PREFACE

In a bid to standardize higher education in the country, the University Grants Commission (UGC) has introduced Choice Based Credit System (CBCS) based on five types of courses viz. *core, discipline specific, generic elective, ability and skill enhancement* for graduate students of all programmes at Honours level. This brings in the semester pattern, which finds efficacy in sync with credit system, credit tranafer, comprehensive continuous assessments and a graded pattern of evaluation. The objective is to offer learners ample flexibility of choose from a wide gamut of courses, as also to provide them lateral mobility between various educational institutions in the country where they can carry their acquired credits. I am happy to note that the University has been recently accredited by National Assessment and Accreditation Council of India (NAAC) with grade "A".

UGC (Open and Distance Learning Programmes and Online Programmes) Regulations, 2020 have mandated compliance with CBCS for U. G. programmes for all the HEIs in this mode. Welcoming this paradigm shift in higher education, Netaji Subhas Open University (NSOU) has resolved to adopt CBCS from the academic session 2021-22 at the Under Graduate Degree Programme level. The present syllabus, framed in the spirit of syllabi recommended by UGC, lays due stress on all aspects envisaged in the curricular framework of the apex body on higher education. It will be imparted to learners over the six semesters of the Programme.

Self Learning Materials (SLMs) are the mainstay of Student Support Services (SSS) of an Open University. From a logistic point of view, NSOU has embarked upon CBCS presently with SLMs in English/Bengali. Eventually, the English version SLMs will be translated into Bengali too, for the benefit of learners. As always, all of our teaching faculties contributed in this process. In addition to this we have also requisitioned the services of best academics in each domain in preparation of the new SLMs. I am sure they will be of commendable academic support. We look forward to proactive feedback from all stakeholders who will participate in the teaching-learning based on these study materials. It has been a very challenging task well executed, and I congratulate all concerned in the preparation of these SLMs.

I wish the venture a grand success.

Professor (Dr.) Subha Sankar Sarkar

Vice-Chancellor

Netaji Subhas Open University

Under Graduate Degree Programme

Choice Based Credit System (CBCS)

Subject : Honours in Mathematics

Course Code : CC-MT-06

Computer Programming & Numerical Methods Lab

First Edition : May, 2022

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

Netaji Subhas Open University

Under Graduate Degree Programme
Choice Based Credit System (CBCS)
Subject : Honours in Mathematics
Course Code : CC-MT-06
Course : Computer Programming & Numerical Methods Lab

: Board of Studies :
: Members :

Professor Kajal De
(Chairperson)
Professor of Mathematics and Director, School of Sciences
Netaji Subhas Open University

Mr. Ratnesh Mishra
Associate Professor of Mathematics
Netaji Subhas Open University

Mr. Chandan Kumar Mondal
Assistant Professor of Mathematics
Netaji Subhas Open University

Dr. Ushnish Sarkar
Assistant Professor of Mathematics
Netaji Subhas Open University

Dr. P. R. Ghosh
Retd. Reader of Mathematics
Vidyasagar Evening College

Professor Buddhadeb Sau
Professor of Mathematics
Jadavpur University

Dr. Diptiman Saha
Associate Professor of Mathematics
St. Xavier's College

Dr. Prasanta Malik
Assistant Professor of Mathematics
Burdwan University

Dr. Rupa Paul
Associate Professor of Mathematics, WBES
Bethune College

: Course Writer :
Mr. Mrinal Nath
Assistant Professor of Computer Science
Netaji Subhas Open University

: Course Editor :
Dr. Tushar Kanti Saha
Associate Professor,
Department of Mathematics
Surendranath College

: Format Editor :
Mr. Mrinal Nath
Assistant Professor of Computer Science
Netaji Subhas Open University

Notification

All rights reserved. No part of this Self-Learning Material (SLM) may be reproduced in any form without permission in writing from Netaji Subhas Open University.

Kishore Sengupta
Registrar

UG : Mathematics
(HMT)

Course Code : CC-MT-06

Computer Programming & Numerical Method Lab under CBCS (BDP)

Unit - 1 ❑ Problem Solving Techniques 7 – 32

Unit - 2 ❑ Introducing C 33 – 46

Unit- 3 ❑ Variables, Constants and Input/Output 47 – 74

Unit - 4 ❑ Expression and Operators 75 – 91

Unit - 5 ❑ Decision and Loop Control Statements 92 – 128

Unit - 6 ❑ Arrays 129 – 152

Unit - 7 ❑ Application of C Programming : Solution of Non-Linear Equations 153 – 188

Unit - 8 ❑ Application of C Programming : Solution of System of Linear Equations by Direct Methods : Gauss Elimination and Gauss Jordan Elimination 189 – 218

Unit - 9 ❑ Application of C Programming : Solution of System of Linear Equations by Direct Methods : Matrix Inverse and LU Decomposition 219 – 240

Unit - 10 ❑ Application of C Programming : Solution of System of Linear Equations by Iterative Methods : Jacobi and Gauss-Siedel Method 241 – 256

Unit - 11 ❑ Application of C Programming : Interpolation 257 – 277

Unit - 12 ❑ Application of C Programming : Integration 278 – 296

Unit - 1 ❑ Problem Solving Techniques

Structure

1.0 Introduction

1.1 Objectives

1.2 Problem Solving Aspects

- **1.2.1 Problem Definition Phase**
- **1.2.2 Getting Started with Problem**
- **1.2.3 The Use of Specific Examples**
- **1.2.4 Similarities Among Problems**
- **1.2.5 Working Backward from the Solution**
- **1.2.6 Using Computer as a Problem-Solving Tool**

1.3 Design of Algorithm

- **1.3.1 Criteria to be Followed by an Algorithm**

1.4 Flowcharts

- **1.4.1 Basic Symbols used in Flowchart Design**

1.5 Fundamental Algorithm

- **1.5.1 Exchanging the Values of Two Variables**
- **1.5.2 Summation of a Set of Numbers**
- **1.5.3 Generation of Fibonacci Sequence**
- **1.5.4 Reversing the Digits of an Integer**

1.6 Factoring Method

- **1.6.1 Finding Square Root of a Number**
- **1.6.2 The Greatest Common Divisor of Two Integers**

1.7 Summary

1.8 References and Further Reading

1.0 Introduction

It's not that I'm so smart it's just that I stay with problems longer.

Albert Einstein.

Developing problem solving skills is like learning to play a musical instrument— a book or a teacher can point someone in the right direction, but only the hard work of the person can take him where he wants to go. Like a musician, the person need to know the underlying concepts but theory is no substitute for practice. Therefore, if someone wish to become a problem solver he/she has to solve problems.

In this unit we introduce the concepts of problem-solving, especially as they pertain to computer programming. In one hand, problem solving using computer is a demanding and intricate process which needs innovative thinking, careful planning, logical precession, perseverance, attention, on the other hand it can be challenging, exciting and satisfying experience with considerable room for personal creativity and expression.

To solve a problem using computer, the programmer needs to create a set of explicit and unambiguous instructions expressed in a particular programming language. This set of instructions is known as program. This set of instructions are also known as algorithm when it is expressed in a form that is independent of any programming language. The program runs on certain input data, supplied by the user and manipulates the data according to the instructions and eventually produces an output which represents the solution to the problem.

1.1 Objectives

After going through this unit the learner should be able to :

- Apply problem solving techniques
- Define an algorithm and its features
- Design flowcharts

1.2 Problem Solving Aspects

Problem solving is a creative process which largely defies systemization and mechanization. Therefore, there is no universal method or recipe for problem solving, rather different strategies appear to work for different people. Even if one is not naturally skilled at problem solving there are a number of steps that can be taken to raise the level of one's performance.

1.2.1 Problem Definition Phase

One needs to fully understand the problem beforehand, to make useful progress in problem solving. This preliminary investigation may be thought of as the problem definition phase. Therefore, the goal of this phase is to find what must be done rather than how to do it.

1.2.2 Getting Started with Problem

There are many ways of solving a problem and there may be several solutions. So, it is difficult to recognize immediately which path could be more productive. Sometimes it is very difficult to find any idea where to begin solving a problem, even if the problem has been defined. Such block occurs particularly when the problem solver overly concerned with the details of the implementation even before completely understanding or working out a solution. The best advice is not to get concerned with the details. Those can come later when the intricacies of the problem have been understood.

1.2.3 The Use of Specific Examples

A useful strategy is to solve a specific example of generic problem before working out the actual solution of the generic problem (To find a maximum of a set of numbers, choose a particular set of numbers and work out the mechanism for finding the maximum in this set). It is usually easier to solve a specific problem because the relationship between mechanism and the particular problem is more clearly defined. This approach can often give us the foothold for making a start on the solution of the general problem.

1.2.4 Similarities Among Problems

Another useful strategy while solving a problem is to bring as much past experience as possible to bear on current problem. Therefore, it is important to see if there are any similarities in current problem and other problems solved so far. But sometimes, it blocks from discovering a desirable or better solution to the problem. A skill that is important to try to develop in problem-solving is the ability to view a problem from a variety of angles. One must be able to metaphorically turn a problem upside down, inside out, sideways, backwards, forwards and so on. Once one has developed this skill it should be possible to get started on any problem.

1.2.5 Working Backward from the Solution

In some cases, if the solution to the problem is already known, it is useful to work backward to the starting point. Even a guess at the solution to the problem may be enough to give us a foothold to start on the problem. We can systematize the investigations and avoid duplicate efforts by writing down the various steps taken and explorations made. Another practice that helps to develop the problem solving skills is, once we have solved a problem, to consciously reflect back on the way we went about discovering the solution.

