the beginning or end of an algorithm. One of the following words BEGIN, START, STOP, EXIT, END, RETURN, etc., is written within the box.

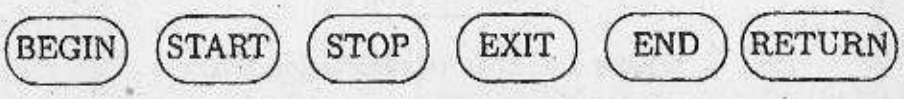


Figure 1.1: Start/Stop box

2. **Input / Output box :** Usually a parallelogram is used to indicate the input or output of an algorithm.

Figure 1.2: Input/Output box

3. **Assignment / Computation box :** Generally a rectangle is used to indicate the assignment of a new value or the value or result of some previous computation.

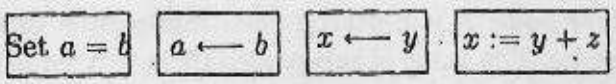


Figure 1.3: Assignment box

4. **Flow indicator :** The arrow is used to indicate the flow/order in which the steps of the algorithm are to proceed.

Figure 1.4: Assignment / Flow indicator symbol

5. **Decision box :** A diamond shaped box is used to indicate a decision position. A logical expression is written within the diamond box. Depending on its value the direction of the flow is considered and is indicated by an arrow.

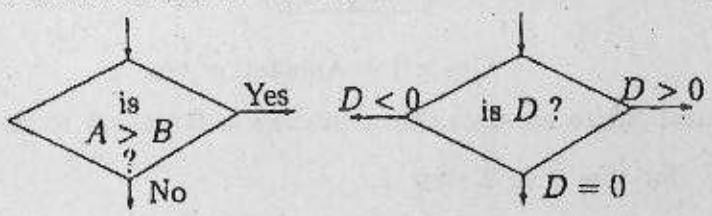


Figure 1.5: Decision box

6. **Loop representation box :** The hexagonal box is used to indicate the beginning of a loop.

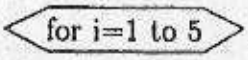


Figure 1.6: Loop representation box

7. **Connection box :** The circle is used as a connection box. When it is inconvenient or impossible to draw a flow chart within a single page, e.g., when a flow chart is so large that it requires more than one page to be drawn, then this symbol is used. The numbers are written within the circles which corresponds the continuation of flow.

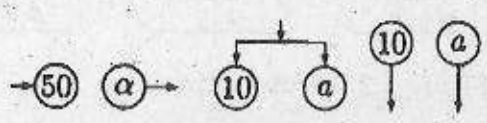


Figure 1.7: Connection box

8. **Sub-procedure box :** A sub-procedure is represented in a flowchart by a double lined rectangle. The name of sub-procedure is written within the box and the details of this sub-procedure are specified else where.

Figure 1.8: Sub-procedure box

9. **Annotation box :** It is used to add comment in a flowchart. It is symbolised by an open-sided rectangle and is connected to the flow chart by a dotted line.

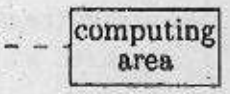


Figure 1.9: Annotation box

Example 1.4.1 Write an algorithm and draw a flowchart to find the values of $y = \sqrt{1+x^2}$ for $x = 0$ to 2 step .2.

Algorithm :

Step 1: Set $x = 0$.

Step 2: Compute $y = \sqrt{1 + x^2}$

Step 3: Print y.

Step 4: Add .2 with x (*i.e.*, $x = x + 0.2$).

Step 5: If $x \leq 2$ then goto Step 2.

Step 6: Stop.

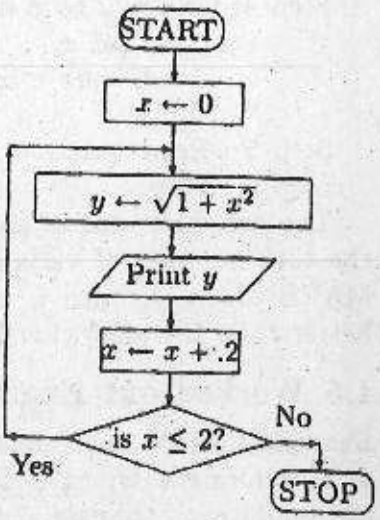


Figure 1.10 : Computation of a function

While writing an algorithm the following two important points are to be noted carefully.

(i) The time required by the algorithm (time complexity) should be as minimum as possible

(ii) The total space required by the algorithm (space complexity) should also be as minimum as possible.

An algorithm is said to be **efficient** if it takes less time and space.

In the following, two algorithms MAXA and MAXB are designed for the same problem, viz., to find the maximum among n numbers and the better one is noted.

Algorithm MAXA

Step 1: Read n (number of elements) and $x_1, x_2, \ldots, x_n$ (the elements).

Step 2: Set $max = x_1$

Step 3: for $i = 2$ to n do

    If $max < x_i$ then $max = x_i$ endif;

endfor;

Step 4: Print max

Step 5: Stop.

Algorithm MAXB

Step 1: Read n (number of elements).

Step 2: Read x.

Step 3: Set $max = x$.

Step 4: for $i = 2$ to n do
read x;
if $max < x$ then $max = x$ endif;
endfor;
Step 5 : Print max.
Step 6 : Stop.

The variables used in algorithm MAXA are n, x_1, x_2, . . . , x_n, max, i, *i.e.*, the total number of variables is $n + 3$ and the variables used in algorithm MAXB are n, x, max, i, *i.e.*, only four variables. Thus algorithm MAXB is better than the algorithm MAXA in respect of space complexity.

1.5 Worked out Examples

Example 1.5.1 Write an algorithm to read three lengths a, b, c and check whether a, b, c forms a triangle. If they forms a triangle then check whether it is (i) equilateral triangle, (ii) isosceles triangle, (iii) right angled triangle, (iv) scalene triangle.

Algorithm **Triangle-test.**
Step 1: Read a, b, c
Step 2: Arrange a, b, c such that $a > b > c$ as
(i) if $a < b$ then swap(a, b)
(ii) if $a < c$ then swap(a, c)
(iii) if $b < c$ then swap(b, c).
/* swap(a, b) interchanges the values between a and b */
Step 3: If $b + c > a$ then
(i) Print 'the sides a, b, c forms a triangle'
else
(i) Print 'the sides a, b, c does not form a triangle'
(ii) Stop
endif;
Step 4: If $a = b = c$ then
(i) Print 'the triangle is equilateral'
(ii) Stop
endif;
Step 5: If $a \neq b$ and $b \neq c$ and $a \neq c$ then
(i) Print 'the triangle is scalene'
endif;
Step 6: If $a = b$ or $b = c$ or $c = a$ then
(i) Print 'the triangle is isosceles'
endif;

Step 7: If $a^2 = b^2 + c^2$ then
(i) Print 'the triangle is right angled'
(ii) Stop
endif;
Step 8: End.

Note : Comments or heading may be put within / * *\/ in an algorithm.

Example 1.5.2 The Fibonacci sequence is defined as follows :

The first and second terms of the sequence are 0 and 1. The third and subsequent terms of the sequence is the sum of the two terms just preceding it, *i.e.*, $t_0 = 0$, $t_1 = 1$, $t_n = t_{n-1} + t_{n-2}$, $n \geq 2$. Write an algorithm and draw a flowchart to obtain all numbers in the Fibonacci sequence less than a fixed number, say 300.

Algorithm **Fibonacci.**

Step 1: Set F0 = 0 and F1 = 1.
Step 2: Compute F2 = F0 + F1.
Step 3: If F2 < 300 then
(i) Print F2.
(ii) Set F0 = F1, F1 = F2.
(iii) Goto Step 2.
Endif;
Step 4: End.

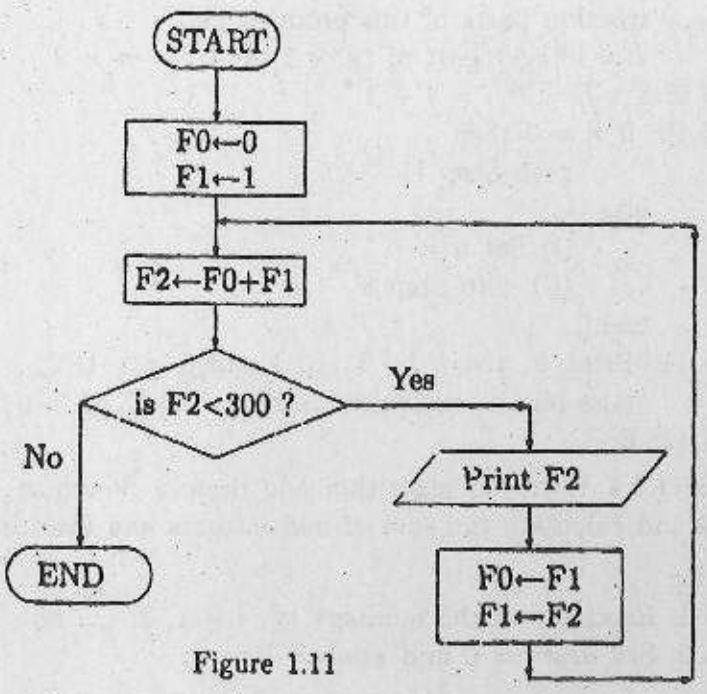


Figure 1.11

Example 1.5.3 Write an algorithm and draw a flowchart to convert a decimal number (integer and fraction) to its binary equivalent.

The following algorithm convert a decimal number whose integer part is m and fraction part is n to its binary equivalent. Step 2 to Step 6 converts m to its binary equivalent while Step 7 to Step 10 converts n to its binary equivalent.

Algorithm **B2D**

Step 1: Read integer part as m and fraction part as n.
Step 2: Set $i = 1$.
Step 3: Divide m by 2. Let q and r be the quotient and remainder,
i.e., q = integer part of (m/2) and $r = m - q \times 2$.
Step 4: Set $b_i = R$ and $i = i + 1$, (b_1 is the least significant digit)
Step 5: If $q = 0$ then
goto Step 6
else
(i) Set $m = q$
(ii) goto Step 3
endif;
Step 6: Set $l = i$; (l is the total number of bits in the integer part)
Step 7: Set $i = 1$
Step 8: Multiply n by 2. Let I, r be the integer and
fraction parts of this product, i.e.,
I = integer part of $(n \times 2)$ and $r = n \times 2 - I$.
Step 9: Set $f_i = I$; $i = i + 1$;
Step 10: If $r = 0$ then
goto Step 11
else
(i) Set $n = r$
(ii) goto Step 8
endif;
Step 11: Print b_i, $i = l, l - 1, \ldots, 1$ and f_j, $j = 1, 2, \ldots, i$.
(the binary equivalent is $b_l b_{l-1} \ldots b_2 b_1 . f_1 f_2 \ldots f_i$)
Step 12: End.

Example 1.5.4 Write an algorithm and draw a flowchart to read a set of n integers and calculate the sum of odd integers and even integers separately.

Algorithm
Step 1: Read n and the numbers m_i, $i = 1, 2, \ldots n$.
Step 2: Set $osum = 0$ and $esum = 0$.

Step 3: For $i = 1$ to n do

if ($int(m_i/2) * 2 = m_i$) then // $int(m_i/2)$ gives the integer

$esum = esum + m_i$ //part when m_i is divided by 2

else

$osum = osum + m_i$

endif;

endfor;

Step 4: Print sum of odd integers *osum* and sum of even integers *esum*.

Step 5: End.

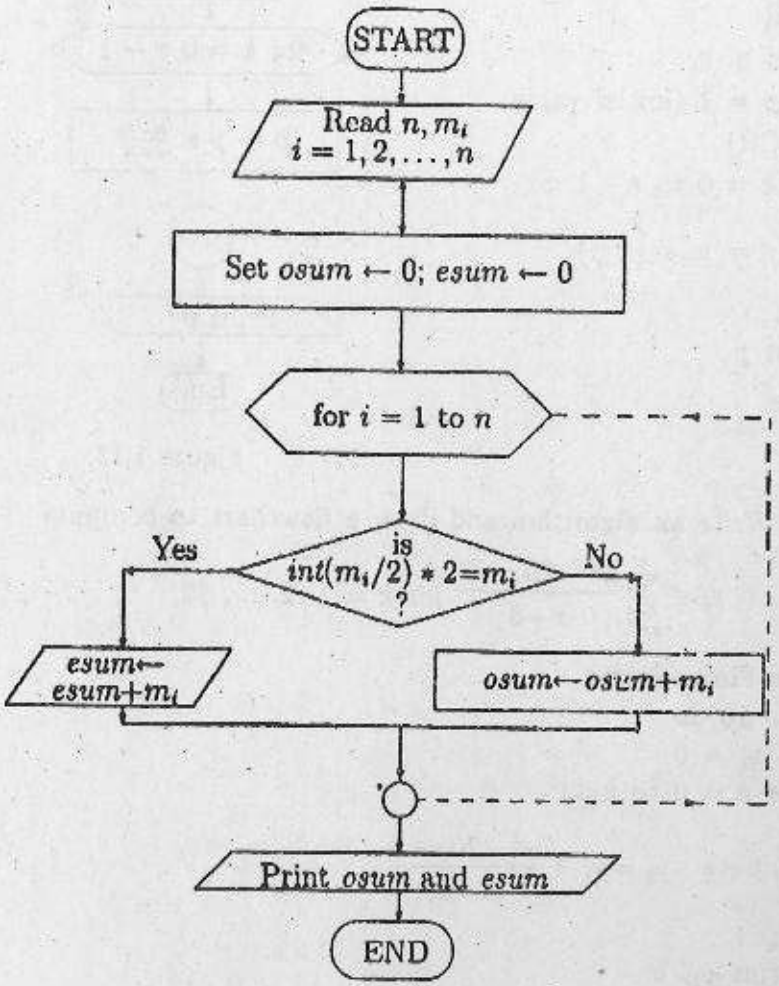


Figure 1.12

Example 1.5.5 Write an algorithm and draw a flowchart to find the value of nCr.

We know

$$nC_r = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)...(n-r-1)(n-r)!}{r!(n-r)!}$$

$$= \frac{n(n-1)...(n-r-1)}{r(r-1)...2.1}$$

$$= \prod_{k=0}^{r-1}\left(\frac{n-k}{r-k}\right)$$

Algorithm NCR

Step 1: Read n, r.

Step 2: Set $p = 1$ (initial value of NCR)

Step 3: For $k = 0$ to $r - 1$ do

$$p = p \times \left(\frac{n-k}{r-k}\right)$$

endfor;

Step 4: Print p.

Step 5: End.

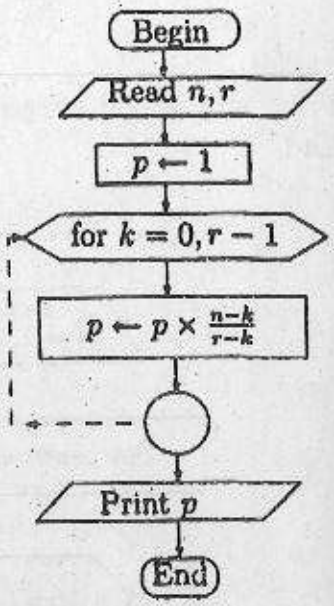


Figure 1.13:

Example 1.5.6 Write an algorithm and draw a flowchart to compute

$$y_k = \sum_{x=0}^{k} \frac{x^2 - 20x + 8}{x + 8} \quad \text{for } k = 1, 2, ..., 10.$$

Algorithm **Sum-Finite-Series**

For $k = 1$ to 10 do

(i) Set $y_k = 0$

(ii) For $x = 0$ to k do

Compute $y_k = y_k + \frac{x^2 - 20x + 8}{x + 8}$

endfor;

(iii) Print y_k, k

endfor;

End.

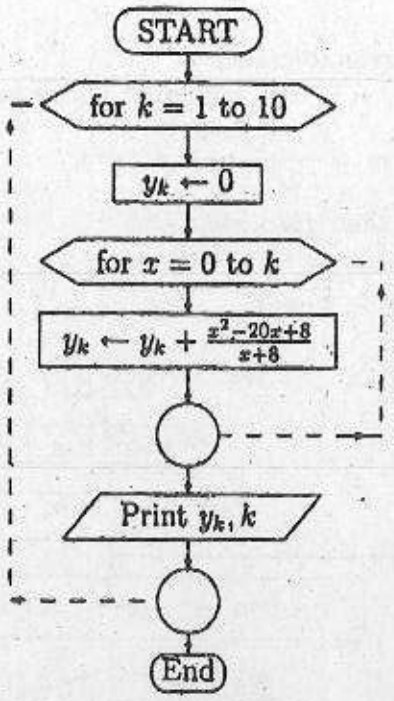


Figure 1.14:

Example 1.5.7 Write an algorithm and draw a flowchart to find the sum of the following series (zero-th order Bessel function):

$$J_0(x) = 1 - \frac{x^2}{2^2.1!.1!} + \frac{x^4}{2^4.2!.2!} - \frac{x^6}{2^6.3!.3!} + \ldots.$$

To calculate the sum of an infinite series, calculate a term for a given x and check whether it is less than a specified small number ε (called the error tolerance). If the value of the term is not small then add it with the sum and calculate next term. Repeat this process until the term is smaller than ε.

The nth and $(n+1)$th terms of the series are respectively

$$t_n = (-1)^n \frac{x^{2n}}{2^{2n}.n!n!}, \quad t_{n+1} = (-1)^{n+1} \frac{x^{2n+2}}{2^{2n+2}.(x+1)!(n+1)!}$$

$$\frac{t_{n+1}}{t_n} = -\frac{x^2}{4(n+1)^2} \quad \text{or} \quad t_{n+1} = -\frac{x^2}{4(n+1)^2} t_n, \; n \geq 1 \text{ and } t_0 = 1.$$

That is, the next term $= -\frac{x^2}{4(n+1)^2} \times$ the previous term.

Algorithm **J0X**

Step 1: Read x and error tolerance ε.

Step 2: Set $term = 1$, $sum = 1$ and $n = 0$ (the initial term and sum).

Step 3 : Compute $term = -\dfrac{x^2}{4(n+1)^2} \times term$.

Step 4: If $|term| < \varepsilon$ then goto Step 7.

Step 5: Compute (i) $sum = sum + term$;

(ii) $n = n + 1$;

Step 6: Goto Step 3.

Step 7: Print sum and x.

Step 8: End.

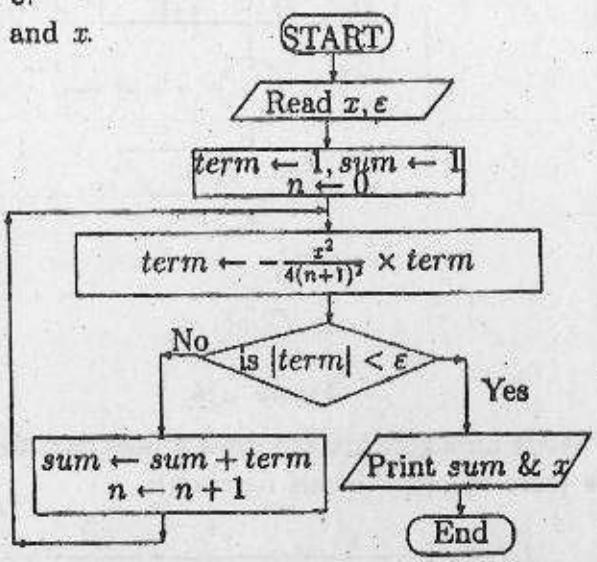


Figure 1.15:

Example 1.5.8 Write an algorithm and draw a flowchart to find the roots of a quadratic equation.

Let $ax^2 + bx + c = 0$ be the quadratic equation, where $a \neq 0$. The roots of this equation are

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

Let $d = b^2 - 4ac$ and $p = \dfrac{-b}{2a}$ and $q = \dfrac{\sqrt{|d|}}{2a}$.

If $d > 0$ then the roots are $p + q$ and $p - q$, if $d = 0$ then the roots are p, p and if $d < 0$ then the roots are $p + iq$ and $p - iq$.

Algorithm **Quadratic**

Step 1: Read the coefficients as a, b, c.

Step 2: Calculate $d = b^2 - 4ac$.

Step 3: Calculate $p = -b/(2a)$ and $q = \sqrt{|d|}/(2a)$

Step 4: If $d > 0$ then

Print 'the roots are real and distinct and they are', $p + q$, $p - q$.

elseif $d = 0$ then

Print 'the roots are real and equal and they are', p, p.

else

Print 'the roots are imaginary and they are', $p + iq$, $p - iq$.

endif;

Step 5: End.

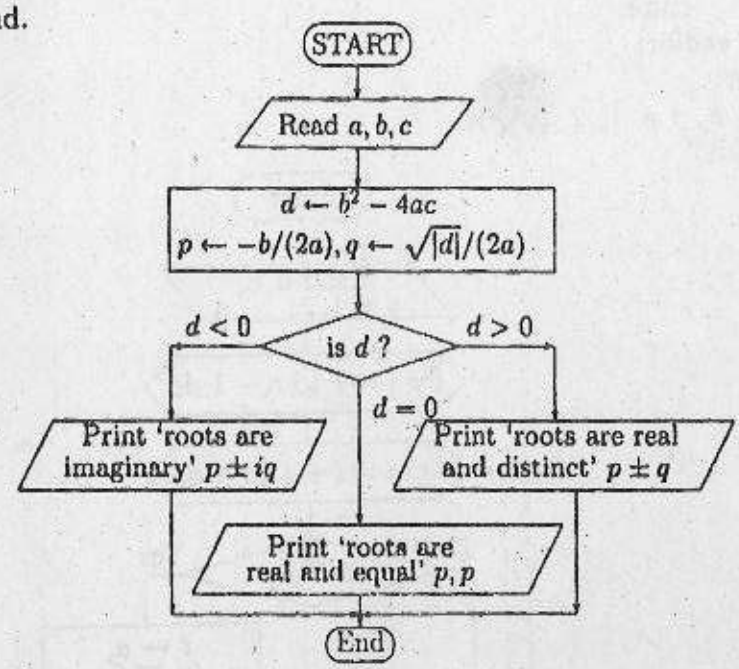


Figure 1.16:

Example 1.5.9 Write a program to arrange n numbers in ascending order.

Solution. Sorting of numbers is a very important problem in Computer Science. Different sorting techniques are available. Here we present straight selection sort algorithm and the program to do it. The idea of this sorting technique is very simple. In the first step, the first element is compared with the remaining elements and the minimum element is placed in the first position. In the second step, the second element is compared with the remaining elements, except the first one, and the minimum one is placed in the second position. This process is continued until the last but one element is compared with the last element. The algorithm is given below.

```
Algorithm SSORT
   /* sorting the array a of size n in non-decreasing order */
   Read n, a_i, i = 1, 2, . . . , n.
   for i = 1 to n - 1 do
         for j = i + 1 to n do
            if (a_i > a_j) then /* interchange a_i and a_j */
                  t = a_i;
                  a_i = a_j ;
                  a_j = t;
            endif;
         endfor;
   endfor;
   Print a_i, i = 1, 2, ... , n.
end SSORT
```

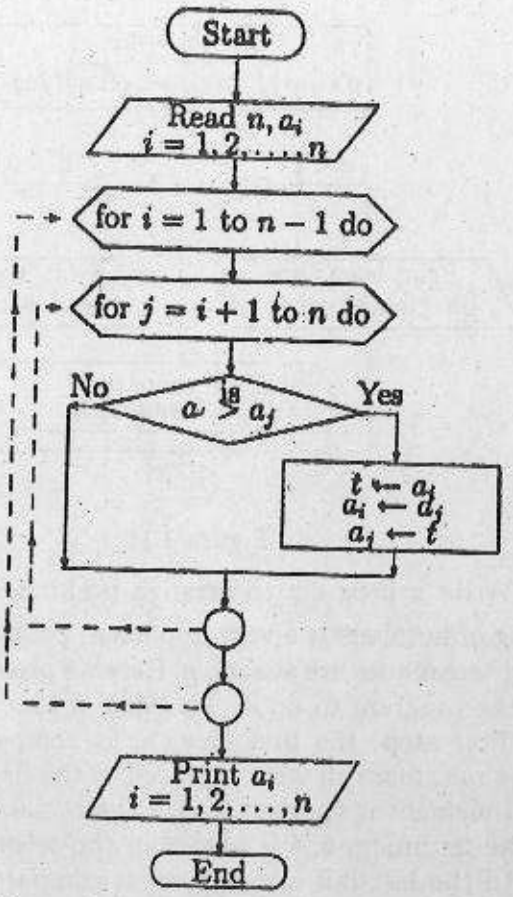


Figure 1.17: Flowchart of sorting

Example 1.5.10 Write an algorithm and draw a flowchart to compute G.C.D and L.C.M of two given integers.

Algorithm **GCD-LCM**

Step 1: Read the numbers as m and n.

Step 2: Calculate the product $p = m \times n$.

Step 3: Compute q (quotient) = integer part of (m/n).

Step 4: Compute q (remainder) $= m - n \times q$.

Step 5: If $r = 0$ then

goto Step 6

else

(i) Set $m = n$

and $n = r$.

(ii) goto Step 3.

endif;

Step 6: Compute l (the L.C.M) $= p/n$.

Step 7: Print n (the G.C.D) and l (the L.C.M).

Step 8: End.

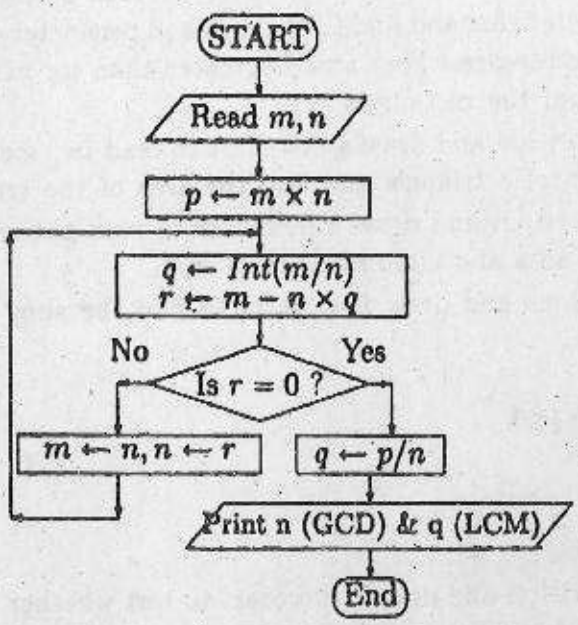


Figure 1.18

1.6 Summary

Here the definition of algorithm is given. Its basic characteristics and design technique are also discussed. The definition of flowchart, different symbols used to draw it and drawing method are provided along with a large number of worked out examples. This unit ends with a good exercise.

Exercise 1

1. Define the term algorithm. What are the basic criteria of an good algorithm?
2. What do you mean by a flowchart? Draw the different symbols used to draw a flowchart.
3. Write an algorithm and draw a flowchart to find the largest among four given numbers.
4. Write an algorithm and draw a flowchart to find the maximum and minimum among n given numbers.
5. Given a set of n integers. Write an algorithm and draw a flowchart to find
   (i) total number of even integers.
   (ii) sum and product of the even integers.
6. Write an algorithm and draw a flowchart to read 50 pairs of length and breadth of rectangles, and find (i) the area and perimeter of each rectangle, (ii) all the rectangles whose area is greater than its perimeter, (iii) the average area of the rectangles.
7. Write an algorithm and draw a flowchart to read two sides and the angle between them of a triangle and find the area of the triangle.
8. Write an algorithm and draw a flowchart to read the radii of 50 circles and find the area and circumference of each.
9. Write algorithms and draw flowcharts to find the sum of the following series:

   (i) $\sum_{r=1}^{100} x^r,\ |x| \le 1$

   (ii) $x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \cdots$
10. Write an algorithm and draw a flowchart to test whether an integer is (i) prime, (ii) perfect square, (iii) even or odd, and (iv) divisible by 7. [In each case separate algorithm and flowchart are required]

11. Write an algorithm and draw a flowchart to generate all prime numbers between two given integers.
12. Write an algorithm and draw a flowchart to compute the value of $n!$.
13. Write an algorithm and draw a flowchart to split a number into digits.
14. Write an algorithm and draw a flowchart to find the value of ${}^nP_r$.
15. Write an algorithm and draw a flowchart to list all three digited numbers which are equal to the sum of the cubes of the digits. [Such numbers are called Armstrong numbers]
16. Write an algorithm to convert a binary number to its decimal equivalent.
17. Write an algorithm and draw a flowchart to solve the following equations

$$a_1x + b_1y = c_1$$
$$a_2x + b_2y = c_2.$$

18. Write an algorithm and draw a flowchart to compute the value of the function

$$f(x) = \begin{cases} x^3e^x & \text{if } x \le 1 \\ x\cos x & \text{if } x > 1 \end{cases}$$

for $x = 0.1, 0.2, \ldots, 1.5$.

19. Write an algorithm and draw a flowchart to compute the sum of (i) $S = 1^2 + 2^2 + \ldots + 100^2$, (ii) $S = 1^3 + 2^3 + \ldots + N^3$ (where N is a given integer).
20. Write an algorithm and draw a flowchart to obtain the scalar product of two vectors $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$ given by

$$\text{scalar product} = \sum_{i=1}^{n} x_i y_i$$

UNIT 2 ❐ PROGRAMMING WITH C

2.1 Introduction to C Programming

The C programming language was originally designed for and implemented on the UNIX operating system by Dennis Ritchie at AT & T Bell Laboratory, USA around 1972. C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than other more powerful languages. The operating system, the C compiler, and essentially all UNIX applications programs are written in C.

During the 1970's, C has undergone many changes for making it reliable and efficient and acceptable by many users working in different areas. During these period a lot of different features and facilities are developed by different people across the world. These different forms of the language become machine dependent. To remove the machine dependence of the language the American National Standard Institute (ANSI) constitute a committee to standardize the language. This resulted in ANSI C and it is machine independent and also operating system independent. Thus, C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.

This book will help the reader learn how to program in C.

2.2 Objectives

After going through this unit you will be able to learn about-

(i) C constant and variables
(ii) Operators and expresions
(iii) Input/output statements of C
(iv) Control statements if, do, do-while, for, etc.
(v) Array and its used
(vi) Functions of C
(vii) Pointers, structures and unions

(viii) Character and string processing
(ix) Use of files in C
(x) Macro and preprocessor.

2.3 Constants and Variables

To solve a problem by computer, the programmer should supply the program and data to the computer. These data may be of numeric type or alphabetic type. The numeric data are also of different types, such as, whole number or fraction. Again, these supplied data are stored at some locations of memory and these stored values are referred by names of variables. The value of a variable at any instant, during the execution of a program, is the value stored at the location identified by the variable name. The variables are also of different types. In this section, we discuss in detail, different types of constants and variables.

2.3.1 Character Set

A computer key board contains many symbols (characters), some of them are used to write a program in C.

The C character set is shown in Table 2.1.

Some characters produce blank space (horizontal or vertical) during printing are called *whitespace*. In C they are *blank space, horizontal tab, carriage return, newline, form feed*, etc.

The digits combined with the letters are called **alphanumeric characters**, and the remaining symbols are termed as **special characters**. In C language, *upper case letter and lower case letter are different*, e.g., *N* and *n* are different. That is, *C is a case sensitive language*. A program in C must be written using only the defined set of characters.

2.3.2 Constant Data

Any entry which remains unchanged during the execution of a program is defined as a **constant**. It may be of numeric, character or logical type. Like other programming languages, C also supports several different types of constants, each of which is represented differently in terms of the computer memory allocation.

Mainly three types of constants are available in C, viz., integer (`int`), floating point (`float`) and character (`char`). These data types also have some extended forms.

A-Z	Alphabets (Upper case)	a-z	Alphabets (Lower case)
0-9	Digits	␣	blank space
,	comma	.	period
;	semicolon	:	colon
?	question mark	'	single quotation
"	double quotation	!	exclamation mark
|	vertical bar	/	slash
\	backslash	~	tilde
-	underscore	$	dollar
%	percent sign	&	ampersand
^	caret	*	asterisk
-	minus sign	+	plus sign
>	greater than sign	<	less than sign
(	left parenthesis	)	right parenthesis
[	left bracket	]	right bracket
{	left brace	}	right brace
#	hass sign	=	equals sign

Table 2.1: The C characters set

The memory requirement of each numerical constant determines the permissible range of values it can take. The memory requirements of different types of constants vary with computers.

Numeric type

Any string of digits preceded by a single algebraic sign (+ or −) is called a **numeric constant**, which may also contain a decimal point.

Numeric type constants are further subdivided as integer and real (floating point)

Integer constant

An **integer constant** is a whole number that is written as a string of decimal digits without a decimal point or an exponent symbol. It can be either positive or negative. It is also called a fixed point constant. The integer constant is of three types–`short int`, `int` and `long int`, in both signed and unsigned forms. The `long` and `unsigned` integers are used to increase the range of values.

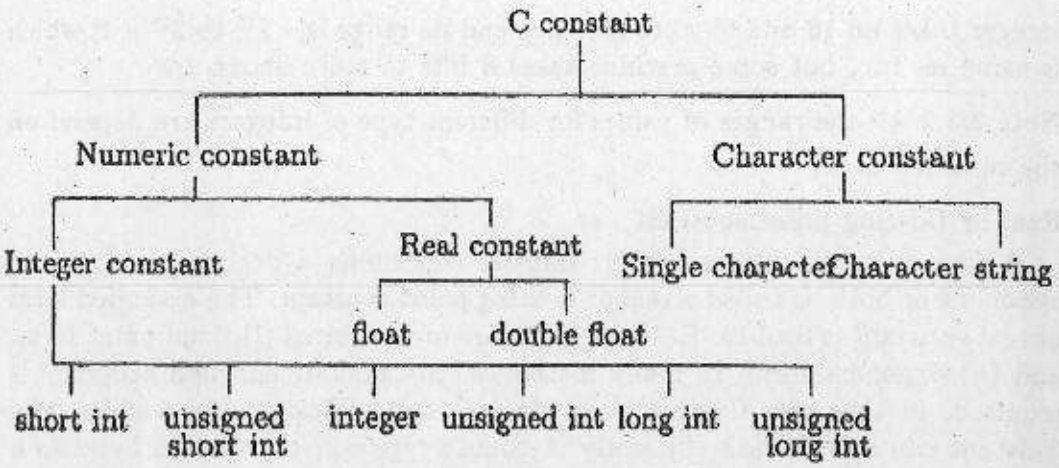


Figure 2.1: Types of constants

Some valid integer constants :

45 2 -66 +34223

Note 2.3.1 For a positive integer '+' sign is optional. The range of the integer constants vary from computer to computer.

C languages also support octal and hexadecimal integers. An octal constant begins with zero (0) and a hexadecimal constant begins with 0x or 0X. For example, the decimal integer 15 is written in octal system in the form 017 and in hexadecimal system it is of the form 0xf or 0XF.

The valid range for integer depends on the word-length of the computer. For a 16-bit word-length computer, the allowable range of integer constant is -2^{15} to $2^{15} - 1$ (i.e. from -32,768 to 32,767). It may be noted that, in *computer arithmetic set of integers is finite but, in algebra it is infinite.* Before storing an integer in a computer, it should be converted to an equivalent binary number.

C also supports other types of integers called short integer, long integer, unsigned integer, etc. The qualifiers u, l and ul (or U, L, UL) are used to represent unsigned integer, long integer and unsigned long integer. For example, 3458U, -98308766L, 89288339UL represent unsigned integer, long integer and unsigned long integer respectively. In unsigned integer, all 16 bits are used to store the magnitude of the number. Thus for a 16-bit computer the range of unsigned integer is 0 to $2^{16} - 1$. A long integer constant uses 32-bit to store its value and the range of the values is -2^{31} to $2^{31} - 1$, i.e. from -2147483648 to 2147483647. However, the range of unsigned long integer is from 0 to $2^{32} - 1$, i.e. from 0 to 4294967295, and needs 32 bits to store it. On the other hand, short

integer takes on 16 bits to store its value and its range is -2^{15} to $2^{15} - 1$, which is same as `int`, but some machine takes 8 bits to store short int.

Note 2.3.2 All the ranges of values for different type of integers are depend on the machine used.

Real or floating point constant

A signed or an unsigned whole number containing a decimal point or an exponent or both, is called a **real or floating point constant**. The extended form of real constant is double. Real constants are of two forms (i) fixed point form, and (ii) exponent form. In many mathematical calculations, high accuracy is required, in this case floating point data is not sufficient, the double type constant can do this task efficiently. A double type constant uses 8 bytes as a storage space. These type of constant is known as **double precision** constant. Double and floating point constants are of similar, but double precision constant gives more precision. Double precision numbers may also be defined as `long float`.

(i) *Real constant of fixed point form*

A real constant of fixed point form is a constant with a decimal point and may have a fractional part.

Some valid real constants :

3.432, 0.34, –0.678, 0.000023, .324, 2.

(ii) *Real constant of exponent form*

Often very large or very small numbers are involved in scientific computations. It is not possible to represent these numbers using fixed point form. These numbers can be represented by a form known as exponent form. A real constant of the exponent form have a format containing a mantissa and an exponent. The mantissa must have at least one digit and a decimal point. It may have a sign. The mantissa is followed by the letter E and an exponent. The exponent must be an integer (without a decimal point) and must have at least one digit. A sign of the exponent is optional.

Some examples of real constants of exponent form are given below:

Scientific form	C floating point form
6.23×10^{23}	6.23E23 or 6.23E + 23
$000345 = .345 \times 10^{-3}$	.345E-3
10^{-14}	1.0E-14

Some valid real constants :

344.e02, 456.0E3, 678.e+5, 456.E+06, 34.1E4, 3467.E5, -.210E-4, 678.234E+14, 02341E03.

The Table 2.2 shows the size and range of constants of different types for the computers having 16-bits as word length.

The memory requirement of each numerical constant determines the permissible range of values it can take. The memory requirements of different types of constants vary with computers. The readers are requested to check the valid range of data in your computer.

Types of constant	Number of bytes used	Range of values	
short int	2	-2^{16-1} to $2^{16-1} - 1$	-32768 to 32767
unsigned short	2	0 to $2^{16} - 1$	0 to 65535
int	2	-2^{16-1} to $2^{16-1} - 1$	-32768 to 32767
unsigned int	2	0 to $2^{16} - 1$	0 to 65535
long int	4	-2^{32-1} to $2^{32-1} - 1$	-2, 147, 483, 648 to 2,147,483,647
unsigned long int	4	0 to $2^{32} - 1$	0 to 42949672965
float	4		8.43×10^{-37} to 3.37×10^{38}
character	1		-128 to 127
double or long float	8		$2.225074 \times 10^{-308}$ to 1.797693×10^{308}

Table 2.2: Memory allocation for constants

Character constant

A character constant is a single character written within single quotations (apostrophes). For example, 'a' 'z' '6' '$' ' ' etc. are the valid character constants. But, '12' (more than two characters), "a" (double quotations) are not valid character constants. Each character constant has unique integer value for a given computer. This code may vary from one computer to another. The integer value corresponding to a character is called ASCII (American Standard Code for Information Interchange) code (most of the computer uses ASCII character set). In ASCII code, each character is coded using 7 bits, and hence $2^7 = 128$

different characters can be coded using ASCII code. It may be noted that one byte space is required to store a character type constant.

Since each character has a unique integer value, it is possible to perform arithmetic operations on character set. For example, the ASCII values of '3' and 'z' are respectively 51 and 122. The the value of the expression '3'-'z'= 51-122 = -71. The value of '3'-2= 51 - 2 = 49.

String constant

A string constant is a string of characters (alphabets, digits or special characters) enclosed in double quotation marks.

The following are valid string constants :

"X = 12" "INDIA" "B.Sc" "C = A + B"

"VIDYASAGAR" "Final Examination" "Sum ="

It may be remembered that the constants "A" and 'A' are different. 'A' represents a character constant and it has an integer value 65, while "A" represents a string constant and it does not have any value.

Escape Sequences

C provides some more characters other than character constants and string constants, which are normally used during printing of output. These non-printable characters are called **escape sequences** or **backslash characters**. An escape sequence always begins with a backslash and is followed by one or more special characters. For example, a line feed (LF, ASCII code 10) is used as a *newline*, which is denoted as \n. Escape sequence represents a single character, even though it is written as a combination of two or more characters.

Some commonly used escape sequences are listed in Table 2.3.

The null character \0 has a particular use. This character is used to indicate the end of a *string* and it may be noted that null character \0 is not equivalent to the character constant '0'.

2.3.3 C Variables

A quantity that varies during program execution is called a **variable**. Each C variable has a specific storage location in memory where its value is stored. The following rules must be followed by every C variable.

Escape sequence	Character/use	ASCII value
\a	bell (alert)	007
\b	backspace	008
\t	horizontal tab	009
\n	newline (line feed)	010
\v	vertical tab	011
\f	form feed (new page)	012
\r	carriage return	013
\"	double quotation	034
\'	single quotation	039
\?	question mark	063
\\	backslash	092
\0	null	000

Table 2.3: Escape sequences

(i) The first character must be an alphabet. However, some systems allow underscore (_) as the first character.

(ii) The variable name can have at most 31 characters, in ANSI C. If a variable name contains more than 31 characters, the first 31 characters are to be recognized by the compiler and the remanning characters will be ignored. Some compilers recognize only the first 8 characters and some other compilers support arbitrary number of characters.

(iii) The variable name should not contain any special character other than underscore (_). Blank spaces are not allowed as valid characters in variable name.

(iv) No C keywords (which have some standard, predefined meaning/action in C, these keywords can be used only for their intended purpose) are allowed as variable name. A list of keywords are given in Table 2.4.

(v) Both upper case and lower case alphabets are used as variable name. But, they are used for naming two different variables names. For example, N and n are two different names. Similarly, PNB, pnb, Pnb, pNB, pnB, etc. are all distinct variable names.

Some compilers may also support the following keywords.

`ada, far, near, asm, fortran, pascal, entry, huge`

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	registrar	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Table 2.4: C keywords

The C language provides different types of variables like constants, viz., (a) integer, (b) real, (c) character, and (iv) string.

Array

Sometimes, it is necessary to represent a group of values by a common variable name. For example, the roll number of 100 students of a class may be represented by a variable, say `roll` and the variables `roll[0]`, `roll[1]`, ..., `roll[99]` represent the rolls of first student, second student, ..., 100th student. A set of data with similar properties and which are stored in consecutive memory locations under a common variable name is called an **array** or a **subscripted variable**. Here `roll` is an array, it is also called the subscripted variable with a subscript. Each element of an array is identified by using subscript (or subscripts) within square brackets after the common variable names (i.e. by the array name). The individual items are called the array elements. For example, `roll[4]` is the fifth element of the array `roll`. An array is called by the array name.

There are several types of arrays such as integer array, floating point array, character array, and also one-dimensional array, two-dimensional array, etc. The one-dimensional character array is known as **string**. Here we concentrate only on string, i.e. on one-dimensional character array.

If an array contains n elements, the subscript will be an integer quantity whose values are 0, 1, ..., n - 1. But, an n-character string contains $(n + 1)$ elements, because of the null character (\0) automatically be placed at the end of the string.

To explain the behaviours of string, let us consider the string "India". Suppose this string is stored into a one-dimensional array x. The string "India" contains 5 characters, so the size of the array x is 6. The value of each array element of x is shown below.

Element no.	Subscript	Array element	String character
1	0	x[0]	I
2	1	x[1]	n
3	2	x[2]	d
4	3	x[3]	i
5	4	x[4]	a
6	5	x[5]	\0

Thus the third element of the character string x is the alphabet d.

2.3.4 Variable Declarations

It is required to declare the type of the variable before it is used in a program.

All variables must be declared before the first executable statement. It is better to declare all the variables at the beginning of each function/program.

A declaration consists of a data type followed by one or more variable names (separated by comma(s), if there are more than one variables) and terminates by a semicolon (;). The array variable must be followed by a pair of square brackets, and within the bracket there is a positive integer mentioning the size of the array. The syntax for declaration of variable $a_1, a_2, \ldots, a_n$ as variables is

data type a_1, a_2, ..., a_n;

Suppose we consider two variables roll and marks. If they are integer variables, then the declaration statement is

```
int roll, marks;
```

or it can also be declared as

```
int roll;
int marks;
```

or as

```
int roll; int marks;
```

It may be noted that, comma (,) represents the separation between the variables and semicolon (;) represents the end of the declaration.

If roll is an integer variable and marks is a real variable, then the declaration statement may be of the following form.

```
int roll; float marks;
```

or

```
int roll;
float marks;
```

Suppose name is an array which represents the name of a country and assumed that the maximum number of characters required to naming the country is 25, then the appropriate declaration is

```
char name[26];
```

Thus, the variable name can take 26 characters at a time as its value. But, if we declare

```
char name;
```

then the variable name will take only one character as its value.

Initialization of Variable During Declaration

The initial values of the variables can also be assigned during type declaration. This declaration must contain a data type, followed by a variable name, an equal sign and an appropriate constant value. The syntax of this declaration is

```
data_type   variable_name = constant;
```

The following are the valid declarations.

```
int      i=0;
float    x=10.5, y=2.5, p=8.54;
char     n='x';
char     country[]="India";
```

Note that the declaration char country[]="India" In this declaration, country is a character array/string containing 6 elements. The first 5 elements represent the alphabet of the word "India" and the 6th element represents the null character (\0). Noted that the size of the array is not mentioned here, computer will automatically determine the size of the array. If we would like to mention the size of the array, then it must be accurate, otherwise the array will store incorrect information.

For example, the declaration char country[3]="India"; gives error; whereas the declaration char country[10]="India"; will store the word "India" and the null character into the first 6 array elements country[0] to country[5], the remaining elements may be assigned zeros or they may be filled by some other arbitrary characters.

2.3.5 Symbolic Constants

Sometimes it happened that a constant, say π, may be used in many places within a program. The value of π is an irrational number and its value correct up to 5 decimal places is 3.14159; and correct up to 8 decimal places is 3.14159265.

Suppose we calculated the value of some expressions (used in the program) using $\pi = 3.14159$. Later we observed that we need more accurate values of the expressions and we consider $\pi = 3.14159265$. If π occurs in many places in the program, then we have to update the value of π in all these places, but this process takes time and there is a chance to overlook one or more positions of π. In this situation, we may use a symbolic name for π, say PI. A **symbolic constant** is a name that substitutes for a sequence of characters. The characters may be a numeric constant, a character constant or a string of characters. During compilation of the program, all the symbolic names are replaced by the corresponding character sequence. The symbolic constants are generally defined at the beginning of a program.

The syntax of this declaration is

```
#define symbolic_name value_of_constant
```

symbolic_name represents a symbolic name, which follow the rules for a variable name, and usually written in upper case letters, to distinguished between symbolic constant and identifier. value_of_constant represents the sequence of characters for the symbolic name. symbolic_names are also known as *constant identifiers*.

Some valid symbolic constants are :

```
#define  PI       3.14159
#define  f(x)     x*x+3*x+5.2
#define  TRUE     1
#define  COUNTRY  "India"
```

The following points keep in mind during declaration of a symbolic constant.

- No blank space between the sign # and the keyword define is allowed.
- The first character of the line must be #.
- Blank spaces are required in between #define and symbolic_name and in between the symbolic_name and the value_of_constant.
- The declaration must not be end with a semicolon, because during compilation symbolic name is replaced by the constant and the semicolon will appear in the wrong position. For example, if N is declared as

  ```
  #define N 10;
  ```

  and the statement `M=N+5;` is in the program, then this statement becomes `M=10;+5;` and it is obviously incorrect.