1.2.6 Using Computer as a Problem-Solving Tool

A computer is a very powerful general-purpose tool that can solve or help to solve many types of problems. There are also many ways in which a computer can enhance the effectiveness of the time and effort that someone is willing to devote to solve a problem. Thus, it will prove to be well worth the time and effort you spend to learn how to make effective use of this tool. To solve a problem using computer involves many steps like understanding of the problem, developing a solution, writing the program, and then testing it. How properly someone is following these steps, determines the overall quality and success of the program. The following are the steps in detail :

1. Specify the problem requirement.
2. Analyse the problem
3. Design an Algorithm and a Flowchart
4. Implement the algorithm in a programming language (Like C language)
5. Test and verify the completed program
6. Run the program on input data and get the output

The main focus of this unit is to discuss the third step i.e. how to design an algorithm and flowchart. The remaining steps will be discussed in subsequent units of this module.

1.3 Design of Algorithm

The primary goal in computer problem solving is an algorithm which is capable of being implemented as a correct and efficient computer program. An algorithm is a finite set of steps defining the solution of a particular problem. It must be noted that an efficient algorithm is one which is capable of giving the solution to the problem by using minimum resources of the system such as memory and processor's time. Algorithm is language independent, well-structured and detailed. It will enable the programmer to translate into a computer program using any high-level language. For each problem or class of problems, there may be different algorithms. For each algorithm, there may be many different implementations or programs (**Figure 1.1**).

1.3.1 Criteria to be Followed by an Algorithm

The following is the criteria to be followed by an algorithm :

- **Input :** An algorithm has zero or more inputs which are to be supplied.

- **Output :** An algorithm has one or more outputs, which have a specified relation to the inputs.
- **Definiteness :** Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- **Finiteness :** If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- **Effectiveness :** Each step must be sufficiently basic that a person using only paper and pencil can in principle carry it out. In addition, not only each step is definite, it must also be feasible.

1.4 Flowcharts

Flowcharts are used in programming to diagram the path in which information is processed through a computer to obtain the desired results. Flowchart is a graphical representation of an algorithm. It makes use of symbols which are connected among them to indicate the flow of information and processing. It will show the general outline of how to solve a problem or perform a task. It is prepared for better understanding of the algorithm. By looking at a Flowchart one can understand the operations or sequence of operations performed in an algorithm. Flowchart is often considered as a blueprint of a design used for solving a specific problem.

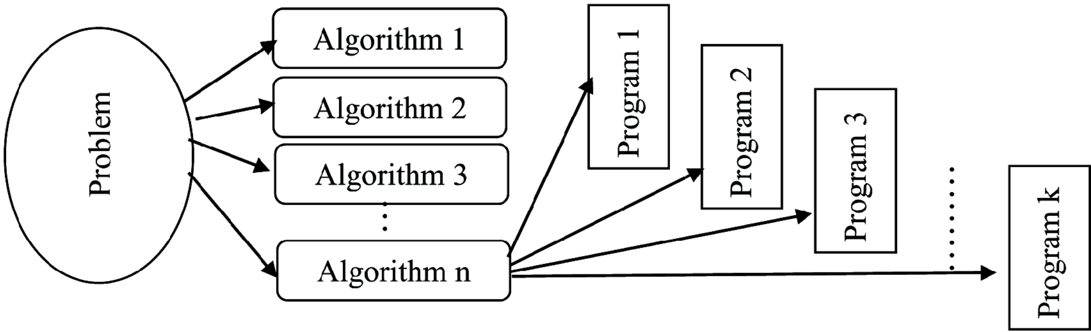


Figure 1.1

1.4.1 Basic Symbols used in Flowchart Design

Symbol Name	Symbol	Function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing : Used for arithmetic operations and data-manipulations
Diamond		Decision making : Used to represent the operation with multiple alternatives.
Arrows		Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
Predefined Process		Used to represent a group of statements performing one processing task.
		Comments, Explanations, Definitions.

Table 1.1

1.5 Fundamental Algorithm

Different types of algorithm are used in practice. Few of them are essential part of many more elaborate computational procedures. Let us discuss these algorithms in the following section.

1.5.1 Exchanging the Values of Two Variables

Problem :

Given two variables a and b, exchange the values assigned to them.

Algorithm Development :

Consider the case when a = 56 and b = 85. Our task is to replace the contents of a with 85 and the contents of b with 56.

Starting Configuration		Target Configuration	
a	b	a	b
56	85	85	56

Figure 1.2

Apparently one can assume that two assignment statements $a = b$ and $b = a$ can be used to achieve the target configuration **(Figure 1.2)**. But to analyse the problem one need to understand how the assignment statement actually work.

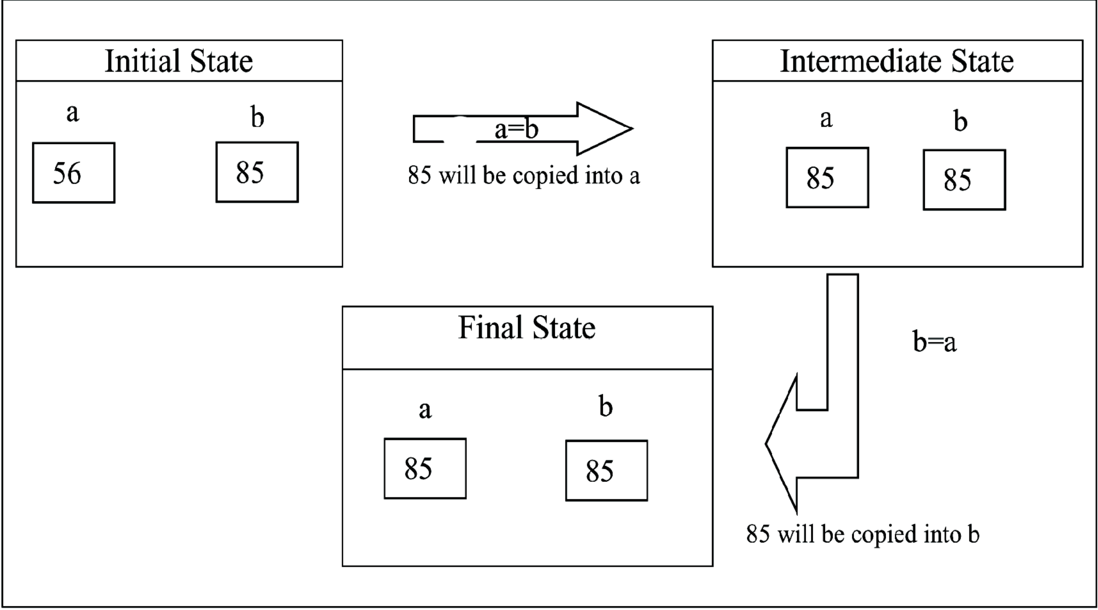


Figure 1.3

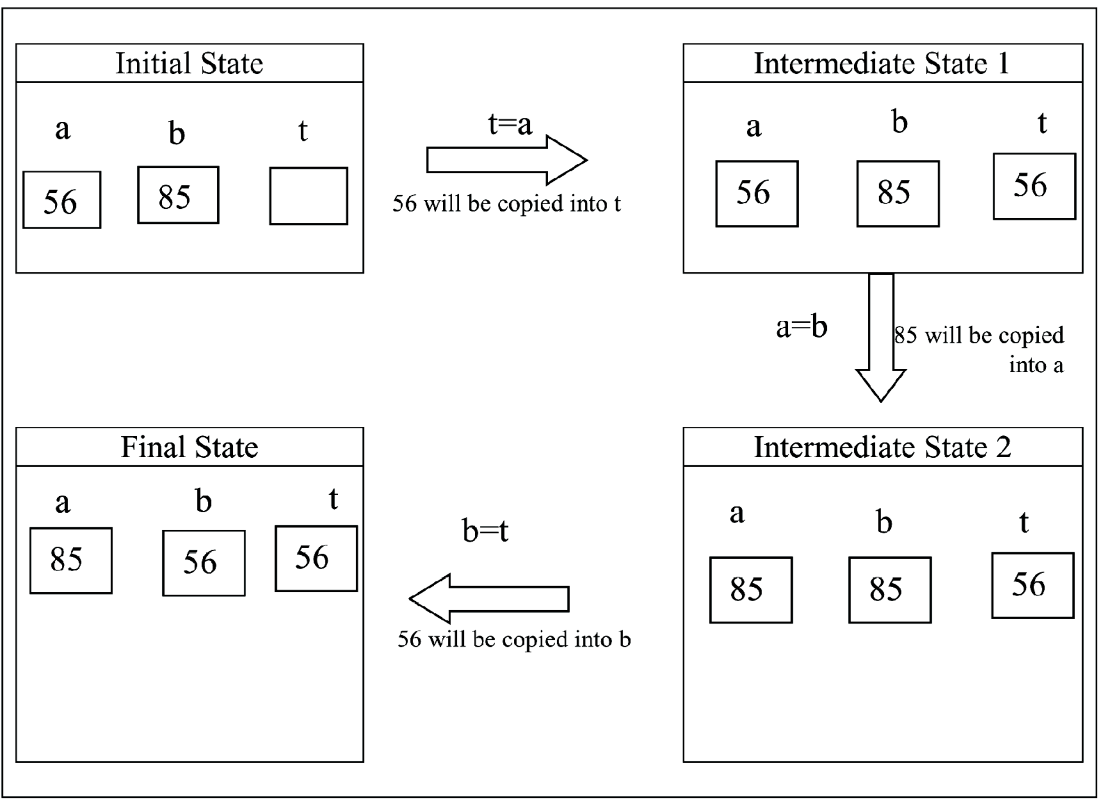


Figure 1.4

Figure 1.3 shows execution of assignment statements $a = b$ and $b = a$ sequentially. After executing the statement $a = b$ on the initial state, the new value of b (= 85) get copied into a and as a result new value of a becomes 85. The old value of a (= 56) is therefore lost from memory as memory cell can't have multiple values of a variable at any particular time instance. This suggests that the older value of variable a needs to be copied to another variable, say t before executing $a = b$. Therefore, to solve the above problem following three assignment statements needs to be executed sequentially. **Figure 1.4** shows how the variable are exchanged by this method.

$$t = a$$
$$a = b$$
$$b = t$$

The algorithm and flowchart of this method is given in **Figure 1.5**.

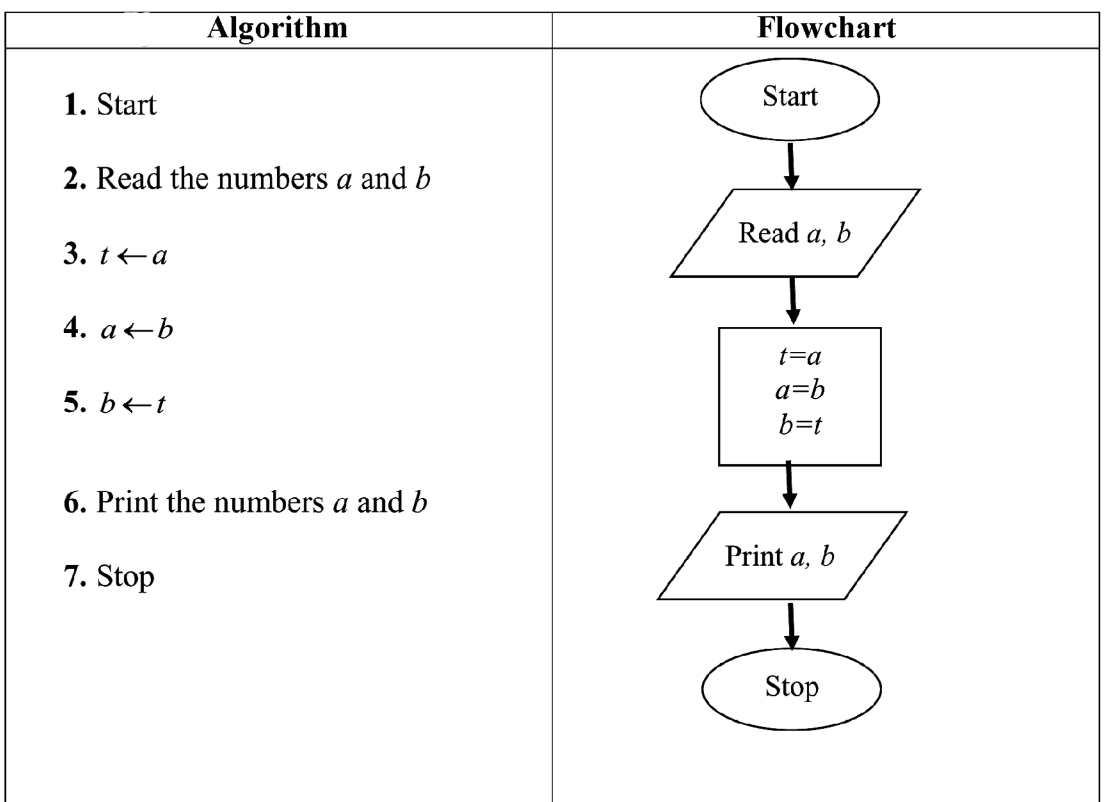
Check Your Progress 1.1

Design an algorithm that makes following exchanges :

$$a \rightarrow b \rightarrow c$$

(with a curved arrow from c back to a)

Algorithm Description and Flowchart :



Algorithm	Flowchart
1. Start **2.** Read the numbers a and b **3.** $t \leftarrow a$ **4.** $a \leftarrow b$ **5.** $b \leftarrow t$ **6.** Print the numbers a and b **7.** Stop	Start Read a, b $t=a$ $a=b$ $b=t$ Print a, b Stop

Figure 1.5

Check Your Progress 1.2

Design an algorithm and flowchart for the following problem :

Given two variables a and b, exchange their values without using a third temporary variable.

1.5.2 Summation of a Set of Numbers

Problem :

Given a set of n numbers, design an algorithm that adds all the numbers and give the sum as output. Assume that $n \geq 0$.

Algorithm Development :

To develop this algorithm at first we need to discuss how addition operation takes place inside the computer processor. Arithmetic processing unit (ALU) of a computer has a specific circuit (Adder) which can take two numbers and return the result after adding them **(Figure 1.6)**. The result can be stored in a memory cell.

Therefore, if s be memory cell which can store the result of adding two numbers a_1 and a_2, then this operation can be written as $s = a_1 + a_2$.

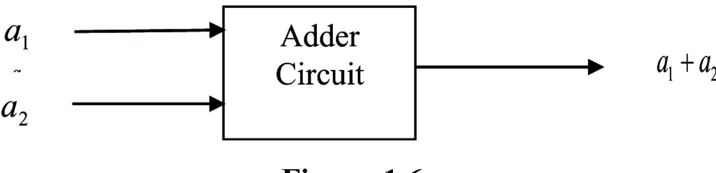


Figure 1.6

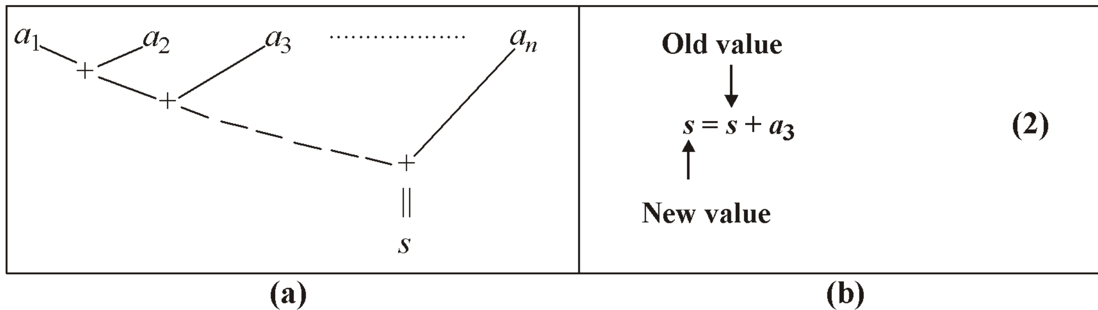
Another important fact is that the assignment (=) operator is right associative (Associativity will be discussed in detail in section 4.3), which means the right hand side will be evaluated first and then the result will be assigned to memory cell in left hand side of the assignment operator. This way, $s = a_1 + a_2$ can add only two numbers and give the result.

A fundamental goal in designing algorithms and implementing programs is to make the programs general enough so that they will successfully handle a wide variety of input conditions. That is, the program should add any n numbers where n can take on a wide range of values. Even when, n is large, computer can still handle the operation $s = a_1 + a_2 + a_3 + \ldots a_n$ by using multiple adders **(Figure 1.7a)**. But this approach needs n memory cells for storing $a_1, a_2, a_3 \ldots, a_n$ to be defined before program execution. If the value of n changes then new variables may need to be defined. But any robust program should be capable of handling different inputs without redefining its variables each time before execution.

One way to do this, that takes note of the fact that the computer adds two numbers at a time is to start by adding first two numbers a_1 and a_2. That is,

$$s = a_1 + a_2 \qquad (1)$$

then proceed by adding a_3 to s computed in step (2) and assign the result to s again **(Figure 1.7b)**.