- The value of the symbolic constant cannot be altered. For example, if the value of the symbolic constant TRUE is defined as 1 then the assignment TRUE=2 or TRUE=2*TRUE, etc. are illegal.
- No separate declaration for data types is required, its data type depends on the value assigned to the symbolic name.
- #define statements may appear in any place of the program, but usually it is placed at the beginning of the program.

Note 2.3.3 It may be noted that the value of a variable can be changed, but the value of a symbolic name cannot be changed. The symbolic name is the representative of some constant, character, string or an expression also. During compilation the value of the symbolic name is replaced by its value. But, variable always stored a single value.

Exercise 2.3

1. Write C characters set.
2. What is the difference between a variable and a constant? What are the rules for naming a C variable?
3. What are the advantages of using double variables?
4. Explain different types of constants used in C.
5. What do you mean by a variable? Explain C variables.
6. Write short notes on fixed point and floating point variables.
7. Which of the followings are not valid variable names in C? State reasons.
   (*i*) total-a (*ii*) end (*iii*) d.a. (*iv*) 5 number (*v*) x5 (*vi*) x + y
8. Write the type declaration statements to declare
   (*i*) roll, marks, subcode as integer variables.
   (*ii*) name, add as string variables each of length 30 characters.
   (*iii*) found, valid, x2 as double variables.
9. What do you mean by symbolic name? What is difference between symbolic name and variable.

2.4 Operators and Expressions

In C, an **expression** is a combination of variables, constants, operators and functions (to be discussed later on). An algebraic expression can not be entered to the computer directly. Before entering an algebraic expression in the computer it should be converted to a C expression. The algebraic and C expressions differs only at their syntax. Any algebraic expression can be converted to a C expression by using certain specific rules. At the time of execution of an expression the values of the variables (stored in memory) are used, to perform necessary operations. C provides three major types of expression namely (i) arithmetic expression, (ii) logical expression and (iii) character/string expression.

2.4.1 Arithmetic Operators

The following are the symbols for binary operators in C corresponding to different arithmetic operators.

Arithmetic operators	C operators
Addition (+)	+
Subtraction (-)	-
Multiplication ($\times$)	*
Division ($\div$)	/
Modulo division	%

The operator % gives the remainder when an integer is divided by another integer. The remainder operator (%) requires that both the operands must be integers and second operand is nonzero. Similarly, in division (/) the second operand should be nonzero. The division of an integer by another integer is called integer division and the result of this division is an integer.

It may be noted that like FORTRAN, BASIC, etc., C does not provide any operator for exponentiation. However, the library function (pow) is used to carried out exponentiation.

2.4.2 Arithmetic Expressions

Three types of arithmetic expressions are defined in C namely (i) integer expression, (ii) real expression and (iii) mixed mode expression.

Integer expression

The mathematical expression obtained by combining integer variables and integer constants with the help of arithmetic operators, is known as integer expression.

Rules for writing an integer expression:

(i) A signed or unsigned integer variable or an integer constant is an integer expression.

(ii) An integer expression connected by an arithmetic operator to an unsigned integer constant is an integer expression.

(iii) An expression enclosed in parentheses is an integer expression.

(iv) Two integer expressions connected by an arithmetic operator is an integer expression.

(v) Two arithmetic operators should not occur in succession in an integer expression.

Note on integer division

When an integer is divided by an another integer then the result may contain afractional part. But in C, the fractional part is discarded during calculation, e.g., the value of 5/4 is 1 and not 1.25, 4/5 is 0 and not .8.

The value of the expression 32/5*5+8/9 is calculated as follows :

32/5*5+8/9 = 6*5+0 = 30+0 = 30

Real expression

The mathematical expression obtained by combining real constants and real variables with the help of arithmetic operators, is known as a real expression.

Rules for writing a real expression :

(i) A signed or unsigned real variable or a real constant is a real expression.

(ii) A real expression connected by an arithmetic operator to an unsigned real variable name or an unsigned real constant is a real expression.

(iii) A real expression enclosed in parentheses is a real expression.

(iv) Two real expressions connected by an arithmetic operator is a real expression.

(v) Two arithmetic operators should not occur in succession in a real expression.

A real expression may be exponentiated by an integer expression to form a real expression.

Note 2.4.1 The expression is scanned from left to right, to complete the execution of all divisions and multiplications in the order of their appearance. Finally, all additions and subtractions are performed again from the left of the expression. If there is any parentheses within the expression then the expression within the parentheses is evaluated first using the above rules. The following expression is evaluated according to the above rules.

e.g., $4.0 * a - b/(c + 6.7)$

$= 4.0 * a - b/e_1$ where $e_1 = c + 6.7$

$= e_2 - e_3$ where $e_2 = 4.0 * a$ and $e_3 = b/e_1$

$= e_4$ where $e_4 = e_2 - e_3$.

At first the value of $c + 6.7$ (within parentheses) is evaluated and stored in e_1. In second step all multiplications and divisions are done from left to right. The values of $4.0 * a$ and b/e_1 are stored in e_2 and e_3 respectively. Finally, addition and subtraction are performed from left to right.

Mixed mode expression

An expression, formed by integer, real or character constants or variables is called **a mixed mode expression**. Modern computers execute mixed mode expressions. Suppose we have to compute the value of A/B. If A is an integer variable and B is real variable then before division A must be converted to real and then A is divided by B and finally the result is stored as a real number.

Examples on arithmetic expression

1. $\dfrac{a+b}{x-y^2}$ A translation of this expression into C is (a+b)/(x - y*y).

2. $23.9 \times 10^{-2}mv^2 + fg/4d$.

In C it becomes 23.9E-2*m*v*v+f*g/(4.*d)

Mode of arithmetic expression

A real variable may be set equal to an integer expression and vice-versa. If a real variable is set equal to an integer expression then the value of the integer expression is converted to real mode before it is stored as a real variable.

e.g., if j = 4, k = 2 and a = j/k then a = 2.0.
if j = 3 and k = 2 then a = 1.0.

If an integer variable or expression is set equal to a real expression then the value of the real expression is 'truncated' and stored as an integer variable. e.g., if l=3./2. then l=1 if k=2./4. then k=0

Hierarchy of arithmetic operators

In general, an arithmetic expression contains numeric constants, numeric variables (integer, real and character), arithmetic operators (addition, subtraction, multiplication, division and modulo division) and parentheses. At the time of execution of an expression only one arithmetic operation is executed at a time. The hierarchy of operators is the order in which the arithmetic operations are executed. The hierarchy of arithmetic operators is shown in Table 2.5.

operator	hierarchy
multiplication (*), division (/) and modulo division (%)	first
addition (+) and subtraction (-)	second

Table 2.5: Hierarchy of arithmetic operators

Example 2.4.1 Evaluate the following arithmetic expression, if a = 2.5, b = 7.2, c = 2.4, i = 5, j = 2

(i) a*b + c/j*i, (ii) i/j + b*c + i*a

where i, j are integer and a, b, c are real variables.

Solution. (i) a*b + c/j*i = 2.5*7.2 + 2.4/2*5
= 18.0 + 2.4/2*5 = 18.0 + 1.2*5 = 18.0 + 6.0
= 24.0

(ii) i/j + b*c + i*a
= 5/2 + 7.2*2.4 + 5*2.5 = 2 + 7.2*2.4 + 5*2.5
= 2 + 17.28 + 5*2.5 = 2 + 17.28 + 12.5
= 2.0 + 17.28 + 12.5 = 19.28 + 12.5
= 31.78

2.4.3 Library Functions

There are certain mathematical functions such as trigonometric, exponential, logarithmic, etc. which are frequently used in programs, especially to solve

scientific problems. These functions are generally referred as **library functions, built-in functions** or **intrinsic functions**. The general form of these functions is

```
function_name  (argument(s))
```

The argument of a function can be a valid variable name or an expression. It can also be a real number, an integer or a character. The argument should be written within parentheses. If a function has more than one argument then they are separated by commas.

A library function is used simply by writing the function name followed by necessary arguments, which are the input of the function. The parentheses must be present after the function name, even if there are no arguments. A function generally returns a value. This is called the output of the function. If there is no output of the function, then the word void is used to indicate that it does not give any output.

To use the library function in a program, certain specific information is to be included in the program. The information is generally stored in some special files, those are available with the software. Such file names are included the beginning of each program using the following preprocessor statement.

```
#include <filename>
```

where `filename` represents the actual name of a specific file. Several such files are available in any C compiler.

Some commonly used files are `stdio.h`, `math.h`, `stdlib.h`, `string.h`, etc. The file extension 'h' designated a 'header' file. Some frequently used library functions are given in Table 2.6.

Note 2.4.2 A straight forward translation of the expression $x^{\frac{2}{3}} - y^{-2}$ in C is `pow(x,2/3)-pow(y,-2)`.

But this expression is incorrect as in the term `pow(x,2/3)`, the division `2/3` is an integer division of 2 by 3. In integer expression evaluation the result can only be an integer. Thus 2/3 gives an integer value 0 (zero). The correct translation of the expression is `pow(x,2.0/3.0)-pow(y,-2)` or `pow(x,2.0/3.0)-1./(y*y)`

The following three header files with their functions are frequently used in several programs.

`stdio.h` (standard I/O header file)

Mathematical function	C equivalent	Mode of input	Mode of output
Power (a^b)	pow(a,b)	double,double	double
Power (10^a)	pow10(a)	double	double
Exponential (e^x)	exp(x)	double	double
Logarithm ($\log_e x$)	log(x)	double ($x > 0$)	double
($\log_{10} x$)	log10(x)	double ($x > 0$)	double
Square root ($\sqrt{x}$)	sqrt(x)	double ($x > 0$)	double
Sine ($\sin x$)	sin(x)	double (x in radian)	double
Cosine ($\cos x$)	cos(x)	double (x in radian)	double
Tangent ($\tan x$)	tan(x)	double (x in radian)	double
Hyperbolic sine ($\sinh x$)	sinh(x)	double	double
Hyperbolic cosine ($\cosh x$)	cosh(x)	double	double
Hyperbolic tangent ($\tanh x$)	tanh(x)	double	double
Arc sine ($\sin^{-1} x$)	asin(x)	double ($-1 \leq x \leq 1$)	double
Arc cosine ($\cos^{-1} x$)	acos(x)	double ($-1 \leq x \leq 1$)	double
Arc tangent ($\tan^{-1} x$)	atan(x)	double	double
Absolute value ($\|x\|$)	abs(x)	int	int
	fabs(x)	double	double

Table 2.6: Some library functions

Function name	Use
gets	Reading a string
puts	Printing a string
getchar	Reading a character
putchar	Printing a character
scanf	Reading multiple variables of different types
printf	Printing multiple variables of different types

string.h (string manipulation header file)

Function name	Use
strcat	Concatenation of two strings
strcmp	Compare two strings
strcpy	Copied a string to another string
strlen	Determines the length of the string

ctype.h (character type header file)

Function name	Use
toascii	Returns the ASCII code of a character
tolower	Converts a character into lower case character
toupper	Converts a character into upper case character

Example 2.4.2 Write the following expressions in C equivalent:

(i) $\frac{\alpha}{\sqrt{\alpha^2+\omega^2}}\cos(\omega t+\phi)+\sin^{-1}(a+b)$

(ii) $\log_e\sqrt{\frac{x}{yz}}+\log_{10}|x|$

(iii) $\frac{1-e^{-\alpha\sqrt{x}}}{1+xe^{-|x|}}+\eta\cos\theta$

(iv) $y=(a+b^2+ab)^5/\sqrt{a+b+1}$

(v) $x_{n+1}=\frac{1}{k}\left[x_n+\frac{a}{x_n^{k-1}}\right]$

Solution.

(i) `alpha* cos(omega*t+phi) / sqrt(alpha*alpha + omega*omega) + asin(a+b)`

(ii) `log(sqrt(x/(y*z))) + log10(fabs(x))`

(iii) `(1.0-exp(-alpha*sqrt(x)))/ (1.0+x*exp(-fabs(x)))+ eta* cos(theta)`

(iv) `y = pow((a+b*b+a*b),5)/sqrt(a+b+1.0)`

(v) `x = (x+a/pow(x,(k-1)))/k`

2.4.4 Unary Operators

C supports different types of unary operators. The negative operator (-) is the most common operator, which is used both as binary and unary operators. The other unary operators are +,++,- -, etc. In unary operation only one operand is required.

Generally, unary operators precede their operands, but some unary operators are written after their operands. C provides two new unary operators called increment (++) and decrement (- -) operators.

The increment operator, increases the value of the operand by 1, whereas decrement operator decreases the value of the operand by 1.

Suppose the value of the variables x and i are respectively 5.234 and 4. The expressions $++x$ and $++i$ are equivalent to $x=x+1$ and $i=i+1$. After execution of these expressions the values of x and i become 6.234 and 5

respectively. Similarly, if $x = 5.234$ and $i = 4$, then after execution of the expressions $--x$ and $--i$ the values of x and i change to 4.234 and 3 respectively.

The increment and decrement operators can also be used in another way. These operators may be used after the operand (postfix form). The use of the forms $++i$ and $i++$ have same effect. In both cases, the value of i will increase by 1. But, if we use these expressions in another way the prefix $(++i)$ and postfix $(i++)$ operators give different results. For example, we consider $j = ++i$ and $k = i++$, and the value of i is 5. The value of i in both cases is equal to 6. After execution of $j = ++i$, the values of j and i are 6 and 6. Again, if $i = 5$ then after execution of $k = i++$, the values of k and i are $k = 5$ and $i = 6$. In the expression $j = ++i$, the operator $++$ is applied on i first and then assign the incremented value of i to j. In this case, the values of i and j are same. Whereas in the expression $k = i++$, the value of i is assign to k first and then increment the value of i by 1. In this case, the value of k is 1 less than i.

Another unary operator available in C which is `sizeof`. This operator gives the size of its operand in bytes. The operand may be a variable or an expression or cast (the data type).

To illustrate the use of `sizeof` operator, the following declarations are considered.

```
short int i;
int j;
long int k;
float a;
double b;
long double c;
```

The values of the expression `sizeof i; sizeof j; sizeof k; sizeof a; sizeof b; sizeof c;` are 2, 2, 4, 4, 8, 10 respectively.

This operator is also used to find out the number of characters in a string. For example, we consider the following string declaration

```
char a[ ]="India";
```

The value of the variable `n` in the statement `n = sizeof a` is 5.

The statement `sizeof (int); sizeof (float); sizeof (char)` gives the output 2, 4, 1 respectively.

It is very important to note that the *precedence of unary operator is higher than the binary operators. The associativity of unary operator is right-to-left.*

There are some other unary operators available in C, which are discussed latter.

2.4.5 Relational and Logical Operators

Often we need the comparison of two arithmetic numbers or expressions. These type of operators which are not arithmetical are called relational operators. The relational operators and their C equivalent are given bellow.

Algebraic form	$a > b$	$a \geq b$	$a < b$	$a \leq b$	$a = b$	$a \neq b$
C form	a > b	a >= b	a <b	a <= b	a == b	a =! b

The operators < . <=,>,>= fall in a same precedence group whereas == and =! fall into another precedence group called **equality operators**. The associativity of these operators is left-to-right.

The precedence of relational operators is lower than unary and arithmetic (binary) operators. Again, the precedence of equality operators is lower than the relational operators. An expression containing these six operators, operands and their combination is called **relational expressions**. For example, $a > y$, $x >= 4.5$ are the relational expressions. The value of the relational expression is either *true* or *false*. The resulting expression is of type integer, since *true* and *false* are represented by integer value 1 and 0 respectively.

Examples:

Let c and d be logical variables.

(i) (5/2 > 3.5 * 2 - 7) (value of this expression is *true*)

(ii) !(a > b)

(iii) c = (a > b) <= (b >= 5.2)

Logical Operators

Two or more relational expressions can be combined by two or more operators *and* and *or*. Also, the negation of a relational expression is done by another operator called *not*. These operators are called **logical operators**. The symbols for logical *and*, *or* and *not* operators are &&, || and ! respectively.

Logical and (&&)

The syntax of this operator is

a && b

where a and b are two relational variables/expressions. The value of the entire expression c = a && b is obtained according to the rule given below.

a	b	logical value of c	integer value of c
true	true	true	1
true	false	false	0
false	true	false	0
false	false	false	0

Logical or (||)

The syntax of this operator is

```
a || b
```

where a and b are two relational variables/expressions. If both the values of a and b are false then the value of this expression is false, otherwise it is equal to true. The value of the expression c=a || b is obtained as per following rule.

a	b	logical value of c	integer value of c
true	true	true	1
true	false	true	1
false	true	true	1
false	false	false	0

Logical not (!)

If a is a logical variable, then the expression

```
!a
```

gives the opposite (negation) value of a. That is, if a is *true*, then !a is *false* and if a is *false* !a is *true*. It may be noted that the logical not operator is a unary operator.

The precedence of logical operators are !,&&, ||.

Let us consider the following logical expression

!((c =='x')||(p >= 5))&&((i < 5)&&(v! = 5)) where the values of p, i, v and c are respectively 6, 4, 5 and 'x'.

The above expression is evaluated as follows :

!(true||true)&&(true&&false) [since the priority of the relational operator is higher than logical operator]

= !true&&&false

= false&&&false [since priority of unary operator (!) is higher]

= false.

2.4.6 Assignment Operators

The assignment operator is essential in every program. This operator assigns a new value to a variable. Its general form is

```
variable =exp;
```

where variable is a variable and exp may be a constant, another variable or an expression to which a value has been assigned previously or a formula which can be evaluated by the computer.

Both variable and exp should be of same type, *i.e.*, both should be either numeric or character, etc. This statement assigns the value of exp to variable. The new value replaces the earlier value, if any, associated with variable.

Some arithmetic assignment statements are x=5.9, z=a, z=a+b*c.

In algebra, the equations $x = 2$ and $2 = x$ both are same, but in C, $2 = x$ is an invalid statement, because the value of x can not be stored in 2 as it is a constant quantity. Similarly, the assignment $b + c = a$ is also invalid because the value of a can not be stored in an expression.

It may be noted that the assignment operator = and the equality operator == are two different operators. The assignment operator, assigns a value to a variable, whereas the equality operator is used to check whether two expressions have the same value. The entire expression containing variable, assignment operator and righthand side expression is also known as assignment statement.

If two operators in an assignment statement are of different data types, then the value of the right hand side expression will automatically be converted to the type of the left hand variable. For example, if i is an integer variable and the assignment statement is $i = 4.5$; then 4 is assigned as the value of i.

The following points are to be noted during automatic type conversion.

- A floating point value may be truncated if it is assigned to an integer variable.
- A double precision value may be truncated if it is assigned to a floating point (single precision) variable.

Multiple assignments are also permissible in C. That is

```
a=b=x=z=2.5;
```

is a valid statement in C. For this statement the value of the variables a, b, x and z become 2.5.

The associativity of multiple assignment is right-to-left. This means, in above example, 2.5 is assigned to z first, then the value of z is assigned to x, then the value of x is assigned to b, and so on. One interesting situation is illustrated below.

Suppose i and a are integer and floating point variables respectively. The multiple assignment statement

```
i=a=3.4;
```

assigns the value 3.4 to a and the truncated value of a, which is 3 is assigned to i, i.e. $a = 3.400000$ and $i = 3$ are the final values of a and i, whereas the assignment statement a=i=3.4, assigns the truncated value 3.4, i.e. 3 to i and then the value of i is assigned to a, i.e. $i = 3$ and $a = 3.000000$.

It is very interesting that C supports five more assignment operators, viz., $+=, -=, *=, /=$ and $\%=$. These operators are basically shortened form of the operators +, −, *, / and %. The use of these operators are shown in Table 2.7.

Expression	Equivalent expression
`i+=6`	`i=i+6`
`i-=5`	`i=i-5`
`i*=4`	`i=i*4`
`i/=7`	`i=i/7`
`i%=8`	`i=i%8`

Table 2.7:

The precedence of these operators is
$=, +=, -=, *=, /=, \%=$ and the *associativity is right-to-left.* The precedence of these operators among all the operators discussed so far is lowest.

2.4.7 Conditional Operator

The only **ternary operator** (needs three operands) in C is ? :, combination of a question mark and a colon. The general syntax of this operator is

```
expression1 ? expression2 : expression3;
```

At the time of execution of this expression, the value of the `expression1` is evaluated first. If the value of `expression1` is true, then `expression2` is performed otherwise `expression3` is performed. The evaluated value is the final value of the conditional expression. It may be noted that exactly one expression, either `expression2` or `expression3` is performed during the execution of this statement.

Suppose i and j are two integer variables and $j = 5$. We consider the following conditional expression

```
k=(i > 5) ? j++ : j--
```

If the condition `i > 5` is true then j is assign to k and then j is incremented by 1, i.e. the value of k and j are respectively $k = 5$, $j = 6$. Otherwise, the value of j is assign to k ($k = 5$) and then j is decremented by 1.

Exercise 2.4

1. Write down the hierarchy of unary, arithmetic, relational, logical and assignment operators.

2. Write short notes on (i) assignment operator, (ii) logical operator, (iii) relational operators, (iv) ternary operator, (v) unary operators, (vi) library function.
3. What do you mean by (i) integer, (ii) real, and (iii) mixed mode expressions.
4. What are the differences between algebraic and C expressions.
5. Write the assignment statements in C for the following algebraic statements.

   (i) Assign the value of $\sqrt{x^2+3}$ to y. (ii) Assign the area of a triangle of base a and height h to AREA.
6. Express the following algebraic expressions into their equivalent C expressions.

   (a) area $= \frac{1}{2} ab \sin\theta$

   (b) $\frac{p^x}{q} + xy$

   (c) $\left(\frac{\sin x+1.2}{1234d}\right)^{3/2}$

   (d) $\frac{p+q}{x+y}$

   (e) $a + 2b - \frac{c}{1+\frac{x}{y}}$
7. Indicate whether the following statements are valid in C and if not, why?

   (i) v=a(b+c)-d/5+b/-3,

   (ii) r=-a/2+b*5-sqrt(-3)+pow(x,-2),

   (iii) x/y=z

   (iv) a = 4(b+pow(c,-.5))
8. Write the following relations as C relational/logical expressions:

   (a) $1 \le x < 2.5$, (b) $|t| + |y| > v$, (c) $a/b + c/d > i * n$, (d) $\sqrt{r} < 10$ or $\sqrt{k} + 5l > 10$, (e) $\sqrt{x} \log_e y < 1/2$ and $\cos y < 1/2$.
9. Find the values of i and a where

   (i) i=3/pow(2,2)*2+3/8*3+pow(3,2)/2, (ii) a=3e02/pow(4.0,0.5)* 2.0+ c/ 8.0*3.0+ pow(3.0,2)/2 (where c=0.8)
10. Find the value of n calculated for the following statement:

    (i) n=pow(j,2)+k*2/5-pow(j,3)/8+(j++) (if j=2, k=5)

    (ii) n=1/2.0+5/a+3*a-b (if a=4.0, b=2.5).

2.5 Input and Output Statements

Every program should take at least one input and must produce some outputs. To solve a problem, it is necessary to transfer the data from the input device to the memory of the computer. This task is performed by the input statement. There are several input devices available with a computer. In this section, we assumed that the data is entered to the computer through the standard input device, such as keyboard.

Using the output statement, the results of the computation are to be transferred from the memory of the computer to the output device.

Several input/output (I/O) statements are available in C language, viz., `getchar`, `putchar`, `scanf`, `printf`, etc. All the I/O functions are available in the header file `stdio.h`.

2.5.1 Reading a Character

The function `getchar` reads a single character from the standard input device followed by a carriage return. This function is a part of standard I/O library. Noted that this function does not take any argument, though a pair of parentheses must be given with the function.

The syntax of this function is

```
char_variable=getchar();
```

where `char_variable` is a pre-defined character type variable.

The function `getchar` reads only one character at a time, but by using repeatedly one can read a string of characters.

2.5.2 Output a Single Character

The `putchar` function is used to print a single character on the screen. This function is also a part of standard C I/O library. This function must need an argument enclosed within parentheses. The value of this argument will display on the screen.

The syntax of this function is

```
putchar(char_variable);
```

where `char_variable` is some pre-defined character variable.

The function `putchar('\n')` will print a carriage return, i.e. cursor move to the beginning of the next line.

Let us consider the following statements

```
char ch;
ch=getchar();
putchar(ch);
```

The first statement declares ch as a character variable. The second statement reads a character from the keyboard and store it to the variable ch. The last statement will print the character ch on the monitor screen.

2.5.3 Input/Output of String

The function getchar and putchar are used to read and print a single character. The functions gets and puts are used to read and print a string of characters including whitespace, tab, etc. The general form of gets function is

```
gets(string_name);
```

and that of puts is

```
puts(string_name);
```

where string_name is a variable of type string (an array of characters). This function reads all characters from keyboard until a newline character is encountered. At end, the function gets adds a null character ('\0') to the end of the string. Note that the newline character is not added to the string.

The puts function will display the string string_name on the screen as it is stored in the memory.

Note that the use of getchar and gets functions. The function getchar does not take any arguments, it reads a character and returns it a character, which can be assigned to a variable. The function gets takes a parameter (of type string), reads a string and assigned to the parameter.

Let us consider the following example.

```
#include<stdio.h>
void main()
  {
    char name[30];
    gets(name);
    puts(name);
  }
```

The input/output of this program is shown below.

```
Netaji Subhas Chandra Bose
Netaji Subhas Chandra Bose
```

2.5.4 The scanf function

The most powerful input statement of C language is scanf. This function is used to read any types of data, viz. integer, character, float, string, etc.

The syntax of scanf function is

```
scanf("control string", arg1, arg2, ..., argn);
```

where the control string specifies a string containing some formatting information and arg1, arg2, . . ., argn represent the address of locations of individual data items. It may be noted that the control string and arguments are separated by commas.

The control string is a complicated group of characters. For each argument there is one group of characters of the form

```
%wp
```

It has three parts, the conversion character %, data type (or type specifier) character p and w, represents the width of the input value, which is optional.

The character groups in control sting may be contiguous or they are separated by blank spaces, tabs, newline characters or any specified characters. If these characters are used in the control string, they are read by the computer but ignored. The commonly used type specifiers are given in Table 2.8.

Format specifier	Meaning
%c	Read a single character
%d or %D	Read a decimal integer
%f	Read a floating-point number
%e or %E	Read a floating-point number of exponent form
%g or %G	Read a floating-point number
%h	Read a short integer
%i or %I	Read a decimal, hexadecimal or octal integer
%o or %O	Read an octal integer
%s	Read a string of characters
%u or %U	Read an unsigned decimal integer
%x or %X	Read a hexadecimal integer
%[...]	Read a string of characters or words

Table 2.8: Format specifiers for input

It is very important to note that the variables or arrays should match with the corresponding character group in control string. Each character group must

be begins with the symbol %. Each argument is the combination of ampersand (&) sign and the variable name. That is, the arguments are basically pointers which represent the addresses of the computer where the values of the variables are stored.

Suppose a,b and x are integer, floating point and character variables. Then the values of these variables can be read using the following input statement.

```
scanf("%d%f%c",&a,&b,&x);
```

In this statement, %d%f%c is the control string or called format specifiers, it contains three character groups. The first group %d indicates that a variable is to be read which is an decimal integer, the second character group %f indicates the type of the variable must be floating-point and the third group %c indicates that this format will read a character.

It may be noted that the data items must corresponding to the arguments in the scanf function in number, in type and in order with the control string. The values to be assigned to the variables (for the above example they are a, b and x) are typed on succession in a line starting from any position. The leading blank spaces are skipped. The values are typed in the same order as in the list and they are separated by blank spaces (other non-standard separators are also available in C, those are discussed latter). If some values are not typed on a line, then they may be typed on the next line. Note that *a single value should not be typed on two or more lines.*

If the values of the variables a,b and x are respectively 58, 67.253 and M, and the statement being

```
scanf("%d%f%c",&a,&b,&x);
```

then they should be typed as

```
58 67.253 M
```

If the values of all the variables are not found on a single line then the lines following it are read until values are assigned to all the variables.

The scanf function is a versatile input function. It contains several facilities using which we can specify the rules to read the data. The scanf function, during its execution, scans a series of input fields, one character at a time, then converts each field according to the format specifier, which is supplied within control string. The format specifiers direct the scanf function to read and convert characters from the input field into specific types of values. This converted input is stored at an address passed as an argument (arg1, arg2, . . ., argn). It may be remembered that, there must be one format specifier and address for each input field.

Some invalid input statements

Here we assumed that a.b are floating point variables. i,j,k are integer variables and p is a character variable.

`scanf("%d%d",&i,&j)`	(; missing at the end)
`scanf("%d",&(i+j));`	(expression is not allowed)
`scanf("%d%d%d",&i,&j,5);`	(constant 5 is not allowed)
`scanf("%d%d%d",i,j,k);`	(missing address operators (&))
`scanf(%d%f%f,&i,&a,&b);`	(quotations missing in control string)
`scanf('%d%f%d',&i,&a,&j);`	(control string must be enclosed by double quotation)
`scanf("%d%d%f,&i,&j,&a");`	(arguments must not be written within control string)
`scanf("%d%d\n",&i,&j);`	(white space is not allowed)
`scanf("A=%fB=%f",&a,&b);`	(only format specifier are allowed within control string)

2.5.5 The printf function

The most powerful output statement in C is printf function. This function is used to print any types of data, such as numeric values single character and strings.

The printf function will display the values of the variable to the standard output device such as monitor screen.

The general form of printf function is

```
printf("control string",arg1, arg2, . . ., argn);
```

where control string is a string that contains formatting information and arg1, arg2, . . ., argn are n variables, constants, array names or expressions. The function calls may also be allowed. It may be noted that the arguments of printf function do not represent memory addresses and hence no ampersand signs are required.

These three types of items may not be present together within a control string, even arguments may be omitted. In the simplest form of control string contains only format specifiers, i.e. its consists of individual groups of characters, with one character group for each argument. Each character group begins with % (similar to scanf function).

Let us consider the following example. Suppose $a = 58$ and $b = 25.652$ and the output statement is

```
printf("a=%d  b=%f\n",a,b);
```

Here %d and %f are format specifier a= and b= are the other characters, they will display as it is and '\n' is the escape sequence.

The output of this statement is

```
a=58  b=25.652000
```

Here the characters a= and b= are printed as they are appear within the control string. The value of a is printed in place of %d. One blank space is inserted in between %d and b within the control string, so one blank space is printed after the value of a. The value of b (=25.652) is printed as 25.652000, because the default precision is 6 (every floating point variable will print its decimal part with 6 decimal places, if it is not mentioned).

After printing this output the cursor move to the first column of the next line, due to the escape sequence '\n'.

The commonly used format specifiers for printf are listed in Table 2.9.

Format specifier	Meaning
%c	Print a single character
%d or %D	Print a decimal integer
%f	Print a floating-point number without exponent
%e or %E	Print a floating-point number of exponent form
%g or %G	Print a floating-point number either e-format or f-format conversion, depending on the magnitude of the value
%i or %I	Print a signed decimal integer
%o or %O	Print an octal integer, without leading zero
%s	Print a string of characters
%u or %U	Print an unsigned decimal integer
%x or %X	Print a hexadecimal integer, without leading 0x

Table 2.9: Format specifiers for output

To print the short integer, long integer and long double the character conversion characters h, l and L are used with the above conversion characters.

Note 2.5.1 The infinite floating point members are printed as +INF and -INF.

Now, from the ongoing discussions we have sufficient resources to write a complete program in C.

2.5.6 Complete Programs

A C program (the main functions) has four parts, viz. inclusion of header files, declaration of main function, declaration of variables and body of the program/main function. The following example illustrates these parts.

Example 2.5.1 Write a program to find Fahrenheit temperature from the corresponding Centigrade temperature.

Solution. The relation between Fahrenheit (f) and Centigrade (c) temperature is

$$\frac{c}{5} = \frac{f-32}{9} \quad \text{or} \quad f = \frac{9c+5\times 32}{5}$$

Inclusion of header file

```
#include<stdio.h>
```

beginning of main

```
void main()
```

beginning of program/main

```
{
```

declaration of variables

```
int c,f;
```

body of the program /main

```
printf("Enter the value of C");
scanf("%f",c);
f=(9*c+160.0)/5.0;
printf("c=%f  f=%f",c,f);
```

end of the program /main

```
}
```

A sample of input/output:

```
39
c=39.000000  f=102.200000
```

2.5.7 Format Specifiers for Input

The very simplest case of scanf function is discussed in previous section. Now, details and variations of different format specifications are discussed here.

%c format specifier

This format specification reads a single character from keyboard. If the control string contains more than one character groups, without white space characters, then c-type conversion gives different results.

To explain this, let us consider the following program segment.

```
char a, b, c;
scanf("%c %c %c",&a,&b,&c);
```

If the input is

```
x y z
```

then the value of x, y and z are assigned as $a = x$, $b = y$ and $c = z$.

In this case, the blank spaces are used as separator between the characters.

For the scanf statement

```
scanf("%c%c%c",&a,&b,&c);
```

with the same input string, the values are assigned as $a = x$, $b =$ ␣, $c = y$.

The blank space between x and y in the input data is taken as the value of b. Since no blank separators are used within the control string, the consecutive characters are taken as input characters.

The other form of %c is %wc, where w indicates the length of the input string. That is, using this specifier a string of characters can be read at most w characters including blank space. The reading will terminate when a newline character is encounter.

Let us consider the statements

```
char a,b,c,p[10];
scanf("%c%c%c%10c",&a,&b,&c,p);
```

and the input string

```
xyzMy India
```

The value of the variables are assigned as a=x, b=y, c=z and p=My India

%s format specifier

This specifier is used to read a string. The general form of this specifier is

```
%ws
```

where w represents the length of the string. But, this specification *terminates reading at the encounter of a blank space* or *newline*, whereas %wc will not terminates when a blank space is encountered.

Let us consider the following example.

```
char s1[20],s2[20];
scanf("%20s",s1);
scanf("%20s",s2);
```

The input strings are

```
My India is great
My_India_is_great
```

The value stored at s1 and s2 are

s1=My and s2=My_India_is_great.

The first scanf terminates at the first blank space (after My). Thus only My is stored to s1. In the second string no blank spaces are included, so entire string is assigned to s2.

%d format specifier

This format specification is used to read a decimal integer data. The general form of this specification is

%wd

where w represents the field width of the data item. Using this specification one can read an integer data at most w digits. The data item may contain fewer than w digits, but it cannot exceed w digits.

If any white space character is encounter in the input, then terminates the reading for that variable, even its length is less than w.

Let us consider the following statements

```
int i,j,k;
scanf("%4d %3d %2d",&i,&j,&k);
```

with the input data

123 45 31

The values of i, j and k are $i = 123$, $j = 45$ and $k = 31$

Note that the reading of values of i and j are terminates at the blank spaces. These blank spaces are used as separators of the data item.

For the input statement

```
scanf("%4d%3d%2d",&i,&j,&k);
```

with input data

12456783456

the values of i, j and k are stored as $i = 1245$, $j = 678$ and $k = 34$.

Note that the last two digits 5 and 6 are ignored.

In C, the standard separator is blank spaces. But, other separators may also be used. These separators must be specified within the control string.

Let us consider the following example.

```
int i,j,k;
scanf("%d,%d,%d",&i,&j,&k);
```

Suppose the input data is

```
123,45,875
```

then i, j, k are stored as $i = 123$, $j = 45$ and $k = 875$.

When separators are used in control string, then the input data must be separated by the same separator, otherwise incorrect assignment will encountered.

%f format specifier

This format specification is used to read floating point data. Its general form is

```
%wf
```

where w represents the width of the data (including sign and decimal point). The w characters will read from input data and it will convert as floating point data before storing it in the memory.

For example, the input statement

```
float a,b,c;
scanf("%5f%3f%4f",&a,&b,&c);
```

reads the values of a, b, c from the input data

```
12.345.678.9
```

as $a = 12.34$, $b = 5.6$, $c = 78.9$.

If the input data is

```
123458903
```

then $a = 12345$, $b = 890$, $c = 3$

and if it is

```
12345890357837
```

then $a = 12345$, $b = 890$, $c = 3578$ and the digits 3 and 7 are ignored.

If the input data is

```
2.3e45.678.3
```

then $a = 2.3e4 = 2.3 \times 10^4$, $b = 5.6$ and $c = 78.3$.

Note that the constant 2.3e4 is written using e specification, though it is entered using f specification.

The double type data, can be read using %lf specification instead of %f specification.

For example, the statement

```
long int i;
double x;
scanf("%ld %lf",&i,&x);
```

will read the long int and double variables i and x.

The format specifiers %e, %E, %g and %G all are used to read floating point values. These format specially significant in case of output statement.

Reading mixed data

It is mentioned earlier that using scanf function one can read mixed mode data, i.e. using a single scanf function different type of data can be read. A careshould be taken to specify format specification and order of variable names. If the, value of a variable does not match with its expected format specification (which is supplied through the control string), the scanf function does not read further.

The statements

```
int i;
char name[15];
scanf("%3d%15c",&i,name);
```

will read the data

```
125Aniket Pal
```

correctly as i=125 and name=Aniket Pal

%[...] and %[^...] format specifiers

C provides a very flexible method to read a string. The format specification %[...] is used to read a string which is defined in a particular domain. Suppose, we would like to read a string which contains only lower case alphabets, then the appropriate input statement is

```
scanf("%[a-z]",s);
```

The reading will terminate at the first occurrence of the other characters (other than lower case alphabets mentioned as a-z).

If the string

```
my india
```

is enter as input for the above input statement, then only my will store at s, since the blank space is not included within the format specification %[a-z]. If we enter the string My India then no character will store to s, because the first character (M) is an upper case alphabet, which is not included within the format specification. To read lower and upper case alphabets, digits and blank space, the following specification is required.

```
%[a-zA-Z0-9' ']
```

This specification is also use to read a specific group of characters. For example,

```
char ch;
printf("Do you want to continue (Y/N)?");
scanf("%[YN]",ch);
```

In this program segment, if the input data is either 'Y' or 'N', then ch will store one of them, otherwise, no character will store against ch.

The other format specification is %[^...].

One or more character may be written after circumflex (i.e. ^). When the program is executed, successive characters will read from the keyboard as long as each input character does not match one of the characters written within the square brackets.

Thus the specification [^...] works opposite to [...] specification. Also, using this specification one can read multiline input and input will terminate when an input character match with any one of the characters written in [^...] If the format [^\n...] is used to read a string, it will read all the characters until a newline character is encounter.

The statements

```
char name[30];
int roll;
scanf("%3d%[^\n]",&roll,name);
```

will read the input

```
250A. K. Sarma
```

correctly as roll=250 and name=A. K. Sarma.

The octal and hexadecimal numbers are read using %o and %x format specifiers. Every octal and hexadecimal number begins with 0 (zero) and 0x respectively. These prefixes indicate that the integers are either octal or hexadecimal. An octal and hexadecimal number can be printed directly in other types of numbers, such as decimal, hexadecimal and octal.

2.5.8 Format Specifiers for Output

All most all format specifiers are used to print an output. But, output format specifiers are more useful rather than input format specifiers.

%d format specifier

This specifier is used to pass the values of integer variables from the computer to the output field. The general form is

%wd

where w is the width of the integer number including sign. If the number of digits of an integer is more than w, then the full integer will print out, overridden the specification. If the number of digits are less than w, then blank spaces are to be added at the beginning to make w width. The leading blank spaces may also be filled up by 0. Normally, the numeric numbers are printed in right justified form. But, left justification can also be possible by using flags.

The use of %d is illustrated in the following examples. Suppose $i = 185$.

Print statement	Output	Remark
`printf("%d",i)`	185	
`printf("%5d",i)`	␣␣185	right justified
`printf("%2d",i)`	185	overridden format specifier
`printf("%-5d",i)`	185␣␣	force to left justified
`printf("%05d",i)`	00185	leading blanks filled by 0
`printf("%+5d",i)`	␣+185	+ flag is used
`printf("%+5d",-i)`	␣-185	+ flag is used

In the above examples three additional characters minus (-), plus (+) and zero (0) have been used, they are known as flags.

%w.pf format specifier

This format specifier is used to print floating point number without exponent. If the number is very large this specification cannot print the number correctly. The general form of this format is

%w.pf

where w is the filed width indicating the total number of characters in the field including sign and decimal point, and p is the number of digits to the right of decimal point called precision.

It should be noted that the width (w) must exceed the number of decimal places by at least three, to make room for a sign and for at least one digit to the left of the decimal point. The output is right justified. p digits will be printed to the right of the decimal point. If the number of digits in the fractional part is less than p, then the remaining positions to the right are filled will zeros. If the fractional part has more than p-digits it will be rounded off. Considering the fractional part in this manner, if we see that the format specification `%w.pf` cannot provide enough number of columns for the number then the full number will be printed, overridden the format specification. The default precision is 6. The leading blanks may be replaced by 0 and printing with left justification is also possible by using minus (-) flag. To demonstrate the feature of `%w.pf` specification, we assume `a=167.2358` and `b=-785.3854`.

Print statement	Output	Remark
`printf("%8.4f",a)`	1 6 7 . 2 3 5 8	
`printf("%8.2f",a)`	1 6 7 . 2 4	right justified
`printf("%-8.2f",a)`	1 6 7 . 2 4	left justified
`printf("%4.4f",a)`	1 6 7 . 2 3 5 8	overridden width
`printf("%08.2f",a)`	0 0 1 6 7 . 2 4	filled by 0
`printf("%6.1f",b)`	- 7 8 5 . 4	
`printf("%8.0f",a)`	1 6 7	decimal part vanish
`printf("%0.3f",a)`	1 6 7 . 2 3 6	note that w is 0

`%e` or `%E` format specifier

This format specification is used to print a large number like 6.23×1023. This type of number is called floating point number with exponent form. The number 6.23×1023 represented in computer as $6.23E + 23$. It has two parts: the left hand side of E called mantissa and the right hand side of E is called exponent. The mantissa, in general is a floating point constant and exponent is an integer constant. The general form of this format is

`%w.pe` or `%w.pE`

where w is the field width that indicates the total number of characters in the field, indicating sign, decimal point and the exponent symbol. p represents the number of digits after the decimal point in the mantissa.

These two format are same. If $e(E)$ is used in the format then e (respectively E) will print in the output.

During printing, p-significant digits are printed to the right of the decimal point and if the number contains more than p-digits, then the fraction will be

rounded. It should be noted that width w must exceeds the number of decimal places p by at least seven to make room for a sign, at least one digit to the left of the decimal point, the decimal point, the exponent symbol, sign of exponent and the two digits exponent.

Let $a = 6.2346 \times 10^{30} = 6.2346E + 30$.

Print statement	Output
`printf("%10.4e",a)`	6 . 2 3 4 6 e + 3 0
`printf("%10.4E",a)`	6 . 2 3 4 6 E + 3 0
`printf("%E",a)`	6 . 2 3 4 6 0 0 E + 3 0
`printf("%10.2e",a)`	6 . 2 3 e + 3 0
`printf("%-10.2e",a)`	6 . 2 3 e + 3 0
`printf("%010.0e",a)`	0 0 0 0 0 6 e + 3 0
`printf("a=%8.2e",a)`	a = 6 . 2 3 e + 3 0

%c format specifier

This format is used to print a single character. The general form is

`%wc`

where w represents the width of the display field. The character will be displayed as right justified. The left justification can be done by using minus flag.

%s format specifier

The general form of this format specification is

`%w.ds`

This format will print first d characters of the string in a display field of length w characters. The default justification is right. The minus (-) flag is used to print left justified data.

Here p is optional. If p is not present in the specifier then the whole string will print within w field width. If w is less than the length of the entire string, then also full string will display as left justified form.

To explain the different forms of this format, let us consider the string "Netaji S O University".

Print statement	Output
%s	Netaji SO University
%21s	Netaji SO University
%25s	Netaji SO University
%25.8s	Netaji S
%.10s	Netaji SO
%10s	Netaji SO University

A list of flags commonly used in output statements is given in Table 2.10.

Flag	Meaning
	Output is left justified. the leading field will be filled by blank spaces, if there be any.
+	The sign (+ or -) of the number must be printed.
0	The leading blank will be filled by 0.
# (o or x)	The octal and hexadecimal numbers will print with the prefixes 0 and 0x respectively.
# (e, f or g)	Decimal point must be place to print a floating point number, even if it is whole number. Also, prevents the truncation of trailing zeros in g format.

Table 2.10: Flags used in print statement

We have discussed how different types of values can be printed on the output device. The output can also be printed with more readable form using appropriate number of blank spaces, newline (`'\n'`) and tab (`'\t'`) characters. Also, some string of characters may be included within control string to clear understanding of the output.

The following example is consider to illustrate the use of other features of printf function.

Example 2.5.2 Write a program to find the surface area and volume of a rectangular parallelepiped the length of whose sides are a, b and c.

Solution. The surface area and volume of a rectangular parallelepiped are respectively $2(ab + bc + ca)$ and abc.

```
/* Volume and surface area of a rectangular parallelepiped */
#include<stdio.h>
main()
{
  float a,b,c;
  float surface,vol;
  printf("Enter three sides\n");
  scanf("%f%f%f",&a,&b,&c);
  surface=2.*(a*b+b*c+c*a);
  vol=a*b*c;
  printf("Surface area is %8.4f and\nVolume is %8.4f", surface,vol);
}
```

A sample of input/output:

```
Enter three sides
2.3  4.5  7.6
  Surface area is 124.0600. and
Volume is 78.6600
```

2.5.9 Worked Out Examples

Example 2.5.3 Indicate the printed form of the output of the following statement.

```
printf("%8.2f\n\n\n%10.2e%10.2E\n%8.4f\n%4d",x,y,z,p,i);
```

where $x = 23.2$, $y = 10.02$, $z = -0.01$, $p = 0.81$, $i = -50$.

Solution. The output is

```
Col. no.     1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
Line no. 1         2 3 . 2 0
Line no. 2
Line no. 3
Line no. 4       1 . 0 0 e + 0 1   - 1 . 0 0 E - 0 2
Line no. 5       0 . 8 1 0 0
Line no. 6     - 5 0
```

Example 2.5.4 Write a program to find the area and circumference of a circle whose radius is r.

Solution. The area and circumference of a circle whose radius r are πr^2 and πr.

```
/* Program to find the area and circumference of a circle */
#include<stdio.h>
main()
{
   float r,area,peri;
   scanf("%f",&r);
   area=3.14159*r*r;
   peri=2.0*3.14159*r;
   printf("The area is %f\n",area);
   printf("The circumference is %f\n",peri);
}
```

A sample of input/output:

```
2.5
The area is 36.316784
The circumference is 21.362812
```

Look at the first line. This line begins with '/*' and ends with '*/', which contains some information. The entire line is called the **comment**. The comment lines are used to identify or explain the various parts of a large program. A comment may be written in any place and any number of lines within the program, but it must begins with '/*' and ends with '*/'. The comment lines are not compiled to produce object code. Thus, we can write any thing as comment. It is optional.

Example 2.5.5 Write a program to find the area of a triangle the coordinates of whose vertices are (x_1, y_1), (x_2, y_2), (x_3, y_3).