(a) **(b)**

Figure 1.7

In similar manner :

$$\left.\begin{array}{l} s = s + a_4 \\ s = s + a_5 \\ \vdots \quad \vdots \quad \vdots \\ \vdots \quad \vdots \quad \vdots \\ s = s + a_n \end{array}\right\} \begin{array}{r} ------(3) \\ ------(4) \\ ------(\vdots) \\ ------(\vdots) \\ ----(n-1) \end{array}$$

From step (2) onwards the steps are repeated with only changes in values of s and a. Therefore for general i^{th} step the operation is

$$s = s + a_{i+1} \qquad (i)$$

Now, according to the given problem the value of $n \geq 0$. Therefore, the algorithm should correctly give the result for boundary/special values like $n = 0$ or $n = 1$. To accommodate $n = 1$ the step (1) needs to be corrected as

$$s = s + a_1 \qquad (1')$$

Now this new step (1′) is also fit into the general step (i) if $i + 1$ is replaced by i. Therefore, the corrected general i^{th} step will become

$$s = s + a_i \qquad (i')$$

Now for $n = 0$, the variable s needs to be initialized to zero before entering into step(1). In summary, the algorithm becomes,

$$s = 0$$
$$s = s + a_i, \quad \text{for } i = 1, 2, \ldots, n$$

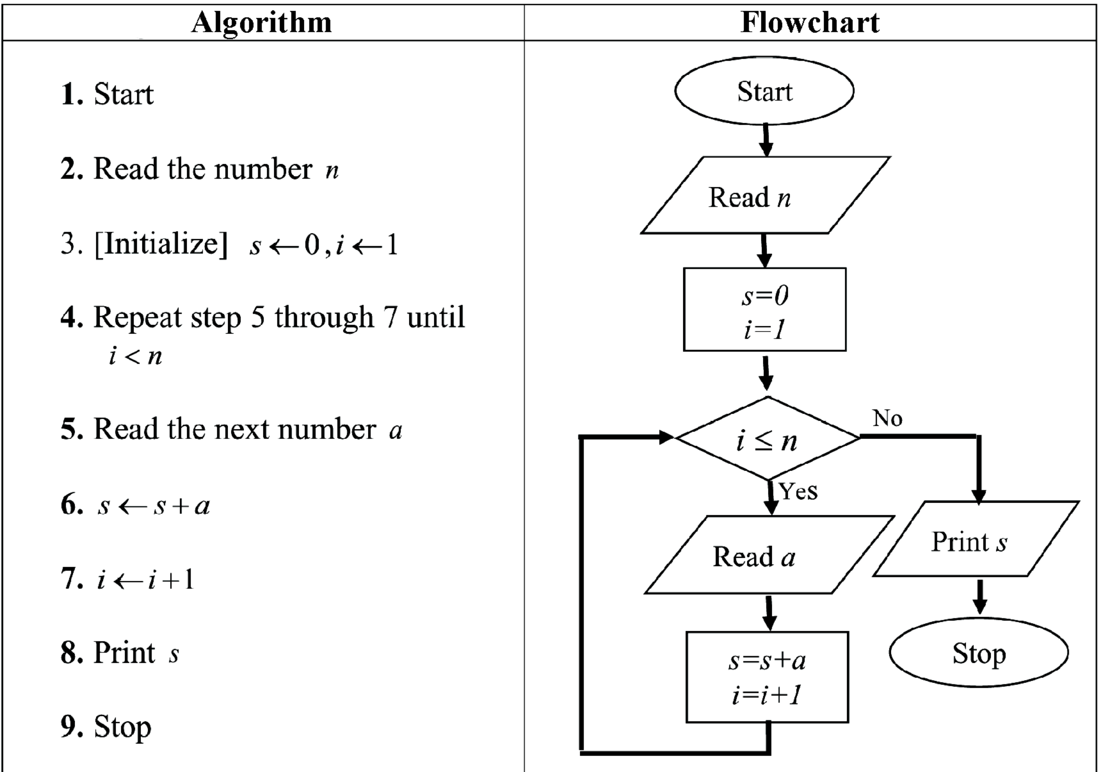
The complete algorithm and flowchart has been given in **Figure 1.8**.

Check Your Progress 1.3

Design an algorithm and flowchart to compute the sum of squares of n numbers. That is,

$$s = \sum_{i=1}^{n} (a_i)^2$$

Algorithm Description and Flowchart :



Algorithm	Flowchart
1. Start **2.** Read the number n 3. [Initialize] $s \leftarrow 0, i \leftarrow 1$ **4.** Repeat step 5 through 7 until $i < n$ **5.** Read the next number a **6.** $s \leftarrow s + a$ **7.** $i \leftarrow i + 1$ **8.** Print s **9.** Stop	

Figure 1.8

Check Your Progress 1.4

Design an algorithm and flowchart to compute harmonic mean H of n data values where

$$H = \frac{n}{\sum_{i=1}^{n} \left(\frac{1}{a_i} \right)}$$

Check Your Progress 1.5

Design an algorithm and flowchart to generate first n terms of the sequence 1, 2, 4, 8, 16, 32, . . . using multiplication operator.

Check Your Progress 1.6

Design an algorithm and flowchart to compute the sum of the first n terms ($n \geq 1$) of the series, $s = 1 - 3 + 5 - 7 + 9 - \ldots$

Check Your Progress 1.7

For a given number n ($n \geq 0$), the factorial of n (written as n !) is given by the formula $n! = 1 \times 2 \times 3 \times 4 \times \ldots n$. Design an algorithm and flowchart to compute the factorial of a given number n ($n \geq 0$).

Check Your Progress 1.8

Design an algorithm and flowchart to determine whether or not a number n is a factorial number (must be equal to the factorial of any other number).

Check Your Progress 1.9

Design an algorithm to simulate multiplication by addition. The program should accept two integers (they may be zero, negative or positive).

1.5.3 Generation of Fibonacci Sequence

Problem :

Generate and print first n terms of Fibonacci sequence where $n \geq 1$. The first few terms are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Each term beyond the first two is derived from the sum of the nearest two predecessors.

Algorithm Development :

From the definition,

new term = preceding term + term before preceding term.

This definition will be used to generate consecutive terms (apart from the first two terms) iteratively. Let us define,

a : *term before preceding term.*

b : *preceding term.*

c : *new term.*

Then to start with,

$a \leftarrow 0$ First Fibonacci number
$b \leftarrow 1$ Second Fibonacci number
$c \leftarrow a + b$ Third Fibonacci number (from definition)

To generate the fourth, same definition will be used. That is, the fourth Fibonacci number will be the sum of the third and second Fibonacci numbers. With regard to the definition, the second Fibonacci number has the roll of *term before preceding term* and the third Fibonacci number has the roll of *preceding term*. This is shown for $n = 8$ in **Table 1.2.** Therefore, before computing the fourth term following steps need to executed.

$a \leftarrow 0$ [1] *term before preceding term.*
$b \leftarrow 1$ [2] *preceding term.*
$c \leftarrow a + b$ [3] *new term.*
$a \leftarrow b$ [4] *term before preceding term becomes preceding term.*
$b \leftarrow c$ [5] *preceding term becomes new term.*

Value of n	***New Term (c) = a + b***	***Preceding Term(b)***	***Term before preceding term(a)***
3	1	1	0
4	2	1	1
5	3	2	1
6	5	3	2
7	8	5	3
8	13	8	5

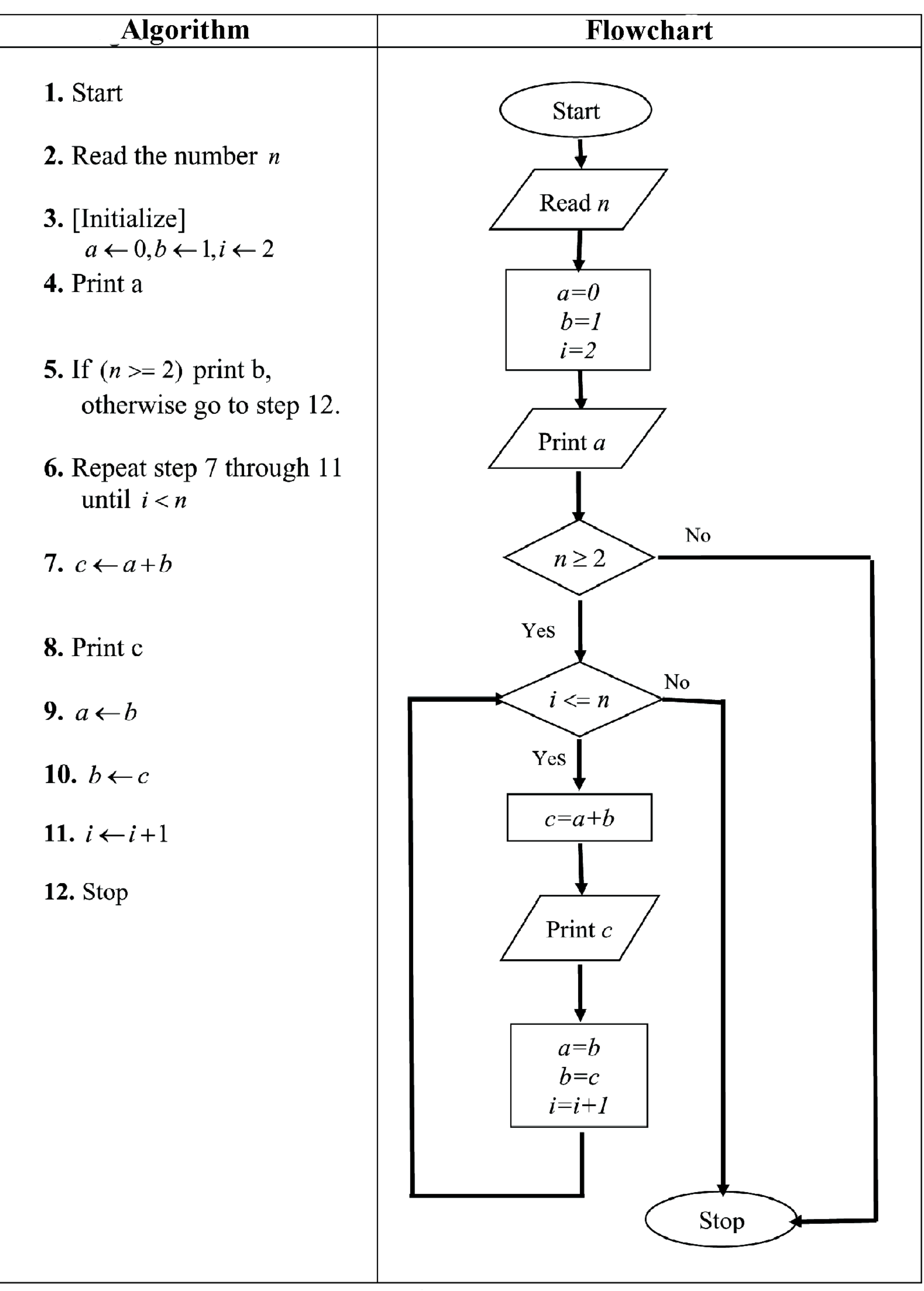
Table 1.2

After making step [5] the definition for next term can be used. To get the Fibonacci sequence for any arbitrary value of n the step [3] to step [5] needs to be executed iteratively. The algorithm and flowchart of Fibonacci numbers are given in **Figure 1.9**.

Remarks :

Italian Mathematician Leonardo of Pisa, commonly known as Leonardo Fibonacci, in the early 13th century, first studied the sequence extensively although this sequence turned up in Indian Mathematics (around 200 BC) in connection with Sanskrit versification. Among those who had commented on the sequence was the scholar Acharya Hemachandra, a poet and polymath who wrote on philosophy, history, grammar, and prosody. For this reason, the Fibonacci numbers are also known as Hemachandra-Fibonacci numbers (See Indubala I. Satija, *Butterfly in the Quantum World*, Morgan & Claypool Publishers, p 4-2).

Algorithm Description and Flowchart :



Algorithm

1. Start

2. Read the number n

3. [Initialize]
$a \leftarrow 0, b \leftarrow 1, i \leftarrow 2$

4. Print a

5. If $(n >= 2)$ print b,
otherwise go to step 12.

6. Repeat step 7 through 11
until $i < n$

7. $c \leftarrow a + b$

8. Print c

9. $a \leftarrow b$

10. $b \leftarrow c$

11. $i \leftarrow i + 1$

12. Stop

Figure 1.9

Check Your Progress 1.10

The first few numbers of the Lucas sequence which is a variation on the Fibonacci sequence are :

1, 3, 4, 7, 11, 18, 29, . . .

Design an algorithm and flowchart to generate Lucas sequence.

Check Your Progress 1.11

Given $a = 0$, $b = 1$ and $c = 1$ are the first three numbers of some sequence. All other numbers of the sequence are generated from the sum of their three most recent predecessors. Design an algorithm and flowchart for the sequence.

Check Your Progress 1.12

Design an algorithm and flowchart to generate the sequence where each member is the sum of adjacent factorials, i.e.

0!, 1!, (0!+1!), (1!+2!), (2!+3!), (3!+4!), . . .

Note that by definition 0! = 1.

1.5.4 Reversing the Digits of an Integer

Problem :

Design an algorithm that accepts a positive integer and reverses the order of its digits.

Algorithm Development :

To analyse the problem let us consider a particular positive integer 34521. The expected output of this problem is 12543. Intuitively, the approach looks very easy. In the first phase, the integer should be read from left to right till the end and all the digits are collected one after another. In the next phase, the collected digits need to be arranged in reverse order. While collecting the individual digits in the first phase, the number needs to be scanned from left end to right end. It is of course very difficult to identify the right end of the number other than seen it visually. One way to do this, is to chop off the least significant digit (rightmost digit) in the number by using integer division and modular arithmetic. i.e.

34521 ***div*** 10 = 3452

34521 ***mod*** 10 = 1

Where x **div** y returns the greatest integer less than or equal to $\frac{x}{y}\left(\left[\frac{x}{y}\right]\right)$ and x

mod y returns the remainder after dividing integer x by integer y. Therefore, following two steps will be applied.

$$r = n \textbf{ mod } 10 \quad [1] \quad \Rightarrow r = 1$$

$$n = n \textbf{ div } 10 \quad [2] \quad \Rightarrow n = 3452$$

The above two steps will be repetitively executed to obtain all individual digit from left to right.

The next step is to carry out the reversal of the digits of the number. To do that, let us take a new variable R which stores the number in reversed order. The initial value of R is set to be zero. The value of R in the next iteration will be generated using the formula $R = R \times 10 + r$ where r is the reminder of current iteration which is initially set to zero. This formula will be executed repeatedly. After first iteration step [1] and [2] produces $n = 3452$ and $r = 1$. Therefore the value of $R = 0 \times 10 + 1 = 1$. Similarly next iteration produces $n = 345$ and $r = 2$, which makes $R = 1 \times 10 + 2 = 12$. It can be noted that after the completion of second iteration the last two digits of n i.e. 21 is reversed and stored in R as 12.

The final thing is to decide the termination condition of the iterative process. The termination condition must in some way be related to the number of digits in the input integer. In fact, as soon as all the digits have been extracted the termination should apply. With each iteration the input integer n is reduced by one digit. Therefore, the iteration stops when n becomes zero. See the steps in **Table 1.3** for $n = 34521$. The algorithm and the flowchart is given in **Figure 1.10**.

Iteration No(i)	Value of $n = \frac{n}{10}$	$r = n$ *mod* 10	Number Reversed $R = R \times 10 + r$
0	34521	0	0
1	3452	1	$0 \times 10 + 1 = 1$
2	345	2	$1 \times 10 + 2 = 12$
3	34	5	$12 \times 10 + 5 = 125$
4	3	4	$125 \times 10 + 4 = 1254$
5	0	3	$1254 \times 10 + 3 = 12543$

Table 1.3

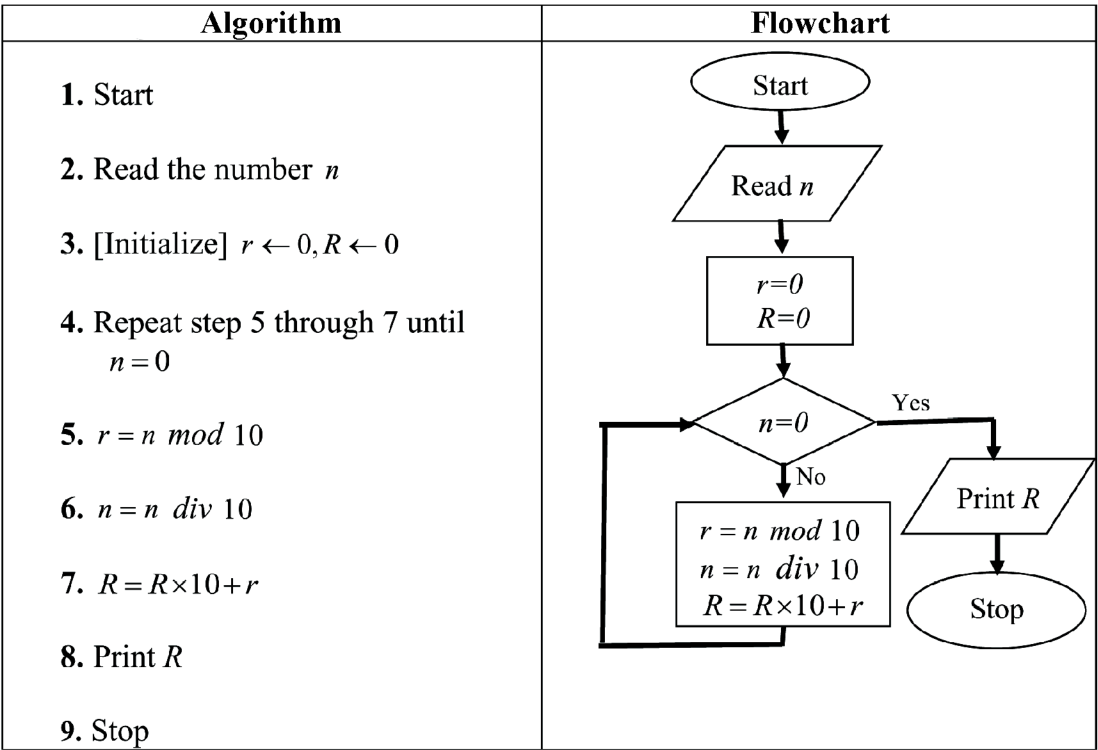
Check Your Progress 1.13

Design an algorithm and flowchart that counts number of digits in an integer.

Check Your Progress 1.14

Design an algorithm and flowchart that sums the digits in an integer.

Algorithm Description and Flowchart :



Algorithm	Flowchart
1. Start **2.** Read the number n **3.** [Initialize] $r \leftarrow 0, R \leftarrow 0$ **4.** Repeat step 5 through 7 until $n = 0$ **5.** $r = n \ mod \ 10$ **6.** $n = n \ div \ 10$ **7.** $R = R \times 10 + r$ **8.** Print R **9.** Stop	

Figure 1.10

Check Your Progress 1.15

Design an algorithm and flowchart that reads in a set of n single digits and convert them into a single decimal integer. For example, the algorithm should convert the set of 5 digits {2, 7, 4, 9, 3} to the integer 27493.

1.6 Factoring Method

Factorization of a number is a very common technique used in different algorithms. In the next section few algorithms will be discussed where factorization is involved.

1.6.1 Finding Square Root of a Number

Problem :

Give a positive number n, device an algorithm to compute its square root.

Algorithm Development :

Let us consider a case when $n = 10,000$ and also assume that square root of n is x. therefore, x must satisfy the equation

$$x^2 = n \quad (1.1)$$

Initially the value of x assumed to be any arbitrary value less than n. Then following approach will solve the root finding problem.