Solution. The area of a triangle is given as $\frac{1}{2}[x_1(y_2-y_3)+x_2(y_3-y_1)+x_3(y_1-y_2)]$.

```
/* The area of a triangle whose vertices are given */
#include<stdio.h>
main()
{
   float x1,y1,x2,y2,x3,y3;
   float area;
   printf("Enter three coordinates\n");
   scanf("%f%f%f%f%f%f",&x1,&y1,&x2,&y2,&x3,&y3);
   area=(x1*(y2-y3)+x2*(y3-y1)+x3*(y1-y2))/2.0;
   printf("The area is %8.3f",area);
}
```

A sample of input/output:

```
Enter three coordinates
2,3,4,5,6,7,3,2,1,2,6,8
The area is 4.345
```

Example 2.5.6 Assuming $a_1b_2 \neq a_2b_1$, the solution of the linear equations $a_1x + b_1y = c_1$, $a_2x + b_2y = c_2$ is given by

$$x = \frac{b_2c_1 - b_1c_2}{a_1b_2 - a_2b_1}, \quad y = \frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}$$

Write a program to read the values of a_1, b_1, c_1 and a_2, b_2, c_2 and to print the values of x and y.

```
/* Solution of a pair of equations containing two variables */
#include<stdio.h>
main()
{
  float a1,b1,c1,a2,b2,c2,x,y;
  printf("Enter the coefficients of first equation\n");
  scanf("%f%f%f",&a1,&b1,&c1);
  printf("Enter the coefficients of second equation\n");
  scanf("%f%f%f",&a2,&b2,&c2);
  x=(b2*c1-b1*c2)/(a1*b2-a2*b1);
  y=(a1*c2-a2*c1)/(a1*b2-a2*b1);
  printf("The solution is x=%6.3f, y=%6.3f",x,y);
}
```

A sample of input/output:

```
Enter the coefficients of first equation
2 3 4
Enter the coefficients of second equation
4 3 2
The solution is x=-1.000, y= 2.000
```

Example 2.5.7 Write a program to read the Basic Pay (BP) of an employee and calculate his/her D.A. (DA), H.R.A. (HRA), Tax, Gross Pay (GP) and Net Pay (NP) and print them, where DA = 43% of BP, HRA = 15% of BP, Tax = 20% of BP, GP = BP + DA + HRA, NP = GP - Tax.

```
/* Program to calculate payment of an employee */
#include<stdio.h>
main()
{
   float  np,bp,da,hra,tax,gp;
   printf("Enter  basic  pay\n");
   scanf("%f",&bp);
   da=bp*0.43;
   hra=bp*0.15;
   tax=bp*0.20;
   gp=bp+da+hra;
   np=gp-tax;
   printf("Basic  pay=%6.0f\nD.A.=%6.0f\nH.R.A.=%8.2f\n",bp,
                                                        da,hra);
   printf("Tax  deducted=%8.2f\nGross  Pay=%6.0f\nNet  Pay=%6.0f",
                                                      tax,gp,np);
}
```

A sample of input/output:

```
Enter basic pay 50890
Basic pay= 50890
D.A.= 21883
H.R.A.= 7633.50
Tax deducted=10178.00
Gross Pay= 80406
Net Pay=70228
```

Exercise 2.5

1. Write short notes on scanf and printf functions.
2. Explain the format specifiers c,d,e,f,g,o,s,x and [...], [^...].
3. Write C statements to perform the following:
   (i) Read the values for X, Y, Z from keyboard.
   (ii) Read the values of A, B, C from first line and M, N from second line.

4. If the input line is

20.5 10

then determine the output of the following program segments

(i)
```
scanf("%f%d",&a,&m);
printf("M= %d A=%f",m,a);
n=m*m+2;
x=a*a;
printf("%f %d",x,n);
```

(ii)
```
scanf("%f%d",&a, &m);
printf("%f %d %f",a,m,a+m);
```

5. Identify the errors, if any, in the following programs.

(i)
```
int b; float a;
scanf("%f%d",a,b);
printf("%f %f %f",A,B,A+B);
printf("%f",a*n);
```

(ii)
```
int a, b,c
scanf(a);
b=a*2;
printf(%d %d %d,a,b,c);
```

(iii)
```
float x,y;
scanf("%f",&x);
y/4=x*x+5;
printf("%f %d",x,y);
```

(iv)
```
float a,b;
scanf("%f %d",&a,5);
b=a+5;
printf("a= b=",a,b);
```

6. Write a program to find the
   (a) area, perimeter and diagonal of a rectangle whose two adjacent sides are a and b.
   (b) area and diagonal of a square whose each side is a.
   (c) area of a triangle whose two sides are a, b and the angle between them is θ (area $= \frac{1}{2} ab \sin\theta$).
7. Write the programs to find
   (a) the volume and surface area of a sphere of radius r. [vol.$= \frac{4}{3}\pi r^3$, surface area $= 4\pi r^2$]
   (b) the volume and surface area of a right circular cylinder of radius r and height h. [vol.$= \pi r^2 h$ and surface area$= 2\pi r h$]
   (c) the volume of a right circular cone of height h and radius of the base is r. [vol.$= \frac{1}{3}\pi r^3 h$].
8. Write a program to read two vectors $\bar{a} = (a_1, a_2, a_3)$ and $\bar{b} = (b_1, b_2, b_3)$.
   (a) Find the scalar product $(\bar{a} . \bar{b})$ and vector product $(\bar{a} \times \bar{b})$. [$\bar{a} . \bar{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$, $\bar{a} \times \bar{b} = (a_2 b_3 - a_3 b_2, b_1 a_3 - a_1 b_3, a_1 b_2 - a_2 b_1)$].
9. (i) A person deposits Rs. P in a bank. The rate of interest (simple) is @Rs. R% per annum. Write a program to determine the total interest (I) and amount (A) at the end of T years. [I=PRT/100, A=P+I] (ii) A person deposits Rs. P in a bank. The bank gives compound interest @Rs. r% compounded n times in a year. Write a program to determine the amount (A) at the end of t years. [$A = P(1 + \frac{r}{100n})^{nt}$].
10. Write a program to read three numbers a, b, c, compute their sum and average and print them.

2.6 Control Statements

Normally, the statements in a C program are executed in the order in which they appear in the program and this is known as normal flow of control. But to solve a problem sometime the sequence of execution of statements are changed under certain conditions. To obtain the solution to a problem decisions, repetitions, branching etc. are sometimes performed. The statements which control the sequence of execution of statements are called **control statements**. C supports several control statements such as `goto`, `while`, `do-while`, `switch`, `for`, etc.

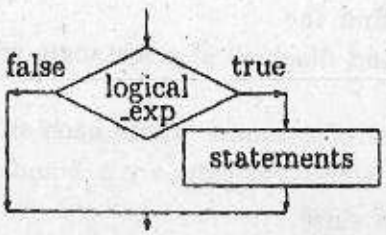


Figure 2.2: Flowchart of if statement

2.6.1 The `if` Statement

This statement is very powerful and essential in any programming language and this is of four types, (i) `if`, (ii) block `if`, (iii) nested `if`, and (iv) multi-alternative `if`.

The simple `if` Statement

Its general form is

```
if (logical expression)
  {
    statements;
  }
```

where `logical_expression` is a logical expression and `statements` are executable C statements. This statement is also known as logical `if` statement.

If the logical expression `logical_expression` is true then the `statements` are executed and after that the next statement will be executed. But, if the expression is false (value 0) then the statements are skipped and the statement just after the `if` statement is executed. The flow chart of the `if` statement is shown in Fig. 2.2.

If the number of statement in `statements` is one then there is no need of opening ({) and closing (}) braces.

Example 2.6.1 The value of y is given by

$$y = \begin{cases} 2x + 3.5 & \text{if } x \leq 2.5 \\ 2x^2 - 1.5 & \text{if } x > 2.5 \end{cases}$$

The if statements to determine y for a given x are

```
if(x <= 2.5) y=2.0*x+3.5;
if(x > 2.5) y=2.0*x*x-1.5;
```

Now, if the problem is to find the value of y and print it for a given x, then the above logical statements are changed to

```
if(x <= 2.5)
   {
     y=2.0*x+3.5;
     printf("y=%f",y);
   }
if(x > 2.5)
   {
     y=2.0*x*x-1.5;
     printf("y=%f",y);
   }
```

Here two logical if statements are used to determine the value of y, but it can be done by using only one if statement called block if statement discussed in the following.

Block if statement

The general form of this statement is

```
if(logical_expression)
   {
     statement_1;
   }
else
   {
   statement_2;
   }
```

where statement_1 and statement_2 are any valid C statements. This statement is also known as if_else statement. The statements statement_1 and statement_2 are called *if block* and *else block* respectively. If the value of logical_expression is true, then statement_1 will be executed and else block will be skipped. If logical_expression is false, then if block will be skipped and statement_2 of the else block will be executed. If there is no statements in the else block then the keyword else is not required and in this case block

if statement becomes a simple if statement. If there is only one statement in any block then the braces are not required.

Example 2.6.2 The block if statement for the function

$$y = \begin{cases} 2x + 3.5, & \text{if } x \le 2.5 \\ 2x^2 - 1.5, & \text{if } x > 2.5 \end{cases}$$

is written below.

```
if(x <= 2.5)
  {
    y=2*x+3.5;
  }
else
  {
    y=2*x*x-1.5;
  }
```

This statement can also be written as

```
if(x <= 2.5)
  y=2*x+3.5;
else
  y=2*x*x-1.5;
```

Nested if statement

In block if statement, the statements statement_1 and statement_2 can be some other block if statement or simple if statement. If there is any block if statement within the initial block if statement in if block or in else block or in both the blocks, then the resulting structure is called a nested block if statement or structure.

Multi-alternative if statement

In block if statement there are only two alternatives, one is if block (to be executed when the logical_expression is true) and other is else block (to be executed when logical_expression is false). Thus, the function $f(x)$ defined by

$$f(x) = \begin{cases} x^2 + 5, & \text{if } x < 0 \\ 2x, & \text{if } 0 \le x < 1 \\ x - 1, & \text{if } x \ge 1 \end{cases}$$

can not easily be implemented using block if statement. A modification of this block if statement is necessary for the implementation of this type of function. This modified if statement is known as **multi-alternative if statement** and is of the form

```
if(expl1) {
     S1
  }
else if(expl2) {
     S2
  }
else if(expl3) {
     S3
  }
.....
else if(expln-1) {
     Sn-1
  }
else
  {
     Sn
  }
```

where $expl_1$, $expl_2$, ... , $expl_{n-1}$ are logical expressions and S_1, S_2, ..., S_n are blocks of C statements.

Here all else if are part of if block. The above function $f(x)$ is implemented using multi-alternative if statement as follows:

```
if(x<0.0)
   fx=x*x+5.;
else if((0.0 ≤ x) && (x<1.0))
   fx=2.0*x;
else
   fx=x-1.0;
```

2.6.2 The goto Statement

The goto statement is used for jumping from one position within the function to other place in the same function. The general form of this statement is

```
goto label;
```

where label is an valid C identifier and it is the statement label of an executable statement (called target statement) in the same function to which the control is transferred and from that label normal sequential execution continues.

The general form of target statement is

```
label : statement;
```

Note that a colon must be given after label. A label can be used against only one statement, i.e. no two statements can have the same label.

Most of the programming languages support goto statement, though the use of this statement is discouraging. The goto statement mainly used in:

2.6.3 The while Statement

Using if statement, repeated execution of a group of statements can be performed. Another useful statement which used to perform such type of execution is while statement. It is a structured statement. The repetition of the loop is defined only on a logical expression. The general form is

```
while (logical_expression)
  {
statement(s);
  {
```

where logical_expression is a logical expression. If the logical expression logical_expression is true then the statements between braces will execute.

The loop will continue when logical_expression is true. If logical_expression is false then the control goes to the next statement of the while loop.

Example 2.6.3 Write a program to evaluate x^n, where n is a positive integer and x is any real number.

```
#include<stdio.h>
main()
  {
    int n,i=1;
    float prod=1.0,x;
    scanf("%f%d",&x,&n);
    while(i<=n)
```

```
    {
      prod*=x;
      i++;
    }
  printf("x=%f  n=%d  Value=%f\n",x,n,prod);
}
```

A sample of input/output:

2 10

```
x=2.000000  n=10  Value=1024.000000
```

Example 2.6.4 Write a program to test whether a given integer is a palindrome or not, by reversing the digits of the number.

Solution. An integer is said to be a palindrome if it is same with its reverse integer, *i.e.*, first digit is equal to last digit, second digit equals to last but one digit and so on. For example, the integer 1234321 is a palindrome but 1234561 is not a palindrome. The following program tests whether an integer is a palindrome or not and also prints its reverse integer.

```
#include<stdio.h>
main()
  {
    long int n,m,revers;
    printf("Enter an integer\n");
    scanf("%d",&n);
    m=n;
    revers=0;
    while(n!=0)
      {
        revers=10*revers+n%10;
        n=(int)n/10;  /* integer part after division by 10 */
      }
    printf("The reserve number is %d\n ",revers);
    if(m==revers)
      printf("%d is a palindrom\n ",m);
    else
      printf("%d is not a palindrom\n",m);
  }
```

A sample of input/output:

```
Enter an integer 23432

The reserve number is 23432
  23432 is a palindrome

Enter an integer 23451

The reserve number is 15432
  23451 is not a palindrome
```

2.6.4 The do-while Statement

Another type of loop is available in C, which is known as do-while statement. This statement is similar to while statement, but one difference is that the logical condition appears in this statement tested at the end of the loop. For this causes this loop is executed at least once.

The general form of this statement is

```
do
{
   statement(s);
}
while(logical_expression);
```

where logical_expression is an logical expression.

The loop will repeat when llogical_expression (the condition) is true. The loop will terminate when logical_expression is false.

Example 2.6.5 Write a program to find the sum of some integers until sum of the integers just greater than or equal to 500.

```
/* sum of integers */
#include<stdio.h>
main()
  {
    int  sum=0,count=0,m;
    do
    {
      scanf("%d",&m);
      sum+=m;
```

```
    count++;
  } while(sum<500);
  printf("Sum=%d  count=%d",sum,count);
}
```

A sample of input/output:

```
100
250
100
120
Sum=570  count=4
```

It may be noted that the logical condition is tested at the end of the loop. The execution is terminated when the logical expression is false and the loop is repeated when the condition is true. Also, this loop is executed for at least one time since the condition is tested at the end of the loop.

2.6.5 The break Statement

This statement is used to terminate loops. It is generally used in while, do-while, for and switch statement.

The general form is

```
break;
```

It may be noted that no condition is required to execute this statement. This statement is frequently used in switch statement; and when the break statement is executed within a switch statement then control goes outside of the switch statement.

The break statement is illustrated in the following program.

Example 2.6.6 Write a program to read a sequence of at most 100 integers and count the number of positive integers up to the first occurrence of negatives integer.

```
#include<stdio.h>
main()
  {
    int  count=0,x;
    do
```

```
    {
      scanf("%d",&x);
      if(x<0) break;
      count++;
    }
    while(count<100);
    printf("Number of positive integers entered is %d",count);
}
```

The do-while loop repeats for 100 times if all the integers are positive or zero, but if there be any negative integer the loop will terminate and then the value of count will be printed.

2.6.6 The continue Statement

The continue statement is generally used within a loop (while, do-while, for, switch, etc) to bypass a portion of the loop. When the continue statement is encounter the control goes to the beginning of the loop and execute the statement within it. Recall that, when break statement is encounter within a loop, the control goes to the outside of the loop.

The general form of the continue statement is

```
continue;
```

The following example is consider to illustrate the continue statement.

Example 2.6.7 Given a set of n real numbers. Find the sum of only positive numbers.

```
#include<stdio.h>
main()
  {
    int n, i=0;
    float x, sum=0;
    printf("Enter number of date ");
    scanf("%d",&n);
    do{
      scanf("%f",&x);
      i++;
      if(x<0) continue; /* read next data */
      sum+=x;
```

```
    }while(i<n);
    printf("The sum of positive numbers is %f",sum);
}
```

Look at the continue statement. If x is negative then the statement sum+=x is not executed, the control directly goes to the beginning of the do-while loop. Reads the next number and again check for its sign.

The break and continue statements are frequently used in switch statement, which is discussed below.

2.6.7 The switch Statement

This statement is mainly used when a particular statement is to be selected from a group of statements, based on the value of an identifier/expression. The general form of this statement is

```
switch (expression)
  {
    case expression1:
      statement(s);
      break;
    case expression2:
      statement(s);
      break;
      .
      .
      .
    case expressionn:
      statement(s);
      break;
    default:
      statement(s);
  }
```

Here expression is an integer variable or integer expression. It may be noted that all these expressions can also be of character type as each individual character has an equivalent integer value.

The expression1, expression2, . . ., expressionn are generally integer or character constants. The expression is either simple or complex.

The values of expression1, expression2, ..., expressionn all are distinct (i.e. deferent alternatives) and they are referred as *case labels*. These case labels identify different groups of statements and they are unique within a switch statement. The default label is optional. The switch statement is executed as follows:

At first the value of expression is determine. If the value of it is equal to expression1 then the control goes to the case expression1, the statements following it are executed and then the control goes to the outside of the switch statement (due to the presence of break statement). If the value of expression is equal to expression2 then the control directly goes to the case expression2, execute the statement(s) of this group (i.e. the statements between second and third cases) and the control goes to the outside the switch statement and so on. If the value of expression does not match with any one of expression1, expression2, ..., expressionn then the statements under default label will be executed, if it is there. If default label is absent and the value of expression does not match with any one of expression1, expression2, ..., expressionn then the switch statement do nothing.

Here we consider an example where the switch statement is appropriately used.

Example 2.6.8 Write a program to compute the value of the Legendre polynomial $P_n(x)$ of degree n defined by

$P_1(x) = x$

$P_2(x) = \frac{1}{2}(3x^2 - 1)$

$P_3(x) = \frac{1}{2}(5x^3 - 3x)$

$P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3)$ for given $n(= 1, 2, 3, 4)$ and x.

```
#include<stdio.h>
main()
  {
    int n;
    float x,p;
    printf("Enter the value of n and x ");
    scanf("%d%f",&n,&x);
    switch (n)
    {
      case 1:
        p=x;
        break;
```

```
    case 2:
      p=(3.0*x*x-1.0)/2.0;
      break;
    case 3:
      p=(5.0*x*x*x-3.0*x)/2.0;
      break;
    case 4:
      p=(35*pow(x,4)-30*x*x+3.0)/8.0;
      break;
    }
  printf("\nP=%f for x=%f",p,x);
}
```

A sample of input/output:

```
Enter the value of n and x 3 .5

P=-0.437500 for x=0.500000
```

If the value of n is 3 then the control directly goes to the label case 3 and the value of p is evaluated from the expression $P_3(x)$ (from the block of case 3) and then exit from the switch statement. Finally, the value of p is printed by the printf statement.

2.6.8 The for Loop

The most useful loop is for loop, it is very powerful statement in C. The simple form of for loop is

```
for(exp1; exp2; exp3)
   {
     statement(s);
   }
```

where exp1 assigns the initial value of the control variable, exp2 generally represents a condition. If this condition is valid then the statements within the for loop execute. exp3 is used to update the value of the control variable. Generally, exp1 is an assignment statement, exp2 is a logical expression and exp3 is a unary expression or an assignment statement. Notice that exp1, exp2 and exp3 are separated by semicolons, this means they are three different statements/expressions.

During the execution of for loop, the value of control variable initialized its value by executing exp1. Then exp2 is evaluated and checked exp2 is true or false. If exp2 is true then all the statement(s) within the for loop are executed,

otherwise the control goes to the outside of the for loop. After execution of all statements of for loop (this is called one pass of the loop) the exp3 is evaluated. exp3 updates the value of control variable and then execute exp2 and check its validity. Repeat this process until exp3 is true.

The flow chart of for loop is shown in Fig. 2.3.

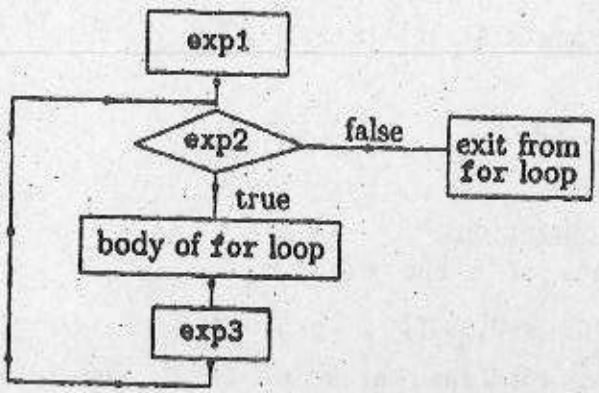


Figure 2.3: Flow chart of for loop

If the number of statements within for loop is one then there is no need to use braces. .

Now, we consider an use of for loop. Let us determine the sum of 10 numbers using for loop.

```
main()
{
  float sum=0, x;
  int i;
  for(i=1;i<=10;i++)
    {
      scanf("%f ",&x);
      sum+=x;
    }
  printf("Sum of 10 numbers is %f",sum);
}
```

Here i is the control variable which is initialized as 1 (since exp1 is $i = 1$). The exp2 is i<= 10, a logical expression. The value of i is known ($i = 1$). Thus the expression i <= 10 is true. Then the for loop reads a number as x. Add

it to the variable sum. Then the value of i is incremented by 1, since exp3 is i++ (an unary operator). Therefore, the value of i now becomes 2. This value satisfies the condition i <= 10. The loop repeats until the condition i <= 10 remains true.

It may be easy to observed that the loop is repeated for 10 times. It may be noted that there is no semicolon (;) at the end of for loop. If there is a semicolon at the end of for statement, then this is also a valid for statement and this loop repeats until exp2 is true, but it does not execute any statement.

Example 2.6.9 Write a program to find the maximum and minimum among N numbers.

```
/* Computation of maximum and minimum among n numbers */
#include<stdio.h>
main()
  {
    int n,i;
    float max,min,x;
    printf("How many numbers\n");
    scanf("%d",&n);
    printf("Enter the numbers\n");
    scanf("%f",&x);
    max=x;
    min=x;
    for(i=2;i<=n;i++)
      {
        scanf("%f",&x);
        if(max<x)  max=x;
        if(min>x)  min=x;
      }
    printf("Maximum=%f,  Minimum=%f\n",max,min);
  }
```

A sample of input/output:

```
How many numbers
5
Enter the numbers
-23 89 56 12 34
Maximum=89.000000,  Minimum=-23.000000
```

In this program the first number is read as x and it is stored in max and min. Since one number is given as input so n is reduced by one. The for loop is repeated for n - 1 times. Each time the loop reads a number as x, it is compared with max and min. If it is greater than max (less than min), then we set max=x (min=x) and control passes to the statement at the end of for loop. If x is not greater than max then the statement max=x is not executed.

Example 2.6.10 An approximate formula to find $n!$ is $n! = \sqrt{2\pi n}.n^n e^{-n}$ (Stirling's formula). Write a program to compute $n!$ exactly and by Stirling formula for $n = 1, 2, \ldots, 10$ and estimate the percentage error in each case.

```
/* Computation of factorial using direct method
  and using stirling's formula */
#include<stdio.h>
#include<math.h>
main()
  {
    long int factd;
    float facts,error;
    int i,n;
    printf("=================================\n");
    printf("n  Exact    by Stirling   Percent\n");
    printf("   value    formula       error\n");
    printf("=================================\n");
    for(n=1;n<=10;n++)
      {
        /* factorial by direct method */
        factd=1;
        for(i=1;i<=n;i++)
             factd*=i;
        /* factorial by Stirling's formula */
        facts=sqrt(2*3.141592653*n)*pow(n,n)*exp(-n);
        error=fabs(factd-facts)*100.0/(float)factd;
        printf("%d\t%ld\t%-9.2f\t%f\n",n,factd,facts,error);
      }
    printf("=================================\n");
  }
```

Output:

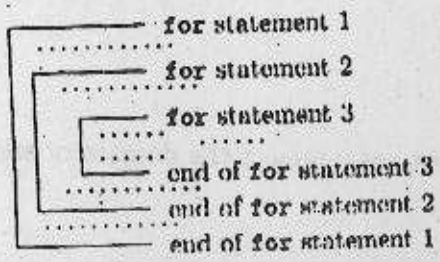
n	Exact value	by Stirling formula	Percent error
1	1	0.92	7.786298
2	2	1.92	4.049784
3	6	5.84	2.729837
4	24	23.51	2.057600
5	120	118.02	1.650696
6	720	710.08	1.378030
7	5040	4980.40	1.182619
8	40320	39902.39	1.035728
9	362880	359536.88	0.921276
10	3628800	3598695.50	0.829599

2.6.9 Nested for Statement

A for loop (outer for loop) may contain one or more for loops (inner loops) within its range. for loops occurring in this fashion are called nested for loops.

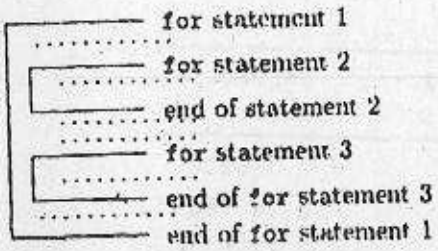
When one for loop is nested inside another, the inner for loop must be entirely contained within the range of the outer for loop. There must be no over-lapping of statements and each loop must have its own unique control variable. However, the loops may have the same terminal statement. The following are valid nested for loops.

(i) A for loop may be contained completely within another for loop.

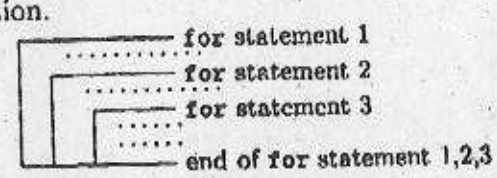


This is the complete nesting of loops.

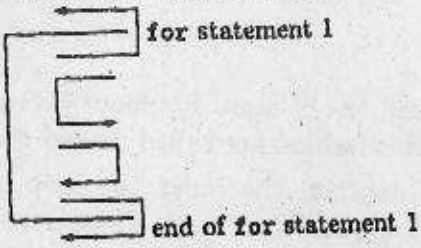
(ii) Two or more for loops may be contained within another for loop.

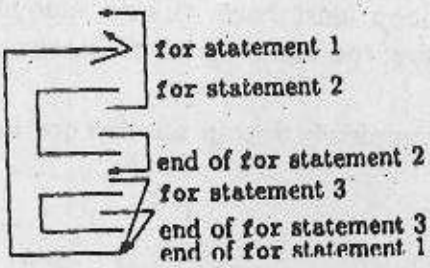


(iii) Two or more for loops starting at different position may end at the same position.



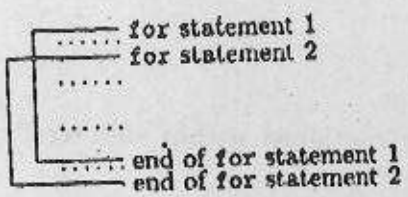
(iv) Transfer of control from any position within for loops to inside and to outside of its domain are allowed.



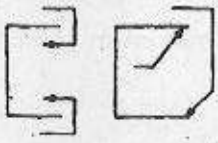


Some examples of invalid for loops

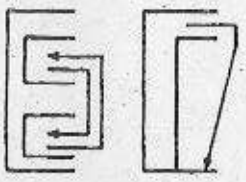
(i) The domain of one for loop must not intersect the domain of another for loop.



(ii) A transfer of control from outside to within the domain of a for loop is not allowed.



(iii) The transfer of control from one for loop to another for loop is not valid.



The comma operator

We have mentioned that exp1, exp2 and exp3 each of these expressions/ statements contains single expression/ statement. But, in general, each of them contains more than one statement/ expression and they are separated by the comma operator (,). For example,

```
for(i=0,j=10; i<=10, j>=5; i++,j--)
  {
    statement1;
    statement2;
    .
    .
    .
    statementn;
  }
```

This statement initialize `i = 0` and `j = 10`. Then check the validity of the expressions `i <= 10` and `j >= 5`. If they are valid then execute all the statements `statement1, statement2, . . ., statementn`. Then the values of *i* (increment) and *j* (decrement) are updated.

It may be noted that the for loop repeats when all the conditions under exp2 are true. If at least one condition is false then the for loop terminates. In the above example, the loop repeats for 6 times, because the condition `j >= 5`), becomes false after 6 times though `i <= 10` remains true.

2.6.10 Worked Out Examples

Example 2.6.11 Write a program to find the maximum among three numbers.

Solution. The following program is designed to find maximum among three numbers using if statements.

```
/* Program to find the maximum among 3 numbers */
#include<stdio.h>
main()
  {
    float a,b,c,big;
    printf("Enter three numbers\n");
    scanf("%f%f%f",&a,&b,&c);
    big=a;
    if(big<b)  big=b;
    if(big<c)  big=c;
    printf("The largest number is %f\n",big);
  }
```

A sample of input/output:

```
Enter three numbers
23.08 45.7 12.8
The largest number is 45.700000
```

Example 2.6.12 Write a program to test whether a given integer is even or odd.

Solution. Let N be an integer. If the remainder (when N is divided by 2) is 0 then N is even otherwise N is odd. The remainder of N is determined by the expression $N\%2$.

```
/* Program to test whether an integer is even or odd */
#include<stdio.h>
main()
  {
    int n;
    printf("Enter an integer\n");
    scanf("%d",&n);
    if(n%2==0)
      printf("%d is even\n",n);
```

```
    else
      printf("%d is odd\n",n);
  }
```

A sample of input/output:

```
Enter an integer
567
567 is odd
```

Example 2.6.13 Write a program to test whether a given integer is perfect square or not.

Solution. A number N is a perfect square if the square of the integer part of its square root is equal to N. That is if $int(\sqrt{N}) \times int(\sqrt{N}) = N$ then the number is a perfect square otherwise the number is not a perfect square.

```
/* Program to test whether an integer is perfect square */
#include<stdio.h>
#include<math.h>
main()
  {
    int n,m;
    printf("Enter an integer\n");
    scanf("%d",&n);
    m=(int)(sqrt(n));
    if(m*m==n)
      printf("%d is perfect square\n",n);
    else
      printf("%d is not perfect square\n",n);
  }
```

A sample of input/output:

```
Enter an integer
224
224 is not perfect square
```

Example 2.6.14 Write a program to test whether an integer is Armstrong number.

Solution. An integer N is said to be an Armstrong number if the sum of cube of its digits is equal to the number itself. For example, the number 153 is an Armstrong number as $1^3 + 5^3 + 3^3 = 153$.

```
/* Program to test whether an integer is Armstrong number */
#include<stdio.h>
#include<math.h>
main()
  {
    int n,m,rem,sum=0;
    printf("Enter an integer\n");
    scanf("%d",&n);
    m=n;
/* Finding sum of cube of the digits */
    while(n!=0)
    {
      rem=n%10; /* finding remainder */
      sum+=pow(rem,3); /* sum of cube of digits */
      n=n/10;
    }
    if(m==sum)
      printf("The integer %d is an Armstrong number",m);
    else
      printf("The integer %d is not an Armstrong number",m);
  }
```

A sample of input/output:

225

The integer 225 is not an armstrong number

Example 2.6.15 Write a program to compute the value of $f(x)$ for a given x where

$$f(x) = \begin{cases} x-1, & \text{for } x < 1 \\ 1-x, & \text{for } 1 \le x < 2 \\ 2-x^2, & \text{for } x \ge 2 \end{cases}$$

```
/* Program to find the value of a function */
#include<stdio.h>
main()
  {
    float x,fx;
    printf("Enter the value of x\n");
    scanf("%f",&x);
    if(x<1.0)
```

```
    fx=x-1.0;
  else if((1.<=x) && (x<2.0))
    fx=1.0-x;
  else
    fx=2.0-x*x;
  printf("f(x)=%f at x=%f\n",fx,x);
}
```

A sample of input/output:

2.5

```
f(x)=-4.250000 at x=2.500000
```

Example 2.6.16 Write a program to identify the nature of the roots only of a quadratic equation.

Solution. The discriminant d of the equation $ax^2 + bx + c = 0$ is $b^2 - 4ac$. If $d < 0$ then the roots are complex; if $d = 0$ then the roots are real and equal; and if $d > 0$ then the roots are real and distinct.

```
/* Program to find the nature of the roots of a quadratic equation */
#include<stdio.h
#include<math.h>
main()
  {
    float a,b,c,d;
    printf("Enter the values of a, b, c\n");
    scanf("%f%f%f",&a,&b,&c);
    d=b*b-4*a*c;
    if(d<0)
      printf("Roots are comples\n");
    else if (d==0)
      printf("Roots are real and equal\n");
    else
      printf("Roots are real and unequal\n");
  }
```

A sample of input/output:

1 7 12

```
Roots are real and unequal
```

Example 2.6.17 Write a program to find the roots of a quadratic equation.

Solution. Let the quadratic equation be $ax^2 + bx + c = 0$. Its roots are $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Let $d = b^2 - 4ac$ and $p = -\frac{b}{2a}$, $q = \frac{\sqrt{|d|}}{2a}$. If $d < 0$ then roots are $p \pm iq$, if $d = 0$ then the roots are p, p and if $d > 0$ then the roots are $p \pm q$

```
/* Program to find the roots of a quadratic equation */
#include<stdio.h>
#include<math.h>
main()
  {
    float a,b,c,d,p,q;
    printf("Enter the values of a, b, c\n");
    scanf("%f%f%f",&a,&b,&c);
    d=b*b-4*a*c;
    p=-b/(a+a);
    q=sqrt(fabs(d))/(a+a);
    if(d<0)
      printf("Roots are (%f,%f) and (%f,%f)\n",p,q,p,-q);
    else if (d==0)
      printf("Roots are %f and %f\n",p,p);
    else
      printf("Roots are %f and %f\n",p+q,p-q);
  }
```

A sample of input/output:

```
Enter the values of a, b, c
1 2 3
Roots are (-1.000000,1.414214) and (-1.000000,-1.414214)
```

Example 2.6.18 Write a program to read N real numbers and count the negative, zero and positive numbers among them.

```
/* Counting of positive, negative and zero */
#include<stdio.h>
main()
```

```
{
  int n, i,neg=0,pos=0,zero=0;
  float a;
  printf("Enter number of numbers\n");
  scanf("%d",&n);
  printf("Enter the numbers\n");
  for(i=1;i<=n;i++)
    {
     scanf("%f",&a);
     if(a<0) neg++;
     else if (a>0) pos++;
     else zero++;
    }
  printf("Negative no.=%d, Positive no.=%d, Zero=%d\n",neg,
                                                    pos,zero);
}
```

A sample of input/output:

```
Enter number of numbers
7
Enter the numbers
12 -89 0 23 -12 0 -27
Negative no.=3, Positive no.=2, Zero=2
```

Example 2.6.19 Write a program to find the sum of the series

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

for given x and n.

Solution. Here $t_n = \frac{x^n}{n!}$ and $t_{n+1} = \frac{x^{n+1}}{(n+1)!}$

Therefore, $\frac{t_{n+1}}{t_n} = \frac{x^{n+1}}{(n+1)!} \times \frac{n!}{x^n} = \frac{x}{n+1}$

Thus, $t_{n+1} = \frac{x}{n+1} \times t_n$

```
/* Program to find the sum of a finite series *
#include<stdio.h>
```

```
#include<math.h>
main()
  {
    int n, i;
    float  x,sum=1,term=1;
    printf("Enter the values of x and n\n");
    scanf("%f%d",&x,&n);
    for(i=1;i<=n;i++)
      {
        term*=x/i;
        sum+=term;
      }
      printf("Sum of the series = %f at x=%5.2f and n=%d\n",sum,x,n);
  }
```

A sample of input/output:

```
Enter the values of x and n
1.2 20
Sum of the series = 3.320117 at x= 1.20 and n=20
```

Example 2.6.20 Write a program to compute

$$J_0(x) = 1 - \frac{x^2}{2^2 1!1!} + \frac{x^4}{2^4 2!2!} - \frac{x^6}{2^6 3!3!} + \ldots$$

for a given x by direct summation of successive terms up to and including the first term that has a magnitude greater than 10^{-8}.

Solution. Here the nth term of the series is

$$t_n = (-1)^n \frac{x^{2n}}{2^{2n}(n!)^2}, \; n = 0, 1, 2, .$$

Therefore $t_{n+1} = (-1)^{n+1} \dfrac{x^{2n+2}}{2^{2n+2}\{(n+1)!\}^2}$

Now, $\dfrac{t_{n+1}}{t_n} = -\dfrac{x^2}{4(n+1)^2}$ or $t_{n+1} = -\dfrac{x^2}{4(n+1)^2} t_n$

That is, if a term is known then the next term of the series is obtain by

$$-\frac{x^2}{4(n+1)^2} \times term$$

where *term* means nth term.

```
/* Program to evaluate Bessel's series */
#include<stdio.h>
#include<math.h>
main()
  {
    float  x,sum,term,eps=1.0e-8;
    int n,m;
    printf("Enter the value of x\n");
    scanf("%f",&x);
    /* initialization of sum, term, n */
    term=1.0;
    sum=1.0;
    n=0;
    while  (fabs(term)>eps)
      {
       term=-x*x*term/(4.*(n+1.)*(n+1.));
       sum+=term;
       n++;
      }
    printf("J0(%5.3f)=  %f\n",x,sum);
  }
```

A sample of input/output:

```
Enter the value of x
.526
J0(0.526)=  0.932018
```

Example 2.6.21 Write a program to compute Greatest Common Divisor (GCD) and Lowest Common Multiple (LCM) of two given integers.

The method to calculate GCD and LCM is described in Unit 1.

```
/* Program to determine GCD and LCM of two integers
   prod, rem represent respectively product and remainder */
#include<stdio.h>
main()
  {
    int  n,m,prod,rem,gcd,lcm;
    printf("Enter two integers\n");
    scanf("%d%d",&m,&n);
```

```
    printf("m=%d  n=%d\n",m,n);
    prod=m*n;
    while(n!=0)
      {
        rem = m % n;
        if(rem == 0)  break;
        m=n;
        n=rem;
      }
    lcm=prod/n;
    gcd=n;
    printf("LCM=  %d  and  GCD=  %d\n",lcm,gcd);
  }
```

A sample of input/output:

```
Enter two integers
8 60
m=8  n=60
LCM=  120  and  GCD=  4
```

Example 2.6.22 Write a program to generate first N terms of the Fibonacci sequence.

Solution. The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, ... It may be noted that the sum of two preceding terms is the next term. Mathematically, $F_{n+2} = F_{n+1} + F_n$ where $F_0 = 0$, $F_1 = 1$. F_n is the nth term of the sequence. The following program generates first N terms of this sequence for a given N.

```
/* Generation of Fibonacci sequence */
#include<stdio.h>
main()
  {
    int f0=0,f1=1,f,n,i;
    printf("How many numbers\n");
    scanf("%d",&n);
    printf("0 1 ");
    for(i=2;i<n;i++)
      {
        f=f0+f1;
```

```
        f0=f1;
        f1=f;
        printf("%4d",f);
      }
  }
```

A sample of input/output:

```
How many numbers
12
0   1   1   2   3   5   8   13   21   34   55   89
```

Example 2.6.23 Write a program to find the prime numbers between two given integers.

```
/* Program to generate prime numbers between two given integers*/
#include<stdio.h>
#include<math.h>
main()
  {
    int  i,j,k,n,m,flag=1;
    printf("Enter two positive integers m and n (m<n)\n");
    scanf("%d%d",&m,&n);
    for(i=m;i<=n;i++)
      {
        flag=1;
        k=(int)sqrt(i);
        /* checking i for prime */
        for(j=2;j<=k;j++)
          {
            if((i%j)==0){
                flag=0; break;
              }
          }
        if(flag==1) /* i is prime */
       printf("%4d ",i);
    }
  }
```

A sample of input/output:

```
Enter two positive integers m and n (m<n)
40 80
41   43   47   53   59   61   67   71   73   79
```

Exercise 2.6

1. Write short notes on (i) goto statement, (ii) if statement, (iii) block if statement, (iv) nested if statement, (v) multi-alternative if statement.
2. Write short notes on (i) for loop, (ii) do loop, (iii) do-while loop, (iv) continue statement, (v) break statement.
3. Draw flowcharts of (i) goto statement, (ii) if statement, (iii) block if statement, (iv) nested if statement, (v) multi-alternative if statement.
4. Compare continue and break statements.
5. Indicate the errors, if any in the following C statements.

   (i) `goto next`

   (ii) `goto p1;`

   (iii) `go to 156`

   (iv) `if(x+y) y=5`

   (v) `if(x<=y) max=y;`

   (vi) `if(x<y) z=x+y;`

   (vii) `if(x<y) then z=x+y;`

6. Assume that at the beginning of each of the following program segment M=5 and N=15. What will be the final values of N and M after the execution of each segment ?

   (i)
   ```
   if(m<n) n=n+5;
   n+=3;
   ```

   (ii)
   ```
   if(2*m ==n) goto 150;
   n++
   goto 120;
   150: n=m;
   120: n+=5;
   ```

7. Write a program to find only the real and unequal roots, if any, of a quadratic equation, with appropriate message.

8. Write a program to check the nature (real and equal, real and distinct or imaginary) of the roots of a quadratic equation $ax^2 + bx + c = 0$.

9. Write a program to find the sum of the following series, correct up to 5 significant digits.

   (a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots\ldots\ldots$ (when x given in degree/radian).

   (b) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots\ldots\ldots$ (when x given in degree/radian).

   (c) $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots\ldots\ldots$ (for any x).

10. Write a program to read N real numbers and count the negative, zero and positive numbers among them.

11. Write a program to test whether an integer is (i) divisible by 2, (ii) divisible by 3, (iii) divisible by 7 and (iv) divisible by 5.

12. A university gives first class degree if the average marks in an examination is 60% or above, second class if the marks 40% or above and the candidate fails if his/her marks is below 40%. Write a program to find the grade of N (=10 say) students.

13. A nationalized bank has a fixed deposit scheme and the rate of interest for different periods are given below :

duration	rate of interest
3 yrs or more	10%
2 yrs to less than 3 yrs	9%
1 yr to less than 2 yrs	8.5%
less than 1 yr	7.5%

The interest is compounded quarterly. Write a program to find the total amount to be returned by the bank, on a fixed capital, at the end of a fixed period.

[total return $A = P(1 + \frac{r}{4\times100})^{4t}$]

14. Indicate the errors, if any, in the following for statements.

(i) `for(i=1,i<=5,++i);`

(ii) `for(i=0;i=5;++i);`

(iii) `for(i=0;i<=10);`

(iv) `for(i=1;i<=5;i+=2)`

(v) `for(j=i;j<=10;i++);`

15. Write programs to find

(i) minimum among n numbers.

(ii) maximum among n numbers.

(iii) maximum and minimum among n numbers.

16. Given a set of n integers. Write programs to find the

(i) total number of even integers.

(ii) total number of odd integers.

(iii) sum and product of even integers.

17. The Fibonacci sequence is defined as follows: The first and second terms of the sequence are 0 and 1. The third and subsequent terms of the sequence are the sum of just preceding two terms. That is, $t_0 = 0$, $t_1 = 1$, and $t_n = t_{n-1} + t_{n-2}$, $n \geq 2$. Write a program to find the first N (=100, say) numbers of the sequence.

18. Write a program to find the sum and average of n numbers.

19. Write the following segment using for loop.

```
        count=0;
        i=1;
next:   scanf("%f",&a);
        if(a>0)  count++;
        i++;
        if(i<=250)  goto  next;
```

20. Rewrite the following program segment without using for loops.

```
for(i=1;i<=4;i++)
    for(j=1;j<=20;j++)
        printf("%d%d",&i,&j);
```

21. Find the output of each of the following program segments:

(i)

```
i=2;
j=5;
for(k=1;k<=5;k++)
   {
      j++;
      i=j*2;
      j+=k;
   }
printf("%d %d %d",i,j,k);
```

(ii)

```
m=100;
n=0;
i=0;
count=0;
for(j=1;j<=5;j++)
   for(k=4;k>=0;k-=2)
      {
         i+=2;
         for(l=1;l<=2;l++)
           {
              n+=2;
              m--;
           }
         count++;
      }
   printf("%d %d %d",m,n,count);
```

2.7 Arrays

Sometimes, it is necessary to represent a group of values by a common variable name. For example, the marks obtained by 200 students in an examination may be represented by a variable, say MARK, and the variables MARK[1], MARK[2], ..., MARK[200] represent the marks of the first student, second student, ..., 200*th* student respectively. A method should be used for numbering

the students. A set of data with similar properties (homogeneous) and which are stored in consecutive memory locations under a common variable name is called an **array or subscripted variable**. Here MARK is an array. It is also called the subscripted variable with one subscript. An individual item is called the **element of the array**. Each element of an array is identified by using subscript (or subscripts) within parentheses after the common variable name (*i.e.*, by the array name). An array is called by the name of the variable part. In C, an array can have one, two or three subscripts. If there is only one subscript, the array is called **one dimensional**. If there occurs two (three) subscripts, then the array is called two (three) dimensional. Some computers allow even more than three subscripts.

The general form of a subscripted variable or array is

```
name[i], name[i][j]
```

where name is a valid C variable and i, j are subscripts. The use of an appropriate type statement a variable name can be declared as real, integer, character, etc. The following rules should be followed when using a subscript.

(i) A subscript may be a valid integer (constant or variable).

(ii) The value of a subscript may be zero or positive.

(iii) A subscript must not be a subscripted variable.

(iv) If a variable is used to represent an array, then it should not be used as an ordinary variable, *i.e.*, if x denotes an array then x should not be used as an ordinary variable.

A subscripted variable should be declared prior to its use.

2.7.1 Declaration of One Dimensional Array

In C, an array is declared as the same way as ordinary variable. Only new thing is that the size of the array must be specified within the square brackets after the array name. The general form of definition of one-dimensional array is

```
data_type array_name[exp];
```

where `data_type` is the type of data (i.e. `int`, `float`, etc.), `array_name` is the valid variable name in C, and exp is a positive integer constant or variable, which represents the size of the array.

Example 2.7.1 Some one-dimensional array declaration is shown below.

```
int x[20];
float a[10];
char name[20];
```

In the first declaration, the array name is x and its size is 20. For this declaration a memory is allocated for x to store 20 integers, i.e. x can store only 20 integer values to the variables x[0], x[1], x[2], ..., x[19].

Similarly, in second declaration a is an array of size 10 and it can store only ten values to the variables a[0], a[1], a[2], . . ., a[9], whereas third declaration indicates name is an array of size 20. It may be noted that the memory is allocated for an array during its declaration.

The array size may also be defined by a symbolic name rather than a fixed integer constant. This makes easier to change the array in future. By changing the value of symbolic constant the size of array can be changed. The value of symbolic constant may defined by using #defined statement as follows.

```
#define MAX 10
```

The following declaration

```
int x[MAX];
```

declares an integer array of size MAX, i.e. of size 10. The constant MAX may be used within the program but its value remains 10.

Initialization of one dimensional array

Like ordinary variable, an array can also be initialized during its declaration. The general form is

```
data_type array_name[exp]={value1, value2, . . ., valuen};
```

where data_type, array_name and exp are same as in previous declaration. The value1 is the value of the first element, value2 is the second element of the array and so on. exp is the optional in this declaration.

Example 2.7.2 The following declarations initialize the three arrays roll, vowel and weight.

```
int roll[6]={10,20,15,20,60,30};
char vowel[5]={'e','i','o','a','u'};
float weight[3]={40.5,50.8,30.9};
```

These declarations initialize three arrays roll, vowel and weight. The value of individual elements are shown below.

```
roll[0]=10    vowel[0]='e'    weight[0]=40.5
roll[1]=20    vowel[1]='i'    weight[1]=50.8
roll[2]=15    vowel[2]='o'    weight[2]=30.9
roll[3]=20    vowel[3]='a'
roll[4]=60    vowel[4]='u'
roll[5]=30
```

A variation of this declaration is also available in C. If the values of some array elements are not specified within braces, then the initial values of these elements are automatically assign to 0. Consider the following declaration.

```
int  roll[6]={10,15,20};
float weight[3]={40.5,2.0};
```

For these declaration the values of individual elements are assigned as follows.

```
roll[0]=10      weight[0]=40.5
roll[1]=15      weight[1]=2.0
roll[2]=20      weight[2]=0.0
roll[3]=0
roll[4]=0
roll[5]=0
```

Note 2.7.1 Comparison, assignment, etc. operations between two similar arrays can not be done with a single statement. These operations are performed element wise with the help of a loop statement such as for loop. For example, the array a can be assigned to the array b by the following statement

```
for(i=0; i<n; i++)
    b[i]=a[i];
```

Input/Output of one-dimensional array

There is no single statement to read an array in C. But, loops (`for`, `while`, etc.) are used to read/write an array. Usually the `scanf/printf` functions are used to read/write an array.

Suppose a is a floating point array of size 50. This array can be read by the following statements.

```
for(i=0; i<50; i++)
 scanf("%f",&a[i]);
```

Similarly, this array can be printed by the following statements.

```
for(i=0; i<50; i++)
 printf("%f",a[i]);
```

Let name be an array of characters containing 20 characters. This array can be read as follows :

```
for(i=0; i<20; i++)
 name[i]=getchar();
```

or this array may be read as

```
k=0;
do
  {
    name[k]=getchar();
    k++;
  }while(k<20);
```

To print the array of characters, putchar function instead of getchar function is used.

Example 2.7.3 Write a program to find the sum and average of N numbers using array.

Solution. Let a_i, $i = 1, 2, \ldots n$ be an array of size n. The sum $\sum_{i=1}^{n} a_i$ is computed in the following program.

```
/* Computation of the sum and average of n numbers */
#include<stdio.h>
main()
  {
    float a[50],sum=0,ave;
    int n,i;
    printf("Enter the value of n\n");
    scanf("%d",&n);
    printf("Enter the numbers\n");
    for(i=1;i<=n;i++){
      scanf("%f",&a[i]);
      sum+=a[i];
     }
    ave=sum/n;
    printf("Sum=%f  Average=%f\n",sum,ave);
  }
```

A sample of input/output:

```
Enter the value of n
6
Enter the numbers
100.45 345.32 879.23 89.45 76.20 78.4
Sum=1569.049927  Average=261.508331
```

2.7.2 Multidimensional Arrays

In multidimensional arrays the number of subscripts is more than one. For example, a_{ij} is a two-dimensional array containing two subscripts i and j usually used to handel matrices. Similarly, a_{ijk} is a three-dimensional array with three indices i, j and k. The declaration of two or more dimensional arrays is similar to one-dimensional array, except that a pair of square brackets is needed for each subscript. That is, for k-dimensional array, k pairs of square brackets are needed. The general form of declaration of multidimensional array is

```
data_type  array_name[exp1][exp2]... [expk];
```

where exp1, exp2, ..., expk are k positive valued integer constants or variables, these indicate the number of elements associated to each subscript.

The following statement

```
float x[4][5]
```

declares that x is a two-dimensional array with the first subscript ranging from 0 through 3 and second subscript ranging from 0 through 4, i.e. it declares the following elements.

```
x[0][0]  x[0][1]  x[0][2]  x[0][3]  x[0][4]
x[1][0]  x[1][1]  x[1][2]  x[1][3]  x[1][4]
x[2][0]  x[2][1]  x[2][2]  x[2][3]  x[2][4]
x[3][0]  x[3][1]  x[3][2]  x[3][3]  x[3][4]
```

Thus the size of this array is 20. It may be noted that the elements of a twodimensional array are store in memory according to the above order (row wise).

In two-dimensional array first and second subscripts are generally referred as row and column respectively.

A three-dimensional array can be defined as

```
float  a[10][5][4];
double  records[T][R][C];
```

The first declaration defines a three-dimensional array a whose first subscript ranging from 0 to 9, second subscript ranging from 0 to 4 and the third one is ranging from 0 to 3. That is, the array a can store $10 \times 5 \times 4 = 200$ elements. Similarly, in second declaration, the array records can store $T \times R \times C$ elements, where T, R and C are symbolic constants.

Initialization of multidimensional array

Like one-dimensional array, the multidimensional array can also be initialized. A two-dimensional array initialize row wise. For example, the array a[3][4] is initialized in the following order.

```
a[0][0]  a[0][1]  a[0][2]  a[0][3]
a[1][0]  a[1][1]  a[1][2]  a[1][3]
a[2][0]  a[2][1]  a[2][2]  a[2][3]
```

That is, keeping the first subscript fixed, the second subscript is varying from 0 to the last value of it, then increase the value of the first subscript by one and repeat all values of second subscript, and so on. Consider the following initialization.

```
int mark[2][3]={50,20,30,40,60,80};
```

The values of all elements of the array mark for this initialization is shown below.

```
mark[0][0]=50   mark[0][1]=20   mark[0][2]=30
mark[1][0]=40   mark[1][1]=60   mark[1][2]=80
```

If the number of elements within the braces is less than the array size, then the remaining elements set to zero. For example, the declaration

```
int mark[2][3]={50,20,30,40};
```

initializes the elements as

```
mark[0][0]=50    mark[0][1]=20    mark[0][2]=30
mark[1][0]=40    mark[1][1]=0     mark[1][2]=0
```

A two-dimensional array can also be initialized as a group of elements. The above array mark is initialized as follows.

```
int mark[2][3]={
                {50,20,30}
                {40,60,80}
               };
                  or
int mark[2][3]={{50,20,30},{40,60,80}};
```

In this initialization, the numbers of the set {50, 20, 30} are assigned to the elements mark[0][j] for $j = 0, 1, 2$ and the number of second set {40, 60, 80} are assigned to the elements mark[1][j] for $j = 0, 1, 2$.

If the number of values within braces are less than the number of elements of the array, then the remaining elements set to zero.

The input/output statements for multidimensional arrays are similar for onedimensional array. For example, the input statement to read the floating point array a[i][j], where $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$ is

```
for(i=0;i<m;i++)
   for(j=0;j<n;j++)
      scanf("%f",&a[i][j]);
```

Similarly, its output statement is

```
for(i=0;i<m;i++)
   for(j=0;j<n;j++)
      printf("%f ",a[i][j]);
```

To print this array row wise the following statements are used.

```
for(i=0;i<m;i++)
 {
   for(j=0;j<n;j++)
      printf("%f ",a[i][j]);
      printf("\n");
 }
```

An example of two-dimensional array is given below.

Example 2.7.4 Write a program to find the transpose of a matrix of order $m \times n$.

Solution. Let $A = [a_{ij}]_{m \times n}$ be a matrix and its transpose be $B = [b_{ij}]_{n \times m}$ where $b_{ji} = a_{ij}$ for $i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$.

```
/* Program to find transpose of a matrix */
#include<stdio.h>
main()
  {
    int i,j,m,n;
    float a[10][10], b[10][10];
    printf("Enter the order of the matrix a\n");
    scanf("%d%d",&m,&n);
```

```
    printf("Enter the elements of the matrix a\n");
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
        scanf("%f",&a[i][j]);
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
        b[j][i]=a[i][j];
    printf("The transpose of a is \n");
    for(i=0;i<n;i++){
      for(j=0;j<m;j++)
        printf("%5.2f  ",b[i][j]);
        printf("\n");
      }
  }
```

A sample of input/output:

```
Enter the order of the matrix a
3 4
Enter the elements of the matrix a
2 3 4 6
7 8 9 0
1 2 3 4
The transpose of a is
   2.00  7.00  1.00
   3.00  8.00  2.00
   4.00  9.00  3.00
   6.00  0.00  4.00
```

2.7.3 Worked Out Examples

Example 2.7.5 (Sine series) Write a program to compute

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots\ldots$$

to three significant digits for $x = 30°$, $50°$ and $73°$ by direct summation of successive terms and print the values of x, number of terms used in the series to obtain the accuracy and value of the series in tabular form as

x No. of terms Value of the series

Solution. Here the series is the sine series and the nth term of the series is,

$$t_n = (-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!},\ n = 1, 2, \ldots$$

Therefore $t_{n-1} = (-1)^n \frac{x^{2n+1}}{(2n+1)!}$

Now, $\frac{t_{n+1}}{t_n} = -\frac{x^2}{(2n+1)(2n)}$, or, $t_{n+1} = -\frac{x^2}{2n(2n+1)}t_n$, $n = 1, 2, \ldots$.

That is, the *newterm* $= -\frac{x^2}{2n(2n+1)} \times$ *oldterm*, where initially *oldterm=first term=x.*

```
/* Program to find the sum of the sine series */
#include<stdio.h>
#include<math.h>
main()
  {
    float  x,term,sum,a[4],eps=1.0e-5;
    int n,i;
    printf("Enter three values in degree\n");
    scanf("%f%f%f",&a[1],&a[2],&a[3]);
    printf("x(in deg) No. of  Value of \n");
    printf("          terms   the series\n");
    for(i=1;i<=3;i++)
      {
       x=a[i]*3.14159/180.; /* change to radian */
       term=x;
       sum=x;
       n=1;
       do
          {
            term=-x*x*term/((2.*n)*(2*n+1));
            sum+=term;
            n++;
          }while(fabs(term)>eps);
           printf("%5.2f\t%5d\t%9.3f\n",a[i],n,sum);
      }
  }
```

A sample of input/output:

```
Enter three values in degree
30 50 73
 x(in deg)  No. of   Value of
            terms    the series
   30.00      4        0.500
   50.00      5        0.766
   73.00      6        0.956
```

Example 2.7.6 (Horner method for polynomial evaluation) Write a program to find the value of a polynomial by Horner's method.

Solution. Let the polynomial of degree n be

$$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \quad + a_1 x + a_0.$$

This can be written as

$$(a_n x + a_{n-1})x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$$
$$= ((a_n x + a^{n-1})x + a^{n-2})x^{n-2} + \ldots + a_1 x + a_0$$
$$= \ldots\ldots = (((\ldots ((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0.$$

The computation begins from the inner parentheses, *i.e.*, from $(a_n x + a_{n-1})x$, and terminates when $n = 1$. It may be noted that to find the value of each inner loop, one addition and one multiplication is required. So, to compute the value of a polynomial of degree n we have to compute only n additions and n multiplications.

```
/* COMPUTATION OF POLYNOMIAL USING HORNER METHOD */
#include<stdio.h>
main()
  {
    float a[20],sum=0.,x;
    int i,n;
    printf("Enter the degree of the polynomial ");
    scanf("%d",&n);
    printf("Enter the coeffn. from highest degree\n");
    for(i=n;i>=0;i--)
      scanf("%f",&a[i]);
    printf("Coefficients of the polynomial are\n");
    for(i=n;i>=0;i--)
      printf("%6.3f ",a[i]);
    printf("\nEnter the value of x ");
```

```
    scanf("%f",&x);
    /* Evaluation of polynomial */
    for(i=n;i>=1;i--)
      sum=(sum+a[i])*x;
    sum+=a[0];
    printf("\nVALUE OF THE POLYNOMIAL AT %6.3f IS %9.4f",x,sum);
  }
```

A sample of input/output:

```
Enter the degree of the polynomial
4
Enter the coeffn. from highest degree
1.2 3.4 5.6 8.1 5
Coefficients of the polynomial are
1.200 3.400 5.600 8.100 5.000
Enter the value of x
1.5
VALUE OF THE POLYNOMIAL AT 1.500 IS 47.3000
```

Example 2.7.7 Write a program to split a number into digits and find the

(i) sum of the digits,

(ii) product of the digits,

(iii) the largest digit in the number with its position of occurrence and print the digits in reverse order with single spacing.

Solution. Let N be the given number. First, of all we divide N by 10. Then the remainder is the right most digit of the given number. Let it be D[1]. The quotient is taken as N. Again we divide N by 10. The remainder is taken as D[2] and the quotient as N. This process is repeated until the remainder becomes 0. Let k be the total number of digits of the given number. Then D[k], D[k-1], ..., D[1] are the digits of the given number in succession. Next, from the array D we compute the largest digit with its position and the sum of the digits. The digits in reverse order are obtained by printing the array D as D[1], D[2], ... D[k].

```
/* Program to split a number into digits and to find
   the sum, the product of digits, maximum digit within the
   number with position and printing of digits in reverse order.
*/
#include<stdio.h>
main()
```

```
{
  int d[10],  n,m,k=0,i,sum=0,prod=1,max,pos;
  printf("Enter an integer ");
  scanf("%d",&n);
  do
    {
     m=n/10;
     k++;
     d[k]=n%10; /* remainder */
     n=m;
    }while(n!=0);
 printf("The digits are \n");
 for(i=k;i>=1;i--)
    printf("%3d",d[i]);
/* Sum and product of digits */
 for(i=1;i<=k;i++)
    {
     sum+=d[i];
     prod*=d[i];
    }
 printf("\nSum of the digits = %4d and product=%5d\n",sum,prod);
/*Computation of maximum digit with position */
 max=d[1];
 pos=1;
 for(i=2;i<=k;i++)
    if(d[i]>max){
    pos=i;
    max=d[i];
   }
   printf("Maximum digits is %d at position %2d\n",max,k-pos+1);
   printf("Digits in reverse order\n");
   for(i=1;i<=k;i++)
     printf("%2d",d[i]);
   return;
 }
```

A sample of input/output:

```
Enter an integer
2873
```

```
The digits are
  2  8  7 3
Sum of the digits = 20 and product= 336
Maximum digits is 8 at position 2
Digits in reverse order
  3  7  8  2
```

Example 2.7.8 (Searching) Write a program to search an element within a list of elements.

Solution. Searching is a process which checks whether an element is in a list of elements or not. The element which is to be searched is called 'key' element. If the key is available in the list then the search is called successful, otherwise it is called unsuccessful search. In searching method every element is compared with the key element and if they are equal, then the search is successful.

```
/* Program to search an element within a list of n elements */
#include<stdio.h>
main()
  {
    int i,n,flag=0;
    float a[100],key;
    printf("Enter number of elements in the list\n");
    scanf("%d",&n);
    printf("Enter the elements\n");
    for(i=1;i<=n;i++)
      scanf("%f",&a[i]);
    printf("Enter the key element\n");
    scanf("%f",&key);
    for(i=1;i<=n;i++)
      if(a[i]==key)
        {
          flag=i;
          break;
        }
    if(flag!=0)
      printf("Number %6.2f is in the list at position %d\n", key,flag);
    else
      printf("Number %6.2f is not in the list\n",key);
```

```
    return;
  }
```

A sample of input/output:

```
Enter number of elements in the list
7
Enter the elements
10 34.5 60 23.9 70 90 -50
Enter the key element
70
Number 70.00 is in the list at position 5
```

Example 2.7.9 (Sorting in ascending order) Write a program to arrange n numbers in ascending order.

Solution. Sorting of numbers is a very important problem in Computer Science. Different sorting techniques are available. Here we present straight selection sort algorithm and the program to do it. The idea of this sorting technique is very simple. In the first step, the first element is compared with the remaining elements and the minimum element is placed in the first position. In the second step, the second element is compared with the remaining elements, except the first one, and the minimum one is placed in the second position. This process is continued until the last but one element is compared with the last element. The algorithm is given below.

```
/* Program to arrange a set of n elements in ascending order */
#include<stdio.h>
main()
  {
    int i,j,n;
    float a[100],temp;
    printf("Enter number of elements in the list\n");
    scanf("%d",&n);
    printf("Enter the elements\n");
    for(i=1;i<=n;i++)
      scanf("%f",&a[i]);
    for(i=1;i<=n-1;i++)
      for(j=i+1;j<=n;j++)
        {
          if(a[i]<a[j]) continue;
```

```
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
          }
      printf("Sorted list is\n");
      for(i=1;i<=n;i++)
        printf("%7.3f ",a[i]);
      return;
  }
```

A sample of input/output:

```
Enter number of elements in the list
6
Enter the elements
45.6 -78.3 89 0 12 67.3
Sorted list is
-78.300    0.000    12.000    45.600    67.300    89.000
```

Note 2.7.2 To sort n elements in descending order, the above program may be used by replacing < by > in the if statement.

Example 2.7.10 (Trace of a matrix) Write a program to find the trace of a square matrix of order $n \times n$.

Solution. The trace of a square matrix is the sum of its diagonal elements, *i.e.*, if $A = [a_{ij}]_{n \times n}$ then trace is $\sum_{i=1}^{n} a_{ii}$

```
/* Program to find the trace of a matrix */
#include<stdio.h>
main()
  {
    int i,j,n;
    float a[10][10],sum=0;
    printf("Enter the order of the square matrix \n");
    scanf("%d",&n);
    printf("Enter the elements of the matrix \n");
    for(i=0;i<n;i++)
      for(j=0;j<n;j++)
```

```
    scanf("%f",&a[i][j]);
  for(i=0;i<n;i++)
    sum+=a[i][i];
  printf("The trace of the matrix is %f6.2\n",sum);
}
```

A sample of input/output:

```
Enter the order of the square matrix
4
Enter the elements of the matrix
3 4 5 6
8 9 01
-4 -5 -6 -7
9 0 1 2
The trace of the matrix is
 8.00
```

Example 2.7.11 (Sum of matrices) Write a program to find the sum of two matrices of order $m \times n$. .

Solution. We know that the addition between two matrices is possible if they are of the same order. Let $A = [a_{ij}]$ and $B = [b_{ij}]$ be two matrices of order $m \times n$. Let $C = [c_{ij}]$ be the sum of A and B, where $c_{ij} = a_{ij} + b_{ij}$, $i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$.

```
/* Program to find the sum of two matrices */
#include<stdio.h>
main()
 {
   int i,j,m,n,p,q;
   float a[10][10], b[10][10],c[10][10];
   printf("Enter the order of the matrix A\n");
   scanf("%d%d",&m,&n);
   printf("Enter the elements of the matrix A\n");
   for(i=0;i<m;i++)
      for(j=0;j<n;j++)
     scanf("%f",&a[i][j]);
   printf("Enter the order of the matrix B\n");
   scanf("%d%d",&p,&q);
```

```
    if((m!=p) || (n!=q)) {
      printf("Order is not compatible! Addition is not possible\n");
      exit(0);
    }
    printf("Enter the elements of the matrix B\n");
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
      scanf("%f",&b[i][j]);
    /* finding sum */
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
        c[i][j]=a[i][j]+b[i][j];
    /* printing of matrix */
    for(i=0;i<m;i++){
      for(j=0;j<n;j++)
        printf("%6.3f  ",c[i][j]);
        printf("\n");
      }
  return;
  }
```

A sample of input/output:

```
Enter the order of the matrix A
2 3
Enter the elements of the matrix A
4 5 6
7 6 9.5
Enter the order of the matrix B
2 3
Enter the elements of the matrix B
4.5 6.7 8
2 1 -3
 8.500 11.700 14.000
 9.000  7.000  6.500
```

Example 2.7.12 (Product of matrices) Write a program to find the product of two matrices of order $m \times n$ and $n \times l$.

Solution. The product AB of two matrices A and B is possible if the number of columns of A is equal to the number of rows of B. Let $A = [a_{ij}]_{m \times n}$ and

$B = [b_{ij}]_{n \times l}$ be two given matrices. If the product AB is equal to C then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \; i = 1, 2, \ldots, m; \; j = 1, 2, \ldots, l.$$

```
/* Program to find the product of two matrices */
#include<stdio.h>
main()
  {
    int i,j,k,m,n,p,q;
    float a[10][10], b[10][10],c[10][10];
    printf("Enter the order of the matrix A\n");
    scanf("%d%d",&m,&n);
    printf("Enter the elements of the matrix A\n");
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
        scanf("%f",&a[i][j]);
    printf("Enter the order of the matrix B\n");
    scanf("%d%d",&p,&q);
    if(n!=p) {
      printf("Matrix product is not possible\n");
      exit(0);
     }
    printf("Enter the elements of the matrix B\n");
    for(i=0;i<p;i++)
      for(j=0;j<q;j++)
        scanf("%f",&b[i][j]);
    for(i=0;i<m;i++)
      for(j=0;j<q;j++)
        {
          c[i][j]=0.0;
          for(k=0;k<n;k++)
            c[i][j]+=a[i][k]*b[k][j];
        }
    printf("Product of the matrices\n");
    for(i=0;i<m;i++){
```

```
        for(j=0;j<q;j++)
          printf("%6.3f  ",c[i][j]);
          printf("\n");
        }
    return;
}
```

A sample of input/output:

```
Enter the order of the matrix A
2 3
Enter the elements of the matrix A
2 3 4
5 4 3
Enter the order of the matrix B
3 3
Enter the elements of the matrix B
4 5 -1
2 3 1
2 0 1
Product of the matrices
22.000    19.000    5.000
34.000    37.000    2.000
```

Exercise 2.7

1. What is an array ? Explain how an array variable is different from an ordinary variable ?
2. Explain the salient features of an array and its uses.
3. How are the individual elements accessed and processed in an array ?
4. What is a multidimensional array and how is it different from an onedimensional array ?
5. Describe how we can read to and print from an array in C.
6. Write short notes on (i) subscripted variable, (ii) array declaration.
7. Suppose an array a is declared as `float a[100]`. What happen if we want to store (i) 40 elements, (ii) 150 elements in a.
8. Write a program to read an integer array x_i, $i = 1, 2, \ldots, n$, and find the sum of even and odd integers in it separately.

9. Write a program to read n integers M_i, $i = 1, 2, \ldots, n$, and find the product of the even integers only.

10. Write a program to arrange n real (or integer) numbers in descending/ascending order.

11. Write a program to test whether a square matrix is orthogonal. [A matrix A of order $n \times n$ is said to be orthogonal if $AA^t = I$, where A^t is the transpose of A and I is the unit matrix of order $n \times n$.]

12. Write a program to find the value of a determinant of order 3.

13. Write a program to find the minimum (maximum) among the elements of a specified row (or column) of a matrix of order $m \times n$.

14. Pascal's triangle is a set of numbers having the following properties. Each row begins and ends with 1's and each number between two 1's is the sum of the numbers on either side of it in the above row. The first six rows of Pascal's triangle is shown below.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
```

Write a program to compute and print the first 15 rows of this triangle.

15. Write a program to read a set of numbers $x_1, x_2, \ldots, x_n$, and a number k, and check whether the number k is present in the array. If it is in the array print out how many times the number k is repeated in the array.

16. Write a program to read a two dimensional array and find the sum of the elements in each row and column separately and display them.

2.8 Functions

In C, every problem is decomposed into functions. The concept of function is the most powerful and convenient way to solve a large problem. The programmer can divide a large problem into several smaller subprograms, which when reassembled constitutes the complete C program. A function is an self-contained

program segment and it can be developed and tested separately. The use of function reduces the complexity and the size of source code of the main program significantly. Also, the same variable names can be used in several functions. Every C program contains one or more functions. One such function has a special name main. The execution of a C program begins from this function. The other functions are called from main function or another function. A function can be called from any position of the main function or from another function.

The functions in C are broadly of two types - library functions or built-infunctions and user-defined functions. The library functions such as abs, sqrt, sin, etc. are already mentioned. In this section only user defined functions are discussed.

In C, there are two major steps to use an user defined functions. The first step is declaration of function in calling (main) function (A function is said to be calling function if another function is called from this function). The second step is the definition of the function at outside of the calling function.

2.8.1 Definition of Function

The general structure of a function definition in C is

```
data_type function_name(arg1, arg2,...., argn)
  {
     declarations of local variables;
     body of the functions;
     return (expression);
  }
```

where data_type represents the type of the value which is returned by the function, function_name is the name of the function (which is user supplied valid C variable), and arg1, arg2,...., argn are called the formal arguments or formal parameters. Only the variables are used as formal arguments. The formal arguments allow information to be transferred from the calling portion of the program to the function. The variables used as formal arguments are 'local' in the sense that they are not identified outside of the function, i.e. these variables may be used as other variables outside the function. It may be noted that the formal arguments are separated by commas and enclosed by a pairs of parentheses. If the function does not required any arguments then only a pair of parentheses or void enclosed by the parentheses must be written after the function name.

The data_type may be omitted if the function returns an integer or a character (a character value is represented as an integer value) value. If the function does not return any value then the keyword void is written as data_type.

To use a function in a calling program (function reference), write its name with a list of **actual arguments** replacing the formal arguments, at the point in an expression where the value of the function is required. The actual arguments are also known as **actual parameters** or simply **arguments**. The actual arguments and formal arguments must correspond in order, number and type. An actual argument may be a constant, an array or an ordinary variable, an arithmetic or logical expression, another function.

The main part of a function is known as **body** of the function. It contains some statements which are required to evaluate the function. The formal arguments are used in these statements. It can access other functions. Also, it can access the function itself (this process is known as **recursion**, to be discussed latter).

The value obtain in a function is transferred to the calling program using the return statement. The return statement also causes control to be returned to the point from which the function was called. The general form is

```
return exp; return(exp);
```

The exp may be constant, variable or any expression. If exp represents an expression, then exp must be written within a pair of parentheses. The value of exp is returned to the calling function. The exp is optional, i.e. return statement may be written without it. If exp is absence then the function does not return any value to the calling function, just control goes to return back to the calling function. Only one exp can be included in the return statement. Thus, a function returns only one value to the calling function. But, return statement may be written in different positions with different expressions if the function contains multiple branches.

The flow between the calling function and a function is shown in Fig. 2.4.

Suppose a function sum is called from the main function then a portion of memory is allocated to store all the identifiers used in the function main and the necessary stack, etc. After completion of execution of the function (sum), the necessary value is returned to the calling function (main) and the memory allocated for the function (sum) becomes free.

Use of a function

Once a function is defined it can be used in several times within a program. It is *called* or *accessed* by specifying its name, followed by a list of actual arguments enclosed in parentheses. The arguments must be separated by commas.

```
/* main function or calling function */
  {
  .........
  v=sum(a,b); /* jump to the function sum */
  ......... /* back to calling function */
  .........
  }
 /* function definition */
 float sum(float a, float b) /* jump from calling function */
  {
  .........

 return exp; /* return back to the calling function */
  }
```

Figure 2.4: The flow between function and calling function.

If a function does not require any arguments, a pair of empty parentheses must be included after function name. The actual arguments may be constant, variable or expression. Each actual argument must be of same data type as its corresponding formal argument.

When a single value is passed to a function through an actual argument, the value of actual argument is *copied* into the corresponding formal argument for the function. Therefore, the value of the formal argument can be changed within the function, but the value of actual arguments remains same within the calling function. This procedure for passing the value of an argument to a function is known as passing by value.

The following program computes the value of ${}^nC_r$ by calling a function fact (which determines factorial) for three times.

Example 2.8.1 (Computation of ${}^nC_r$) Write a program to find the value of ${}^nC_r$ for given n and r using function.

Solution. The value of ${}^nC_r$ is

$$\frac{n!}{r!(n-r)!}$$

where $0! = 1$ and $n! = n.(n-1)(n-2)\ldots 2.1$. n and r are positive integers. To compute the value of ${}^nCr$, we have to compute the values of $n!$, $r!$ and $(n-r)!$, *i.e.*, same computations are required for different values, such as n, r and n-r. Thus, use of function is appropriate to compute the value of ${}^nCr$. A function

fact(m) is written with argument m and it is called three times for $m = n$, r and $n - r$.

```
/* Computation of n-c-r using function */
#include<stdio.h>
main()
  {
    int n,r,ncr;
    int fact(int m); /* declaration of function fact
                        as integer with integer argument */
    printf("Enter the values of n and r\n");
    scanf("%d%d",&n,&r);
    ncr=fact(n)/(fact(r)*fact(n-r)); /* calling of function fact
                                        with actual arguments n,r,n-r*/
    printf("N=%d R=%d NCR=%d\n",n,r,ncr);
    return;
  }/* end of main function */
/* definition of function fact */
int fact(int m) /* heading of the function fact with
                   formal argument m */
  {
    int i,mp=1; /* declaration of local variables */
    if((m==0) || (m==1))
      mp=1;
    else
      for(i=2;i<=m;i++)
        mp*=i;
      return mp;
  }/* end of function fact */
```

A sample of input/output:

```
Enter the values of n and r
7 3
N=7 R=3 NCR=35
```

In this program, the value of ncr is obtained by calling the function fact for first time with the actual argument n. The function is executed by taking $m = n$ and return the value of mp (the value of $n!$) to the main function. Again, the function fact is called for the argument r. The function is evaluated for $m = r$ and returned its value to the main function. Lastly, the function fact is

called for the argument n - r. Thus, the same function fact is called for three times with three different arguments and the value of ncr is evaluated using the expression `fact(n)/(fact(r)*fact(n-r));`.

2.8.2 Passing Arrays to a Function

Like ordinary variables, an array can also be passed to a function. To pass an array to a function, it is sufficient to list the name of arrays *without any subscripts* or *brackets* as an actual argument. The corresponding formal arguments are written in the same order and then must be declared as array within the formal argument declarations. During declaration of array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array *is not specified* within the square bracket in the formal argument declaration.

Suppose x is an integer array of size 50. If we calculate the maximum among n integers using a function, the declaration of function and array in main function is as follows.

```
main()
 {
   int n; /* declaration of variable */
   float max; /* declaration of variable */
   int x[50]; /* declaration of array. */
   int maximum(int[],int); /* declaration of function */
   . . . . . .
   . . . . . .
   max=maximum(x,n); /* calling of function with actual arguments
*/
   . . . . . .
   . . . . . .
 }
```

The definition of the function maximum is shown below.

```
int maximum(y,m) /* function definition */
int m;           /* definition of formal argument */
int y[ ];        /* definition of formal argument (array) */
  {
    . . . . .
    . . . . .
  }
```

From the main function the function maximum is called with two actual arguments x and n. Observed that x appears as an ordinary variable. In the definition of function, it is observed that two formal arguments y and m are used. The formal arguments y and m are declared as integer array and integer variable. Thus, there is a correspondence between the formal and actual arguments.

The formal arguments can also be declared in the first line of the function definition. For the function maximum, the first line may be written as

```
int maximum(int y[], int m)
   {
      . . . . . .
      . . . . . .
   }
```

It is mentioned earlier that the arguments are passed to a function by *value* when the formal arguments are ordinary variables. But, when an array is passed to a function, the values of the array elements *are not* passed to the function directly.

In this case, the *address* of the first array element is passed to the corresponding formal argument. That is, rather than passing the actual values of the array elements, a reference (called *pointer*) is passed to the function. Passing of argument in this way is called passed by reference rather than by value.

When the pointer of an array is passed to the function, the values of all elements of the array can be accessed by some memory management procedure.

Example 2.8.2 Write a program to find the maximum and minimum values among n numbers.

Solution. Let x be a floating point array and n be the number of elements from which the maximum and minimum are to be determined. Two functions maximum and maximum are defined to find the maximum and minimum values of the array x.

```
/* Computation of maximum and minimum from an array */
#include<stdio.h>
main()
 {
   int i,n;
   float x[100],max,min; /* declaration of array */
   float maximum(float[],int); /* declaration of function
                                  with type of arguments */
```

```
    float minimum(float[],int);
    printf("Enter array size\n");
    scanf("%d",&n);
    printf("Enter the elements of the array\n");
    for(i=0;i<n;i++) scanf("%f",&x[i]);
    max=maximum(x,n); /* calling of function */
    min=minimum(x,n); /* calling of function */
    printf("The maximum is %f\n",max);
    printf("The minimum is %f\n",min);
    return;
    }/* end of main function */
/* definition of the function maximum */
float maximum(float a[], int m) /* definition of function
                                with two formal arguments */
  {
    int i; /* declaration of local variable */
    float max;
    max=a[0];
    for(i=1;i<m;i++)
      if(max<a[i]) max=a[i];
    return max;
  }/* end of the function maximum */
/* definition of the function minimum */
float minimum(float a[], int m)
  {
    int i; /* declaration of local variable */
    float min;
    min=a[0];
    for(i=1;i<m;i++)
      if(min>a[i]) min=a[i];
    return min;
  }/* end of the function minimum */
```

A sample of input/output:

```
Enter array size
7
Enter the elements of the array
```

```
-90 -12 34 56 78 892 73
The maximum is 892.000000
The minimum is -90.000000
```

2.8.3 Recursion

We have seen that a function is called from main function and also mentioned that from any function one can call another function. Now, if a function is called from the same function, then the procedure is called **recursion**. A few high level languages support this facility. In recursion, a function calls *itself* repeatedly, until some termination condition has been satisfied.

This process reduces the effort to write a program to solve a problem.

The very common problem to illustrate the recursion is determination of $n!$. Normally, the value of $n!$ is determined by $n \times (n-1) \times \ldots \times 2 \times 1$. But, its recursive form is $n \times (n-1)!$. That is, the value of $n!$ depends on the value of $(n-1)!$, and the value of $(n-1)!$ depends on $(n-2)!$ and so on. That is, when the value of $(n-1)!$ is known, then one can calculate the value of $n!$. Also, we know that $1! = 1$. This is the termination condition for this problem.

The following program computes the value of $n!$ for a given n.

Example 2.8.3 ($n!$ using recursion) Calculate factorial of a positive integer n using recursion.

```
/* Computation of n! using recursion */
#include<stdio.h>
main()
 {
   int n;
   long int fact(int n); /* function declaration */
   printf("Enter an integer ");
   scanf("%d",&n);
   printf("n!=%ld\n",fact(n)); /* calling of function fact */
 }
 /* function definition */
 long int fact(int n)
   {
     long int f;
     if(n<=1)
      return 1;
```

```
    else
      f=n*fact(n-1); /* function fact is called with arg. n-1*/
}
```

A sample of input/output:

```
Enter an integer
10
n!=3628800
```

Here the function fact is declared as long int, because the value of factorial is large even for small value of n.

It is observed that the function fact is called from main for a single time and it is called for $(n-1)$ times from itself. In each call the value of actual argument of the function fact is reduces by 1. The repetition terminates when the argument becomes 1. The body of the function is not simple like other functions. To illustrate the mechanism of this function, we assume that $n = 4$.

Since $n \neq 1$, the statement f=n*fact(n-1) will be executed with $n = 4$. That is, the value of f is

```
f=4*fact(3);
```

and f depends on the value of fact(3). The value of fact(3) will be evaluated using the formula

```
f=3*fact(2);
```

Here also fact(2) is not known. Again, the function fact is called for $n = 2$. In this case,

```
f=2*fact(1);
```

Again, the function fact is called for $n = 1$. In this case, termination condition has been satisfied and fact returns 1. All the intermediate values of $n(= 3, 2, 1)$ are stored on a data structure called *stack*[1] until the termination condition has been satisfied. After reaching the termination condition, the actual values will be returned according to the following reverse order.

```
fact(1)=1
fact(2)=2*fact(1)=2×1=2
fact(3)=3*fact(2)=3×2=6
fact(4)=4*fact(3)=4×6=24
```

If a recursive function contains one or more local variables (the variables declared within the recursive function), then in each call a different set of local

[1] The stack is an order list of data structure in which all insertion and deletion are made in one end called top.

variables is created. The names of the local variables remain same (as they declared) within the function, but their values form different set in each call. Each set of values will store on the stack and they will be used in appropriate reversed call.

The variables used in a C program are broadly categorized, depending on the place of their declaration, as **local** (or **internal**) or **global** (or **external**). The local variables are those which are declared within a particular function, while external variables are declared outside of any function. The local variables are accessed within the function only where it is declared, where as the global variables can be accessed from any function within the program.

The use of local and global variables are illustrated in the following program.

Example 2.8.4 Write a program to find the sum and average of n numbers, without passing the array to the function.

```
/* Computation of sum and average using global variable */
#include<stdio.h>
float x[100]; /* declaration of global array, before main() */
float sum(int); /* declaration of global function */
main()
 {
   int i,n;
   float average(int); /* declaration of local function */
   printf("Enter array size\n");
   scanf("%d",&n);
   printf("Enter the elements of the array\n");
   for(i=0;i<n;i++) scanf("%f",&x[i]);
   printf("The sum of the elements is %f\n",sum(n));
   printf("The average is %f\n",average(n));
   return;
 }/* end of main function */
/* definition of the function sum */
float sum(int n)
 {
   int i;
   float s=0; /* i,s are local variables */
   for(i=0;i<n;i++)
```

```
        s+=x[i]; /* use of global variable */
      return s;
    }/* end of the function sum */

/* definition of the function average */
float average(int n)
 {
    float ave;
    ave=sum(n)/n; /* calling of global function sum */
    return ave;
 }/* end of the function average */
```

A sample of input/output:

```
Enter array size
7
Enter the elements of the array
10 23.5 67.4 90.2 -87.3 45 23
The sum of the elements is 171.800003
The average is 24.542858
```

2.8.4 Storage Classes

Recall that when a function is called from main or any other functions, the space is allocated for the variables declared within the function. The values of the variables are stored within the allocated space for this function and when the function returned a value to the main or other calling functions, then the allocated space becomes free and as a result all the variables declared within the function loss their values. But, the values of the variables of main (or other calling function) do not alter during execution of the called function. Thus we observed that some variables are active and some are not. This leads to the concept of scope and longevity of the variables, while using function in a program. The scope of variable means for what portion of the program a variable is available for use i.e. the variable is active. The longevity or lifetime means the period for which a variable retains its value during execution of a program, i.e. the variable is alive. To indicate the scope and longevity of variable we have to mention the storage class of it. Thus, each C variable has a data type and a storage class.

There are four different storage classes in C. They are automatic, external,

static and register. They are mentioned by the keywords auto extern, static and register respectively.

All the C variables are broadly classified into two categories depending on their place of declaration. These are **local** (*internal*) or **global** (*external*).

The local variables are created and used in a particular function and they are useful only within the function where they are defined. Whereas global variables are created outside any function and they can be used by any function with the program. The value of global variables can be changed from any place of the program. but, the value of a local variable can only be altered within the function where it is defined.

Some variable declarations with storage class are shown below.

```
auto int x,y;
static int z;
extern char c;
register float a, b;
```

Example 2.8.5 (Swapping) Write a program to interchange two values using function call.

Solution. Let x and y be two numbers. We have to interchange their values, i.e. x gets y value and y gets x value. A third variable *temp* is used to do this. The basic idea is that, *temp* will store the value of x and x will copy the value of y and finally the value of *temp* will assign to y.

```
#include<stdio.h>
float x,y; /* global declaration */
main()
 {
   void swap(); /* the function does not return any value */
   printf("Enter two numbers \n");
   scanf("%f%f",&x,&y);
   printf("Before call x=%f y=%f\n",x,y);
   swap();
   printf("After call x=%f y=%f\n",x,y);
   return;
 }
 void swap()
 {
   float temp;
   temp=x;
```

```
    x=y;
    y=temp;
}
```

A sample of input/output:

```
Enter two numbers
6785 984
Before call x=6785.000000 y=984.000000
After call x=984.000000 y=6785.000000
```

The swap interchanges two values and it returns two values implicitly with the help of global variables.

2.8.5 Worked Out Examples

Example 2.8.6 Write a function to compute the value of $f(x)$ defined as

$$f(x) = \begin{cases} e^x, & x < 0 \\ \sin x, & 0 \le x < 1 \\ \log_e x, & x \ge 1 \end{cases}$$

Then use this function in the main segment to compute and print $f(x)$ for x = -1 to 2 with spacing 0.5 in tabular form.

```
/* Evaluation of function using function */
#include<stdio.h>
#include<math.h>
main()
 {
   float x;
   float f(float x);
   printf("..............................\n");
   printf("  x                    f(x)\n");
   printf("..............................\n");
   for(x=-1.0;x<=2.0;x+=0.5)
     printf(" %4.1f  %8.3f\n",x,f(x));
   printf("..............................\n");
 }
 /* Definition of the function f(x) */
   float f(float x)
```

```
{
  float fx;
  if(x<0)
    fx=exp(x);
  else if((0<=x) && (x<1))
    fx=sin(x);
  else
    fx=log(x);
  return fx;
}
```

Output:

```
.....................
   x       f(x)
.....................
 -1.0     0.368
 -0.5     0.607
  0.0     0.000
  0.5     0.479
  1.0     0.000
  1.5     0.405
  2.0     0.693
.....................
```

Example 2.8.7 Write a function subprogram to find the sum of three numbers and use it to compute the sum of nine numbers.

Solution. Let `SUM3(x,y,z)` computes the sum of the numbers x, y, z. Using it four times we can compute the sum of nine numbers.

```
/* Sum of nine numbers using function */
#include<stdio.h>
main()
 {
   float  a1,a2,a3,a4,a5,a6,a7,a8,a9;
   float  u,v,w;
   float  sum3(float,float,float);
   printf("Enter nine values\n");
```

```
    scanf("%f%f%f%f%f%f%f%f%f",&a1,&a2,
        &a3,&a4,&a5,&a6,&a7,&a8,&a9);
    u=sum3(a1,a2,a3);
    v=sum3(a4,a5,a6);
    w=sum3(a7,a8,a9);
    printf("Sum=%f\n",sum3(u,v,w));
}
/* Definition of the function sum3() */
    float sum3(float x, float y, float z)
    {
      return (x+y+z);
    }
```

A sample of input/output:

```
Enter nine values
3 5 10 30 15 -10 4 8 13
Sum=78.000000
```

Example 2.8.8 Write a statement function to compute the value of a determinant of order 2 and use it to find the value of a determinant of order 3.

Solution. Let the determinant be

$$D = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

Its value is

$$D = a_1 \begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} - b_1 \begin{vmatrix} a_2 & c_2 \\ a_3 & c_3 \end{vmatrix} + c_1 \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix}$$

$$= a_1.DET2(b_2,c_2,b_3,c_3) - b_1.DET2(a_2,c_2,a_3,c_3) + c_1.DET2(a_2,b_2,a_3,b_3)$$

where $DET2(a,b,c,d) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = a.d - b.c$

```
/* Evaluation of determinant of order 3 using function */
#include<stdio.h>
main()
{
  float  a1,b1,c1,a2,b2,c2,a3,b3,c3,det3;
  float  det2(float,float,float,float);
  printf("ENTER THE ELEMENTS OF THE DETERMINANT ROW WISE\n");
  scanf("%f%f%f%f%f%f%f%f%f",&a1,&b1,&c1,&a2,&b2,&c2,&a3,&b3,&c3);
  det3=a1*det2(b2,c2,b3,c3)-b1*det2(a2,c2,a3,c3)+c1*det2(a2,b2,a3,b3);
  printf("The value of the determinant = %f\n",det3);
}
float det2(float a, float b, float c, float d)
  {
    return(a*d-b*c);
  }
```

A sample of input/output:

```
Enter the elements of the determinant row wise
3 4 5 8 9 1 4 9 -3 -4
The value of the determinant = -442.000000
```

Exercise 2.8

1. What is function? List out the advantages and disadvantages of using function.
2. How function is declared within a program?
3. What is the purpose of return statement?
4. What do you mean by arguments of function (actual and formal arguments)?
5. What are the differences between built-in function and user-defined function?
6. How a global variables is declared in C?

7. Write a function to find the product $\sum_{i=1}^{n} A(i)B(i)$ of two vectors $A(i)$ and $B(i)$, $i = 1, 2, \ldots, n$, and use it to find the value of

$$R = \sum_{i=1}^{n} [A(i)B(i) + B(i)C(i) + C(i)A(i)]$$

8. Write a function to find the sum (product) of four numbers and use it to find the sum (product) of sixteen numbers.
9. Write a function to find the maximum (minimum) among three numbers and use it to find the maximum (minimum) among nine numbers.
10. Write a function to test whether a matrix is null (symmetric). If it is null (symmetric) then the value of the function will be 1, otherwise, the value will be 0. Complete the program to test the matrix.
11. Write a function sump to find the value of the expression $\sum_{i=1}^{n} x_i^p$ with the array x_i and p (a positive integer) as dummy arguments. Use it find the value of

(i) $S1 = \sum_{i=1}^{n} x_i$, (ii) $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, (iii) $S2 = \sum_{i=1}^{n} x_i^2$

(iv) $S3 = \sum_{i=1}^{n} x_i^3$, (iv) $S = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$

12. Write a function GCD(i, j) to find the greatest common divisor of two integers i, j. Use this function to find the G.C.D among a finite set of integers (say 50). [The following relation may be used: if n =GCD($m_1, m_2, \ldots, m_k$) then GCD($m_1, m_2, \ldots, m_k, m_{k+1}$) = GCD($n, m_{k+1}$).]

2.9 Pointers

It is well known that every variable occupies one or more memory locations (cells) to store its value. A single cell can store one byte of information. A single character needs one byte space, so it can store in a single memory cell. An integer needs two bytes space, to store it two contiguous memory cells are required. A floating point number needs four contiguous cells. Each memory cell

is recognized by a number associated with it called address of the memory cell. The addresses of cells are usually done by hexadecimal numbers starting from 0. The compiler assigns the memory cells for all variables used in the program, without knowledge of the user and forms a table containing variable names and their address, called hash table. Thus during the execution of a program computer will determine the address of a variable from hash table and then finds the value of the variable from the located address. Therefore, with each variable two quantities are associated - one is the value of the variable and other is the address of the memory cell where the value is stored. If a variable needs one or more cells to store it, then the address of the variable means the address of the first cell. The value stored in a cell called its content. In C, one can easily determines the address of a variable. The technique and related topics regarding the address of a variable are discussed in the next section.

2.9.1 Address Operator

Suppose *v* is a variable. The address of this variable is obtained by the expression

```
&v
```

where & is a unary operator, called the address operator and read as address of. The address of a variable can store to another variable. For example, the statement

```
pv = &v;
```

stores the address of the variable v to the variable pv. This variable pv is called a pointer to v, as it points to the location where v is stored. The variable pv is called pointer variable. It may be noted that pv stores its address not its value. The content of a memory cell, i.e. the value (of v) stores in a memory cell is obtained by the expression

```
*pv
```

where * is a unary operator, called the indirection operation. Thus, if pv is a pointer variable then pv gives the address of a variable and *pv gives the value stored it in.

The following program illustrates the use of address operator.

```
#include<stdio.h>
void main()
 {
   int i=100;
   float a=7.5;
```

```
    char c='b';
    printf("Address of i=%x and its value =%d\n",&i,i);
    printf("Address of a=%x and its value =%f\n",&a,a);
    printf("Address of c=%x and its value =%c\n",&c,c);
}
```

The output of this program is

```
Address of i=fff4 and its value =100
Address of a=fff0 and its value =7.500000
Address of c=ffef and its value =b
```

Note that the addresses of the variables i, a and c may be changed for other computer and different instances. Because, the addresses printed above are available for a particular moment for a particular computer. The free addresses depend on many factors, such as, size of operating system, other programs stored in the memory, the data and program sizes of current program, etc. The values 100, 7.500000, b are the contents of the memory address `fff4`, `fff0`, `ffef` respectively. The contents and addresses are shown in Fig. 2.5.

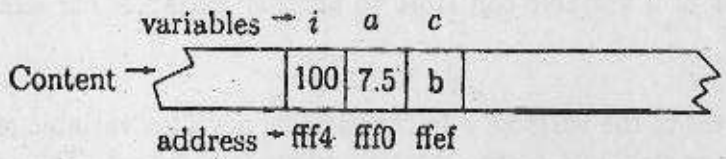


Figure 2.5: Addresses and contents of variables.

2.9.2 Pointer Declaration

Like other variables, each pointer variable must be declared before its use. The rules for declaration of a pointer variable are similar to ordinary variable. The difference is that, when a pointer variable is declared, the variable name must be preceded by an asterisk (*). The asterisk makes the difference between the ordinary variable and the pointer variable. The general form of pointer declaration is

```
data_type *ptvar;
```

where ptvar is the name of the pointer variable and data_type is any one of the C data types like `int`, `long`, `char`, `float`, etc. Note that ptvar is the pointer, its value is the address of certain memory location and it is a hexadecimal

number. Whereas *ptvar is the value stored at the address ptvar. So the value of *ptvar is either an int or long or char and so on, which is same as data_type.

Let us consider the following program to illustrate the use of pointer variable.