Approach 1 :

At the beginning, based on the chosen initial value of x three scenario may arise

Scenario 1 : If the value of $x^2 = n$ then x is the square root of n.

Scenario 2 : If the value of x^2 is greater than n then decrease the value of x by 1 until the value of x^2 becomes less than n. Once the value of x^2 is less than n, increase the value of x by 0.1 until the value of x^2 becomes greater than n. Once the value of x^2 is greater than n, decrease the value of x by 0.01 until the value of x^2 becomes less than n. This process will continue till the given accuracy is reached.

Scenario 3 : If the value of x^2 is less than n then increase the value of x by 1 until the value of x^2 becomes greater than n. This process then in similar to scenario 2 and continues till the given accuracy is reached. How this approach converges to the desired solution is diagrammatically represented in **Figure 1.11**. Now the important point to be noted that this approach heavily depends on the initial value of x. If $n = 10,000$ and the initial guess of x is chosen as 500 then the approach takes 400 iterations before converging to the actual root 100.

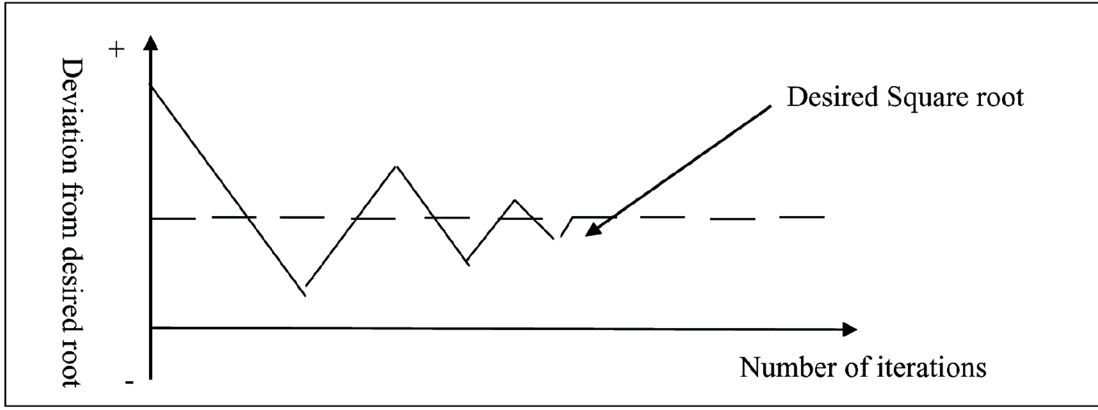


Figure 1.11

Approach 2 (Babylonian Method) :

Another better approach is known as Babylonian method which was developed by Babylonians in about 1800 B.C. This method was explicitly stated by Herons of Alexandria after many centuries.

To understand the Babylonian Method let us take the same example where n = 10000 and the initial guess is $x = 500$. Then, $x^2 = 2{,}50{,}000$ greater than 10,000. It is clear from the equation (1.1) that 500 should divide 10,000 to give a quotient of 500 if 500 is truly the square root of 10,000. Instead 500 divides 10,000 to give 20. If 20 had been chosen as initial guess of x then, $x^2 = 400$ less than 10,000.

Therefore, it suggests that if the chosen square root is too large then another candidate for the square root derived from starting guess, is too small. This shows that actual square root will lie in between these two values. **Table 1.4** establish this fact for n = 10,000 where the actual square root of n is in between 500 and 20.

Square	Square Root
2,50,000	500
10,000	??
400	20

Table 1.4

Therefore, to have a better square root of n, the average of above two values will be chosen. As a result, the new estimate of square root is (500 + 20)/2 = 260. The square of this estimate also may again be either greater than, equal to, or less than 10,000. Using this value as next approximated value of square root of n, the method will continue until the desired root is obtained. **Table 1.5** shows all the intermediate estimates for n = 10,000.

Square	Square Root
2,50,000	500
67,600	260
22,269.59	149.23
11689.93	108.12
10,057	100.29

Square	Square Root
10,000	100
9942.25	99.71
8554.36	92.45
4490.43	67.01
1469.29	38.46
400	20

Table 1.5

The advantage of Babylonian method is that it converges faster than the method mentioned earlier. For n = 10,000 and initial guess of x = 500, the number of iteration is 5, much less than the previous method where number of iteration is 400. In summary, Babylonian method has following key steps.

Step 1 : Consider the value of n, initial guess of x and a (*accuracy*)

Step 2 : If initial guess is not accurate then calculate other estimate = (n/x) and determine better estimate by the formula $(x + n/x))/2$. Repeat step 2 until the desired accuracy.

Figure 1.12 provides the algorithm and flowchart of Babylonian method.

Algorithm Description and Flowchart :

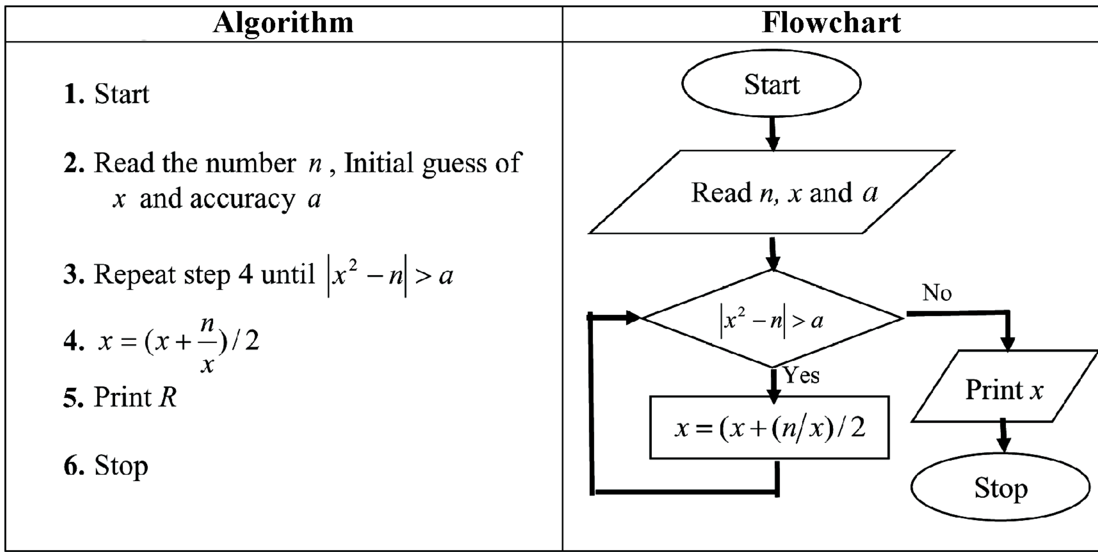


Figure 1.12

Speed of convergence of Babylonian method :

Let us consider x_n is the approximate root of n, found after n^{th} iteration. Also assume that the actual root of n is s. Then if the relative error after n^{th} iteration is e_n, then

$$e_n = \frac{s - x_n}{s} \Rightarrow x_n = s(1 - e_n) \qquad (1.2)$$

The next approximate root $x_{n+1} = \left(x_n + \frac{n}{x_n}\right)/2$. Substituting the value of x_n from equation 1.2, $x_{n+1} = \left[s(1-e_n) + \frac{n}{s(1-e_n)}\right]/2 \Rightarrow x_{n+1} = \frac{s^2(1-e_n)^2 + n}{2s(1-e_n)}$

$$\Rightarrow x_{n+1} = \frac{s^2(1-e_n)^2 + s^2}{2s(1-e_n)} \qquad \text{[replacing } n \text{ by } s^2\text{]}$$

$$\Rightarrow x_{n+1} = \frac{s^2(1-2e_n+e_n{}^2)+s^2}{2s(1-e_n)} \quad \Rightarrow x_{n+1} = \frac{s(2-2e_n+e_n{}^2)}{2(1-e_n)}$$

$$\Rightarrow x_{n+1} = \left[s + \frac{se_n{}^2}{2(1-e_n)}\right] \Rightarrow x_{n+1} = \left[s + \frac{se_n{}^2}{2}\right] \quad \text{[since } e \text{ small, } 1 - e \approx 1\text{]}$$

$$\Rightarrow \quad x_{n+1} = s + \frac{se_n{}^2}{2} \Rightarrow \frac{x_{n+1} - s}{s} = \frac{e_n{}^2}{2}$$

$$\Rightarrow \quad e_{n+1} = \frac{e_n{}^2}{2}$$

The quadratic term shows that relative error decreases rapidly and method will converge to the desired root.

Check Your Progress 1.16

Design an algorithm and flowchart for square root finding algorithm using the approach 1 mentioned in section 1.6.1.

Check Your Progress 1.17

The geometric mean, used to measure central tendency, is calculated by the following formula :

Geometric Mean = $\sqrt[n]{x_1 x_2 x_3 \ldots x_n}$

Design an algorithm and flowchart to determine the geometric mean of n numbers.

Check Your Progress 1.18

Design an algorithm and flowchart that finds the integer whose square is closest to but greater than the integer number as input data.

1.6.2 The Greatest Common Divisor of Two Integers

Problem :

Given two positive non-zero integers n and m, devise an algorithm to compute their greatest common divisor (GCD).

Algorithm Development :

Let us assume two number $n = 84$ and $m = 186$. Also assume that GCD of n and m is x which is to be determined. The GCD x is the largest integer which can divide both n and m. Alternatively it can be said that there is no integer larger than n and m that can divide simultaneously both the integers. Therefore, the value of x must be lesser or equal to the smaller of n and m. To find out the GCD multiple approach can be taken.

Approach 1 :

First take the value of two integers and store the smaller number in n and the bigger number in m. If the number n divides m, then n is the desired GCD. If not, then repeatedly decrease the value of n by 1 and check whether the updated n divides both original n and m or not. After finite no of iterations eventually updated n will divide original n and m both and return the value of updated n as the GCD.

It is very easy to observe that this process is not very efficient. If the value of the smaller number n is very large then the iteration may run up to n times. As an example if $n = 84$ and $m = 186$ then this algorithm runs for 78 iterations and return 6 as the GCD of 84 and 186.

Approach 2 :

Approach 1, discussed so far can be modified to produce better result if value of n is decreased by a larger amount instead of 1 in every iteration. But even though the approach may need huge number of iteration in certain problems. A better approach published by Greek Philosopher Euclid more than 2000 years ago known as Euclid's algorithm to find GCD of two integers. This algorithm was probably invented by a predecessor of Euclid called Eudorus. The ancient Chinese also discovered the algorithm.

To elaborate the Euclid's algorithm let's start with the same example where m

= 84 and n = 186 and the greatest common divisor x needs to be determined. Now x is common divisor of m and n. Let us also calculate the remainder r when n divides m. If the value of r is zero then x is assigned the value of n which is the GCD of m, n. If not, then it can be proved that x will divide the remainder r as well, algebraically in the following manner :

$r = m - kn$ where k is a positive integer greater than zero.
x divides m and $n \Rightarrow m = px$ and $n = qx$ where p, q ($p \geq q$) are positive integers.
$\Rightarrow r = px - kqx = x(p - kq)$.
Since x is a factor in r, so x also divides r.
Therefore, x divides m, $n \Rightarrow x$ divides n, r i.e. GCD of $(m, n) \Rightarrow$ GCD of (n, r).

The problem of finding GCD of (m, n) now becomes equivalent to the problem of finding GCD of (n, r) which of course is easier than the original problem. Now by reducing the problem continuously using the above formula, finally the desired GCD can be found when n will divide m. Therefore, the key step to reduce the GCD problem is following.

The value of m in i^{th} iteration = the value of n in $(i - 1)^{th}$ iteration (1)
The value of n in i^{th} iteration = the value of r in $(i - 1)^{th}$ iteration (2)

The algorithm will generate following sequence of sub-problem for the example chosen at the beginning.

GCD (186,84) $\Rightarrow$ *GCD* (84,186 mod 84) $\Rightarrow$ *GCD* (84, 18) $\Rightarrow$ *GCD* (18, 84 mod 18) $\Rightarrow$ *GCD* (18, 12) = *GCD* (12, 18 mod 12) $\Rightarrow$ *GCD* (12, 6) **(Table 1.6)**

No. of iterations	First (larger) Integer (m)	Second Integer (n)	Remainder $r = m \bmod n$
0	186	84	18
1	84	18	12
2	18	12	6
3	12	6	0

GCD

Table 1.6

At the end, the sub problem *GCD* (6, 12) is easiest to solve since 6 divides 12, so the GCD of (84, 186) = 6. The important fact to be noted that Euclid's algorithm takes only 4 iterations to give the result which is very small compared to the 78 iterations taken by the approach 1. **Figure 1.13** provides the algorithm and flowchart of Euclid's GCD algorithm.

One of the other advantages of Euclid's algorithm is, it doesn't require the explicit ordering of the two values m, n. In the last example, the first integer m

(initial value is 186) is assumed to be the larger integer in all the sub problems. In spite of that, if the value of m and n are such that $m = 84$, $n = 186$ then after the first iteration m and n will interchange their values to make the assumption of m being larger valid from next iteration onwards. $GCD\,(84, 186) \Rightarrow GCD\,(186, 84 \text{ mod } 186) \Rightarrow GCD\,(186, 84)$.

Check Your Progress 1.19

Design an algorithm and flowchart to find the GCD using the approach 1 mentioned in section 1.6.2.

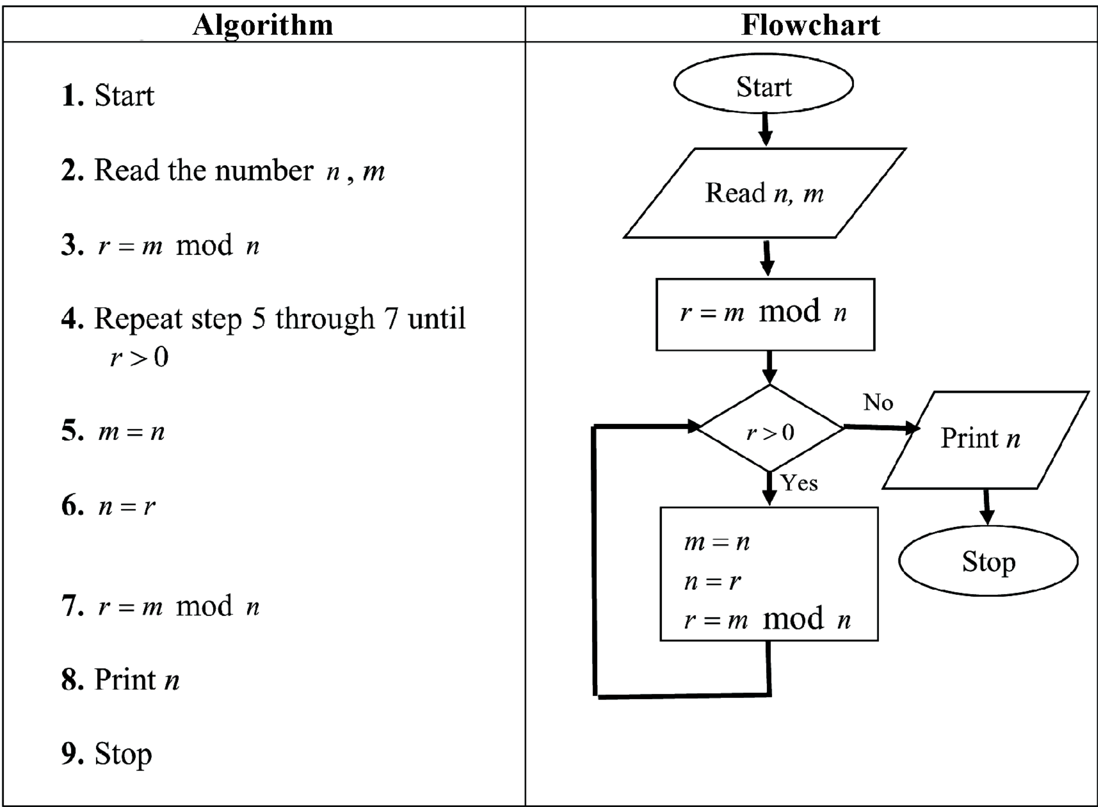
Check Your Progress 1.20

Design a GCD algorithm which does not use either division or mod function.

Check Your Progress 1.21

Design an algorithm and flowchart that will find the GCD of n positive integers.

Algorithm Description and Flowchart :



Algorithm	Flowchart
1. Start **2.** Read the number n, m **3.** $r = m$ mod n **4.** Repeat step 5 through 7 until $r > 0$ **5.** $m = n$ **6.** $n = r$ **7.** $r = m$ mod n **8.** Print n **9.** Stop	

Figure 1.13

Check Your Progress 1.22

Design an algorithm to compute smallest common divisor, other than one, of two positive non-zero integers.

Check Your Progress 1.23

Design an algorithm to compute lowest common multiple (LCM) of two non-zero positive integers n and p. The LCM is defined as the lowest integer m such that n and p divide exactly into m.

1.7 Summary

There are many different aspects for problem solving. It is a common practice to analyze these aspects before solving new problems. These aspects will help and guide to find the solution by systematic way. Steps should be followed to solve the problem that includes writing the algorithm and drawing the flowchart for the solution to the stated problem. The important point that needs to be remembered while developing the algorithm is the efficiency. There are certain criteria of any good algorithm. When these criteria are followed the algorithm becomes efficient and useful.

1.8 References and Further Reading

1. How to solve it by Computer, 5th Edition, R G Dromey, PHI, 1992.
2. Introduction to Computer Algorithms, Second Edition, Thomas H. Cormen, MIT press, 2001.
3. Fundamental Algorithms, Third Edition, Donald E Knuth, Addison-Wesley, 1997.
4. How to solve it, Polya G, Princeton University Press, 1971.

Unit - 2 ❑ Introducing C

Structure

2.0 Introduction

2.1 Objectives

2.2 What is a Programming Language and What is a Program?

2.3 C Language

- **2.3.1 Some Features of C Language**
- **2.3.2 Writing a Simple C Program**
- **2.3.3 Compiling a C program**
- **2.3.4 Running a C Program**
- **2.3.5 Integrated Development Environment**
- **2.3.6 Installation steps for Code::Blocks IDE**

2.4 Summary

2.5 References and Further Reading

2.0 Introduction

If someone claims to have the perfect programming language, he is either a fool or a salesman or both.

—Bjarne Stroustrup.

In the earlier unit, the problem-solving aspects along with few very famous algorithms and their flowcharts were discussed. Now the next step is to implement those algorithms so that the computer can execute them. In this unit the topic that is going to be covered is C language — a standardized programming language known for its power and portability as an implementation vehicle for these problem solving techniques using computer.