```
#include<stdio.h>
main()
 {
   int *ptvar; /* pointer declaration */
   int u, v;
   u=10;
   v=50;
   printf("Value of ptvar=%X\n",ptvar);
   ptvar=&u; /* address of u is assigned to the pointer ptvar */
   printf("Value of ptvar=%X and its content=%d\n",ptvar,*ptvar);
   ptvar=&v; /* address of v is assigned to the pointer ptvar. */
   printf("Value of ptvar=%X and its content=%d\n",ptvar,*ptvar);
   *ptvar=500; /* 500 is assigned as content of ptvar */
   printf("Value of ptvar=%X and its content=%d\n",ptvar,*ptvar);
 }
```

The output of this program is

```
Value of ptvar=3C2
Value of ptvar=FFF4 and its content=10
Value of ptvar=FFF2 and its content=50
Value of ptvar=FFF2 and its content=500
```

Initially, the value of u and v are 10 and 50 respectively and the values of u and v are stored in some memory locations. The value of ptvar is 3C2 (in the current execution, it may be changed in another execution). The statement ptvar=&u copied the address of u to the pointer variable ptvar and it is changed to FFF4. And the subsequent printf function prints the content of the address FFF4, i.e. the value of u. The statement ptvar=&v assigns the address of v to the pointer variable ptvar and hence the printf function prints the contents of ptvar (FFF2) which is 50. The statement *ptvar=500 assigns the numeric value 500 to the content of ptvar, does not change the value of ptvar. The value of ptvar remains same (FFF2), see the last print statement.

The pointer variables and ordinary variables can be declared together, i.e. the following statements are valid.

```
int a, b, *pv, *pu;
float *a, *b, x, y;
```

Note that the initialization of pointer is a very sensitive task. Arbitrary initialization may cause the serious error in the program, as the value of a pointer variable is unpredictable. it depends on computer, execution instant, etc. However, we can assign null pointer as

```
#define NULL 0
float *ptvar=NULL;
```

The variable ptvar is initially assigned a value 0 to indicate some special task. The arithmetic operations can also be done on pointer variables.

```
int *pv, v=10, u=5;
pv=&v; /* address of v is assigned to pv,
          thus the value of *pv becomes 10 */
*pv+=2; /* the value of *pv changed to 10+2=12 */
v=u* *pv; /* the value of v=5*12=60 */
v=*pv * 8; /* v=480 */
```

The value of *pv after second line becomes 10 (=v). In third line the content of pv, i.e. the value of *pv changed to 12. The value of v (in the fourth line) is 5 * 12 = 60. That is, the content of &v is also changed to 60. Thus, the content of pv is changed to 60. That is, the value of *pv is now 60 (not 12). Therefore, the statement v=*pv*8 changes the value of v to 480.

2.9.3 Void pointer

It is seen that a pointer variable can hold the address of same type of data. That is, an integer pointer variable can hold only the address of an integer variable, it does not hold the address of any other type of variables. This fact is illustrated in the following.

```
int *ipv;
float x;
ipv=&x; /* illegal assignment */
```

The assignment ipv=&x is not valid as ipv is an integer pointer variable whereas x is a floating point variable. To store the value of x needs four bytes space, while two bytes space is allocated for ipv.

The void pointer eliminates this restriction. If a pointer variable is declared as void, then any type of pointer assignment can be done with this variable. For example,

```
void *vptr;
int i;
char c;
float f;
vptr=&i;
vptr=&c;
vptr=&f;
```

All these assignments are valid. But, when arithmetic operations are done on the contents of pointer variables, it must be suitably type cast to the required data type.

2.9.4 Passing Pointers to a Function

Like ordinary variables, pointers can also be passed to a function as arguments, i.e. an address is passed as arguments. This process is referred as passing arguments by reference or by address. When an argument is passed by value to a function, then the actual argument is copied to the formal argument. If the values of this formal argument is changed within the function it does not alter the value of the actual argument as formal arguments are local within the function. But, when an argument is passed by reference, then the address of the variable is passed to the function. Since the address of a memory location is unique, so the content of the passing address can be accessed from any place (from the function and also from the calling function). Thus, any change in the content of an address can be identified from any place, i.e. the content of an data item are changed globally.

In formal pointer arguments declaration, each argument must be preceded by an asterisk. It is obvious that the actual arguments are the addresses of the variables.

The following example illustrates the use of transfer of arguments by reference.

Example 2.9.1 Write a program to interchange the values of two variables using pointers.

Solution. A function in C can not return more than one value. But, to interchange the values of two variables using function we have to return two values, it is not possible by passing values. This can be done by passing reference, shown below.

```
/* Program to interchange two values using pointer */
#include<stdio.h>
main()
  {
    float a,b;
    void swap(float *, float*); /* declaration of function */
    printf("Enter two values\n");
    scanf("%f%f",&a,&b);
    printf("The given values of a=%f b=%f\n",a,b);
    swap(&a,&b);
    printf("The changed values are a=%f b=%f\n",a,b);
    return;
  }
  void swap(float *u, float *v)
    {
      float *temp;
      *temp=*u;
      *u=*v;
      *v=*temp;
    }
```

The input/output of this program is shown below.

```
Enter two values
89.44999 456.32
The given values of a=89.449990 b=456.320000
The changed values are a=456.320000 b=89.449990
```

In the function swap all the variables u, v and temp are pointer. The addresses of a and b are passed to swap and they are copied as u and v respectively. Then the contents of u and v are interchanged. Then automatically the values of a and b are changed.

Passing array to a function

It is already discussed how an array can be transferred to a function. Since the array name of an array itself a pointer, the array name is sufficient to use as actual argument, no ampersand sign is required preceding array name. Thus, if a is an array of size n, then this array can be transferred to the function sum by the statement

```
sum(a);
```

In this call the address of the base element, i.e. address of the element a[0] is passed as reference to the function, and the entire array is passed to the function. The first line of the function definition must be of the form

```
float sum(float *x);
```

Also, one can pass a portion of an array, rather entire array to a function by passing the address of appropriate array element. Suppose a is an array of size 50 and if we pass a[20] to a[40] (last element) to the function sum, then the following call is necessary.

```
sum(&a[20]);
```

In this call the elements a[20] to a[49] are available for the function sum. Here noted that the address of the element a[20] is passed to the function.

A function can also return a pointer. This can be done by appropriate function declaration.

2.9.5 Pointers and One-Dimensional Array

Now, we extent the idea of arithmetic on pointers to array. Recall that the address of the first element of an array is the address of the entire array. Thus, if a is an integer array (one-dimension), then the address of the first element is given by either &a[0] or by simply a. The address of the second element is obtain by &a[1] or by a+1. In general, the address of the i+1th element of the array a can be determined either from the expression &a[i] or from the expression a+i, i is called the offset.

One important point is that the array elements are always stored in contiguous memory locations irrespective of the size of the array. That is, if the address of a[0] (a is an integer array) is FFF0, then the address of the element a[1] is FFF2, that of a[2] is FFF4, and so on. In general, a pointer when incremented, always points to a location after skipping the number of bytes required for the data type pointed to by it.

The following program illustrates the above fact.

```
#include<stdio.h>
main()
 {
   int a[]={10,12,8,5};
   int i, *pa;
   pa=&a[0]; /* address of the array is assigned to pa */
   for(i=0;i<4;i++)
```

```
    {
      printf("%X %d %X \n",pa,*pa,a+i);
      pa++; /* pointer incremented */
    }
}
```

The output of this program is

```
FFEE    10    FFEE
FFF0    12    FFF0
FFF2    8     FFF2
FFF4    5     FFF4
```

Note that the output. The first and third columns are identical. This means, the address of an array element can be obtained by incrementing pointer variable and also by simple addition of the position (i.e. subscript) of the array element with array name. The status of the memory, its contents and array elements are shown in Fig. 2.6.

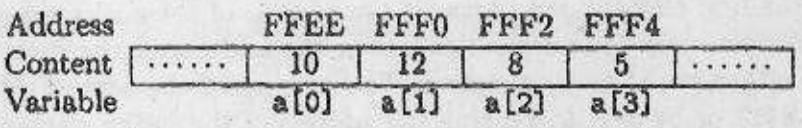


Figure 2.6: The array elements, corresponding address and the contents.

It is very interesting that, if the address of the first element of the array a is FFF0 (say), then the value of the expression a+2 is not necessarily FFF2, it may be FFF4 or FFF8 or something else depends on the data type of the array a. The value of the expression a+i will automatically determined by the C compiler, without the knowledge of user regarding the number of memory cells required for different data types.

Thus &a[i] and (a+i) both represent the same memory address for the i+1th element of the array a, and a[i] and *(a+i) both represent the contents (i.e. the value of the i + 1th element) of the same address. These two terms are interchangeable.

The following program illustrates the relationship between the array elements and their address.

```
#include<stdio.h>
main()
```

```
{
   int  a[5]={10,12,20,25,30};
   int i;
   for(i=0;i<5;i++)
     {
       /* printing of values */
       printf("a[%d]=%d  *(a+%d)=%d",i,a[i],i,*(a+i));
       /* printing of address */
       printf("&a[%d]=%X  (a+%d)=%X\n",i,&a[i],i,(a+i));
       pa++;  /* pointer incremented */
     }
}
```

The output of the above program is shown below.

```
a[0]=10   *(a+0)=10     &a[0]=FFEC      (a+0)=FFEC
a[1]=12   *(a+1)=12     &a[1]=FFEE      (a+1)=FFEE
a[2]=20   *(a+2)=20     &a[2]=FFE0      (a+2)=FFE0
a[3]=25   *(a+3)=25     &a[3]=FFE2      (a+3)=FFE2
a[4]=30   *(a+4)=30     &a[4]=FFE4      (a+4)=FFE4
```

From this output it is observed that a[i] and &a[i] are different, first one represent the value and second one represent the address of the $i+1$th element. Similarly, *(a+i) and (a+i) are also different, *(a+i) represents the value and a+i represents the address of the $i+1$th element. Thus, a[i] and *(a+i) are same and &a[i] and (a+i) are same.

It is sometimes necessary to assign an address to a variable, but, it is not possible to assign an arbitrary address to an array name or to an array element. Also, the address of one array element cannot be assigned to some other array element, i.e. the statement &a[2]=&a[1] is not valid.

Example 2.9.2 Write a program to add two arrays using pointer.

Solution. Let a and b be two array of size 10. We add these two arrays and store it to the array c.

```
/* Addition of two arrays using pointers */
#include<stdio.h>
void main()
```

```
    int a[10],b[10],c[10];
    int i,n;
    void sum(int*,int*,int*,int);
    printf("Enter array size\n");
    scanf("%d",&n);
    printf("Enter elements of the first array\n");
    for(i=0;i<n;i++) scanf("%d",&a[i]);
    printf("Enter elements of the second array\n");
    for(i=0;i<n;i++) scanf("%d",&b[i]);
    sum(a,b,c,n);
    printf("The sum of two arrays\n");
    for(i=0;i<n;i++) printf("%d ",c[i]);
  }
void sum(int *a,int *b,int *c, int n)
  {
    int i;
    for(i=0;i<n;i++)
    *(c+i)=*(a+i)+ *(b+i); /* addition of pointers */
  }
```

A sample of input/output:

```
Enter array size
5
Enter elements of the first array
3 7 9 12 6
Enter elements of the second array
-4 2 6 -2 9
The sum of two arrays -1 9 15 10 15
```

In the function sum no formal array subscript is used to find the sum of two elements. The contents of the addresses of the elements are added.

2.9.6 Dynamic Memory Allocation

In conventional array declaration, the size of an array must be mentioned. For this declaration a fixed amount of memory is reserved for this array. For example, for the declaration

```
int a[10];
```

total ($10 \times 2 =$) 20 bytes memory space is reserved for the array a. Also, at most 10 values can be stored against a. Suppose in a program the array size is

declared as large enough say 100 and only 40 elements are used in an execution. In this case, the space for remaining (100 - 40 =) 60 elements is unnecessary occupied. This is the wastage of memory.

These two draw backs to use array are removed in dynamic memory allocation. In dynamic allocation an array is define as a pointer variable. When an array is define as a pointer variable, then the space for the array is not fixed, it can be changed as per requirement. A special declaration is required to allocate the space for the array. This is called **dynamic memory allocation.** The library function `malloc` is generally used for this purpose.

Suppose a is a one-dimensional integer array containing 15 elements. The conventional method to define a as an array is

```
int a[15];
```

But, a can be defined as a pointer variable as

```
int *a;
```

This type of declaration does not mean a is an array, it look likes an ordinary integer pointer variable. Therefore, this statement does not allocate a memory block for 15 elements. To assign appropriate memory for a, the following statement is necessary.

```
a=(int *) malloc (15*sizeof(int));
```

The function malloc allocates a block of memory for 15 integers and the allocated space is (15 $\times$ 2 =) 30 bytes (if the size of an integer is 2). This function returns a pointer which points to the first element of the array a (i.e. the address of `a[0]`). The cast operator (`int *`) is required, because this pointer points to an integer array.

If a is an floating point array then the above statement must be of the following form.

```
a=(float *) malloc (15*sizeof(float));
```

If the `malloc` function returns 0, i.e. NULL pointer, then memory allocation is failed. This happens when the sufficient amount of space is not available in the memory.

If an initialization is required for an array, then the array must be defined as an ordinary array rather than a pointer variable.

Example 2.9.3 Write a program to read a string of characters (only alphabets and blank spaces) and count the number of blank spaces, vowels and consonants in the string.

Solution. Suppose x is an array containing n characters. The array is defined as a pointer variable. The space for this character array is allocated by the function `malloc`.

```
/* Use of pointer in a string of characters */
#include<stdio.h>
void main()
 {
   char *x,c;
   int  n,i,vcount=0,bcount=0,ccount=0;
   printf("Enter string size\n");
   scanf("%d",&n);
   x=(char *) malloc (n*sizeof(char)); /* allocation of memory */
   printf("Enter the string\n");
   for(i=0;i<=n;i++)
     *(x+i)=getchar();
   printf("Given string is\n");
   for(i=1;i<=n;i++) printf("%c",*(x+i));
   for(i=1;i<=n;i++)
     {
       c=toupper(*(x+i)); /* converted to upper case */
       if(c==' ') bcount++;
       if(c=='A'||c=='E'||c=='O'||c=='U'||c=='I') vcount++;
     }
   ccount=n-vcount-bcount;
   printf("\nBlank space=%d Vowel=%d Consonent=%d\n",
                                   bcount,vcount,ccount);
 }
```

A sample of input/output:

```
Enter string size
18
Enter the string
I read in class iv

Given string is
I read in class iv
Blank space=4 Vowel=6 Consonent=8
```

The space for the array x of size n is allocated by

```
x=(char *) malloc (n*sizeof(char));
```

A character is read by getchar() and it stores at the address (x+i). To check the type of a character, first it is converted to the upper case letter by the function toupper. Then this character is checked for vowel and blank space.

The memory allocated for a pointer array can be destroyed (if it is not required in future) by the function free. The occupied memory becomes free and this free space may be used within the same program, if needed. Let x be a pointer array. The following statement will free the space occupied for the array x.

```
free(x);
```

Exercise 2.9

1. What is meant by 'the address of a memory cell'? How are addresses usually numbered?
2. What kind of information is represented by a pointer variable?
3. What is the relationship between the address of a variable *v* and the corresponding pointer variable *pv*?
4. How many memory cells are required to store a single character? An integer? A long integer? A floating-point number?
5. How is a variable's address determine?
6. How is a pointer variable declared? What is the purpose of the data type included in the declaration?
7. What is the relationship between an array name and a pointer? How is an array name interpreted when it appears as an argument to a function?
8. In what way can the assignment of an initial value be included in the declaration of a pointer variable?
9. What is the purpose of the indirection operator? To what type of operand must the indirection operator be applied?
10. Explain the meaning of each of the following declarations.

    (a)
    ```
    int *p;
    ```
    (b)
    ```
    float a,b;
    float *pa,*pb;
    ```
    (c)
    ```
    float a=1.45;
    float *pa=&a;
    ```
11. Write a function which will interchange the values between two floating point numbers. Demonstrate this function by calling this function from main. Pointer may be used.

2.10 String Manipulation

Manipulation of strings is a very common problem in any programming language. Different techniques and functions are used in manipulation of strings. C provides several functions which are used to manipulation of strings. The common string manipulation functions are listed below.

Function	Action
`strlen(str1)`	determines the length of the string str1
`strcat(str1,str2)`	concatenates (joins) two strings str1 and str2
`strcmp(str1,str2)`	compares two strings str1 and str2
`strcpy(str1,str2)`	copies the string str2 over the string str1

The details of the string manipulation functions are explain below.

`strlen()`

This function determines the length of a string, i.e. counts the number of characters presents in the string excluding null character. The general form of this function is

```
strlen(str1);
```

where `str1` is a string constant or a string variable. This function returns an non-negative integer. For example, let `str1="Good"` and `str2="Good"`. Then the output of the execution of the functions strlen(str1) and strlen(str2) are respectively 4 and 5.

`strcat()`

This function concatenates two strings. The general form of this function is

```
strcat(str1,str2);
```

where `str1` and `str2` are two strings. This function joins the string `str2` to the end of the string `str1`. This function removes the null character from the end of the string `str1` and `str2` is added to str1 from this point. The string `str1` remains unchanged. To illustrate the use of this function, consider the following strings

```
str1="Good"
str2="Day"
str3="Good"
```

The strings after execution of the function

```
strcat(str1,str2);
```

are str1="GoodDay" and str2="Day" and the strings after execution of the function

```
strcat(str3,str2);
```

are str3="Good Day" and str2="Day".

`strcmp()`

This function compares two strings and returns an integer value. If the two strings are identical it returns 0, otherwise returns a non-zero integer value. Its general form is

```
strcmp(str1,str2);
```

where str1 and str2 are two string variables or constants.

Two strings are compared with respect to the ASCII values of the individual characters of the strings. Since the ASCII values of upper case and lower case alphabets are different. therefore the strings str1="aBc" and str2="abc" are different. The ASCII value of a is 97 and that of B and b are respectively 66 and 98. In these strings, the ASCII codes of first characters viz. a are same, but the ASCII codes of second characters are 66 and 98. Then "aBc" comes before "abc" in dictionary order. It may be noted that the value of the function strcmp("aBc","abc") is 66 - 98 = -32. Thus the negative value of the function strcmp(str1,str2) indicates that the string str1 comes before str2 with respect to dictionary order.

`strcpy()`

This function is used to assign a string (constant or variable or expression) to another string. Its general form is

```
strcpy(str1,str2);
```

where str1 is a string variable and str2 is a string constant, variable or an expression. The value of str2 is assigned to the string str1.

That is, if str1 and str2 are "Ram" and "India" respectively, then the execution of the function

```
Strcpy(str1,str2);
```

results str1="India" and str2="India". It may be noted that str1="India" is not a valid assignment statement. If any string constant or variable or expression is to be assign to another string variable, then the function strcpy must be used.

Example 2.10.1 Write a program to arrange the names of some students in alphabetic order.

```
/* Sorting of names in alphabetic order */
#include<stdio.h>
#include<string.h>
void main()
 {
    char name[10][25], temp[25];
    int i,j,n;
    printf("Enter number of students\n");
    scanf("%d",&n);
    printf("Enter names line by line \n");
    for(i=1;i<=n;i++) /* reading of names */
      scanf("%[^\n]",name[i]);
    /* sorting begins */
    for(i=1;i<=n-1;i++)
    for(j=i+1;j<=n;j++)
      {
        if(strcmp(name[i],name[j])>0)
          { /* interchange between the strings */
            strcpy(temp,name[i]);
            strcpy(name[i],name[j]);
            strcpy(name[j],temp);
          }
      } /* end of sorting */
    printf("\nSorting list is\n");
    for(i=1;i<=n;i++)
      printf("%s\n",name[i]);
 }
```

A sample of input/output:

```
Enter number of students
6
Enter names line by line
Raja
Aniket
Pratik
Kajal
```

Rani
Moni

```
Sorting list is
Aniket
Kajal
Moni
Pratik
Raja
Rani
```

Counting the number of words in a string

The following program reads a string of characters (up to 50) as st and counts the number of words in it. A word is a sequence of consecutive characters (containing alphabets, digits or any special character) without blank space.

```
#include<stdio.h>
void main()
{
 char st[30];
 int k=1,i,l;
 printf("Enter the string\n");
 gets(st);
 l=strlen(st); /* determines length of the string */
 for(i=1;i<l;i++)
   if(st[i-1]==' ' && st[i] !=' ') k++;
 if(st[0]==' ') k--; /* if leading blank */
 printf("No. of words in the string is %d ",k);
}
```

A sample of input/output:

```
Enter the string
I am a very good student
No. of words in the string is 6
```

Rewrite the name in short form

The following program reads the full name (up to 50 characters) of a person in the usual form (*i.e.* in the form GURU PRASANNA BHATTACHARJEE) and rewrite it in short form (*i.e.* in the form G.P.BHATTACHARJEE).

```
#include<stdio.h>
#include<string.h>
void main()
{
 char st[50], t[40]; /* st is the given string, t is the short form */
 int k=0,i,l,m=0;
 printf("Enter the string\n");
 gets(st);
 l=strlen(st);
 t[0]=st[0]; /* first character puts to t*/
 for(i=1;i<=l;i++) /* finds other blank position */
   if(st[i-1]==' ' && st[i] !=' ')
     {
      k+=2;
      t[k-1]='.';
      t[k]=st[i];
      m=i; /* m is the last blank position within the string */
     }
 /* finds surname */
 if(m==0) k=0;
 for(i=m+1;i<=l;i++)
   t[++k]=st[i];
 printf("The short name is\n");
 puts(t);
}
```

A sample of input/output:

```
Enter the string
Guru Prasanna Bhattacharjee
The short name is
G.P.Bhattacharjee
```

Rewrite the name with their surname first

The following program reads the name of a person (e.g. Nitya Lal Saha) and then prints its surname first and the other parts of the name at the end, i.e. in the form Saha Nitya Lal.

```
#include<stdio.h>
#include<string.h>
```

```
void main()
{
  char st[50];
  int k=0,i,l;
  printf("Enter the string\n");
  gets(st);
  l=strlen(st);
  /* determines last blank position */
  for(i=1;i<=l;i++)
    if(st[i-1]==' ' && st[i] !=' ') k=i;
  printf("The change name is\n");
  if(k==0)
    printf("%s",st);
  else
    {
      for(i=k;i<=l;i++) printf("%c",st[i]); /* prints surname */
      for(i=0;i<k;i++) printf("%c",st[i]); /* prints other parts */
    }
}
```

A sample of input/output:

```
Enter the string
Nitya Lal Saha
The change name is
Saha Nitya Lal
```

2.11 Structures and Unions

The structure is a very important and useful topic in C language. A group of data are binding together to form a single data with the help of structure. A single structure may contains different types of data, viz., `int`, `float`, `char`, etc. Other types of data such as array, pointer even another structure can also be included in a single structure. With the help of structure one can define a new type of data and their associated functions. The individual elements are called members of the structure.

The union also contains multiple members and it is closely associated with the structure. The members of union share the same memory location, even they are in different by type. The advantage of union is that it can store different data items in same portion of the memory in different instances.

2.11.1 Definition of Structure

Several data items of different types are grouped into a structure. The general form to define a structure is

```
struct  struct_name
 {
   data_type  member1;
   data_type  member2;
   . . . . . .
   data_type  membern;
 };
```

Here `struct` is the keyword to declare a structure, `struct_name` is the name of the structure called the tag (this is user defined), `member1, member2, ...., membern` are *n* different members (variables) called members of the `struct_name`.

It may be remembered that definition and declaration of structure are same. Note that the semicolon (;) after closing brace (}).

The members of a structure may be ordinary variables (of type `int, char, float`, etc.) arrays, pointers and also other structures. But, storage class can not be assigned to an individual member and no member can be initialize within a structure. If a structure includes another structure, its name must be distinct, but member name may be same.

Suppose a class contains, say, 100 students and an institute would like to process the student records. For simplicity, we assume that the institute will store only roll number, name, sex and height of each student for their record. A structure for these fields can be defined as

```
struct  student
  {
    int  roll_no;      /* integer variable */
    char name[30];     /* an array of char */
    char sex;          /* a character variable /*
    float height       /* a floating point variable */
  }
```

Here the structure name is student and the members are `roll_no, name` (30-element array), sex and height. The diagrammatic representation is shown in Fig. 2.7.

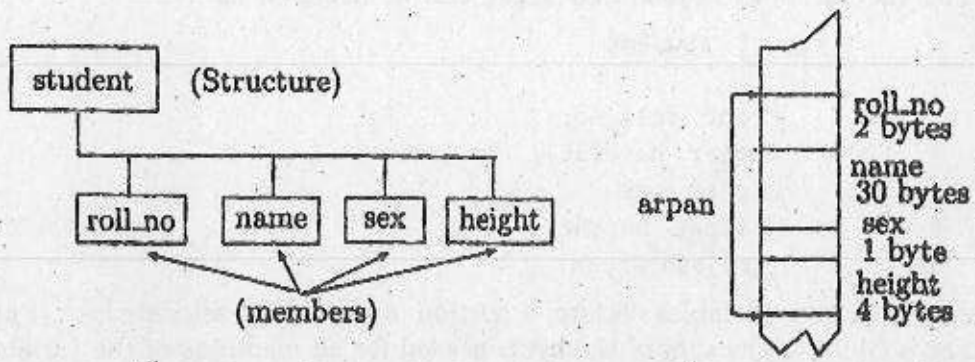


Figure 2.7: Memory allocation for structure variable

By this declaration, the structure student becomes a new type of data and one can define variables of type student. The following statement declares two variables rajesh and arpan of type student.

```
struct student rajesh, arpan;
```

Note that the keyword struct must be present during the declaration and rajesh and arpan become two structure type variables.

The general form to declare structure type variables is

```
storage_class struct struct_name var1,var2, ..., varn;
```

where storage_class represents storage class specifier and it is optional, struct is a keyword, var1, var2,..., varn are n variables of type struct_name.

The declaration of structure type variables and the definition of structure are written together as

```
storege_class struct struct_name
  {
    data_type member1;
    data_type member2;
    . . . . . .
    data_type membern;
  } var1,var2,...,varm;
```

In this declaration, the m variables var1, var2, ..., varm are declared together and their type is struct_name, i.e. each variable contains n data items member1, member2, ..., membern. The m variables are separated by commas and the declaration ends with semicolon. The storage_class is optional. Also, struct_name is optional for this type declaration.

Thus the variables rajesh and arpan can be declared as

```
struct student
  {
     int roll_no;
     char name[30];
     char sex;
     float height;
  } rajesh,arpan;
```

When a structure variables declare, a portion of memory is allocated for it and its size is equal to the sum of the bytes needed for all members of the variable. For example, 37 bytes (2 bytes for roll_no, 30 bytes for name, 1 byte for sex and 4 bytes for height) memory space will allocate for the variable arpan and another 37 bytes is also allocate for rajesh.

2.11.2 Accessing a Structure

Though a structure grouped a number of data together even then an individual element can be accessed separately, i.e. an individual element can be read, write, update, etc. separately. An individual member of a structure is access by

```
variable.member
```

where variable is a structure type variable (not structure name) and member is a member which is defined within the structure. The dot(.) separates the variable name and the member name. This is *an operator in C. its precedence is high and associativity is left-to-right.*

Now, consider the previous structure student. The member roll_no of the variable arpan is written as arpan.roll_no.

Similarly, arpan.name represents the name of arpan, etc. The input and output of a structure variable can be done by reading and writing all members individually. For example, the following program segment is used to read the (structure) variable arpan.

```
scanf("%d%s%c%f",&arpan.roll_no,arpan.name,&arpan.sex,&arpan.height);
```

Similarly, the necessary output statements to print the (structure) variable arpan are

```
printf("The record for arpan \n");
printf("Roll No. is     : \%d\n ",arpan.roll_no);
printf("Name is         :.\%s\n ",arpan.name);
printf("Sex is          : \%c\n ",arpan.sex);
printf("Height is       : \%6.2f\n ",arpan.height);
```

Notice that the expressions &arpan.roll_no and &(arpan.roll_no) are identical. Because, the precedence of dot(.) is higher compare to the unary operators and arithmetic, relational, logical and assignment operators. Thus, the value of the expressions ++arpan.roll_no and ++(arpan.roll_no) are same. First the value of arpan.roll_no is determine and then it will be incremented. This is a meaningful expression. Obviously, ++arpan does not give any value as arpan is a group of data and it is an erroneous expression. A particular element, say 5th, of the member name (it is an array) can be accessed as arpan.name[5]. It gives the fifth character of the member name of the structure variable arpan.

2.11.3 Nested Structure

A structure can be defined within another structure. This process is called **nesting of structure.**

Now, we add the date of birth of the students to the structure student. A date has three parts-day, month and year, and a structure for it can be defined as

```
struct date
  {
    int day;
    int month;
    int year;
  };
```

Now we include this structure to the structure student. The necessary declaration is

```
struct student
   {
     int roll_no;
     char name[30];
     char sex;
     float height;
     struct date date_of_birth;
   };
```

Here the structure date is nested within the structure student. The structure student contain another structure date and this new structure.contains more information than the old one.

If arpan is a variable of this new structure student, then the three individual members of date_of_birth can be accessed by writing

arpan.date_of_birth.day, arpan.date_of_birth.month,
arpan.date_of_birth.year

These members behave as ordinary variables and one can apply any unary, binary or any other operations as required. The memory allocation for the structure variable arpan is shown in Fig. 2.8.

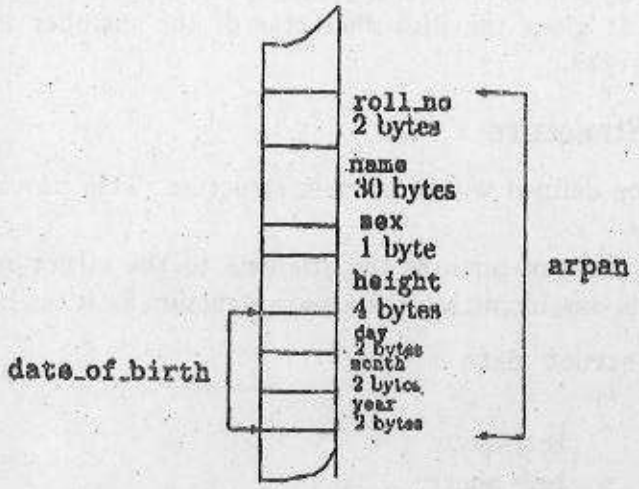


Figure 2.8: Memory allocation for the variable arpan.

In general, a member of nested structure can be accessed as

variable.outer_memebr.inner_member

where variable is a structure type variable, outer_memebr is the direct member of the structure and inner_member is the member of the structure defined within the structure.

The above nested structure can also be defined as

```
struct student
  {
    int roll_no;
    char name[30];
    char sex;
    struct date
      {
        int day;
```

```
        int month;
        int year;
    } dob;
};
```

2.11.4 Initialization of Structure Variable

Like ordinary variables, the members of a structure variable can also be initialized. The general form is

```
storege_class struct struct_name
        variable={value1, value2, ..., valuen};
```

where variable is a variable of type struct_name. The quantities value1, value2, ..., valuen are the values of the first, second, and so on members of the variable. The initial values of (value1, value2,..., valuen) must appear in the order in which they are declared within the structure. A structure variable can be initialize only if its storage class is either external or static.

Suppose arpan is a variable of the modified structure student (containing the structure variable data_of_birth). The assignment of initial value can be done by the statement

```
static struct student arpan={777,"Arpan Pal",'m',48,6,7,2001};
```

This declaration initializes the members as follow.

```
arpan.roll no=777;
arpan.name="Arpan Pal";
arpan.sex='m';
arpan.height=48;
arpan.date_of_birth.day=6;
arpan.date_of_birth.month=7;
arpan.date_of_birth.year=2001;
```

That is the date of birth is 6 July 2001. The above statements can also be used to initialize the variable arpan.

2.11.5 Array of Structure

Since structure variable is a new type (user define) of data, like integer array one can define array of structure variable. Each element of the array is a structure. The general form to declare such array is

```
struct student cs_student[50];
```

In this declaration cs_student is an array of 50 elements and each element is a structure of type student. This array can also be defined as

```
struct date                    struct student
  {                              {
    int day;                       int roll_no;
    int month;                     char name[30];
    int year;                      char sex;
  };                                   float height;
                                   struct date date_of_birth;
                                 }cs_student[50];
```

The date of (i + 1)th cs_student can be accessed as

```
cs_student[i].roll_no,
cs_student[i].name,
cs_student[i].sex,
cs_student[i].height,
cs_student[i].date_of_birth.day,
cs_student[i].date_of_birth.month,
cs_student[i].date_of_birth.year,
```

These represent respectively the roll_no, name, sex, height and date of birth (day, month, year). A particular character, say 5th, of the member name of the 10th student is obtained from the expression

```
cs_student[9].name[4];
```

Initialization of array of structure

An array of structures can be initialize in the same way as a single structure. The process is very simple and illustrated by the following example.

```
struct date                    struct student
  {                              {
    int day;                       int roll_no;
    int month;                     char name[30];
    int year;                      char sex;
  };                               float height;
                                   struct date dob;
                                 }cs_student[50];
```

The following statement initializes the records of 5 students.

```
struct student cs_student[5]=
  {
     777,"Aniket  Pal",'m',48,6,7,2001,
     555,"Sanjay  Roy",'m',50,7,2,2002,
     333,"Sonia  Gandhi",'f',58,15,3,2001,
     333,"Asha  Khan",'f',78,12,12,2003,
     222,"R.K.Sharma",'m',57,10,5,2002
  };
```

Here cs_student is an array of 5 elements of data type student, cs_student[0] will be assigned the first set of values, cs_student[1] the second set of values and so on. Here the 5 sets of data are written in five different lines for clear understanding, but, these values can be written as a single line also. Again, values for a member can be enclosed within braces. This representation is more better than other forms. That is, the above initialization can be rewritten as

```
struct student cs_student[5]=
  {
     {777,"Aniket  Pal",'m',48,6,7,2001},
     {555,"Sanjay  Roy",'m',50,7,2,2002},
     {333,"Sonia  Gandhi",'f',58,15,3,2001},
     {333,"Asha  Khan",'f',78,12,12,2003},
     {222,"R.K.Sharma",'m',57,10,5,2002}
  };
```

Now, we consider an example to explain the advantage of structure data.

Example 2.11.1 (Sorting of record) Suppose the roll number, name and total marks of some students are known. Write a program to read the roll number, name and mark of all students and prepare a merit list for them (i.e. sort the records in descending order of marks).

```
/* Sorting of record w.r.t. to descending order of marks */
#include<stdio.h>
void main()
 {
   struct student
     {
       int roll_no;
```

```
    char name[15];
    int mark;
  };
struct student cs_student[40], temp;
int i,j,n;
printf("How many students \n");
scanf("%d",&n);
/* reading of records */
printf("Enter the records\n");
for(i=0;i<n;i++)
  {
    scanf("%d",&cs_student[i].roll_no);
    scanf("%s",cs_student[i].name);
    scanf("%d",&cs_student[i].mark);
  }
/* sorting of records */
for(i=0;i<=n-2;i++)
  for(j=i+1;j<=n-1;j++)
    {
      if(cs_student[i].mark<cs_student[j].mark)
        {
          /* interchange between two records */
          temp=cs_student[i];
          cs_student[i]=cs_student[j];
          cs_student[j]=temp;
        }
    }
/* printing of records */
printf("Sorted record is\n");
for(i=0;i<n;i++)
  printf("%4d %-15s %5d\n",cs_student[i].roll_no,
         cs_student[i].name,cs_student[i].mark);
}
```

A sample of input/output:

```
How many students
6
Enter the records
11 Aniket 90
```

22 Rahul 34
33 Sanjib 50
44 Tapan 60
55 Kalyani 35
66 Monoranjan 47

```
Sorted record is
  11   Aniket        90
  44   Tapan         60
  33   Sanjib        50
  66   Monoranjan    47
  55   Kalyani       35
  22   Rahul         34
```

A structure named student is define having three data members roll number, name and mark. An array cs_student of size 40 is declared whose type is student. The records for all students are read by scanf functions. The sorting of records with respect to mark is done by exchange sorting technique. The new thing is that the interchange between two records. The statement cs_student[i]=cs_student[j] copies all the members of the $(j+1)$th element of the array cs student to the corresponding members of the $(i+1)$th element of the same array, i.e. entire structure is copied by this single statement. This statement is equivalent to the following three statements

```
cs_student[i].roll_no=cs_student[j].roll_no;
cs_student[i].name=cs_student[j].name;
cs_student[i].maek=cs_student[j].mark;
```

Instead of these three statements only one statement is used to copied a structure and this is the main advantage to use structure.

2.11.6 User Defined Data Types

Using the concept of structure we have seen that a new data type can be defined in C. But, to declare a variable of structure type the keyword struct is written before the structure name. This can be removed by using the keyword typedef. Using this keyword a new data type can be defined that are equivalent to the standard data type. Once a new data type is defined, the ordinary variables, arrays, pointers, etc. can be declared like standard data type.

The general form to define new data type is

```
typedef data_type new_data_type;
```

where data_type is either a standard data type or other user defined data type, new_data_type is the name of new data type and typedef is the required keyword.

For example,

```
typedef int mark;
```

defines a new data type mark which is basically an integer. Now, one can declare the variables of type mark as follows:

```
mark phy_mark, chem_mark;
```

Thus like integer (or any other standard) variable one can define variables of type mark. Here phy_mark and chem_mark are two variables of type mark, these are actually of type integer because type of mark is int.

The following declaration is interesting.

```
typedef char string[50];
string name, address;
```

The first line defines a data type string which is a 50-element array. The second line declares two variables name and address of type string. Since string is a 50-element array of type char, therefore name and address are also arrays of length 50 of type char. This type of data are useful in some particular applications.

The above definition is equivalent to the following one.

```
typedef char string;
string name[50], address[50];
```

The feature of typedef is significant when it is used in structure. A new data type which contains multiple members can be defined with the help of typedef and structure.

The following syntax is used to define a new data type whose base is a structure

```
typedef struct
  {
    . . . . . .
    . . . . . .
  } new_data_type;
```

where new_data_type (this is supplied by user) is the user defined data type. Remember that this is not the name of the variable.

Now, the variable of new_data_type is declared as same as standard data type.

Let us consider the following structure.

```
typedef struct
  {
    int roll_no;
    char name[30];
    char sex;
    float height;
  } student;
```

The variables of type student is declared as

```
student cs_student, me_student,ee_student[4];
```

Note that the keyword struct is not required to declared a variable. but the keyword typedef is used to define structure.

The more complicated data type can be defined by repeated use of typedef. For example,

```
typedef struct
  {
    int day;
    int month;
    int year;
  }date;
typedef struct
  {
    int roll_no;
    char name[30];
    char sex;
    float height;
    date date_of_birth;
  } student;
```

Here date and student are user defined data type. The data type date is used to define the data type student.

2.11.7 Structures and Pointers

Like integer and other standard pointer variables, structure type pointer variables can also be declared.

If var is a structure type variable, then &var represents the starting address

of that variable. A pointer for a structure type variable can be declared as

```
data_type *pv;
```

where data_type is the type of data. In this case, it is of type structure. *pv represents the name of the pointer variable.

To illustrate the structure type pointer variable, let us consider the following structure.

```
typedef struct
  {
    int roll_no;
    char name[30];
    char sex;
    float height;
  }student;
student cs_student, *ps;
```

Here cs_student is a variable of type student, but, ps is a pointer variable which points to student (structure data type).

It is obvious that when the variable cs_student is declared of type student, a space is allocated for this variable and the amount of the space is calculated as follows:

Variable name	Required space (bytes)
roll_no	2
name	30
sex	1
height	4
Total	37

Thus for the variable cs_student, 37 bytes is allocated in computer memory.

Now, when the statement

```
pv=&cs_student;
```

is executed, then the beginning address of the allocated space for cs_student is assigned to pv.

The above declaration (the definition of structure as typedef and declaration of variable of type student) may be written as

```
struct
  {
    int roll_no;
```

```
    char name[30];
    char sex;
    float height;
} cs_student, *pv;
```

A care is to be taken to access the individual structure member. An individual member is accessed with the help of the operator -> (composition of minus operator and greater than sign). If pv is a structure type pointer variable then a member can be accessed by the expression

```
pv->member;
```

Note that -> is comparable to dot (.) operator and this expression is equivalent to

```
structure_variable.member;
```

Let us consider the following structure to illustrate the use of individual member.

```
typedef struct
  {
    int day;
    int month;
    int year;
  } date;
typedef struct
  {
    int roll_no;
    char name[30];
    char sex;
    float height;
    date date_of_birth;
  } student;
```

Now, we consider

```
student cs_student, *pv;
```

and the address of cs_student is assign to pv as

```
pv=&cs_student;
```

By this statement pv points to cs_student.

The roll_no of the cs_student can be accessed by any one of the following statements

```
cs_student.roll_no;      (here pointer is not used)
pv->roll_no;             (pointer pv is used)
(*pv).roll_no;
```

In the last expression parentheses are required as the precedence of dot is higher than parentheses. If parentheses are not used then dot operator is applied first. i.e. the computer will try to determine the value of pv.roll_no, but it does not mean anything, so an error will occur.

The name of cs_student can be accessed by any one of the following expressions

cs_student.name pv->name (*pv).name

Let us consider the following program to use the pointer variable as structure member.

```
#include<stdio.h>
main()
 {
   int r=45;
   char nam[]="Aniket";
   char s='m';
   float h=48;
   int d=6,m=7,y=2001;
   typedef struct
     {
       int day;
       int month;
       int year;
     } date;
   typedef struct
     {
       int *roll_no;
       char *name;
       char *sex;
       float *height;
       date dob;
     } student;
   student *pv, cs_student;
   cs_student.roll_no=&r;
   cs_student.name=nam;
   cs_student.sex=&s;
   cs_student.height=&h;
   cs_student.dob.day=d;
   cs_student.dob.month=m;
```

```
    cs_student.dob.year=y;
    pv=&cs_student; /* pv points to cs_student */
    printf("%d %s %c %f DOB=%d.%d.%d\n", *cs_student.roll_no,
      cs_student.name, *cs_student.sex,
      *cs_student.height,cs_student.dob.day,cs_student.dob.month,
      cs_student.dob.year);
    printf("%d %s %c %f DOB=%d.%d.%d\n", *pv.roll_no,
      pv.name, *pv.sex,*pv.height,pv.dob.day,pv.dob.month,
      pv.dob.year);
} /* main */
```

The output of this program is

```
45 Aniket m 48 DOB=6.7.2001
45 Aniket m 48 DOB=6.7.2001
```

Note that the assignment statement roll_no is a pointer variable, so cs_student.roll_no is a pointer variable and & operator is used to assign the value of r to cs_student.roll_no. In this case the address is assigned. nam is an array, so it points to a string and to assign it to cs_student.name no & operator is needed. cs_student.dob.day, cs_student.dob.month and cs_student.dob.year are ordinary variables, so & operator is not required at these assignment.

Again, since cs_student.roll_no is a pointer, therefore *cs_student.roll_no represents its content. Note that * is not required to print a string of characters. Again, since cs_student.dob.day is an ordinary variable, no * is required to print its value.

Observe that second print statement. pr->roll_no, pv->sex, pv->height represent pointer, to print their contents * is required. Note that to access the content of pv->roll_no, we use the expression *pv->roll_no not pv->*roll_no. The latter one is incorrect. dob.day, dob.month, dob.year are ordinary variables. Therefore, no asterisk is required to print the values of these variables. But, the operator -> is used to indicate that dob.day is a member of pv.

Note that the precedence of the operator -> is high, i.e. the precedence of both the operator -> and . (dot) are same. Also, their associativity is left to right. Moreover, the precedence of the operator -> is higher than any unary operator, arithmetic, relational and logical or assignment operators. Thus, a care should be taken while using this operator.

2.11.8 Passing Structure to Function

Like ordinary variable, structure type variable can also be passed to a function. There are several ways to pass a structure type variable to a function or return a structure type result from a function. The simplest way to pass a structure type data is passing of individual member values (like ordinary variable), and the value of only one member can be returned through return statement. In other way a complete structure can be passed as argument.

Let us consider the following program to illustrate the transformation of individual arguments.

```
#include<stdio.h>
void main()
 {
   typedef struct
    {
      int roll_no;
      char name[30];
      int ph_mark,chem_mark,math_mark,total_mark;
    } student;
   student ankit;
   int total;
   int calculate_total(int,int,int);
   ankit.roll_no=100;
   ankit.name="Ankit Sarkar";
   ankit.ph_mark=50;
   ankit.chem_mark=60;
   ankit.math_mark=70;
   ankit.total_mark=calculate_total(ankit.ph_mark,
                                    ankit.chem_mark,ankit.math_mark);
   printf("Total marks of %s is %d\n",ankit.name,ankit.total_mark);
 }
/* definition of function */
int calculate_total(int p, int c, int m)
 {
   return(p+c+m);
 }
```

In this program a portion of structure (the fields ph_mark, chem_mark and math_mark) ankit is transferred to the function calculate_total and the single value, sum of these marks is returned to the main function.

An entire structure can be transferred to a function by passing a pointer argument which points to a structure type variable. This logic is similar to passing an array to a function. This process (argument passed by pointer) is called passed by reference rather by value.

2.11.9 Union

In a program all the variables occupied different memory locations to store their values. Also, the values of all the members of a structure are stored in different memory locations. But, sometimes we observed that a variable or an array is used for the first part (or a portion) of the program and another variable or array is used in the next part (other portion) of the program. To minimize the space one can assign these two or more variables or arrays into the same memory location. This can be done with the help of union. That is, the members within a union share the same storage area. This is useful when a program involves multiple data and the values need not be assigned to all of the members simultaneously. Only one memory of a union is active at any particular time instant. A mechanism is required to manage the space and to activated a particular member. All these are done by compiler automatically, however, the use must keep track of what type of information is activated at any given time.

The declaration of a union is similar to structure. The general form is

```
union union_name
  {
     data_type member1;
     data_type member2;
     . . . . . .
     data_type membern;
  };
```

where union is the required keyword, data_type is the data type of each member member1, member2, ..., membern. union name is the name of union, it is also called tag.

The variable of union type is declared as

storage class union union name variable1, variable2, ..., variablem where variable1, variable2,..., variablem are the variables of type union_name, storage_class is optional, it specifies storage class of union_name.

The above two declaration may also be combined as

```
storage_class union union_name
```

```
{
    data_type  member1;
    data_type  member2;
    . . . . . .
    data_type  membern;
} variable1, variable2,..., variablem;
```

The union_name is optional for this type of declaration.

Since all the members of a union share same memory space, the compiler will allocate sufficient space for the variables of union to accommodate the largest member in the union. Other members also use the same space.

Define a union named emp with a variable var and a pointer *pu.

```
union emp
  {
     int emp_id;
     char name[30];
  };
union emp var, *pu;
```

The above declaration may be combined as

```
union emp
  {
     int emp_id;
     char name[30];
  } var, *pu;
```

Here union is defined along with a variable var and a pointer variable *pu. In this case, 30 bytes of memory space is allocated for the variable var.

The members of union can be accessed as in similar way of structure, i.e. they can be accessed by either dot or - > operator.

Let us consider the above structure emp and the following declaration.

```
union emp e1, *e2;
e2=&e1;
```

The individual members can be accessed by the following statements

```
e1.emp_id; /* access the emp id */
e1.name;
e2->emp_id;
e2->name; /* access the name */
```

Assignments are also similar to assignment of structure.

Note 2.11.1

(i) A union variable can be assigned to another union variable, provided both the variables have the same composition.

(ii) The address of a union variable can be determined by the operator &.

(iii) A union variable can pass to a function as an argument.

(iv) A function can accept and returns a union or a pointer to a union.

(v) The arithmetic and logical operators on a union variable (as a whole) are not allowed.

Exercise 2.11

1. What is a structure? How does a structure differ from an array?
2. What is a structure member? What is the relationship between a structure member and a structure?
3. How can structure variables be declared? How do structure variable declarations differ from structure type declarations?
4. Describe the syntax for defining the composition of a structure. Can individual members be initialized within a structure type declaration?
5. Can a structure variable be defined as a member of another structure? Can an array be included as a member of a structure? Can an array have structure as elements? Explain.
6. How is an array of structures initialized?
7. How is a structure member accessed? How can a structure member be processed?
8. How can the size of a structure be determined? In what units is the size reported?
9. What is the purpose of the typedef feature? How is this feature used in conjunction with structures?

10. How can an entire structure be passed to a function? How can an entire structure be returned from a function?
11. What is union? How does a union differ from a structure?
12. How is a union accessed? How can a union member be processed? How is a member of a union variable assigned an initial value?
13. In what way does the initialization of a union variable differ from the initialization of a structure variable?
14. Write a program that reads several different names and addresses into the computer, rearranges the names into alphabetical order, and then writes out the alphabetized list. A structure may be defined which includes name and address of each people.

2.12 File Processing

In the programs of previous sections it is assumed that the input is supplied from the keyboard (the standard input device) and output is displayed on the monitor (the standard output device). But, when the size of input or output is large, then it is time consuming to enter data or to read output from the display terminal (monitor) at a time. Many business and scientific applications require large amount of data to be entered and saved for latter use. Instead of reading (or writing) data from keyboard (on monitor) one can read (or write) from (to) a unit called **data file** or simply **file**. The file primary reside in primarily memory and when it is saved it will stored in secondary memory. Thus data files allow us to store information permanently and one can access and modified whenever necessary.

Depending on the storing technique, the data file are classified into two categories- **stream-oriented** (or **standard** or **ASCII**) files and **system-oriented** (or **low-level** or **binary**) files. The stream-oriented files are readable and opened in any text editor and so easy to work with these types of files. These files are commonly used. On the other hand, system-oriented (binary) files are not readable by any arbitrary text editor. To store an information in ASCII file needs more space then binary file.

The binary file generally used to design and development of a complex database or to read and write a binary information. The binary file is more accurate because it stores the exact internal representation of a value. There are no conversion errors or round off errors. Storing data in binary form takes less time as there is no conversion is required during storing data to a file. But,

binary data file cannot easily be transferred from one computer to another due of variations in the internal representation of the data in the computer. On the other hand the text file can easily be transferred from one computer to another.

2.12.1 File Pointer, Opening and Closing a File

A computer cannot read or write data directly from (or to) the file which reside in secondary storage. During reading from a data file, the information is transferred from data file to a special portion of primary memory called buffer area, and then it is used in program. The use of buffer area speed up the reading from or writing to the data file.

To use data file, at first we have to declare a special pointer of type FILE (all characters must be upper case) which points to the beginning of the buffer area. FILE is a structure defined in the header file stdio.h. A pointer of type FILE is declare as

```
FILE *fp;
```

The pointer fp referred to as a **stream pointer** or simply a **stream**. This pointer is very important during use of data file.

After declaration of file pointer, a file must be defined for use (reading or writing or both for reading and writing). The function fopen is used to open a file. The general form is

```
fp=fopen(file_name, file_type);
```

where file_name and file_type are two strings (constants or variables) they represent the name of the data file and the manner in which it must be opened (reading, writing, both reading and writing, appending, overwriting, etc.). The file_name is a valid name of a file which is allowed by the operating system. For ASCII file, the file_type must be one of the strings shown in Table 2.11.

The fopen function returns a pointer which points to a buffer. If fopen returns a NULL pointer, then it indicates that the file associated in fopen function is not opened successfully.

The two statements

```
FILE *fp;
fp=fopen("sample.dat","r");
```

can be written as

```
FILE *fp=fopen("sample.dat","r");
```

In the first declaration, fp is declared as a file pointer and second statement opens a file named sample.dat for reading. While in the second declaration, the file pointer fp is declared as well as the function fopen is assigned to it.

file_type	Use
"r"	Open an existing file for reading only
"r+"	Open an existing file for both reading and writing
"w"	Open a new file for writing only. If a file with the file name file_name exists, it will be destroyed and then a new file will be created with name file name.
"w+"	Open a new file for both reading and writing. If the file having name file_name exists, it will be destroyed and a new file will be created for both reading and writing.
"a"	Open an existing file for appending. That is, for adding new data at the end of file. If no such file exists, then a new file will be created.
"a+"	Open an existing file for both reading and appending. If no such file exists, then a new file will be created.

Table 2.11:

If a file is opened, it must be closed. This can be done by the following statement

```
fclose(fp);
```

where fp is the file pointer associated to the file which is opened for use. When a file is closed, then the buffer created for it, will be destroyed and the memory allocated for buffer becomes free. If the file is not closed, the compiler will automatically closes the file at the end of the program. Some times it may happen that if the data file is not properly closed, then all the data are erased from the file. So it is good practice to close a file explicitly using the function fclose.

2.12.2 File Handling Functions

Some functions are specially defined to handel data file. These are discussed in the following.

(a) fopen

The general form of this function is

```
fopen(file_name,file_type);
```

This function returns a pointer file_name is a character string represents name of the file to be opened. If the file does not exist in the working folder, the full path name is required. The double backslash (for DOS) or a simple slash (for UNIX) are to be given in the path name. For example, if the data file

test.dat reside in the folder cprog of d drive, then the file_name will be d:\\cprog\\test.dat (for DOS) or d:/cprog/test.dat (for UNIX).

The file_type represents the mode in which a file is to be opened. These values are r, w, r+, w+, a, a+. If the file is being opened or created as a text mode (ASCII) file, the character t (for text) may be write at the end of above mentioned modes. For example, rt, wt, r+t, etc. The use of t is optional for text file. If the file is open as binary file, then the character b must be added at the end of above modes. For example, rb, wb, r+b, etc. If neither t nor b is used with the modes, then the file will open as text mode.

(b) fclose

The general form of this function is

```
fclose(file_pointer);
```

where file_pointer is a pointer of type FILE. This function releases the buffer associated with this pointer and closes the file. The standard file pointers stdin, stdout, etc. can also be closed using the function fclose. This function returns an integer 0 on success and EOF on error.

(c) fgetc or getc

These two functions have same utility. fgetc is the function version of getc. The general form are

```
getc(file_pointer);
fgetc(file_pointer);
```

These functions read a character from the file which is associated to the pointer file_pointer. On success these functions returns a character and on end-of-file or error return EOF.

(d) fputc or putc

These two functions also have the same use. fputc is the function version of putc. The general form are

```
putc(ch, file_pointer);
fputc(ch, file_pointer);
```

These two functions write the character ch to the file associated to the file pointer file_pointer. On success these functions return the character ch and on error they return EOF.

(e) fscanf

This function is similar to scanf function, except that the file pointer must be specified as first argument. Its general form is

```
fscanf(file_pointer, character_stream);
```

This function reads the values of the variables specified in character_stream from the file associated to the pointer file_pointer. Themeaningof character_stream is same as that of scanf function.

(f) fprintf

This function is similar to printf function, except that the first argument must be the file pointer. Its general form is

```
fprintf(file_pointer, character_stream);
```

The file_pointer is the pointer which is associated to a file. The meaning of character_stream is same as that of printf function.

If stdin, stdout, etc. are used as file pointer the effect will be the same as using printf or scanf function. The fscanf and fprintf functions are used to read and write formatted data file (the file contains different type of data, viz., int, float, char, etc., in a particular order).

2.12.3 Writing to a File

A data file can be created into two different ways. The first method is, one can create the file directly by using any text editor (for example, C editor). The process is same as creating a program file. The second method is read the data from the keyboard using the functions getchar, gets, scanf and write them to a specified file using the functions fputc, fputs and fprintf. The last three functions are similar to putchar, puts and printf, except that the functions fputc, fputs and fprintf accept a pointer to the FILE structure as the first parameter.

The following program reads the roll number, name and sex of some students and write them to a file named student.dat.

```
#include<stdio.h>
main()
 {
    int roll,i;
    char name[15];
    char sex;
```

```
    FILE *fp; /* declaration of file pointer */
    fp=fopen("student.dat","w"); /* opens the file student.dat
                                    as writing mode */
    for(i=0;i<5;i++)
      {
        printf("Enter record of student-%d",i+1);
        sex=getchar();
        scanf("%d",&roll);
        gets(name);
        /* writing to the file */
        fprintf(fp,"%d %s %c",roll,name,sex);
      }
    fclose(fp);
}
```

If the input for this program are

```
m   10  Rahul
m   20  Sachin
f   27  Sumita
m   27  Monojit
m   62  Nitu
```

then the content of the file student.dat is

```
10  Rahul    m
20  Sachin   m
27  Sumita   f
27  Monojit  m
62  Nitu     m
```

The file student.dat can be viewed in many different ways. The file can be viewed by opening it by C editor or by DOS command type or print or by UNIX command cat.

2.12.4 Reading From a File

The standard input functions are getchar, gets, scanf. To read data from a file, the functions getc, fgets and fscanf are used. The first argument of these functions is file pointer, otherwise these functions are similar to above functions.

Example 2.12.1 Suppose a file `triangle.in` contains three sides of some triangles, say n. The first line of the file `triangle.in` contains the value of n and next each line contains the value of three sides. Write a program to read the sides of the triangles and display three sides and corresponding area on the screen.

```
#include<stdio.h>
#include<math.h>
main()
 {
   float a,b,c,s,t,area;
   int n,i;
   FILE *fp; /* file pointer */
   fp=fopen("triangle.in","r");
   /* checking for existence of a file */
   if(fp==NULL) /* if error, terminates the program */
     {
       printf("Error-opening file\n");
       exit(1);
     }
   fscanf("%d",&n);
   for(i=0;i<n;i++)
     {
       fscanf("%f%f%f",&a,&b,&c);
       s=(a+b+c)/2;
       t=s*(s-a)*(s-b)*(s-c);
       if(t>0)
         {
           area=sqrt(a);
           printf("%0.2f %0.2f %0.2f %f \n",a,b,c,area);
         }
       else
         printf("The sides do not form a triangle\n");
     }
   fclose(fp);
   return;
 }
```

Let the content of the file triangle.in be

```
3
4 5 6
2 3 4
1 2 3
```

The output of this program is

```
4.00   5.00   6.00   9.921567
2.00   3.00   4.00   2.904737
The sides do not form a triangle
```

Example 2.12.2 Write a program to read a text file (ASCII) and display its content on the monitor.

```
#include<stdio.h>
main()
 {
   char ch,fname[13];
   FILE *ifile;
   printf("Enter file name \n");
   scanf("%s", fname);
   ifile=fopen(fname,"r");
   /* checking the existence of the file */
   if(ifile==NULL)
     {
      printf("No such file exists!\n");
      exit(1);
     }
   /* reading a character from a file and printing it on screen*/
   while(!feof(ifile)) /* while not end of file ifile */
      {
        ch=fgetc(ifile); /* reads a character from ifile */
        putchar(ch); /* prints on the screen */
      }
   fclose(ifile);
   return;
 }
```

The above program reads all the characters from a file. The file name supplied externally. The characters are displayed on the monitor screen.

The two lines

```
ch=fgetc(ifile);
putchar(ch);
```

can be written together as putchar(fgetc(ifile));

The above program reads a file and displayed its content to the screen. Using this program (with a minor extension) one can save a file to another file. The details are given in the following program.

Example 2.12.3 Write a program which will save a text file to another file. The file names are to be supplied externally.

```
#include<stdio.h>
#include<io.h>
main()
 {
   char  ch,ifname[13],ofname[13];
   FILE  *ifile,*ofile;
   int  count=0;
   /* source file */
   printf("Enter source file name \n");
   scanf("%s", ifname);
   ifile=fopen(ifname,"r"); /* open for reading */
   /* checking the existence of the file */
   if(ifile==NULL)
     {
       printf("Error - No such file exists!\n");
       exit(1);
     }
   /* target file */
   printf("Enter target file name \n");
   scanf("%s", ofname);
   ofile=fopen(ofname,"w"); /* open for writing */
   /* checking the existence of the file */
   if(ofile==NULL) /* error checking */
     {
       printf("Error in file creation !\n");
       exit(0);
     }
```

```
    /* reading a character from a file and writing it to another file
*/
    while(!feof(ifile)) /* while not end of file source file */
      {
        ch=fgetc(ifile); /* reads a character from ifile */
        count++;
        fputc(ch,ofile); /* prints on the screen */
      }
    fclose(ifile);
    fclose(ofile);
    printf("The number of characters in the source file
                                                    is %d",count);
    return;
  }
```

This program reads two files - one is source (the file from which the data are to be transferred) and other is target (the file to which the data are to be written). A character is read out from the source file whose file pointer is ifile and it is written to the target file whose file pointer is ofile. This process is repeated until the source file reaches end-of-file condition. The variable count determines the number of characters in the source (also in target) file.

Example 2.12.4 Write a program to merge two files into a single file.

Solution. Suppose source1 and source2 are the given files which are to be merged and the new file target is created. At first source1 is to be copied to target and then source2 is appended to target. The program is shown below.

```
#include<stdio.h>
void main()
{
  char  ch,source1[13],source2[13],target[13];
  FILE *ifile,*ofile;
  printf("Enter first source file \n");
  scanf("%s", source1);
  printf("Enter second source file \n");
  scanf("%s", source2);
  printf("Enter target file name \n");
  scanf("%s", target);
```

```
  ifile=fopen(source1,"r"); /* open for reading */
  ofile=fopen(target,"w"); /* open for writing */
  while(!feof(ifile)) /* while not end of file ifile */
    fputc(fgetc(ifile),ofile); /* reading a character from source
                                  file and writing it to target */
  fclose(ifile); /* source1 is closed now */
  ifile=fopen(source2,"r"); /* open second file for reading */
  while(!feof(ifile)) /* while not end of file ifile */
    fputc(fgetc(ifile),ofile);
  fclose(ofile);
  fclose(ofile);
}
```

2.12.5 Operations on Data Files

Assume that the records (roll_no and name) of some students are stored in the file stu.dat. Let the key value of the record to be searched, be read as s_roll (the roll number). Now, a record from the file stu.dat is read and the key (roll_no) of the current record is compared with the search key (s_roll). If they are equal then the record is printed and the search is terminated. If the search key s_roll does not match with any record then a message 'Record does not exist' is printed and the program is halted there after.

```
#include<stdio.h>
void main()
{
  char name[25],fname[13];
  int roll_no,s_roll;
  FILE *fp;
  printf("Enter file name \n");
  scanf("%s",fname); /* input of record file */
  fp=fopen(fname,"r");
  printf("Enter roll number to be searched\n");
  scanf("%d",&s_roll);
  while(!feof(fp)) /* while not end of file */
    {
      fscanf(fp,"%d%[^\n]",&roll_no,name);
      if(roll_no==s_roll)
        {
```

```
            printf("Record found\n");
            printf("%d %s\n",roll_no,name);
            fclose(fp);
            exit(1);
        }
    }
  printf("Record does not exist\n");
}
```

Assumed that the file stu.dat contains the following information.

```
125 ANU
333 ANIKET PAL
401 SUMITA BERA
504 PARTHA SAHA
```

The input/output of the above program is

```
Enter file name
stu.dat
Enter roll number to be searched
333
Record found
333 ANIKET PAL
```

Updating a file

Suppose roll_no and name of some students are stored in the file stu.dat. The following program will read roll_no and name of each student from the file stu.dat and marks of four subjects from keyboard. Then the total mark, percentage of marks and class obtained are computed and the result are stored into a new file stunew.dat. The class is computed according to the following rule.

% of marks	Class
60 and above	I
45 to below 60	II
30 to below 45	III
below 30	FAIL

Finally, the name, roll no, marks of four subjects, total mark, percentage of marks and the class of each student are printed.

```
/* Updating a file */
#include<stdio.h>
void main()
{
   char  name[20],remark[4];
   int  roll_no,mark1,mark2,mark3,mark4,total_mark;
   float  percent;
   FILE  *fpi,*fpo;
   fpi=fopen("stu.dat","r");
   fpo=fopen("stunew.dat","w");
   while(!feof(fpi))
     {
       fscanf(fpi,"%d%[^\n]",&roll_no,name);
       printf("Enter marks of four subjects of %d %s\n",
                                                    roll_no,name);
       scanf("%d%d%d%d",&mark1,&mark2,&mark3,&mark4);
       total_mark=mark1+mark2+mark3+mark4;
       percent=total_mark/4.0;
       /* calculation of class */
       if(percent>=60)
         strcpy(remark,"I");
       else if((percent>=45) && (percent<60))
         strcpy(remark,"II");
       else if((percent>=30) && (percent<45))
         strcpy(remark,"III");
       else
         strcpy(remark,"FAIL");
       fprintf(fpo,"%d %s %d %d %d %d %d %0.2f %s\n",
           roll_no,name,mark1,mark2,mark3,mark4,
           total_mark,percent,remark);
     } /* end of while */
   fclose(fpi);
   fclose(fpo);
}
```

Suppose the records of the file stu.dat are

```
125  ANU
333  ANIKET  PAL
401  SUMITA  BERA
504  PARTHA  SAHA
```

```
Enter marks of four subjects of 125 ANU
50 20 30 40
Enter marks of four subjects of 333 ANIKET PAL
40 60 90 87
Enter marks of four subjects of 401 SUMITA BERA
10 20 30 40
Enter marks of four subjects of 504 PARTHA SAHA
45 35 56 47
```

Finally, the file stunew.dat updated as

```
125 ANU             50  20  30  40  140  35.00  III
333 ANIKET  PAL     40  60  90  87  277  69.25  I
401 SUMITA  BERA    10  20  30  40  100  25.00  FAIL
504 PARTHA  SAHA    45  35  56  47  183  45.75  II
```

Exercise 2.12

1. Explain the merits and demerits of sequential file and random access files processing.
2. Explain the terms: sequential file, random access file.
3. Write short notes on : opening a file, closing a file, appending a file.
4. Explain the functions: rewind, ftell, fseek.
5. Suppose the file TRIANGLE.DAT contains the three sides of some triangles. Write a program to read the sides of each triangle from the file TRIANGLE.DAT and calculate perimeter and area of the triangle and store them along with three sides, to the file TRIANGLE.OUT.
6. Suppose a file ABC.DAT has 1000 numbers, 10 numbers per line. Write a program to read these members and sort them and store the sorted numbers in the file XYZ.DAT.
7. Write a program to read a file and to display the contents of the file on the screen with line number.
8. Write a program to merge two files into one file.
9. Write a program to open a sequential file named STD.DAT at unit 20 and read the roll number, name and total marks obtained by 100 students from keyboard and store them into the file STD.DAT.

10. Write a program to create a sequential file EMP.DAT and read the empcode, name and salary of employees of an organization from keyboard and store them in EMP.DAT. Also introduce the scope of checking of error in the input. If there is error on the input then again read that record.

11. Suppose a file SEM1 contains the roll number, name and total marks (out of 400) of some students of Semester-I Examination. Another file SEM2 contains the roll number, name of those students in the same order as in the file SEM1 with marks of four papers PAPER5, PAPER6, PAPER7, PAPER8 of Semester-II Examination. Write a program to prepare the result which to be stored in the file RESULT of this examination containing roll number, name, marks in Semester-I, total percentage of marks and class. The grade will be determined as

$$\begin{aligned}
&\%\text{ of marks} \geq 90\% \rightarrow \text{Ex}\\
80\% \leq\ &\%\text{ of marks} < 90\% \rightarrow \text{A}\\
70\% \leq\ &\%\text{ of marks} < 80\% \rightarrow \text{B}\\
60\% \leq\ &\%\text{ of marks} < 70\% \rightarrow \text{C}\\
50\% \leq\ &\%\text{ of marks} < 60\% \rightarrow \text{D}\\
&\%\text{ of marks} < 50\% \rightarrow \text{FAIL.}
\end{aligned}$$

2.13 Macro and Preprocessor

The C language provides some special features which are not available in any other high level programming language. Here some more advanced features are presented. We have already used the `#define` statement. This statement is used to defined (in earlier sections) a symbolic constant within a program. The advantage to use this statement is that, before compilation of the program, all symbolic constants are replaced by their equivalent expressions. For example, let us consider the following statement

```
#define MAX 100
```

Now, if the constant `MAX` contains in five places in the program, then before compilation `MAX` is replaced by 100 in such five places.

The `#define` statement can also be used in many purposes. It can be used to define macros, i.e. a single variable is used to define a constant, an expression, a complete statement or a group of statements. More precisely, using a single identifier one can define a function. Other than `#define` statement there are more such statements. For example, `#if`, `#else`, `#elif`, `#endif`, `#ifdef`, etc. These are called preprocessors. The preprocessor is a program that processes

the source code before passing it to the compiler. It is a collection of special statements called **directives** or **preprocessors command line.**

All the preprocessor directives are placed in the program before the main function. Before the source code passes through the compiler, it is examined by the preprocessor. After examined by the preprocessor and their appropriate action, the source code is submitted for compilation.

2.13.1 Macros

It is mentioned that using the concept of macro we define some constants by some symbolic name and the values of the symbolic names are replaced by the constants. The macro substitution is very important feature in C programming language and we can use a macro to define many things.

A macro is defined by the preprocessor directive #define. The general form is

```
#define identifier string
```

where identifier is a symbolic name (valid C variable) and string may be a constant, an expression of any type (int, float, char, etc). The three terms are separated by blank spaces. Note that no semicolon is used at the end of this statement.

Some examples of macros are

```
#define PI 3.1415926
#define country "INDIA"
#define email "mmaplvu@gmail.com"
#define NULL 0
#define begin {
#define EQ ==
```

Usually, in macro definition the identifiers are written in upper case alphabets to distinguished between symbolic name and other identifiers, but both upper case and lower case alphabets may be used.

Let us consider the following example

```
#include<stdio.h>
#define MAX 10
#define NAME "ARJUN"
main()
 {
   int a[MAX],i;
   printf("Name=%s",NAME);
```

```
    for(i=0;i<MAX;i++)
      scanf("%d",&a[i]);
}
```

In this program two macros MAX and NAME are defined. During preprocessing the value of NAME and MAX are replaced by ARJUN and 10 respectively. The preprocessed program thus becomes

```
#include<stdio.h>
main()
 {
    int a[10],i;
    printf("Name=%s","ARJUN");
    for(i=0;i<10;i++)
      scanf("%d",&a[i]);
 }
```

Note that NAME outside the double quotations (in printf) mark is replaced by "ARJUN" and within quotations it is unaffected by macro substitution.

Macro definitions are placed at the beginning of the program, before any function declaration, even main function. The scope of macro definition is over the whole program. It may be remembered that a macro defined in one file is not recognized within another file.

In the above program, some constants are defined as macros. But, a single statement, multiple statements, operators, function and any other valid string may be defined as macro.

The following are valid declarations.

```
#define BEGIN {
#define END }
#define AND &&
#define OR ||
#define SPACE
```

But, some care should be taken to define macro. For example, macros X and Y are defined as

```
#define X a+b
#define Y c-d
```

and Z=X/Y is a statement in the program. When macros are substituted the expression for Z becomes $a + b/c - d$ and obviously it is not correct expression. The following definition resolved this drawback.

```
#define X (a+b)
#define Y (c-d)
```

Multiline macros can also be defined by placing a backward slash (\) at the end of each line except the last line. That means, a single macro can define a group of statements and other things.

This feature is explained by the following example.

```
#include<stdio.h>
#define READ   scanf("%d",&n);    \
               for(i=0;i<n;i++){ \
                 scanf("%d",&a[i]);    \
                 printf("%d ",a[i]);   \
                 }
#define        SUM sum=0;         \
               for(i=0;i<n;i++)   \
                 sum+=a[i];
#define PRINT printf("Sum=%d\n",sum);
main()
 {
   int n, a[10],i,sum;
   READ
   SUM
   PRINT
 }
```

This program contains three multiline macros and using these macros the main function becomes shortened. Note that no semicolon is needed at the end of three macros READ, SUM and PRINT (within main), because in these macros the semicolons are already given.

The preprocessed form of the above program is given below.

```
#include<stdio.h>
main()
{
 int n, a[10],i,sum;
 scanf("%d",&n);
 for(i=0;i<n;i++)
```

```
    {
      scanf("%d",&a[i]);
      printf("%d ",a[i]);
    }
  sum=0;
  for(i=0;i<n;i++)
    sum+=a[i];
  printf("Sum=%d\n",sum);
}
```

Macros with Arguments

Macros are also used in place of functions. A function needs some arguments, so macros to define functions use arguments. The general form of such macro is

```
#define identifier(arg1,arg2,...,argn) string
```

where identifier is a valid C variable which is also used as function name. arg1, arg2, ..., argn are the formal arguments without their data type and string is the expression for this function.

Let us consider the following simple example.

```
#defin f(x) x*x+2*x+5.0
```

Suppose the following statements are included in the program.

```
val1=f(2.5);
val2=f(2+3);
a=4;
val3=f(a++);
```

During preprocessing these three statements transferred to

val1 = 2.5 * 2.5 + 2 * 2.5 + 5.0 = 16.25

val2 = 2 + 3 * 2 + 3 + 2 * 2 + 3 + 5.0 = 2 + 6 + 3 + 4 + 3 + 5.0 = 23.0

val3 = (a++) * (a++) + 2 * (a++) + 5.0 = 4. 5 + 2 * 6. + 5.0 = 37.0.

Look at these three values val1, val2 and val3. There is no problem in val1, this is absolutely correct. When val2 is evaluated, x is replaced by the expression 2+3 in straight forward way, shown in the expression of val2 and gives incorrect result. This error can be removed by defining the function $f(x)$ as

```
#define f(x) (x)*(x)+2*(x)+5.0
```

But, the third case is very serious. When f(a ++) for a = 4 is called, the value of x is replaced by 4 (=a) and after replacement a is incremented to 5, this incremented value replaces x. Again, a is incremented to 6 and this is used in third places of x.

The use of macro in place of function reduces the running time of the program. Because, when macro is used, during preprocessing the function (defined in macro) is substituted in all places where it appears within the program and then it is compiled. That is, in this process no functions are called during program execution. Thus, if a macro is substituted in many places, then the object code becomes long. The use of macro becomes economic where there are relatively few function calls within a loop.

Example 2.13.1 Write a program to find the value of $f(x)$ and $g(x)$ for $x = 0$, 0.2, 0.4,..., 1.0 where $f(x) = \sqrt{1+x}$ and $g(x) = \sin x$. Use macros for $f(x)$ and $g(x)$.

```
#include<stdio.h>
#include<math.h>
#define f(x) sqrt(1+(x))
#define g(x) sin((x))
void main()
 {
   float x;
   printf(" f(x) \t g(x)\n");
   for(x=0.0;x<=1.0;x+=0.2)
     printf("%7.4f\t%7.4f\n",f(x),g(x));
 }
```

The output of the above program is shown below.

```
f(x)      g(x)
1.0000    0.0000
1.0954    0.1987
1.1832    0.3894
1.2649    0.5646
1.3416    0.7174
1.4142    0.8415
```

2.13.2 The C Preprocessor

The statements #include and #define discussed earlier are preprocessor directives. The other preprocessor directives are #if, #elif, #else, #endif, #ifdef, #ifndef, #line and #undef. The more preprocessor directives are #pragma, #error, # and ##. Among them the directives #if, #elif, #else and #endif are used frequently.

The preprocessor directives generally appear at the beginning of the program, but this is not mandatory. A preprocessor can be defined any where in the program. The directive will be active from the place of its appearance to the end of the program.

#ifdef, #ifndef preprocessors

These directives along with #else, #elif and #endif directives are frequently used as preprocessors. These are used in conditional compilation of the source program, depending on the value of test condition.

Let us consider the following example. A vector is a one-dimensional array and its number of rows and columns are denoted by 1 and COLUMNS (> 1). If number of rows becomes greater than one, then it is considered as a matrix. Thus, the number of rows and columns in a matrix are denoted by ROWS and COLUMNS, where ROWS (> 1) and COLUMNS (≥ 1). Also, assumed that the objects (OBJECT) vector and matrix are denoted by VECTOR and MATRIX respectively. Now, we would like to define the size of vector or matrix by preprocess directives.

```
#if defined (ROWS)
    #define COLUMNS 5
#endif
```

The preprocessor #if defined() is equivalent to #ifdef. Thus the above statements are equivalent to

```
#ifdef ROWS
    #define COLUMNS 5
#endif
```

Thus, if ROWS is already defined, then 5 is assigned to the symbolic constant COLUMNS. The #else directive can also be used within #ifdef directive, illustrated below.

```
#ifdef ROWS                    /* if ROWS is defined */
   #define COLUMNS 5
#else
   #define ROWS 2
   #define COLUMNS 5
#endif
```

These statements check whether ROWS is defined or not. If ROWS is defined

then COLUMNS is set to 5, otherwise the symbolic constants ROWS and COLUMNS are set to 2 and 5 respectively.

In the directives #if, #ifdef, #ifndef the last directive must be #endif. Thus, the general form of #ifdef, #ifndef and #if are

```
#if (or #ifdef or #ifndef) symbolic_constant
        ..............
        ..............
#else
        ..............
        ..............
#endif
```

If the symbolic_constant is true, then the statements between #if and #else will execute, otherwise the statements between #else and #endif will execute.

The following preprocessors directives illustrate the use of #if.

```
#if OBJECT==VECTOR
    #define ROWS 1
    #define COLUMNS 5
#else
    #define ROWS 2
    #define COLUMNS 5
#endif
```

That is, if the object is defined as vector then the number of rows and columns are set to 1 and 5 respectively, otherwise (the object is matrix) the number of rows and columns are set to 2 and 5 respectively.

In C, the #if statement checks only one condition. But, if we have to check multi-condition, then else if statement is used. Like C's control statement else if there is a similar preprocessor directive #elif. With #if directive, any number of #elif directives can be used, but only one #else directive is used. The #else directive is optional, depends on the program logic.

```
#if ROWS==1
    #define COLUMNS 5
#elif ROWS==2
    #define COLUMNS 10
```

```
#elif ROWS==3
    #define COLUMNS 12
#else
    #define COLUMNS 14
#endif
```

In the above examples we have use #define to defined some symbolic constants within #if, #elif and #endif blocks.

Exercise 2.13

1. What is macro? Summarize the similarities and differences between macros and functions.
2. How is a multiline macro defined?
3. Describe the use of arguments within a macro.
4. What is the main advantage in the use of macro rather than a function? What is the main disadvantage of it?
5. What is the scope of a preprocessor directive within a program file?

2.14 Summary

This is the largest unit of this module. In this unit, a brief overview of C programming language is given. The basic elements of a programming language, viz. constants, variables, expressions, arrays, input/output functions, control statements are discussed and illustrated by several examples. The use of function is a very special feature, and it is discussed here. The use of pointers, structures and unions are very important features of C, these make easier to writing a program. These features are incorporated with simple examples. The use of data files is discussed in this module. Some advanced topics of C language are also briefly introduced with examples.

UNIT 3 ❐ PROBLEMS ON NUMERICAL ANALYSIS

In this unit, some fundamental problems of numerical analysis have been considered. The algorithms and programs to solve such problems are designed.

3.1 Objectives

After going through this unit you will be able to learn about–

(i) Design of algorithms to solve numerical problems

(ii) Development of C programs for some basic problems of numerical analysis.

3.2 Solution of Algebraic and Transcendental Equations

3.2.1 Bisection Method

Let ξ be a root of the equation $f(x) = 0$ lies in the interval $[a, b]$, i.e., $f(a).f(b) < 0$, and $(b - a)$ is not sufficiently small. The interval $[a, b]$ is divided into two equal intervals $[a, c]$ and $[c, b]$, each of length $\frac{b-a}{2}$ and $c = \frac{a+b}{2}$. If $f(c) = 0$, then c is an exact root.

Now, if $f(c) \neq 0$, then the root lies either in the interval $[a, c]$ or in the interval $[c, b]$. If $f(a).f(c) < 0$ then the interval $[a, c]$ is taken as new interval, otherwise $[c, b]$ is taken as the next interval. Let the new interval be $[a_1, b_1]$ and use the same process to select the next new interval. In the next step, let the new interval be $[a_2, b_2]$. The process of bisection is continued until either the midpoint of the interval is a root, or the length $(b_n - a_n)$ of the interval $[a_n, b_n]$ (at nth step) is sufficiently small. The number an and bn are the approximate roots of the equation $f(x) = 0$. Finally, $x_n = \frac{a_n + b_n}{2}$ is taken as the approximate value of the root ξ.

An algorithm of bisection method is presented below.

Algorithm 3.1 (Bisection method). This algorithm finds a real root of the equation $f(x) = 0$ which lies in $[a, b]$.

Algorithm Bisection

Input function $f(x)$;

```
// Assume that f(x) is continuous within [a, b] and a root lies on [a, b].//
Read ε;          //tolerance for width of the interval//
Read a, b;       //input of the interval//
Compute fa = f(a); fb = f(b);      //compute the function values//
if sign(fa) = sign(fb) then
//sign(fa) gives the sign of the value of fa.//
  Print 'f(a) · f(b) > 0, so there is no guarantee for a root within [a, b]';
  Stop;
endif;
do
  Compute c = (a + b)/2;
  Compute fc = f(c);
  if fc = 0 or |fc| < ε then
     a = c and b = c;
  else if sign(fb) = sign(fc) then
     b = c; fb = fc;
  else
     a = c; fa = fc;
  endif;
while (|b - a| > ε);
Print 'the desired root is' c;
end Bisection
```

```
/* Program Bisection
   Program to find a root of the equation x*x*x-2x-1=0 by
   bisection method.
   Assume that a root lies between a and b. */
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define f(x) x*x*x-2*x-1 /* definition of the function f(x) */
void main()
{
 float a,b,fa,fb,c,fc;
 float eps=1e-5; /* error tolerance */
 printf("\nEnter the value of a and b ");
 scanf("%f %f",&a,&b);
 fa=f(a); fb=f(b);
 if(fa*fb>0)
```

```
    {
      printf("There is no guarantee for a root within [a,b]");
      exit(0);
    }
  do
  {
    c=(a+b)/2.;
    fc=f(c);
    if((fc==0) || (fabs(fc)<eps))
      {
        a=c;b=c;
      }
    else if(fb*fc>0)
      {
        b=c; fb=fc;
      }
    else
      {
        a=c; fa=fc;
      }
  }while(fabs(b-a)>eps);
  printf("\nThe desired root is %8.5f ",c);
} /* main */.
```

```
Enter the value of a and b 0 2
The desired root is 1.61803
```

3.2.2 Iteration Method or Fixed Point Iteration

The **iteration method** or the **method of successive approximations**, is one of the most important methods in numerical mathematics. This method is also known as **fixed-point iteration**.

Let $f(x)$ be a function continuous on the interval $[a, b]$ and the equation $f(x) = 0$ has at least one root on $[a, b]$. The equation $f(x) = 0$ can be written in the form

$$x = \phi(x). \tag{3.1}$$

Suppose $x_0 \in [a, b]$ be an initial guess to the desired root ξ. Then $\phi(x_0)$ is evaluated and this value is denoted by x_1. It is the first approximation of the root ξ. Again, x_1 is substituted for x to the right side of (3.1) and obtained a new

value $x_2 = \phi(x_1)$. This process is continued to generate the sequence of numbers $x_0, x_1, x_2, \ldots, x_n, \ldots$, those are defined by the following relation:

$$x_{n+1} = \phi(x_n), \quad n = 0, 1, 2, \ldots \tag{3.2}$$

This successive iterations are repeated till the approximate numbers xn's converges to the root with desired accuracy, i.e., $|x_{n+1} - x_n| < \varepsilon$, where ε is a sufficiently small number. The function $\phi(x)$ is called the iteration function.

```
/* Program Fixed-Point Iteration
   Program to find a root of the equation x*x*x-3x+1=0
   by fixed point iteration method. phi(x) is obtained
   by rewrite f(x)=0 as x=phi(x), which is to be supplied.*/
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define phi(x)  (3*x-1)/(x*x)
/* definition of the function phi(x) and it to be
   changed accordingly */
void main()
{
 int k=0; /* counts number of iterations */
 float x1,x0; /* initial guess */
 float eps=1e-5; /* error tolerance */
 printf("\nEnter the initial guess x0 ");
 scanf("%f",&x0);
 x1=x0;
 do
 {
  k++;
  x0=x1;
  x1=phi(x0);
 }while(fabs(x1-x0)>eps);
 printf("One root is %8.5f obtained at %d th iteration ",x1,k);
} /* main */
```

```
Enter the initial guess x0 1
One root is 1.53209 obtained at 37th iteration
```

3.2.3 Newton-Raphson Method or Method of Tangent

Let $x0$ be an approximate root of the equation $f(x) = 0$. Suppose $x_1 = x_0 + h$ be the exact root of the equation, where h is the correction of the root (error). Then $f(x_1) = 0$.

Using Taylor's series, $f(x_1) = f(x_0 + h)$ is expanded in the following form

$$f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(x_0) + \ldots\ldots = 0$$

Neglecting the second and higher order derivatives the above equation reduces to

$$f(x_0) + hf'(x_0) = 0 \text{ or, } h = -\frac{f(x_0)}{f'(x_0)}$$

Hence, $x_1 = x_0 + h = x_0 - \dfrac{f(x_0)}{f'(x_0)}$

To compute the value of h, the second and higher powers of h are neglected so the value of $h = -\dfrac{f(x_0)}{f'(x_0)}$ is not exact, it is an approximate value. So, x_1, obtained from (3.3) is not a root of the equation, but it is a better approximation of x than x_0.

In general,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.4}$$

This expression generates a sequence of approximate values $x_1, x_2, \ldots, x_n, \ldots$ each successive term of which is closer to the exact value of the root ℓ than its predecessor. The method will terminate when $|x_{n+1} - x_n|$ becomes very small. In Newton-Raphson method the arc of the curve $y = f(x)$ is replaced by a tangent to the curve, hence, this method is sometimes called the **method of tangents.**

```
/* Program Newton-Raphson
   Program to find a root of the equation x*x*x-3x+1=0 by Newton-
   Raphson method. f(x) and its derivative fd(x) are to be supplied. */
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
```

```
void main()
{
  int k=0; /* counts number of iterations */
  float x1,x0; /* x0 is the initial guess */
  float eps=1e-5; /* error tolerance */
  float f(float x);
  float fd(float x);
  printf("\nEnter the initial guess x0 ");
  scanf("%f",&x0);
  x1=x0;
  do
  {
   k++;
   x0=x1;
   x1=x0-f(x0)/fd(x0);
  }while(fabs(x1-x0)>eps);
  printf("One root is %8.5f obtained at %d th iteration ",x1,k);
} /* main */
/* definition of the function f(x) */
float f(float x)
  {
   return(x*x*x-3*x+1);
  }
/* definition of the function fd(x) */
float fd(float x)
  {
   return(3*x*x-3);
  }
```

```
Enter the initial guess x0 1.1
One root is 1.53209 obtained at 7 th iteration
```

3.3 Solution of System of Linear Equations

Generally, two types of methods are used to solve a system of linear equations, viz., direct and iteration. If the system of equations has a large number of variables, then the direct methods are not much suitable. In this case, the approximate numerical methods are used to determine the variables of the system.

The approximate methods for solving system of linear equations make it possible to obtain the values of the roots of the system with the specified accuracy as the limit of the sequence of some vectors. The process of constructing such a sequence is known as the iterative process.

The efficiency of the application of approximate methods depends on the choice of the initial vector and the rate of convergence of the process.

Here two iteration methods - Jacobi's iteration and Gauss-Seidal's iteration are discussed along with the direct method LU-decomposition.

3.3.1 Jacobi's Iteration Method

Let us consider a system of n linear equations containing n variables:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \quad & + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \quad & + a_{2n}x_n = b_2 \\ \cdots\cdots\cdots\cdots & \\ a_{n1}x_1 + a_{n2}x_2 + \quad & + a_{nn}x_n = b_n. \end{aligned} \tag{3.5}$$

Also, we assume that the quantities a_{ii} are pivot elements.

The above equations can be written as

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}x_n) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \ldots - a_{2n}x_n) \\ &\cdots\cdots\cdots\cdots \\ x_n &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{nn-1}x_{n-1}). \end{aligned} \tag{3.6}$$

Let $x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}$ be the initial guess to the variables $x_1, x_2, \ldots, x_n$ respectively (initial guess may be taken as zeros). Substituting these values in the right hand side of (3.6), which yields the first approximation as follows.

$$\begin{aligned} x_1^{(0)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - \ldots - a_{1n}x_n^{(0)}) \\ x_2^{(1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - \ldots - a_{2n}x_n^{(0)}) \\ &\cdots\cdots\cdots\cdots \\ x_n^{(1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(0)} - a_{n2}x_2^{(0)} - \ldots - a_{nn-1}x_{n-1}^{(0)}) \end{aligned} \tag{3.7}$$

Again, substituting $x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)}$ in the right hand side of (3.6) and obtain the second approximation $x_1^{(2)}, x_2^{(2)}, \ldots, x_n^{(2)}$.

In general, if $x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}$ be the kth approximate roots then the next approximate roots are given by

$$\begin{aligned}
x_1^{(k+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \ldots - a_{1n}x_n^{(k)}) \\
x_2^{(k+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \ldots - a_{2n}x_n^{(k)}) \qquad (3.8) \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
x_n^{(k+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \ldots - a_{nn-1}x_{n-1}^{(k)})
\end{aligned}$$

$k = 0, 1, 2, \ldots$

The iteration process is continued until all the roots converge to the required number of significant figures. This iteration method is called **Jacobi's iteration** or simply the **method of iteration.**

The Jacobi's iteration method surely converges if the coefficient matrix is diagonally dominant.

Algorithm 3.2 (Gauss-Jacobi's). This algorithm finds the solution of a system of linear equations by Gauss-Jacobi's iteration method. The method will terminate when $|x_i^{(k+1)} - x_i^{(k)}| < \varepsilon$ where ε is the supplied error tolerance, for all i.

Algorithm Gauss Jacobi

Step 1. Read the coefficients a_{ij}, $i, j = 1, 2, \ldots, n$ and the right hand vector b_i, $i = 1, 2, \ldots, n$ of the system of equations and error tolerance ε.

Step 2. Rearrange the given equations, if possible, such that the system becomes diagonally dominant.

Step 3. Rewrite the ith equation as

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}x_j\right) \text{ for } i = 1, 2, \ldots, n.$$

Step 4. Set the initial solution as
$x_i = 0$, $i = 1, 2, 3, \ldots, n$.

Step 5. Calculate the new values xn_i of x_i as

$$xn_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}x_j\right) \text{ for } i = 1, 2, \ldots, n.$$

Step 6. If $|x_i - xn_i| < \varepsilon$ (ε is an error tolerance) for all i, then goto Step 7 else set $x_i = xn_i$ for all i and goto Step 5.

Step 7. Print xn_i, $i = 1, 2, \ldots, n$ as solution.

end Gauss Jacobi

```
/*Program Gauss_Jacobi
Solution of a system of linear equations by Gauss-Jacobi's iteration
method. Testing of diagonal dominance is
also incorporated.*/
 #include<stdio.h>
 #include<math.h>
 #include<stdlib.h>
void main()
{
 float a[10][10],b[10],x[10],xn[10],epp=0.00001,sum;
 int i,j,n,flag;
 printf("Enter number of variables ");
 scanf("%d",&n);
 printf("\nEnter the coefficients rowwise ");
 for(i=1;i<=n;i++)
   for(j=1;j<=n;j++) scanf("%f",&a[i][j]);
 printf("\nEnter right hand vector ");
 for(i=1;i<=n;i++)
   scanf("%f",&b[i]);
 for(i=1;i<=n;i++) x[i]=0; /* initialize */
/* checking for row dominance */
flag=0;
for(i=1;i<=n;i++)
 {
   sum=0;
   for(j=1;j<=n;j++)
     if(i!=j) sum+=fabs(a[i][j]);
   if(sum>fabs(a[i][i])) flag=1;
 }
/* checking for column dominance */
if(flag==1)
```

```
    flag=0;
    for(j=1;j<=n;j++)
      {
        sum=0;
        for(i=1;i<=n;i++)
          if(i!=j)  sum+=fabs(a[i][j]);
        if(sum>fabs(a[j][j]))  flag=1;
      }
  }
if(flag==1)
  {
    printf("The coefficient matrix is not diagonally dominant\n");
    printf("The Gauss-Jacobi method does not converge surely");
    exit(0);
  }
for(i=1;i<=n;i++)  printf(" x[%d]  ",i);printf("\n");
do
{
for(i=1;i<=n;i++)
  {
    sum=b[i];
    for(j=1;j<=n;j++)
      if(j!=i)  sum-=a[i][j]*x[j];
    xn[i]=sum/a[i][i];
  }
for(i=1;i<=n;i++)  printf("%8.5f  ",xn[i]);printf("\n");

flag=0; /* indicates |x[i]-xn[i]|<epp for all i */
for(i=1;i<=n;i++)  if(fabs(x[i]-xn[i])>epp)  flag=1;
if(flag==1)  for(i=1;i<=n;i++)  x[i]=xn[i]; /* reset x[i]  */
}while(flag==1);

 printf("Solution is \n");
 for(i=1;i<=n;i++)  printf("%8.5f  ",xn[i]);
} /* main */

Enter number of variables 3
Enter the coefficients rowwise
```

```
9 2  4
1 10 4
2 -4 10
Enter right hand vector
20 6 -15
   x[1]      x[2]      x[3]
  2.22222 0.60000 -1.50000
  2.75556 0.97778 -1.70444
  2.76247 1.00622 -1.66000
  2.73640 0.98775 -1.65000
  2.73606 0.98636 -1.65218
  2.73733 0.98727 -1.65267
  2.73735 0.98733 -1.65256
  2.73729 0.98729 -1.65254
  2.73729 0.98729 -1.65254
Solution is
 2.73729 0.98729 -1.65254
```

3.3.2 Gauss-Seidal's Iteration Method

A simple modification of Jacobi's iteration sometimes give faster convergence. The modified method is known as Gauss-Seidal's iteration method.

Let us consider a system of n linear equations with n variables.

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \qquad\qquad + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \qquad\qquad + a_{2n}x_n = b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \qquad\qquad + a_{nn}x_n = b_n. \end{array} \tag{3.9}$$

Assume that the diagonal coefficients a_{ii}, $i = 1, 2, \ldots, n$ are diagonally dominant. If this is not the case then the above system of equations are re-arranged in such a way that the above condition holds.

The equations (3.9) are rewritten in the following form :

$$\begin{array}{l} x_1 = \dfrac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}x_n) \\ x_2 = \dfrac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \ldots - a_{2n}x_n) \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ x_n = \dfrac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{nn-1}x_{n-1}) \end{array} \tag{3.10}$$

To solve these equations an initial approximation $x_2^{(0)}, x_3^{(0)}, \ldots, x_n^{(0)}$ for the variables $x_2, x_3, \ldots, x_n$ respectively is considered. Substituting these values to the above system and get the first approximate value of x_1, denoted by $x_1^{(1)}$. Now, substituting $x_1^{(1)}$ for x_1 and $x_3^{(0)}, x_4^{(0)}, \ldots, x_n^{(0)}$ for $x_3, x_4, \ldots, x_n$ respectively and we find $x_2^{(1)}$ from second equation of (3.10), the first approximate value of x_2. Then substituting $x_1^{(1)}, x_2^{(1)}, \ldots x_{i-1}^{(1)}, x_{i+1}^{(0)}, \ldots, x_n^{(0)}$ for $x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ to the ith equation of (3.10) respectively and obtain $x_i^{(1)}$, and so on.

If $x_i^{(k)}$, $i = 1, 2, \ldots, n$ be the kth approximate value of x_i, then the $(k+1)$th approximate value of $x_1, x_2, \ldots, x_n$ are given by

$$
\begin{aligned}
x_1^{(k+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \ldots - a_{1n}x_n^{(k)}) \\
x_2^{(k+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \ldots - a_{2n}x_n^{(k)}) \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
x_i^{(k+1)} &= \frac{1}{a_{ii}}(b_i - a_{i1}x_1^{(k+1)} - \ldots - a_{ii-1}x_{i-1}^{(k+1)} - a_{ii+1}x_{i+1}^{(k)} - \ldots - a_{in-1}x_{n-1}^{(k)}) \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
x_n^{(k+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k+1)} - a_{n2}x_2^{(k+1)} - \ldots - a_{nn-1}x_{n-1}^{(k+1)}) \\
k &= 0, 1, 2, \ldots .
\end{aligned}
\tag{3.11}
$$

That is,

$$
x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \; i = 1, 2, \ldots, n \text{ and } k = 0, 1, 2, \ldots
$$

The method is repeated until $|x_i^{(k+1)} - x_i^{(k)}| < \varepsilon$ for all $i = 1, 2, \ldots, n$, where $\varepsilon > 0$ is any pre-assigned number called the error tolerance. This method is called Gauss-Seidal's iteration method.

Algorithm 3.3 (Gauss-Seidal's). This algorithm finds the solution of a system of linear equations by Gauss-Seidal's iteration method. The method will terminate when $|x_i^{(k+1)} - x_i^{(k)}| < \varepsilon$, where ε is the supplied error tolerance, for all i.

Algorithm Gauss Seidal

Step 1. Read the coefficients a_{ij}, $i, j = 1, 2, \ldots, n$ and the right hand vector b_i, $i = 1, 2, \ldots, n$ of the system of equations and error tolerance ε.

Step 2. Rearrange the given equations, if possible, such that the system becomes diagonally dominant.

Step 3. Rewrite the ith equation as

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j - \sum_{j>i} a_{ij}x_j\right) \text{ for } i = 1, 2, \ldots, n.$$

Step 4. Set the initial solution as
$x_i = 0$, $i = 1, 2, 3, \ldots, n$.

Step 5. Calculate the new values xn_i of x_i as

$$xn_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}xn_j - \sum_{j>i} a_{ij}x_j\right) \text{ for } i = 1, 2, \ldots, n.$$

Step 6. If $|x_i - xn_i| < \varepsilon$ (ε is an error tolerance) for all i then goto Step 7 else set $x_i = xn_i$ for all i and goto Step 5.

Step 7. Print xn_i, $i = 1, 2, \ldots, n$ as solution.

end Gauss Seidal

```
/* Program Gauss-Seidal
Solution of a system of linear equations by Gauss-Seidal's
iteration method. Assume that the coefficient matrix
satisfies the condition of convergence. */
 #include<stdio.h>
 #include<math:h>
 void main()
 {
   float a[10][10],b[10],x[10],xn[10],epp=0.00001,sum;
   int i,j,n,flag;
   printf("Enter number of variables ");
   scanf("%d",&n);
   printf("\nEnter the coefficients rowwise ");
   for(i=1;i<=n;i++)
     for(j=1;j<=n;j++)  scanf("%f",&a[i][j]);
   printf("\nEnter right hand vector ");
   for(i=1;i<=n;i++)
     scanf("%f",&b[i]);
   for(i=1;i<=n;i++) x[i]=0;  /* initialize */
   /* testing of diagonal dominance may be included here
     from the program of Gauss-Jacobi's method */
```

```
do
{
for(i=1;i<=n;i++)
  {
    sum=b[i];
    for(j=1;j<=n;j++)
     {
    if(j<i)
      sum-=a[i][j]*xn[j];
    else if(j>i)
      sum-=a[i][j]*x[j];
    xn[i]=sum/a[i][i];
      }
  }
flag=0; /* indicates |x[i]-xn[i]|<epp for all i */
for(i=1;i<=n;i++) if(fabs(x[i]-xn[i])>epp) flag=1;
if(flag==1) for(i=1;i<=n;i++) x[i]=xn[i]; /* reset x[i] */
}
while(flag==1);
printf("Solution is \n");
for(i=1;i<=n;i++) printf("%8.5f ",xn[i]);
} /* main */
```

```
Enter number of variables 3
Enter the coefficients rowwise
3 1 -1
2 5 2
2 4 6

Enter right hand vector
7 9 8
Solution is
  2.00000   1.00000   0.00000
```

3.3.3 LU Decomposition Method

This method is also known as factorization or **Crout's reduction method.** Let the system of linear equations be

$$\mathbf{Ax = b} \tag{3.12}$$

where **A**, **x**, **b** are respectively coefficient matrix, variable vector and right hand side vector.

The matrix **A** can be factorized into the form **A** = **LU**, where **L** and **U** are the lower and upper triangular matrices respectively. *If the principal minors of* **A** *are non-singular, then this factorization is possible and it is unique.*

The matrices **L** and **U** are of the form

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \text{ and } \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{23} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \quad (3.13)$$

The equation **Ax** = **b** becomes **LUx** = **b**. Let **Ux** = **z** then **Lz** = **b**, where **z** $= (z_1, z_2, \ldots, z_n)^t$ is an intermediate variable vector. The value of **z** i.e., $z_1, z_2, \ldots, z_n$ can be determined by forward substitution in the following equations.

$$\begin{aligned} l_{11}z_1 &= b_1 \\ l_{21}z_1 + l_{22}z_2 &= b_2 \\ l_{31}z_1 + l_{32}z_2 + l_{33}z_3 &= b_3 \\ \cdots\cdots\cdots\cdots\cdots\cdots &\cdots \\ l_{n1}z_1 + l_{n2}z_2 + l_{n3}z_3 + \ldots + l_{nn}z_n &= b_n. \end{aligned} \quad (3.14)$$

After determination of **z**, one can compute the value of **x** i.e., $x_1, x_2, \ldots, x_n$ from the equation **Ux** = **z** i.e., from the following equations by the backward substitution.

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + \cdots + u_{1n}x_n &= z_1 \\ u_{22}x_2 + u_{23}x_3 \cdots + z_{2n}x_n &= z_2 \\ u_{33}x_3 + u_{23}x_3 \cdots + u_{3n}x_n &= z_3 \\ \cdots\cdots\cdots\cdots\cdots\cdots &\cdots \\ u_{n-1n-1}x_{n-1} + u_{n-1n}x_n &= z_{n-1} \\ u_{nn}x_n &= z_n. \end{aligned} \quad (3.15)$$

When $u_{ii} = 1$, for $i = 1, 2, \ldots, n$, then the method is known as **Crout's decomposition method.**

Procedure to compute L and U

Here, we assume that $u_{ii} = 1$ for $i = 1, 2, \ldots, n$. From the relation **LU** = **A**, i.e., from

$$\begin{bmatrix} l_{11} & l_{11}u_{12} & l_{11}u_{13} & \cdots & l_{11}u_{1n} \\ l_{21} & l_{21}u_{12}+l_{22} & l_{21}u_{13}+l_{22}u_{23} & \cdots & l_{21}u_{1n}+l_{22}u_{2n} \\ l_{31} & l_{31}u_{12}+l_{32} & l_{31}u_{13}+l_{32}u_{23}+l_{33} & \cdots & l_{31}u_{1n}+l_{32}+u_{2n}+l_{33}u_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ l_{n1} & l_{n1}u_{12}+l_{n2} & l_{n1}u_{13}+l_{n2}u_{23}+l_{n3} & \cdots & l_{n1}u_{1n}+l_{n2}u_{2n}+\cdots+l_{nn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

we obtain

$$l_{i1} = a_{i1},\ i = 1, 2, \ldots, n \text{ and } u_{1j} = \frac{a_{1j}}{l_{11}},\ j = 2, 3, \ldots, n.$$

The second column of L and the second row of U are determined from the relations

$$l_{i2} = a_{i2} - l_{i1}u_{12}, \text{ for } i = 2, 3, \ldots, n,$$

$$u_{2j} = \frac{a_{2j} - l_{21}u_{1j}}{l_{22}} \text{ for } j = 3, 4, \ldots, n.$$

Next, third column of L and third row of U are determined in a similar way. In general, l_{ij} and u_{ij} are given by

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj},\ i \geq j \tag{3.16}$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}}{l_{ii}},\ i < j \tag{3.17}$$

$$u_{ii} = 1,\ l_{ij} = 0,\ j > i \text{ and } u_{ij} = 0,\ i > j.$$

Alternatively, the vectors **z** and **x** can be determined from the equations

$$\mathbf{z} = \mathbf{L}^{-1}\mathbf{b} \tag{3.18}$$

$$\text{and} \quad \mathbf{x} = \mathbf{U}^{-1}\mathbf{z}. \tag{3.19}$$

It may be noted that the computation of inverse of a triangular matrix is easier than an arbitrary matrix.

The inverse of A can also be determined from the relation

$$\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}. \tag{3.20}$$

Algorithm 3.4 (LU decomposition). This algorithm finds the solution of a system of linear equations using LU decomposition method. Assume that the principal minors of all order are non-zero.

Algorithm LU-decomposition

Let $\mathbf{Ax} = \mathbf{b}$ be the systems of equations and $\mathbf{A} = [a_{ij}]$, $\mathbf{b} = (b_1, b_2, \ldots, b_n)^t$, $\mathbf{x} = (x_1, x_2, \ldots, x_n)^t$.

//Assume that the principal minors of all order are non-zero.//

//Determine the matrices **L** and **U**.//

Step 1. Read the matrix $\mathbf{A} = [a_{ij}]$, $i, j = 1, 2, \ldots, n$ and the right hand vector $\mathbf{b} = (b_1, b_2, \ldots, b_n)^t$.

Step 2. $l_{i1} = a_{i1}$ for $i = 1, 2, \ldots, n$; $u_{1j} = \dfrac{a_{1j}}{l_{11}}$ for $j = 2, 3, \ldots, n$;

$u_{ii} = 1$ for $i = 1, 2, \ldots, n$.

Step 3. For $i, j = 2, 3, \ldots, n$ compute the following

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \quad i \geq j$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}}{l_{ii}}, \quad i < j$$

Step 4. //Solve the system $\mathbf{Lz} = \mathbf{b}$ by forward substitution.//

$$z_1 = \frac{b_1}{l_{11}}, \quad z_i = \frac{1}{l_{ii}}\left(b_i - \sum_{j=1}^{i-1} l_{ij} z_j\right) \text{ for } i = 2, 3, \ldots, n.$$

Step 5. //Solve the system $\mathbf{Ux} = \mathbf{z}$ by backward substitution.//

Set $x_n = z_n$;

$$x_i = z_i - \sum_{j=i+1}^{n} u_{ij} x_j \text{ for } i = n-1, n-2, \ldots, 1.$$

Print $x_1, x_2, \ldots, x_n$ as solution.

end LU-decomposition

```
/* Program LU-decomposition. Solution of a system of
equations by LU decomposition method. Assume that all
order principal minors are non-zero. */
#include<stdio.h>
void main()
{
  float a[10][10],l[10][10],u[10][10],z[10],x[10],b[10];
  int i,j,k,n;
```

```
printf("\nEnter the size of the coefficient matrix ");
scanf("%d",&n);
printf("Enter the elements rowwise ");
for(i=1;i<=n;i++)  for(j=1;j<=n;j++)  scanf("%f",&a[i][j]);
printf("Enter the right hand vector ");
for(i=1;i<=n;i++)  scanf("%f",&b[i]);
/* computations of L and U matrices */
for(i=1;i<=n;i++)  l[i][1]=a[i][1];
for(j=2;j<=n;j++)  u[1][j]=a[1][j]/l[1][1];
for(i=1;i<=n;i++)  u[i][i]=1;
for(i=2;i<=n;i++)
  for(j=2;j<=n;j++)
    if(i>=j)
      {
        l[i][j]=a[i][j];
        for(k=1;k<=j-1;k++). l[i][j]-=l[i][k]*u[k][j];
      }
    else
      {
        u[i][j]=a[i][j];
        for(k=1;k<=i-1;k++)  u[i][j]-=l[i][k]*u[k][j];
        u[i][j]/=l[i][i];
      }
printf("\nThe lower triangular matrix L\n");
for(i=1;i<=n;i++)
  {
    for(j=1;j<=i;j++) printf("%f ",l[i][j]);
    printf("\n");
  }
printf("\nThe upper triangular matrix U\n");
for(i=1;i<=n;i++)
  {
    for(j=1;j<i;j++) printf(" ");
    for(j=i;j<=n;j++) printf("%f ",u[i][j]);
    printf("\n");
  }
/* solve Lz=b by forward substitution */
z[1]=b[1]/l[1][1];
for(i=2;i<=n;i++)
```

```
    {
      z[i]=b[i];
      for(j=1;j<=i-1;j++)  z[i]-=l[i][j]*z[j];
      z[i]/=l[i][i];
    }
  /* solve Ux=z by backward substitution */
  x[n]=z[n];
  for(i=n-1;i>=1;i--)
    {
      x[i]=z[i];
      for(j=i+1;j<=n;j++)  x[i]-=u[i][j]*x[j];
    }
  printf("The solution is ");
  for(i=1;i<=n;i++) printf("%f ",x[i]);
} /* main */
```

A sample of input/output:

```
Enter the size of the coefficient matrix 3
Enter the elements rowwise
4 2 1
2 5 -2
1 -2 7

Enter the right hand vector
3 4 5

The lower triangular matrix L
4.000000
2.000000   4.000000
1.000000   -2.500000   5.187500

The upper triangular matrix U
1.000000 0.500000 0.250000
         1.000000 -0.625000
                  1.000000
The solution is -0.192771 1.325301 1.120482
```

3.4 Integration

At first we deduce the general integration formula based on Newton's forward interpolation formula. The Newton's forward interpolation formula for the equispaced points x_i, $i = 0, 1, \ldots, n$, $x_i = x_0 + ih$ is

$$\phi(x) = y_0 + u\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \frac{u(u-1)(u-2)}{3!}\Delta^3 y_0 + \cdots, \tag{3.21}$$

where $u = u = \dfrac{x - x_0}{h}$, h is the spacing.

Let the interval $[a, b]$ be divided into n equal subintervals such that $a = x_0 < x_1 < x_2 < \ldots < x_n = b$. Then

$$I = \int_a^b f(x)dx = \int_{x_0}^{x_n} \phi((x)dx$$

$$= \int_{x_0}^{x_n} \left[y_0 + u\Delta y_0 + \frac{u^2 - u}{2!}\Delta^2 y_0 + \frac{u^3 - 3u^2 + 2u}{3!}\Delta^3 y_0 + \cdots\right]dx$$

Since $x = x_0 + uh$, $dx = hdu$, when $x = x_0$ then $u = 0$ and when $x = x_n$ then $u = n$.

Thus,

$$I = \int_0^n \left[y_0 + u\Delta y_0 + \frac{u^2 - u}{2!}\Delta^2 y_0 + \frac{u^3 - 3u^2 + 2u}{3!}\Delta^3 y_0 + \cdots\right] hdu$$

$$= h\left[y_0[u]_0^n + \Delta y_0\left[\frac{u^2}{2}\right]_0^n + \frac{\Delta^2 y_0}{2!}\left[\frac{u^3}{3} - \frac{u^2}{2}\right]_0^n + \frac{\Delta^3 y_0}{3!}\left[\frac{u^4}{4} - u^3 + u^2\right]_0^n + \cdots\right]$$

$$= nh\left[y_0 + \frac{n}{2}\Delta y_0 + \frac{2n^2 - 3n}{12}\Delta^2 y_0 + \frac{n^3 - 4n^2 + 4n}{24}\Delta^3 y_0 + \cdots\right] \tag{3.22}$$

From this formula, one can generate different integration formulae by substituting $n = 1, 2, 3, \ldots$.

3.4.1 Trapezoidal Rule

Substituting $n = 1$ in the equation (3.22). In this case all differences higher than the first difference become zero. Then

$$\int_{x_0}^{x_n} f(x)dx = h\left[y_0 + \tfrac{1}{2}\Delta y_0\right] = h\left[y_0 + \tfrac{1}{2}(y_1 - y_0)\right] = \frac{h}{2}\left(y_0 + y_1\right) \tag{3.23}$$

The formula (3.23) is known as the **trapezoidal rule.**

In this formula, the interval $[a, b]$ is considered as a single interval, and it gives a very rough answer. But, if the interval $[a, b]$ is divided into several subintervals and this formula is applied to each of these subintervals then a better approximate result may be obtained. This formula is known as composite formula, deduced below.

Composite trapezoidal rule

Let the interval $[a, b]$ be divided into n equal subintervals by the points $a = x_0, x_1, x_2, \ldots, x_n = b$, where $x_i = x_0 + ih$, $i = 1, 2, \ldots, n$.

Applying the trapezoidal rule to each of the subintervals, one can find the composite formula as

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx$$

$$= \frac{h}{2}[y_0 + y_1] + \frac{h}{2}[y_1 + y_2] + \frac{h}{2}[y_2 + y_3] + \cdots + \frac{h}{2}[y_{n-1} + y_n]$$

$$= \frac{h}{2}[y_0 + 2(y_1 + y_2 + \cdots + y_{n-1}) + y_n] \qquad (3.24)$$

Algorithm 3.5 (Trapezoidal). This algorithm finds the value of $\int_a^b f(x)dx$ based on the tabulated values (x_i, y_i), $y_i = f(x_i)$, $i = 0, 1, 2, \ldots, n$, using trapezoidal rule.

Algorithm Trapezoidal
Input function $f(x)$;
Read a, b, n; //the lower and upper limits and number of subintervals.//
Compute $h = (b - a)/n$;
Set $sum = \frac{1}{2}[f(a) + f(a + nh)]$;
for $i = 1$ to $n - 1$ do
  Compute $sum = sum + f(a + ih)$;
endfor;
Compute $result = sum * h$;
Print *result*;
end Trapezoidal

```
/* This program finds the value of integration of a function
by trapezoidal rule. Here we assume that f(x)=x^3. */
  #include<stdio.h>
  void main()
  {
    float a,b,h,sum; int n,i;
    float f(float);
    printf("Enter the values of a, b ");
```

```
    scanf("%f %f",&a,&b);
    printf("Enter the value of n ");
    scanf("%d",&n);
    h=(b-a)/n;
    sum=(f(a)+f(a+n*h))/2.;
    for(i=1;i<=n-1;i++)  sum+=f(a+i*h);
    sum=sum*h;
    printf("The value of the integration is %8.5f ",sum);
  }
/* definition of the function f(x) */
float f(float x)
  {
    return(x*x*x);
  }
```

```
Enter the values of a, b 0 1
Enter the value of n 100
The value of the integration is 0.25002
```

3.4.2 Simpson's 1/3 rule

In this formula the interval $[a, b]$ is divided into two equal subintervals by the points x_0, x_1,, x_2, where $h = (b - a)/2$, $x_1 = x_0 + h$ and $x_2 = x_1 + h$.

This rule is obtained by putting $n = 2$ in (3.22). In this case, the third and higher order differences do not exist.

The equation (3.22) is simplified as

$$\int_{x_0}^{x_n} f(x) \simeq 2h\left[y_0 + \Delta y_0 + \frac{1}{6}\Delta^2 y_0\right] = 2h\left[y_0 + (y_1 - y_0) + \frac{1}{6}(y_2 - 2y_1 + y_0)\right]$$

$$= \frac{h}{3}\left[y_0 + 4y_1 + y_2\right] \qquad (3.25)$$

The above rule is known as **Simpson's 1/3 rule** or simply **Simpson's rule**.

Composite Simpson's 1/3 rule

Let the interval $[a, b]$ be divided into n (an *even number*) equal subintervals by the points x_0, x_1, x_2, ..., xn, where $x_i = x_0 + ih$, $i = 1, 2, ..., n$.

Then

$$\int_a^b f(x)dx = \int_{x_0}^{x_2} f(x)dx + \int_{x_2}^{x_4} f(x)dx + \cdots + \int_{x_{n-2}}^{x_n} f(x)dx$$

$$= \frac{h}{3}[y_0 + 4y_1 + y_2] + \frac{h}{3}[y_2 + 4y_3 + y_4] + \cdots + \frac{h}{3}[y_{n-2} + 4y_{n-1} + y_n]$$

$$= \frac{h}{3}[y_0 + 4(y_1 + y_3 + \cdots + y_{n-1}) + 2(y_2 + y_4 + \cdots + y_{n-2}) + y_n] \qquad (3.26)$$

This formula is known as **Simpson's 1/3 composite rule** for numerical integration.

Algorithm 3.6 (Simpson's 1/3). This algorithm determines the value of $\int_a^b f(x)dx$ using Simpson's 1/3 rule.

Algorithm Simpson_One_Third
Input function $f(x)$;
Read a, b, n; //the lower and upper limits and number of subintervals.//
Compute $h = (b - a)/n$;
Set $sum = [f(a) - f(a + nh)]$;
for $i = 1$ to $n - 1$ step 2 do
  Compute $sum = sum + 4 * f(a + ih) + 2 * f(a + (i + 1)h)$;
endfor;
Compute $result = sum * h/3$;
Print *result*;
end Simpson_One_Third.

```
/* Program Simpson's 1/3
   Program to find the value of integration of a function
   f(x) using Simpson's 1/3 rule. Here we assume that f(x)=x^3.*/
#include<stdio.h>
void main()
{
 float f(float);
 float a,b,h,sum;
 int i,n;
 printf("\nEnter the values of a, b ");
 scanf("%f %f",&a,&b);
 printf("Enter the value of subintervals n ");
```

```
scanf("%d",&n);
if(n%2!=0) {
   printf("Number of subdivision should be even");
   exit(0);
 }
h=(b-a)/n;
sum=f(a)-f(a+n*h);
for(i=1;i<=n-1;i+=2)
   sum+=4*f(a+i*h)+2*f(a+(i+1)*h);
 sum*=h/3.;
 printf("Value of the integration is %f ",sum);
} /* main */
/* definition of the function f(x) */
float f(float x)
 {
  return(x*x*x);
 }
```

```
Enter the values of a, b 0 1
Enter the value of subintervals n 100
Value of the integration is 3.750000
```

3.5 Ordinary Differential Equations

3.5.1 Euler's Method

This is the most simple but crude method to solve differential equation of the form

$$\frac{dy}{dx} = f(x,y),\ y(x_0) = y_0 \tag{3.27}$$

Let $x_1 = x_0 + h$, where h is small. Then by Taylor's series

$$y_1 = y(x_0 + h) = y_0 + h\left(\frac{dy}{dx}\right)_{x0} + \frac{h^2}{2}\left(\frac{d^2y}{dx^2}\right)_{c_1},$$

where c_1 lies between x_0 and x

$$= y_0 + hf(x_0, y_0) + \frac{h^2}{2}y''(c_1) \tag{3.28}$$

If the step size h is chosen small enough, then the second-order term may be neglected and hence y_1 is given by

$$y_1 = y_0 + hf(x_0, y_0). \tag{3.29}$$

Similarly,

$$y_2 = y_1 + hf(x_1, y_1) \tag{3.30}$$

$$y_3 = y_2 + hf(x_2, y_2) \tag{3.31}$$

and so on.

In general,

$$y_{n+1} = y_n + hf(x_n, y_n),\ n = 0, 1, 2, \ldots \tag{3.32}$$

This method is very slow. To get a reasonable accuracy with Euler's methods, the value of h should be taken as small.

Algorithm 3.7 (Euler's method). This algorithm finds the solution of the equation $y' = f(x, y)$ with $y(x_0) = y_0$ over the interval $[x_0, x_n]$, by Euler's method

$$y_{i+1} = y_i + hf(x_i, y_i),\ i = 0, 1, 2, \ldots, n - 1.$$

Algorithm Euler
Input function $f(x, y)$
Read x_0, y_0, x_n, h //x_0, y_0 are the initial values and x_n is the last value of x//
//where the process will terminate; h is the step size//
for $x = x_0$ to x_n step h do
$y = y_0 + h * f(x, y0)$;
Print x, y;
$y_0 = y$;
endfor;
end Euler

```
/* Program Euler
   Solution of a differential equation of the form y'=f(x,y),
   y(x0)=y0 by Euler's method. */
#include<stdio.h>
#include<math.h>
void main()
{
  float x0,y0,xn,h,x,y;
  float f(float x, float y);
```

```
 printf("Enter the initial (x0) and final (xn) values of x ");
 scanf("%f %f",&x0,&xn);
 printf("Enter initial value of y ");
 scanf("%f",&y0);
 printf("Enter step length h ");
 scanf("%f",&h);
 printf(" x-value y-value\n");
 for(x=x0;x<xn;x+=h)
  {
    y=y0+h*f(x,y0);
    printf("%f %f \n",x+h,y);
    y0=y;
  }
} /* main */
/* definition of the function f(x,y) */
float f(float x, float y)
{
 return(x*x+x*y+2);
}
```

```
Enter the initial (x0) and final (xn) values of x
0 .2
Enter initial value of y 1
Enter step length h .05
  x-value   y-value
  0.050000  1.100000
  0.100000  1.202875
  0.150000  1.309389
  0.200000  1.420335
```

3.5.2 Runge-Kutta Methods

The Euler's method is less efficient in practical problems because if h is not sufficiently small then this method gives inaccurate result.

The Runge-Kutta methods give more accurate result. One advantage of this method is it requires only the value of the function at some selected points on the subinterval and it is stable, and easy to program.

The Runge-Kutta methods perform several function evaluations at each step and avoid the computation of higher order derivatives. These methods can be constructed for any order, i.e., second, third, fourth, fifth, etc. The fourth-order Runge-Kutta method is more popular.

Second-order Runge-Kutta method

The second-order Runge-Kutta formula is

$$y_1 = y_0 + \tfrac{1}{2}(k_1 + k_2) \tag{3.33}$$

where

$k_1 = hf(x_0, y_0)$ and

$k_2 = hf(x_0 + h, y_0 + hf(x_0, y_0)) = hf(x_0 + h, y_0 + k_1)$.

The local truncation error of this formula is of $O(h^3)$.

Fourth-order Runge-Kutta Method

The fourth-order Runge-Kutta method is

$$y_1 = y_0 + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{3.34}$$

where

$$k_1 = hf(x_0, y_0)$$
$$k_2 = hf(x_0 + h/2, y_0 + k_1/2)$$
$$k_3 = hf(x_0 + h/2, y_0 + k_2/2)$$
$$k_4 = hf(x_0 + h, y_0 + k_3).$$

Starting with the initial point (x_0, y_0), one can generate the sequence of solutions at $x_1, x_2, \ldots$ using the formula

$$y_{i+1} = y_i + \tfrac{1}{6}(k_1^{(i)} + 2k_2^{(i)} + 2k_3^{(i)} + k_4^{(i)}) \tag{3.35}$$

where

$$k_1^{(i)} = hf(x_i, y_i)$$

$$k_2^{(i)} = hf(x_i + h/2, y_i + k_1^{(i)}/2)$$

$$k_3^{(i)} = hf(x_i + h/2, y_i + k_2^{(i)}/2)$$

$$k_4^{(i)} = hf(x_i + h, y_i + k_3^{(i)})$$

Algorithm 3.8 (Fourth-order Runge-Kutta method). This algorithm finds the solution of the differential equation $y' = f(x, y)$ with $y(x_0) = y_0$ using fourth-order Runge-Kutta method, i.e., using the formula

$$y_{i+1} = y_i + \frac{1}{6}[k_1 + 2(k_2 + k_3) + k_4]$$

within the interval $[x_1, x_n]$ at step h.

Algorithm RK4
Input function $f(x, y)$;
Read x_0, x_n, y_0, h; //initial and final value of x, initial value of y and step size.//
Set $y = y_0$;
for $x = x_0$ to x_n step h do
    Compute $k_1 = h * f(x, y)$;
    Compute $k_2 = h * f(x + h/2, y + k_1/2)$;
    Compute $k_3 = h * f(x + h/2, y + k_2/2)$;
    Compute $k_4 = h * f(x + h, y + k_3)$;
    Compute $y = y + [k_1 + 2(k_2 + k_3) + k_4]/6$;
    Print x, y;
endfor;
end RK4

```
/* Program Fourth_Order_Runge-Kutta
   Solution of a differential equation of the form y'=f(x,y),
   y(x0)=y0 by fourth order Runge-Kutta method. */
#include<stdio.h>
#include<math.h>
void main()
{
 float x0,y0,xn,h,x,y,k1,k2,k3,k4;
 float f(float x, float y);
 printf("Enter the initial values of x and y ");
 scanf("%f %f",&x0,&y0);
 printf("Enter last value of x ");
 scanf("%f",&xn);
 printf("Enter step length h ");
 scanf("%f",&h);
 y=y0;
 printf(" x-value y-value\n");
 for(x=x0;x<xn;x+=h)
```

```
    {
     k1=h*f(x,y);
     k2=h*f(x+h/2,y+k1/2);
     k3=h*f(x+h/2,y+k2/2);
     k4=h*f(x+h,y+k3);
     y=y+(k1+2*(k2+k3)+k4)/6;
     printf("%f  %f\n",x+h,y);
    }
} /* main */
/* definition of the function f(x,y). */
float f(float x, float y)
 {
  return(x*x-y*y+y);
 }
```

```
Enter the initial values of x and y 0 2
Enter last value of x 0.5
Enter step length h 0.1
  x-value  y-value
  0.100000 1.826528
  0.200000 1.695464
  0.300000 1.595978
  0.400000 1.521567
  0.500000 1.468221
```

3.6 Fitting of a Straight Line

Let

$$y = a + bx \tag{3.36}$$

be the equation of a straight line, where a and b are two parameters whose values are to be determined. Let (x_i, y_i), $i = 1, 2, \ldots, n$, be a given sample of size n.

Here S is given by

$$S = \sum_{i=1}^{n} (y_i - Y_i)^2 = \sum_{i=1}^{n} (y_i - a - bx_i)^2$$

The normal equations are

$$\frac{\partial S}{\partial a} = -2\sum(y_i - a - bx_i) = 0$$

$$\frac{\partial S}{\partial b} = -2\sum(y_i - a - bx_i)x_i = 0$$

which give on simplification,

$$\sum y_i = na + b\sum x_i$$

$$\sum x_i y_i = a\sum x_i + b\sum x_i^2 . \tag{3.37}$$

The solution of these equations is

$$b = \frac{n\Sigma x_i y_i - \Sigma x_i \Sigma y_i}{n\Sigma x_i^2 - (\Sigma x_i)^2} \text{ and } a = \frac{1}{n}\left[\sum y_i - b\sum x_i\right]. \tag{3.38}$$

But, when the sample size is large or the data are large then there is a chance for data overflow while computing $\Sigma x_i y_i$, Σx_i^2 and $(\Sigma x_i)^2$. Then the suggested expression for b is

$$b = \frac{\Sigma(x - \bar{x})(y - \bar{y})}{\Sigma(x_i - \bar{x})^2} \text{ , where } \bar{x} = \frac{1}{n}\sum x_i,\ \bar{y} = \frac{1}{n}\sum y_i \tag{3.39}$$

Let this solution be denoted by $a = a^*$, $b = b^*$. Then the fitted straight line is

$$y = a^* + b^* x. \tag{3.40}$$

Algorithm 3.9 (Straight line fit). This algorithm fits a straight line for the given data points (x_i, y_i), $i = 1, 2, \ldots, n$, by least squares method.

Algorithm Straight_Line

Step 1. Read (x_i, y_i), $i = 1, 2, \ldots, n$.

Step 2. //Computation of x and y//
Set $sx = 0$, $sy = 0$.
for $i = 1$ to n do
$sx = sx + x_i$ and $sy = sy + y_i$
endfor;
Compute $\bar{x} = sx/n$ and $\bar{y} = sy/n$;

Step 3. Set $sxy = 0$, $sx2 = 0$;
for $i = 1$ to n do
$sxy = sxy + (x_i - \bar{x})(y_i - \bar{y})$;
$sx2 = sx2 + (x_i - \bar{x})^2$;
endfor;

Step 4. Compute $b = sxy/sx2$; and $a = \bar{y} - b\bar{x}$;

Step 5. Print 'The fitted line is y=',a,'+', b, 'x'.
end Straight_Line

```
/* Program Straight Line Fit
 Program to fit a straight line for the given data points
 (x[i],y[i]), i=1, 2, . . ., n, by least squares method.*/
#include<stdio.h>
#include<math.h>
void main()
{
 int n,i; float x[50], y[50], sx=0,sy=0,sxy=0,sx2=0,xb,yb,a,b;
 char sign;
 printf("Enter the sample size and the sample (x[i],y[i]) ");
 scanf("%d",&n);
 for(i=1;i<=n;i++) scanf("%f %f",&x[i],&y[i]);
 for(i=1;i<=n;i++){
   sx+=x[i]; sy+=y[i];
  }
 xb=sx/n; yb=sy/n;
 for(i=1;i<=n;i++){
   sxy+=(x[i]-xb)*(y[i]-yb);
   sx2+=(x[i]-xb)*(x[i]-xb);
  }
 b=sxy/sx2; a=yb-b*xb;
 sign=(b<0)? '-':'+';
 printf("\nThe fitted line is y = %f %c %f x",a,sign,fabs(b));
 } /* main */
```

```
Enter the sample size and the sample (x[i],y[i]) 5
1   12
4   13
6   10
7   8
10  3
The fitted line is y = 15.097345 - 1.053097 x
```

3.7 Summary

The algorithms and programs are designed to solve the following problems of numerical analysis. Finding of roots of an equation by bisection, fixed point iteration and Newton-Raphson method, solution of a system of linear algebraic equations by Jacobi's, Gauss-Seidal iterations methods and LU-decomposition method, integration by trapezoidal and Simpson 1/3 rules, solution of an ordinary differential equation by Euler's and Runge-Kutta methods, and fitting of a straight line based on a bivariate data.

Exercise 3

1. Write a program to find a root of the equation $f(x) = 0$ by bisection method. Use your program to find a root from each of the following equations.
   (a) $x^3 + 2x^2 - x + 7 = 0$, (b) $x^3 - 4x - 9 = 0$, (c) $\cos x = 3x - 1$.

2. Write a program to find a root of the equation $f(x) = 0$ by iteration method. Use your program to find a root from each of the following equations.
   (a) $x^3 - 5.2x^2 - 17.4x + 21.6 = 0$, (b) $x^7 + 28x^4 - 480 = 0$,
   (c) $(x - 1)(x - 2)(x - 3) = 0$, (d) $x - \cos x = 0$, (e) $x + \log x = 2$.

3. Write a program to find a root of the equation $f(x) = 0$ by Newton-Raphson method. Use your program to find a root from each of the following equations.
   (a) $2x - \cos x - 1 = 0$, (b) $x^5 + 3x^2 - 1 = 0$, (c) $x^2 - 2 = 0$.

4. Write a program to find the value of $\sqrt[5]{a}$ using Newton-Raphson method.

5. Write a program to solve a system of linear equations using Jacobi's iteration method. Test your program for the following equations
   (a) $9x + 2y + 4z = 20$
   $x + 10y + 4z = 6$
   $2x - 4y + 10z = -15$.

   (b) $5x - y + 2z = 10$
   $-2x + 10y + 4z = 8$
   $x - 3y + 8z = 12$.

6. Write a program to solve a system of linear equations using Gauss-Seidal's iteration method. Test your program for the following equations

(a) $-9x + 2y + 2z = 5$
$x + 10y - 4z = 7$
$x - 5y + 12z = 8.$

(b) $7x - y + z = 9$
$3x + 8y + 2z = 9$
$2x - 4y - 4z = 10.$

7. Write a program to evaluate an integration by trapezoidal rule. Use this program to find the following integrations.

(a) $\int_0^2 (1 + e^{-x} \sin 4x)dx$ taking $n = 100$.

(b) $\int_0^5 e^{-x^2} dx$, taking $h = 0.01$.

(c) $\int_0^1 x^4 e^{x-1} dx$, taking $n = 60$.

8. Write a program to evaluate an integration by Simpson 1/3 rule. Use this program to find the following integrations.

(a) $\int_0^{\pi/2} \frac{dx}{\sin^2 x + 2\cos^2 x}$, taking $n = 100$.

(b) $\int_0^1 e^{x+1} dx$, taking $n = 200$.

(c) $\int_{1.0}^{1.8} \frac{e^x + e^{-x}}{2} dx$, taking $h = 0.05$.

9. Write a program to solve the following differential equations by Euler's method.

(a) $\frac{dy}{dx} = 3x^2 + y$, $y(0) = 4$ for the range $0.1 \leq x \leq 0.5$, by taking $h = 0.1$.

(b) $y' = x^2 + y^2$, $y(0) = 0.5$, find y at $x = 0.1$ and 0.2.

10. Write a program to solve the following differential equations by fourth order Runge-Kutta's method.

(a) $5\frac{dy}{dx} = x^2 + y^2$, $y(0) = 1$, find y in the interval $0 \leq x \leq 0.4$, taking $h = 0.1$.

(b) $\frac{dy}{dx} = xy + y^2$, given that $y(0) = 1$. Taking $h = 0.2$, find y at $x = 0.2$, 0.4. 0.6.

(c) $y' = x + y$, $y(0) = 1$ within the interval [0,0.1] taking $h = 0.02$.

11. Write a program to fit a straight line from the following data.

(a)

x :	1	3	5	7	9
y :	10	13	25	23	33

(b)

x :	1951	1961	1971	1981	1991
y :	33	43	60	78	96

Find y when $x = 2001$.

(c)

x :	2	4	6	7	9
y :	6	10	8	15	20

UNIT 4 ❐ DATA STRUCTURES

Data structure is the main part to develop an algorithm. An efficient algorithm cannot be designed without appropriate use of data structure. Different kinds of data structures are available to design an algorithm and each data structure has a special feature. So we have to learnt all the commonly used data structures before development of an efficient algorithm. In this book commonly used data structures, viz., arrays, stacks, queues and linked lists are studied.

4.1 Objectives

After going through this unit you will be able to learn

(i) What is data structure?
(ii) Asymptotic notations
(iii) The data structure array
(iv) Stack and its applications
(v) Queue and its applications
(vi) Linked list its applications.

4.2 Asymptotic Notations

To analysis the performance of an algorithm there are several kinds of mathematical notations are used. Some of them are presented here.

Def. 4.2.1 ***O*-notation (big-oh).** *To represent* **asymptotic upper bound** *the O-notation is used. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}. \quad (4.1)$$

In other words, $f(n) = O(g(n))$ iff

$$\lim_{x \to \infty} \frac{f(n)}{g(n)} = \text{a non-zero constant.}$$

Thus O-notation is used to represents an upper bound of a function, to within a constant factor.

The *O*-notation is used to express an upper bound, we might also wish to determine a function which is a lower bound.

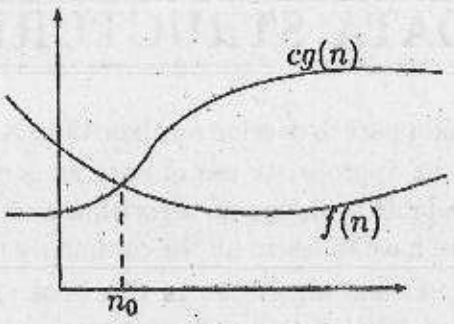


Figure 4.1: $f(n) = O(g(n))$.

Some properties of O-notation

(i) If $A(n) = a_k n^k + \ldots + a_1 n + a_0$ then $A(n) = O(n^k)$, k is independent of n.

(ii) $O(f(n)) + O(g(n)) = O(\max_n\{f(n), g(n)\})$

(iii) $O(f_1(n)) + O(f_2(n)) + \ldots + O(f_k(n)) = O(\max_n\{f_1(n), f_2(n), \ldots, f_k(n)\})$, k is independent of n.

(iv) $c\ O(f(n)) = O(f(n))$. c is independent of n.

(v) $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$.

(vi) $n^k \times O(f(n)) = O(n^k f(n))$. k is independent of n.

(vii) $O(n^k) \times O(n^l) = O(n^{k+l})$, k and l are independent of n.

Def. 4.2.2 Ω-notation. *The Ω-notation provides an* **asymptotic lower bound**. *For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions*

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}. \quad (4.2)$$

Def. 4.2.3 θ-notation (tight bound). *For a given function $g(n)$, we denote by $\theta(g(n))$ the set of functions*

$$\theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}. \quad (4.3)$$

In other words, for all $n \ge n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an asymptotically tight bound for $f(n)$.

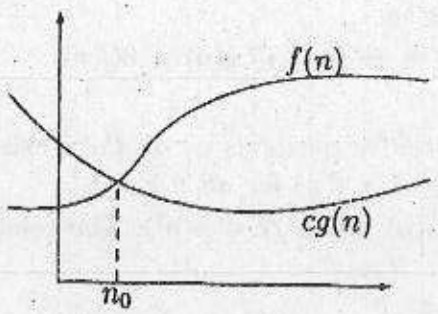


Figure 4.2 : $f(n) = \Omega(g(n))$.

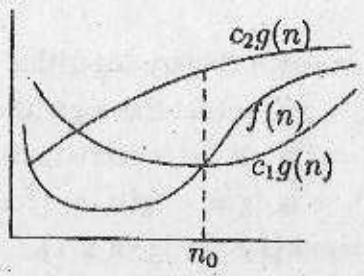


Figure 4.3 : $f(n) = \theta(g(n))$.

Def. 4.2.4 o-notation (little-oh). *The o-notation is used to denote an upper bound that is not asymptotically tight. We define $o(g(n))$ as the set*

$o(g(n)) = \{f(n)$: *for any positive constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$}.* (4.4)

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for some constants $c > 0$ but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for all constants $c > 0$, i.e.,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \tag{4.5}$$

Def. 4.2.5 ω-notation. *Similar to O-notation and o-notation, ω-notation is defined in contrast of Ω-notation. The ω-notation is used to denote a lower bound that is not asymptotically tight.*

One way to define it is by

$$f(n) \in \omega(g(n)) \text{ iff } g(n) \in o(f(n)).$$

Formally,

$\omega(g(n))$= {$f(n)$: *for any positive constants* $c > 0$, *there exists a constant* $n_0 > 0$ *such that* $0 \leq cg(n) < f(n)$ *for all* $n \geq n_0$}. (4.6)

For example, $n^2/2 = \omega(n)$ but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{x \to \infty} \frac{f(n)}{g(n)} = \infty .$$

Different kind of logarithms are used to analysis the algorithms. They are defined in the following.

$\lg n = \log_2 n$ (binary logarithm)

$\ln n = \log_e n$ (natural logarithm)

$\lg^k n = (\lg n)^k$ (exponentiation)

$\lg^{(2)} n = \lg \lg n = \lg(\lg n)$ (composition)

$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$.

4.3 Time and Space Complexities

The **time complexity** of an algorithm is given by a function $T(n)$ which is the maximum over all input of size n, of the sum of the time required by each instruction executed. For a problem of size n, the maximum time required to solve the problem over all input sequences of that size is called the **worst case** time complexity. The space complexity of an algorithm is the number of memory units used. The space required by an algorithm is, therefore, the maximum number of memory required at any time during the course of execution of the algorithm.

If $T(n)$ is a polynomial function of n then the algorithm is said to take a polynomial time (space). If $T(n)$ is a polynomial of degree one then we say that the algorithm takes linear time (space).

An algorithm is considered to be **efficient** if it takes a time which is a polynomial function of the problem size. The problems for which such algorithms exist belong to the class P. Problems for which such algorithms do not seem to exist form the class of NP-complete problems.

4.4 Data Structure

Before presenting the definition of data structure we consider an example of a data structure, 'natural number'.

Suppose we want to define the data structure Naturel Number (abbreviated by NatNo) where NatNo= {0, 1, 2, ...,} with the four operations being a test for zero, addition, equality and successor. The following notations can be used.

Structure NatNo

declare

1 ZERO() → NatNo
2 ISZERO(NatNo) → boolean
3 SUCC(NatNo) → NatNo
4 ADD(NatNo,NatNo) → NatNo
5 EQ(NatNo,NatNo) → boolean
6 for all x, y ∈ NatNo let
7 ISZERO(ZERO)::=true; ISZERO(SUCC(x))::=false;
8 ADD(ZERO,y)::=y; ADD(SUCC(x),y)::=SUCC(ADD(x, y));
9 EQ(x,ZERO)::=if ISZERO(x) then true else false;
10 EQ(ZERO,SUCC(y))::=false; EQ(SUCC(x),SUCC(y))::=EQ(x, y);
11 Endfor;

End NatNo.

In the declaration section five functions are defined by giving their names, inputs and outputs. ZERO is a constant function which means it takes no input arguments and its result is the natural number zero, written as ZERO. ISZERO is a boolean function whose result is either true or false. SUCC stands for successor. Using the functions ZERO and SUCC we can define all of the natural numbers as :

ZERO, 1=SUCC(ZERO), 2=SUCC(SUCC(ZERO)), and so on.

The rules on line 8 tell us exactly how addition operation works e.g., if we want to add two and three we would get the following sequence of expressions : ADD(SUCC(SUCC(ZERO)), SUCC(SUCC(ZERO))) which, by line 8 equals SUCC(ADD(SUCC(ZERO), SUCC(SUCC(SUCC(ZERO))) which by line 8 equals SUCC(SUCC(SUCC(SUCC(SUCC(ZERO))))) of course, this is not the way to implement addition. In practice we use bit string which is a data structure that is usually provided on our computers. But, however the ADD operation is implemented, it must obey these rules. Hopefully, this motivates the following definition of data structure.

Def. 4.4.1 *A* **data structure** *is a set of domains D, a designation domain* $d \in D$*, a set of functions F and a set of axioms A. The triplets (D, F.A) denotes the data structure d and it will usually be abbreviated by writing d.*

In previous example,

d = NatNo,

D = {NatNo, boolean},

F = {ZERO, ISZERO, SUCC, ADD},

A = {lines 7 through 10 of the structure NatNo}.

There are different kind of data structures are used to solve problems with the help of computer. The most useful data structures are array (one and two dimensions), stack, queue, linked lists, etc.

4.5 Arrays

One of the simplest generalizations of a linear list is a two-dimensional or higher dimensional arrays of information, e.g., consider the case of an $m \times n$ matrix

$$\begin{bmatrix} A[1,1] & A[1,2] & \cdots & A[1,n] \\ \vdots & \vdots & \vdots & \vdots \\ A[m,1] & A[m,2] & \cdots & A[m,n] \end{bmatrix} \tag{4.7}$$

In this two-dimensional array, each node $A[j, k]$ belongs to two linear lists: the 'row j' list $A[j, 1], A[j, 2], \ldots, A[j, n]$ and the 'column k' list $A[1, k], A[2, k], \ldots, A[m, k]$. These orthogonal row and column lists essentially account for the two-dimensional structure of a matrix. Similar, remarks apply to higher dimensional arrays of information.

Sequential allocation

When an array is stored in sequential memory locations, storage is usually allocated so that,

$$LOC(A[j, k]) = a_0 + a_1 j + a_2 k \tag{4.8}$$

where a_0, a_1, a_2 are constants and $LOC(A[j, k])$ represents the location of the element $A[j, k]$ in the memory. Let us consider a more general case. Suppose

we have a four dimensional array with one word elements $Q[i, j, k, l]$ for $0 \le i \le 2$, $0 \le j \le 4$, $0 \le k \le 10$, $0 \le l \le 2$ we should like to allocate storage so that

$$LOC(Q[i, j, k, l]) = a_0 + a_1 i + a_2 j + a_3 k + a_4 l. \tag{4.9}$$

This means that a change in i, j, k or l leads to a readily calculated change in the location of $Q[i, j, k, l]$. The most natural (and most commonly used) way to allocate storage is to be the array applied in memory in the 'lexicographic order' of its indices, sometime called 'row major order'.

The elements of the array Q are

$Q[0, 0, 0, 0]$, $Q[0, 0, 0, 1]$, $Q[0, 0, 0, 2]$, $Q[0, 0, 1, 0]$, $Q[0, 0, 1, 1]$, ..., $Q[0, 0, 10, 2]$, $Q[0, 1, 0, 0]$, ..., $Q[0, 4, 10, 2]$, $Q[1, 0, 0, 0]$, ..., $Q[2, 4, 10, 2]$.

It is easy to see that this order satisfies the requirements of (4.9), and we have

$$LOC(Q[i, j, k, l]) = LOC(Q[0, 0, 0, 0]) + 165i + 33j + 3k + l. \tag{4.10}$$

In general, given a k-dimensional array with c-word elements $A[i_1, i_2, \ldots, i_k]$ for $0 \le i_1 \le d_1$, $0 \le i_2 \le d_2$, ..., $0 \le i_k \le d_k$, we can store it in memory as

$$\begin{aligned} LOC(A[i_1, i_2, \ldots, i_k]) &= LOC(A[0, 0, \ldots, 0]) + c(d_2 + 1) \ldots (d_k + 1)i_1 \\ &\quad + \ldots + c(d_k + 1)i_{k-1} + ci_k \\ &= LOC(A[0, 0, 0, \ldots, 0]) + \sum_{1 \le r \le k} a_r i_r \end{aligned} \tag{4.11}$$

where $a_r = c \prod_{r < s \le k} (d_s + 1)$

We want to store the triangular matrix $A[j, k]$ for say $0 \le k \le j \le n$; (column major order)

$$\begin{bmatrix} A[0,0] & & & \\ A[1,0] & & & \\ A[2,0] & A[2,1] & A[2,2] & \\ \cdots & \cdots & \cdots & \\ A[n,0] & A[n,1] & \cdots & A[n,n] \end{bmatrix} \tag{4.12}$$

We may know that all other entries are zero, or that $A[j, k] = A[k, j]$, so only half of the values need to be stored. If we want to store the lower triangular matrix of the form (4.12) we need only $\frac{1}{2}(n + 1)(n + 2)$ consecutive memory positions. We are forced to give up the possibility of linear allocation as in equation (4.8), but we can now ask instead for an allocation arrangement of the form

$$LOC(A[j, k]) = a_0 + f_1(j) + f_2(k) \tag{4.13}$$

where f_1 and f_2 are functions of one variable.

In turns out that lexicographic condition (4.13) and assuming one-word entries we have in fact the rather simple formula

$$LOC(A[j,k]) = LOC(A[0,0]) + \frac{j(j+1)}{2} + k \tag{4.14}$$

If we store the matrix A in the array B then the element $A[j, k]$ is stored in $B[p]$, where

$$p = LOC(A[0,0]) + \frac{j(j+1)}{2} + k \quad \text{for given } j \text{ and } k. \tag{4.15}$$

There is a far better way to store two triangular matrices if we are fortunate enough to have two of them with the same size. If $A[j, k]$ and $B[j, k]$ are both to be stored for $0 \le k \le j \le n$ we can fit them both into a single matrix $C[i, j]$ for $0 \le j \le n$, $0 \le k \le n + 1$ using the convention

$$A[j, k] = C[j, k], B[j, k] = C[k, j + 1]. \tag{4.16}$$

Thus

$$\begin{bmatrix} C_{00} & C_{01} & \cdots & C_{0n+1} \\ C_{10} & C_{11} & \cdots & C_{1n+1} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n0} & C_{n1} & \cdots & C_{nn+1} \end{bmatrix} = \begin{bmatrix} A_{00} & B_{00} & B_{10} & \cdots & B_{n0} \\ A_{10} & A_{11} & B_{11} & \cdots & B_{n1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{n0} & A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix}$$

The two triangular matrices are packed together tightly within the space of $(n+1)(n+2)$ locations and we have linear addressing as in (4.16). The generalization of triangular matrices to higher dimensions is called a tetrahedral array.

4.6 Stacks

A stack is an ordered list in which all insertions and deletions are made at one end, called the top. The stacks are sometimes referred to as Last In First Out (LIFO) lists.

Usually the following operations are performed in the stack.

`create(S)` which creates S as an empty stack.

`push(i,S)` which inserts the elements i onto the stack S and returns the modified stack.

`pop(S)` which removes the top elements of stack S and returns the modified stack.

`top(S)` which returns the top element of stack S.

`isempty(S)` which returns true if S is empty else false.

These five functions constitute a working definition of a stack. However, we choose to represents a stack, it must be possible to build these operations.

The simplest way to represent a stack is by using a one-dimensional array, say STACK(1 : n), where n is the maximum number of allowable entries. The first or bottom element in the stack will be stored at STACK(1), the second at STACK(2) and ith at STACK(i).

Associated with the array there will be a variable 'top', which points to the top element in the stack. With this assumption, the functions used in stack are implemented below.

`create()` declare the array STACK(1 : n) and set top $\leftarrow$ 0;

`isempty(STACK)` = true (if top=0) false otherwise.

`top(STACK)` = error (if top=0) STACK(top) otherwise.

The algorithms for the functions push and pop are presented in the following.

```
Procedure push(x, S)
//Inserts an object x into the stack S of maximum size n; 'top' is the number
of elements currently in the stack S.//
if top ≥ n then
    print 'stack full'; stop;
else
    top ← top+1;
    S(top) ← x;
endif;
end push.
```

```
Procedure pop(S)
//Removes the top element from the stack S and stores it in x unless stack S
is empty.//
if top ≤ 0 then
    print 'stack empty'; stop;
else
    x ← S(top);
    top ← top-1;
endif;
end pop.
```

One ideal example of application of stack is presented in the next section.

Operators	Priority
^	3
* /	2
+ -	1
( )	0

Table 4.1: Priority of operators

4.7 Evaluation of Expression

An expression is made up of operands and operators. The expression $A/B \wedge C + D * E - A * C$ has five operands A,B,C,D,E. Though these are all one letter variables, operands can be any legal variable name or constant in any programming language. In any expression the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. There are the five basic arithmetic operators viz., plus (+), minus (-), times (*), divide (/) and exponent ($\wedge$).

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out.

To fix the order of evaluation, we assign to each operator a priority. Then within any pair of parentheses we understand that operators with the highest priority will be evaluated first. Operators ($+, -, *, /, \wedge$) the following is the order of precedence (highest to lowest):

Compiler can't proceed the mathematical expression or logical expression which is written in conventional way. Before evaluation of an expression the given (conventional) form is converted into another expression called **postfix** expression. If E is an expression with operators and operands, the conventional way of writing E is called **infix**, because the operators come in-between the operands. The postfix (**prefix**) form of an expression calls for each operator to appear after (before) its operands, e.g., the infix expression $A * B/C$ has postfix form $AB * C/$.

If we study the postfix form of $A * B/C$ we see that the multiplication comes immediately after its two operands A and B. Now, imagine that $A*B$ is computed and stored in T. Then we have the division operator /, coming immediately after its two arguments T and C. That is,

$$\begin{aligned}(A * B)/C &= T/C \text{ where } T = AB * \\ &= TC/ \\ &= AB * C/\end{aligned}$$

Consider another expression

$$(A + B) * C = T * C \text{ where } T = AB +$$
$$= TC *$$
$$= AB + C*$$

Example 4.7.1 Convert the following infix expression to postfix expression:

$$A/B \wedge C + D * E - A * C.$$

Solution. In this expression $\wedge$ has highest priority, so $B \wedge C$ is computed first and stored it into T_1 as $T_1 = BC \wedge$. That is,

$A/B \wedge C + B * E - A * C$
$= A/T_1 + B * E - A * C$ where $T_1 = BC \wedge$
$= T_2 + T_3 - T_4$ $T_2 = AT_1/$, $T_3 = BE*$, $T_4 = AC*$
$= T_5 - T_4$ $T_5 = T_2T_3+$
$= T_5T_4-$
$= T_2T_3 + T_4-$
$= AT_1/BE * +AC * -$
$= ABC \wedge /BE * +AC * -$

Some more examples:

Infix	Postfix
$A + B$	$AB+$
$A + B - C$	$AB + C -$
$(A + B) * (C - D)$	$AB + CD - *$
$A \wedge B * C - D + E/F/(G + H)$	$AB \wedge C * D - EF/GH + /+$
$((A + B) * C - (D - E)) \wedge (F + G)$	$AB + C * DE - - FG + \wedge$
$A - B/(C * D \wedge E)$	$ABCDE \wedge / -$

It may be noted that the parentheses presented in the infix expression are removed from the postfix expression. That is, the length (number of characters) of the postfix expression is less than the corresponding infix expression.

4.7.1 Evaluation of a postfix expression

Each operator in a postfix string refers to the previous two operands in the string. Obviously, one of these two operands may itself be the result of applying a previous operator. Suppose that each time we read an operand we push it onto a stack. When we reach an operator, its operands will be the top two elements

on the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack so that it will be available for use as an operand of the next operator. The following algorithm evaluates an expression in postfix using this method.

Algorithm **Eval-Postfix**

```
//The array 'postfix[]' is the postfix expression and 'opstack' is the stack of
operands.//
Initially opstack=empty;
for i = 1 to N //where N is the length of the array postfix[]//
    symb=postfix[i];
    if(symb=operand)
      push(symb,opstack);
    else //symb=operator//
      //removes top element from the stack opstack and stores in opnd2//
        opnd2=pop(opstack);
      //removes next element from the stack opstack and stores in opnd1//
        opnd1=pop(opstack);
        value=result of applying symb to opnd1 and opnd2;
          //value = (opnd1) symb (opnd2) //
          push(value, opstack); //value placed on the top of the stack//
      end //else//
    endif;
endfor;
value of expression = pop(opstack);
end Eval-postfix.
```

Limitations of the algorithm Eval-Postfix

Each character of postfix string is assumed to be either an operator or an operand. Thus, if the expression contains a variable whose length is more than one, then this algorithm does not work. Also, this algorithm fails when the expression consists any functions.

4.7.2 Conversion of an infix expression to a postfix expression

It is well known that expressions within innermost parentheses must first be converted to postfix so that they can then be treated as single operands. In this fashion parentheses can be successively eliminated until the entire expression

is converted. The last pair of parentheses to be opened within a group of parentheses encloses the first expression within that group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack.

Since priority of operators plays an important role in transforming infix to postfix, let us assume the existence of a function P(op), where 'op' is a character representing operator.

Here we define the function P as follows.

$P('(')=P(')')=0$
$P('+')=P('-')=1$
$P('*')=P('/')=2$
$P('\wedge')=3.$

Algorithm **Infix2Postfix**

```
//Converts an infix expression to a postfix expression.//
Initially top=0; i = 0; q = 0; //'top' is the top of the stack//
N = length(infix[]);
while (i < N) do
    i = i + 1;
    case infix[i] of
    letter : q = q + 1; postfix[q]=infix[i];
             //puts infix character directly to postfix expression//
    '('    : push('(',stack); //puts left parenthesis directly to the stack//
    ')'    : x=pop(stack);
             while x ≠ (' do //removes elements from stack and put them to
                             the postfix expression until x is a closing
                             parenthesis//

             q = q + 1; postfix[q]=x;// adds to the postfix expression//
             x=pop(stack);//removes top element from the stack//
           endwhile;

operator   : while {P(infix[i]) ≤ P(stack(top))} and (top > 0) do
             x =pop(stack);//removes top element from the stack//
             q = q + 1; postfix[q]=x; // puts to the postfix expression//
           endwhile;
           push(infix[i], stack); //inserts to the stack//
```

```
        endcase;
    endwhile;
    while top > 0 do //puts remaining elements to postfix expression//
      x=pop(stack);
      q = q + 1; postfix[q]=x;
endwhile;
end Infix2Postfix.
```

4.8 Queues

A queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front.

A minimal set of useful operations on a queue includes the following :

`createq(Q)` which creates Q as empty queue;

`addq(i,Q)` which adds the element i to the rear of the queue Q and returns the new queue;

`deleteq(Q)` which removes the front element from the queue Q and returns the resulting queue;

`front(Q)` which returns the front element of Q;

`isempty(Q)` which returns true if Q is empty else false.

The representation of a finite queue in sequential locations is somewhat more difficult than a stack. In addition to a one dimensional array $Q(1 : n)$, we need two more variables, front and rear. The conventions we shall adopt for these two variables are the front is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear iff there are no elements in the queue. The initial condition then is front = rear =0.

The following implementation of the `createq`, `isemptyq` and front operations results for a queue with capacity n.

`createq(Q)` declares the array $Q(1 : n)$ and sets front $\leftarrow$ rear $\leftarrow$ 0;

`isempty(Q)` is true if (front = rear) false otherwise;

`front(Q)` gives error if (`isempty(Q)`) is true else it will return Q(front+1).

The following algorithms for addq and deleteq result :

```
function addq(x, Q)
//Inserts an object x into the queue Q represented by Q(1 : n) whose capacity n;
rear is the pointer at the end of the queue.//
if (rear=n) then
    print 'queue-full'; stop;
else
```

```
    rear ← rear+1;
    Q(rear) ← x;
endif;
end addq.
```

```
function deleteq(Q)
//Deletes an element from the queue Q represented by Q(1 : n): rear and front
are respectively the pointers at the end and front of the queue.//
if (rear = front) then
    print 'queue-empty'; stop;
else
    front ← front+1;
    x ← Q(front);
endif;
end deleteq.
```

With this setup, notice that unless the front regularly catches up with the rear and both pointers are reset to zero, then the queue-full signal does not necessarily imply that there are n elements in the queue.

A more efficient queue representation is obtained by regarding the array $Q(1 : n)$ as circular. It now becomes more convenient to declare the array as $Q(0 : n - 1)$. When $rear = n - 1$, the next element is entered at $Q(0)$ in case that spot is free. *front* will always point one position counter-clockwise from the first element in the queue. Again, $front = rear$ iff the queue is empty. Initially, we have $front = rear = 0$. Figure 4.4 illustrates some of the possible configurations for a circular queue containing the four elements A, B, C, D with $n > 4$.

In order to add an element, it will be necessary to move rear one position clockwise, i.e.,

if $rear = n - 1$ then $rear \leftarrow 0$ else $rear \leftarrow rear + 1$.

Using the modulo operator which computes remainders, that is just

$$rear \leftarrow (rear + 1) \bmod n.$$

Similarly, it will be necessary to move front one position clockwise each time a deletion is made. Again, using the modulo operation, this can be accomplished by

$$front \leftarrow (front + 1) \bmod n.$$

An examination of the algorithms indicates that addition and delation can now be carried out in a fixed amount of time or $O(1)$.

```
function addcq(x, Q)
//Inserts an item x in the circular queue stored in Q(0 : n - 1) of size n; rear
```

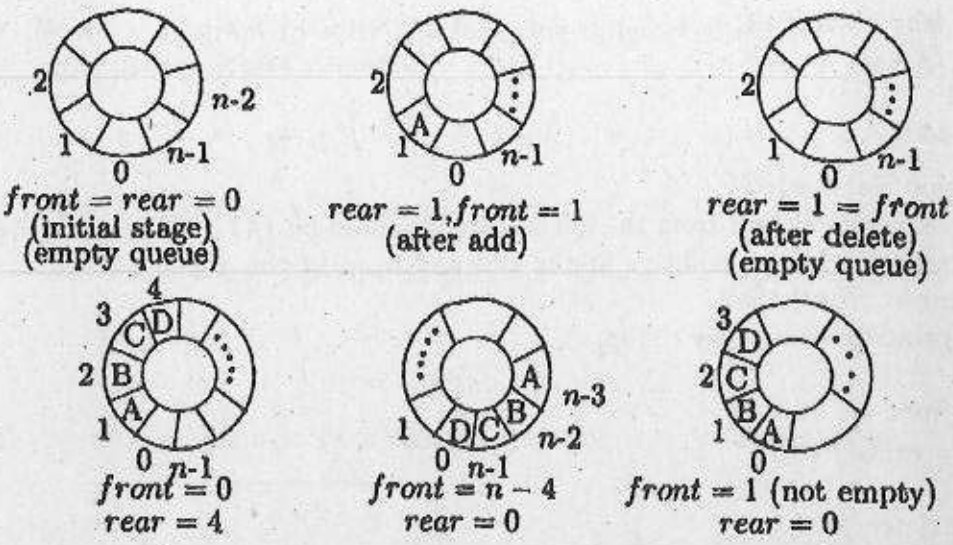


Figure 4.4: Circular representation of queue.

points to the last item and *front* is one position counterclockwise from the first item in Q.//

```
rear ← (rear + 1) mod n //advance rear clockwise
if front = rear then
    print 'queue-full'; stop;
else
    Q(rear) ← x;
endif;
end addcq.
```

```
function deletecq(Q)
//Removes the front element of the queue Q(0 : n - 1).//
if front = rear then
    print 'queue-empty'; stop;
else
    front ← (front + 1) mod n        //advance front clockwise
    x ← Q(front); //save front of queue as x for further use
endif;
end deletecq.
```

One surprising point in the two algorithms is that the test for queue full in addcq and the test for queue empty in deletecq are the same. In the case of

addcq, however, when *front* = *rear* there is actually one space free, i.e., *Q*(*rear*), since the first element in the queue is not at *Q*(*front*) but is one position clockwise from this point. However, if we insert an item here, then we will not be able to distinguish between the cases full and empty, since this insertion would leave *front* = *rear*. To avoid this, we signal queue-full, thus permitting a maximum of n - 1 rather than n elements to be in the queue at any time.

4.9 Linked Lists

One major drawback of stack and queue in array implementation is that a fixed amount of storage remains allocated to the stack or queue even when the structure is actually using a smaller amount or possibly no storage at all. Further, no more than that fixed amount of storage may be allocated, thus introducing the possibility of overflow.

In a sequential representation, the items of a stack or queue are implicitly ordered by the sequential order of storage. Thus, if *item*[x] represents an element of a queue, the next element will be *item*[x+1]. Suppose that the items of a stack or a queue were explicitly ordered, that is, each item contained within itself the address of the next item. Such an explicit ordering gives rise to a data structure, which is known as a **linear linked list**. Each item in the list is called a **node** and contains two fields, an **data** field and a **link** field (address of the next node) (see Figure 4.5). The data field holds the actual element on the list.

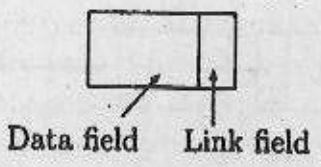


Figure 4.5: A node.

The link field contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a **pointer**. The entire linked list is accessed from an external pointer **list**, which points to (contains the address of) the first node in the list. The link field of the last node in the list contains a special value, known as **null** or **nil** or 0, which is not a valid address. This **null pointer** is used to signal the end of a list. The list with no nodes on it is called the **empty list** or the **null list**. The value of the external pointer **list** to such a list is the null pointer. Thus a list can be initialized to the empty list by the operation *list* = *null*.

It is conventionally to draw linked lists as an ordered sequence of nodes with links being represented by arrows (see Figure 4.6).

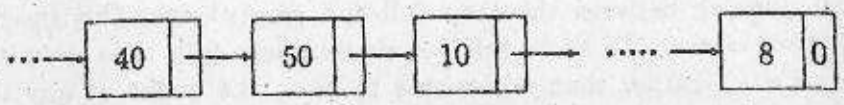


Figure 4.6: A portion of a list.

Notice that we do not explicitly put in the values of the pointers but simply draw arrows to indicate they are there. This is so that we reinforce in our own mind the facts that (i) the nodes do not actually reside in sequential locations, and that (ii) the locations of nodes may change on different runs. Therefore, when we write a program which works with lists, we almost never look for a specific address except when we test for null or 0.

Let us now see why it is easier to make arbitrary insertions and deletions using a linked list rather than a sequential list. To insert the data item 60 between 50 and 10 the following steps are needed.

(i) get a node which is currently unused, let its address be X.

(ii) set the data field of this node to 60.

(iii) set the link field of X to point to the node after 50 which contains 10.

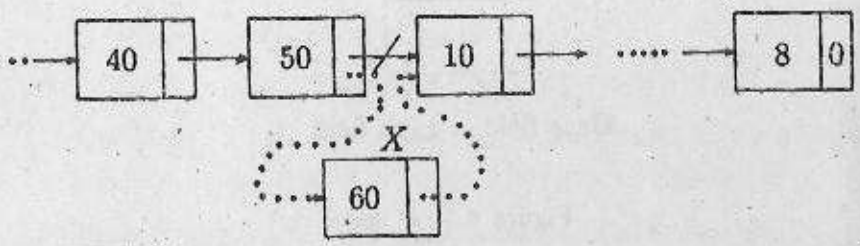


Figure 4.7: The list after insertion of the node X after the node containing 50.

The Figure 4.7 shows how we can draw the insertion using our arrow notation. The new arrows are dotted. The important thing to notice is that when we insert 60 we do not have to move any other elements which are already in the list. We have overcome the need to move data at the expense of storage needed for the second field link. But, we will see that this is not too severe a penalty.

Now, suppose we want to delete 60 from the list. All we need to do is find the element which immediately precedes 60, which is 50. Again, there is no need to move the data around. Even though the link field of 60 still contains a pointer to 10, 60 is no longer in the list.

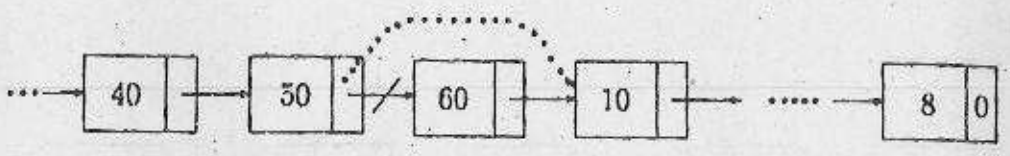


Figure 4.8: List after deletion of the node containing 60.

From our brief discussion of linked lists we see that the following capabilities are needed to make linked representations possible:

(i) A mechanism for dividing memory into nodes each having at least one link field.

(ii) A mechanism to determine which nodes are in use and which are free.

(iii) A mechanism to transfer nodes from the reserved pool to the free pool and vice-versa.

Though data and link look like conventional one dimensional arrays, it is not necessary to implement linked lists using them. For the time being let us assume that all free nodes are kept in a 'black box' called the storage pool and that there exist subalgorithms:

(i) GETNODE(X) which provides in X a pointer to a free node but if no node is free, it prints an error message and stops;

(ii) RET(X) which returns node X to the storage pool.

Example 4.9.1 Let T be a pointer to the first node to a linked list. $T = 0$ if the list has no nodes. Let X be a pointer to some arbitrary node in the list T. The following algorithms inserts a node with data field Z following the node pointed at by X.

```
Procedure insertm(T,X) //inserts Z after the node X in the list T
    call GETNODE(I);
    data(I) ← Z; //data(I) is the data field of the node I
    if T = 0 (null) then
```

$T \leftarrow I$; //inserts into an empty list
$link(I) \leftarrow 0$; //$link(I)$ is the link field of the node I
else
$link(I) \leftarrow link(X)$;
$link(X) \leftarrow I$; //inserts after X
endif;
end insertm.

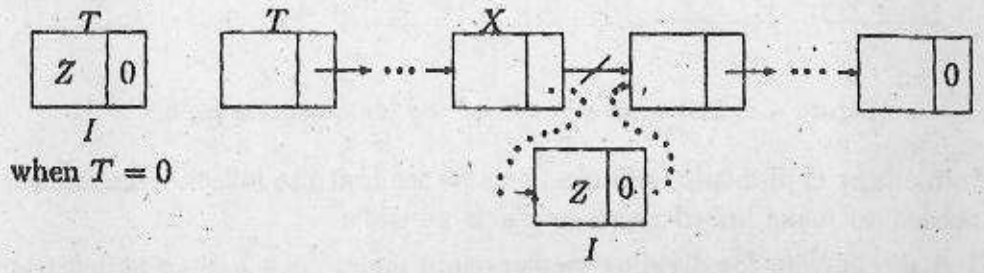


Figure 4.9: Insertion of a node into the list T.

The following algorithm inserts an element at the beginning of the list T.

Procedure insertb(T)
//insert Z at the beginning of the list.//
call GETNODE(I);
$data(I) \leftarrow Z$;
$link(I) \leftarrow T$;
$T \leftarrow I$;
end insertb.

The following algorithm inserts an element Z at the end of the list T, where X is the last element of the list.

Procedure inserte(T,X)
//inserts Z at the end of the list T.//
call GETNODE(I);

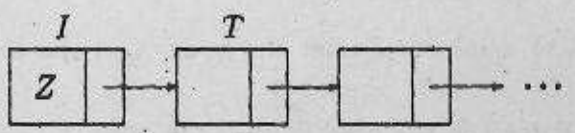


Figure 4.10: Insertion at the beginning of the list.

```
    data(I) ← Z;
    link(X) ← I;
    link(I) ← 0;
end inserte.
```

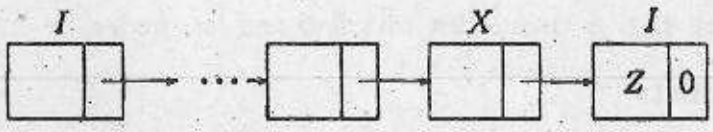


Figure 4.11: Insertion at the end of the list.

Example 4.9.2 Let X be a pointer to some node in a linked list T. Let Y be the node preceding X. $Y = 0$ if X is the first node in T (i.e., if $X = T$). The following algorithm deletes node X from T.

```
Procedure delete(X, Y, T)
if Y = 0 then
    T ← link(T); //removes the first node
elseif
    link(X) = 0 then link(Y) ← 0; //removes last node
else
    link(Y) ← link(X); //removes an interior node
endif;
call RET(X); //returns node to storage poll.
end delete.
```

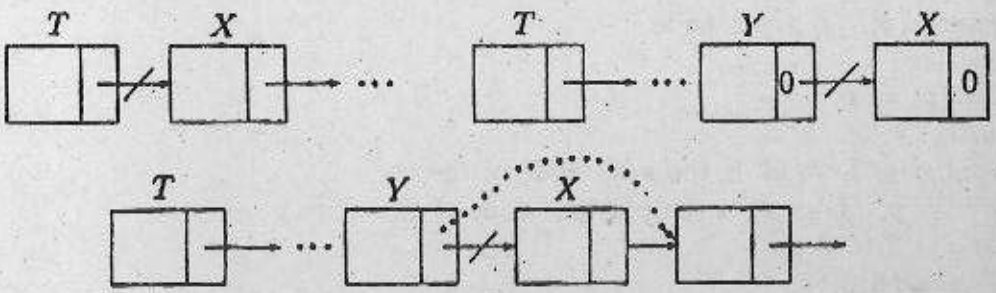


Figure 4.12: Deletion of the node from the list T.

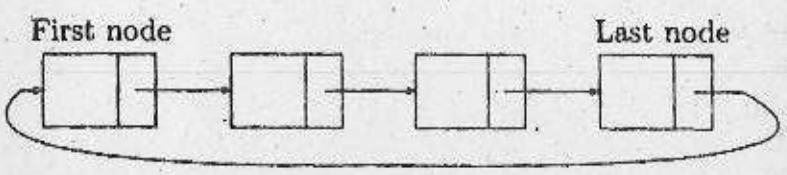


Figure 4.13: A circular list with first and last nodes.

4.9.1 Circular lists

Given a pointer p to a node in a linear list, we cannot reach any of the nodes that precede $node(p)$. If a list is traversed, the external pointer to the list must be preserved to be able to reference the list again.

Suppose that a small change is made to the structure of a linear list, so that the next field in the last node contains a pointer back to the first node rather than the null pointer. Such a list is called a circular list.

From any point in such a list it is possible to reach any other point in the list. If we begin at a given node and traverse the entire list, we ultimately end up at the starting point. A circular list does not have a natural 'first' or 'last' nodes.

4.9.2 Applications of linked lists

Example 4.9.3 Write an algorithm to create a single non-circular linked list containing n elements $x_1, x_2, \ldots, x_n$ and find the maximum among them.

Solution.

Algorithm MAX

//Creates a linked list of size n and finds maximum among them.//

```
    //creates the head node//
    getnode(p); // p is a node
    read x;
    data(p) = x;
    link(p) = 0;
    head = p; //'head' is the head node of the list
    prev = p; // 'prev' is the previous node of the node p
    for i = 2 to n do
      getnode(p);
      read x;
      data(p) = x;
      link(p) = 0;
```

```
        link(prev) = p;
        prev = p;
    endfor;
//end of creation of the list.
    p = head;
    max = data(p); //max represents the maximum value of the data
    p = link(p); //moves to the next node
    for i = 2 to n do
        if data(p) > max then max = data(p);
        p = link(p);
    endfor;
end MAX.
```

Example 4.9.4 Write an algorithm for performing linear search over a linked list with one link field in each node.

Solution. We assumed that the linked list is already created containing certain data. Let *head* is the head node of the list.

```
Algorithm Search-Linear
//head is the head node of the list.//
Input: a linked list and a key.
Output: true or false;
// if the key contain in the list then we set found=true otherwise found=false;
found is a boolean variable.//
    found = false;
    p = head;
    pos = 0;//pos represents the position of the node contains key in the list and
p is the address of the node containing key//
    while p ≠ 0 do
        pos = pos + 1;
        if data(p) = key then found = true and exit;
        p = link(p);
    endwhile;
    if found then write 'key is in the list at pos position of the list'.
end Search-Linear.
```

Example 4.9.5 Write an algorithm for insertion sort using single linked list (non-circular).

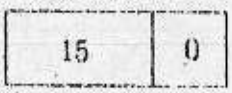
Solution. The basic step in this method is to insert a record R into a sequence of ordered records in such a way that the resulting sequence is also ordered. Here we sorting the data in ascending order.

In insertion sort we always insert a node to the list in such a way that the resulting list becomes a sorted sequence. Insertions are made in two ways. If the data of the head node is less than the input value x (at a particular instant) then we insert a new node containing x, before the head node and this new node becomes the head of the appended list, otherwise we search the appropriate position for the input value and we insert a new node containing this value.

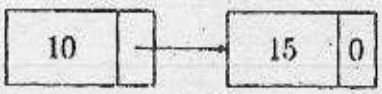
```
Algorithm Insort
//Sorts the data in ascending order by insertion sort using linked list.//
    getnode(p);
    read(x); //x is the input value//
    head = p; //create a new head node//
    data(p) = x;
    link(p) = 0;
    prev = p;
    for i = 2 to n do //insert remaining n - 1 data//
      p = head;
      read(x);
      if data(p) ≥ x then //insert a node before the head node//
        getnode(p);
        data(p) = x;
        link(p) = head;
        head = p;
      else
        while (data(p) < x) and (p ≠ 0) do
          prev = p; //search appropriate position for x
          p = link(p);
        endwhile;
        getnode(q);
        data(q) = x;
        link(prev) = q;
        link(q) = p;
      endif;
    endfor;
end Insort.
```

For illustration, let us consider the data 15, 10, 25, 20.

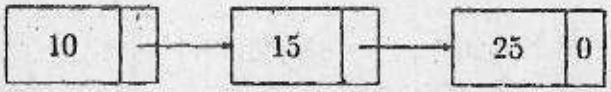
At the initial stage the list is as follows:



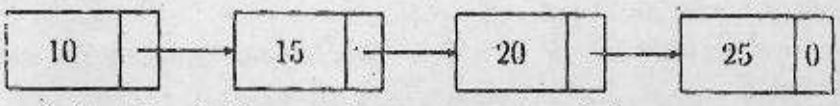
When $x = 10$ then a new node is inserted at the beginning of the list as follows :



When $x = 25$ then a new node containing $x = 25$ is inserted at the end of the list.



When $x = 20$ then a new node whose data field 20 is inserted between the nodes containing 15 and 25 as follows:



Observed that the list is sorted sequence at each instant.

Time complexity of Insort (using linked list)

At any instant, assume that i elements are in the list. To insert a new item to this sorted list takes at most $O(i)$ time (while loop in else block). So for a fixed i the for loop takes $O(i)$ time. Hence the overall time complexity is

$$\sum_{i=2}^{n} O(i) = O(\sum_{i=2}^{n} i) = O(n^2).$$

4.9.3 Addition of two polynomials

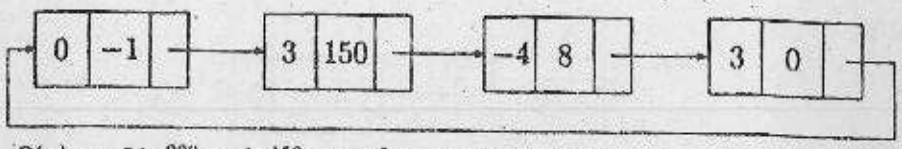
The manipulation of symbolic polynomial has becomes a classical example of the use of list processing. In general, we want to represent the polynomial

$$P(x) = a_m x^{e_m} + a_{m-1} x^{e_{m-1}} + \cdots + a_1 x^{e_1}$$

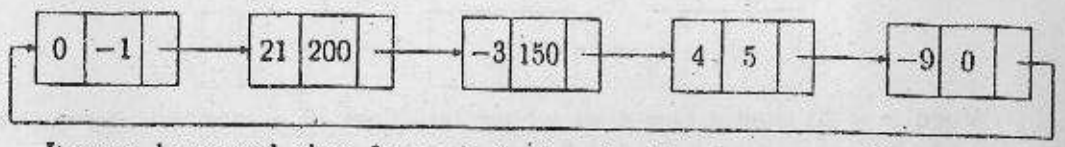
where the a_i are non-zero coefficients with exponents ei such that $e_m > e_{m-1} > \ldots > e_2 > e_1 \geq 0$. Each term will be represented by a node. A node will be of fixed size having three fields which represent the coefficient and exponent of a term plus a pointer to the next term as shown below:

COEF	EXP	LINK

For instance, the polynomial $P(x) = 3x^{150} - 4x^8 + 3$ would be stored as
Head node = P .



while $Q(x) = 21x^{200} - 3x^{150} + 4x^5 - 9$ would look like
Head node = Q



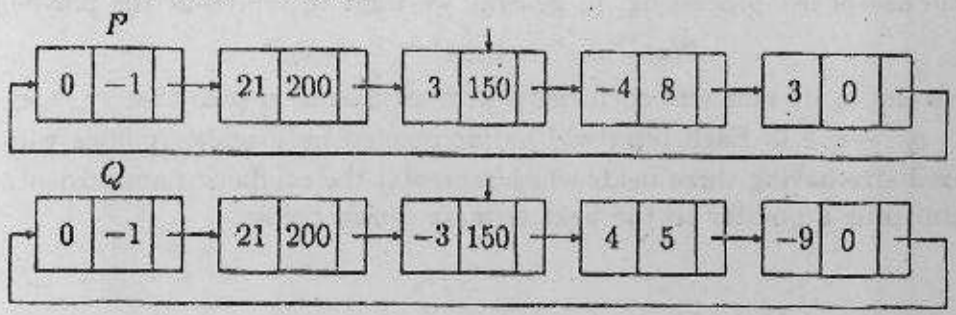
It may be noted that for each head node the dummy coefficient 0 and exponent – 1 are initialized.

Here two polynomials P and Q are added and replaced the value in the polynomial P.

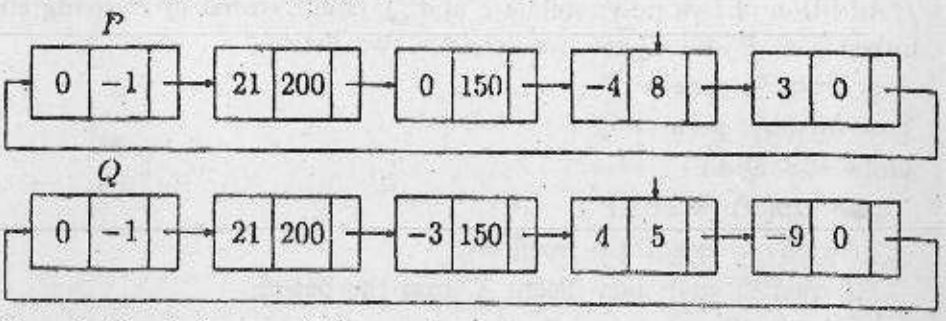
Addition of polynomials without third list

In order to add two polynomials together we examine their terms starting at the nodes after head nodes pointed by P and Q. Two pointers x and y are used to move along the nodes of P and Q. If the exponents of two terms are equal, then the coefficients are added and replace it to the node of P and pointers x and y are advanced to the next terms. If the exponent of the current term in P is less than the exponent of the current term of Q, then a duplicate term of Q is created and attached to P before the current term of P. The pointer y is advanced to the current term of P. The pointer y is advanced to the next term. If the exponent of current term of P is greater than the exponent of the current term of Q, then advance the pointer x to the next term.

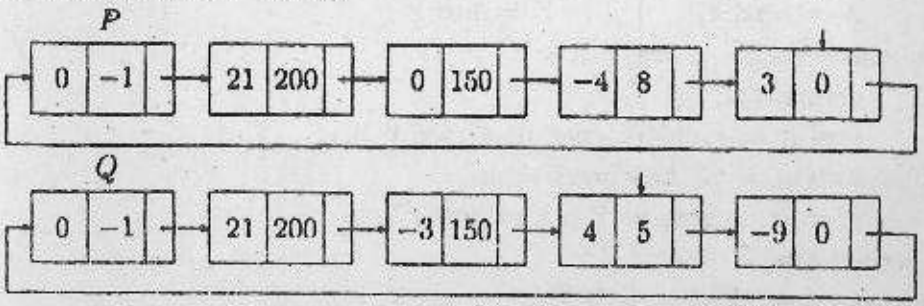
Step 1 : $(exp(x) < exp(y))$



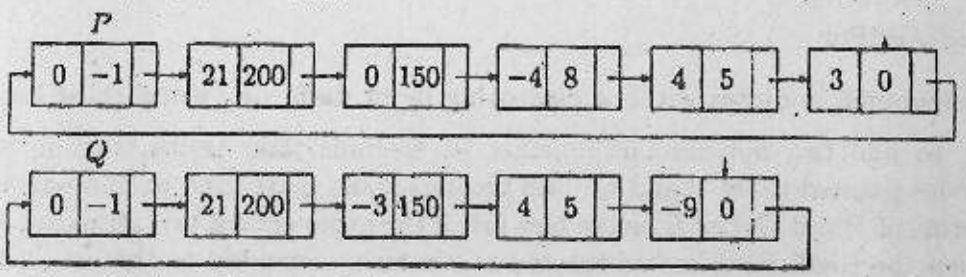
Step 2 : $(exp(x) = exp(y))$



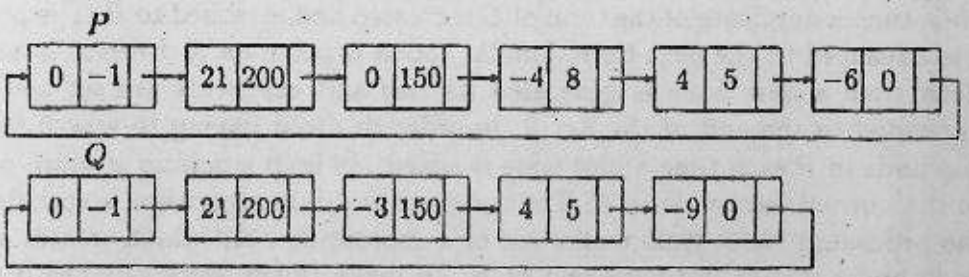
Step 3: $(exp(x) > exp(y))$



Step 4: $(exp(x) < exp(y))$



Step 5: $(exp(x) = exp(y))$



The sum of the polynomials P and Q is given by

$$P(x) = 21x^{200} + 0.x^{150} - 4x^8 + 4x^5 - 6.$$

```
Algorithm AddPoly (P ← P + Q)
    //Addition of two polynomials P and Q; result stored in P; using circular
    linked list; P and Q are the heads of two lists.//
    X = link(P); prex = P;
    Y = link(Q); prey = Q;
    while Y ≠ Q do
      case exp(X) = exp(Y):
        coef(X) = coef(X) + coef(Y);
        if coef(X) = 0 then delete X from the list P;
        prex = X;    prey = Y;
        X = link(X);          Y = link(Y);
      case exp(X) < exp(Y):
        getnode(u);
        exp(u) = exp(Y); coef(u) = coef(Y);
        link(u) = X; link(prex) = u;
        prex = u; prey = Y; Y = link(Y);
      case exp(X) > exp(Y):
        prex = X; X = link(X);
    endwhile;
end AddPoly
```

Polynomial addition without destroying input data, i.e., using third list

To add two polynomials together we examine their terms starting at the nodes pointed to by P and Q. Two pointers x and y are used to move along the terms of P and Q. Let R be the new list. If the exponents of two terms are equal, then the coefficients are added and a new term is created to the list R. If the exponent of the current term in P is less than the exponent of the current term of Q, then a duplicate of the term of Q is created and attached to R. The pointer y is advanced to the next term. Similar action is taken on P if $exp(x) > exp(y)$. Each time a new node is generated its *coef* and *exp* fields are set and it is appended to the end of the list R. In order to avoid having to search for the last node in R each time a new node is added, we keep a pointer d which points to the current last node in R. The complete addition algorithm is specified by the procedure PADD. PADD makes use of a subroutine JOIN which creates a new node and appends it to the end of R. To make things work out neatly, R is initially given a single node with no values which is deleted at the end of the algorithm.

```
Procedure JOIN(C,E, d)
//Create a new term with coef = C and exp = E and attach it to the node
    pointed at by d.//
    getnode(z);
    exp(z) ← E;
    coef(z) ← C;
    link(d) ← z;
    d ← z;
end JOIN

Algorithm PADD(P,Q,R)
//Polynomials P and Q represented as single linked lists are summed
the new list named R.//
    x ← P;   y ← Q; //x, y points to next terms of P,Q//
    getnode(R); d ← R; //initial node for R//
    while x ≠ 0 and y ≠ 0 do //while there are more terms in P and Q.//
      case exp(x) = exp(y) :
         t ← coef(x) + coef(y);
         if t ≠ 0 then JOIN(t, exp(x), d);
         x ← link(x); y ← link(y) //advance to next terms//
      case exp(x) < exp(y) ;
         JOIN(coef(y), exp(y), d);
         y ← link(y); //advanced to next term of Q//
      case exp(x) > exp(y) :
         JOIN(coef(x), exp(x), d);
         x ← link(x); //advanced to next term of P//
      endcase;
    endwhile;
    while x ≠ 0 do //copying remaining terms of P//
      JOIN(coef(x), exp(x), d);
      x ← link(x);
    endwhile;
    while y ≠ 0 do //copying remaining terms of Q//
      JOIN(coef(y), exp(y), d);
      y ← link(y);
    endwhile;
    link(d) ← 0; z ← R;R ← link(R) //delete extra initial node.//
RET(z);
end PADD
```

4.10 Summary

Some fundamental notations used to analysis of an algorithm and a data structure are introduced in this unit. Time and space complexities of an algorithm are defined. The commonly used data structures, viz. arrays, stacks, queues and linked lists are discussed along with very simple examples. Evaluation of postfix expression and conversion of infix expression to postfix expression are discussed with examples. The data structure stack is used to solve these two problems. Determination of maximum element from a list, insertion sort and addition of polynomials are performed by linked lists/circular linked lists.

Exercise 4

1. Explain the following terms: (i) O (big oh), (ii) θ, (iii) Ω, (iv) ω, (v) o (little oh), (vi) time complexity, (vii) space complexity.
2. Write an algorithm to store a set of real numbers into a stack and find their sum.
3. Define infix, prefix and postfix expressions. Convert the following infix expressions into prefix and postfix expressions:
   (a) $A + B * C - (D + E) * C$,
   (b) $A * ((B + C) + G/(D + A)) - G$.
4. Write an algorithm to evaluate a postfix expression.
5. Write an algorithm to convert an infix expression to postfix expression.
6. Write an algorithm to create a queue. Also, write algorithms for insertion and deletion operations on queue.
7. Discuss about the implementation of queue as a circular array.
8. What do you mean by a linked list? Write algorithms to insert a node to the linked list and to remove a node from the linked list.
9. Write an algorithm to create a linked list containing n numbers and find maximum among them.
10. Suppose two linked lists L_1 and L_2 are given. Write an algorithm to concatenate these two lists.
11. Suppose a linked list is given. Write a procedure to add a node containing the value, say, X after a given node.
12. Write an algorithm to add two polynomials without using a third list.
13. Write an algorithm to add two polynomials using a third list.

REFERENCES

1. E.Balagurusamy, ***Programming in ANSI C, 4e***, The McGraw-Hill Companies, New Delhi (2009).
2. B.Gottfried, ***Programming with C***, Schaum's Outlines, The McGraw-Hill Edition, New Delhi (2001).
3. K.R.Venugopal and S.R.Prasad, ***Programming with C***, The McGraw-Hill Companies, New Delhi (2005).
4. B.W.Kernighan and D.M.Ritchie, ***The C Programming Language***, Prentice-Hall, (1977).
5. Horowitz and Sahani, ***Fundamental of Computer Algorithms***, Galgotia, New Delhi (1995).
6. Horowitz and Sahani, ***Fundamental of Data Structure***, Galgotia, New Delhi (1995).
7. M.Pal, ***Numerical Analysis for Scientists and Engineers***, Narosa, New Delhi (2007).