A language is a mode of communication between two people. It is necessary for those two people to understand the language in order to communicate. But even if the two people do not understand the same language, a translator can help to convert one language to the other, understood by the second person. Similar to a translator a computer language is the mode of communication between a user and a computer. One form of the computer language is understood by the user, while in the other form it is understood by the computer. A translator (or compiler) is needed to convert from user's form to computer's form. Like other languages, a computer language also follows a particular grammar known as the syntax.

2.1 Objectives

After going through this unit the learner should be able to :

- Define what is a program?
- Understand what is C programming language?
- Understand step-by-step process of compilation and execution of C program.
- Write small C program.
- Understand what is an integrated development environment (IDE).
- Install Code::Block IDE in their personal computer.

2.2 What is a Programming Language and What is a Program?

The detailed set of steps for solving a problem, known as Algorithm and their pictorial representation, known as Flowchart have been discussed in the previous unit. Now the next step is to express the algorithm in programming language. A programming language is a vocabulary and set of grammatical rules for instructing a computer or computing device to perform specific tasks.

A procedure expressed in a programming language is known as a computer program. A computer program tells the computer how to do what programmers want. Just as a chef needs a recipe to make a dish, a program needs instructions to produce results. A recipe is nothing more than a set of detailed instructions that, if properly written, describes that proper sequence and the contents of the steps needed to prepare a certain dish. That's exactly what a computer program is to computer.

Programming languages can be divided into two categories :

1. Low Level Language or Machine Language

A machine language consists of the numeric codes for the operations that a particular computer can execute directly. The codes are strings of 0s and 1s, or binary digits ("bits"). Machine language instructions typically use some bits to represent operations, such as addition, and some to represent operands, or perhaps the location of the next instruction. Machine language is difficult to read and write, since it does not resemble conventional mathematical notation or human language, and its codes vary from computer to computer and therefore the language is machine dependent. Another type of Low-Level Language is the Assembly Language. It uses short

mnemonic codes for instructions and allows the programmer to introduce names for blocks of memory that hold data. Every machine provides a different set of mnemonics to be used for that machine only depending upon the processor that the machine is using.

2. High Level Language

These languages are particularly oriented towards describing the procedures for solving the problem in a concise, precise and unambiguous manner. Every high level language follows a precise set of rules. They are developed to allow application programs to be run on a variety of computers. These languages are machine-independent. Languages falling in this category are FORTRAN, BASIC, PASCAL, etc. They are easy to learn and programs may be written in these languages with much less effort. However, the computer cannot understand them and they need to be translated into machine language with the help of other programs known as Compilers or Translators (Interpretors).

2.3 C Language

C is a general-purpose computer programming language initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs. Most of its constructs map efficiently to typical machine instructions, and therefore it has been found very useful in applications that had formerly been coded in assembly language, most notably system software like the UNIX operating system. By 1980s, however, C has expanded beyond the narrow confines of UNIX world, undergone lot of standardisation process and formally approved by International Organization for Standardisation. This version of C language is usually referred to as C89 or C90 which is used in this book.

2.3.1 Some Features of C Language

C is a general purpose, structured programming language. Structured language has following distinguishing features.

1. The data and code are physically separated.
2. The entire program is divided into modules using top-down approach which helps in debugging, testing and maintenance of the code.
3. It supports several control structures like *while, do-while, for* and several data structures *like structure, array, file* etc.

Among the two types of programming languages discussed earlier, C lies in between these two categories. Therefore, C is often called a middle level language. It combines the elements of high level languages with the functionality of assembly language. It provides relatively good programming efficiency (as compared to machine oriented language) and relatively good machine efficiency as compared to high level languages). As a middle level language, C allows the manipulation of bits, bytes and addresses — the basic elements with which the computer executes the inbuilt and memory management functions. C language has lot of other important features. Though most of these are strength of the language, some features still show certain weakness also.

1. Efficiency :

C was intended for application written in assembly language (low level programming language for microprocessors and other programmable devices), therefore special care had been taken to design the language from the beginning so that it could run quickly in limited amount of memory.

2. Portability :

C is highly portable, this means that programs once written can be run on another machine with little or no modification.

3. Power :

C has large collections of data types and operators which make it a powerful language. These collections can be used to write a complex program with just few lines of code.

4. Flexibility :

Though C was originally designed for system programming, C is now capable of handling all kinds of application like embedded system, commercial software application etc.

5. Standard Library :

C's great strength lays on the standard library which contains hundreds of functions for input/output, string handling, storage allocation and other useful operations.

6. Integration with UNIX :

C is very powerful in combination with UNIX.

7. C program can be error-prone.

C's flexibility sometimes makes it error-prone. Programming mistakes sometimes remain undetected in C, which is normally easy to detect in some other language.

8. C Programs can be difficult to modify :

Large programs written in C can be difficult to modify if they have not been designed with maintenance plan in mind. Most of the modern programming languages like C++, Java, Python have certain features like classes, packages that can divide large program into more manageable pieces. **Figure 2.1** shows all these features in a diagram.

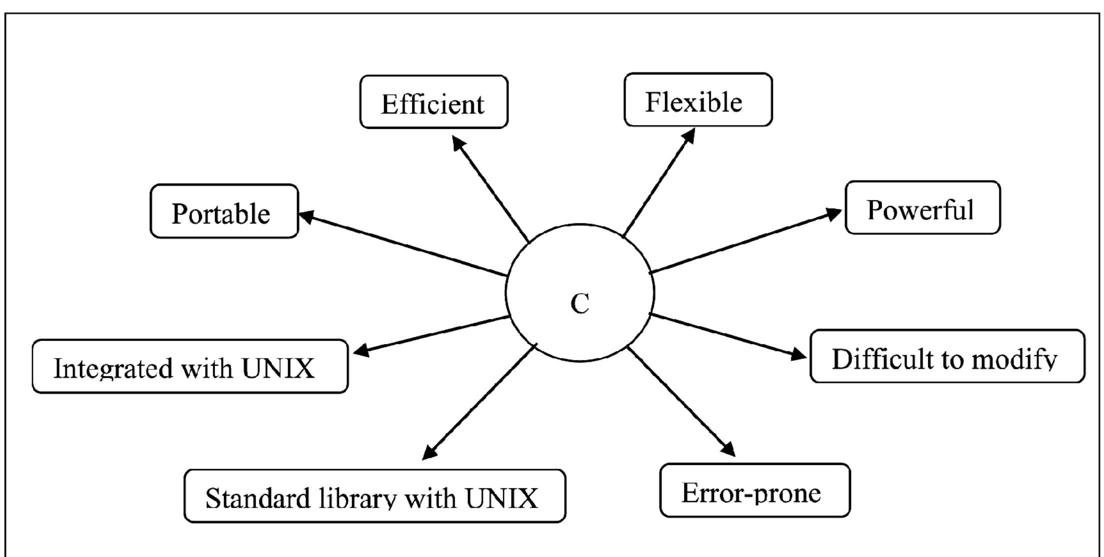


Figure 2.1

2.3.2 Writing a Simple C Program

The C program in **Table 2.1** displays a message *"Welcome to C"*.

Line No	Code
1	/*Program to print a message*/
2	#include<stdio.h> /* header file*/
3	int main() /* main function*/
4	{
5	printf("Welcome to C"); /* output statement*/
6	return 0;
7	}

Table 2.1

The lines of code written in **Table 2.1** are briefly explained in **Table 2.2**. The more detailed explanation will be provided in subsequent units with relevant examples.

Line No	Code	Brief Description
2	#include <stdio.h>	This is known as preprocessor directive. This directive states that the information in <stdio.h> is to be included into the program at the beginning. <stdio.h> contains information about C's standard I/O (Input Output) library. C performs input (read) and output (write) operations using functions in the standard library instead of using any built-in "read" and "write" command like few other languages. In this case, the preprocessor directive is used to use the printf() function in line 4.
3	int main()	This declares the start of the function *main()*. This function is the entry point of the program. Functions are the building blocks by which programs are constructed. C program is little more than a collection of functions. Functions fall into two categories: those written by the programmers, known as *User Defined Functions* and those provided as a part of the C language, known as *Library Functions*. The term "function" comes from mathematics, where a function is a rule for computing a value when given one or more arguments : $f(x) = x^2 + 5$ $g(x, y) = x^2 + y^3$ C uses "function" more loosely. In C, a function is a series of statements that have been grouped together and given a name. Some functions compute a value and some don't. A function that compute a value uses a *return* statement to specify what value it returns. *Int* before *main()* function indicates that *main* will return an integer. Although C program may have many functions, only the *main()* function is mandatory. It gets called automatically when the program is executed. The name *main* is critical; it cannot be *begin* or *start* or even *MAIN*. Since functions are outside the scope of this course, only the *main()* function will be used in subsequent chapters.

Line No	Code	Brief Description
4	{	This opening curly bracket shows the start of *main* function.
5	printf ("Welcome to C");	This is first statement in the main function. The statement is a command to be executed when the program runs. C requires each statement end with a semicolon. *printf* () is a function from the standard I/O library that can produce nice formatted output. *printf* () will be discussed in section 3.7 in more detail.
6	return 0;	This indicates that the *main*() function will return 0 to the operating system when the program terminates. The value returned by *main*() is a status code that can be tested when the program terminates. *main*() should return 0 if the program terminates normally; to indicate abnormal termination *main*() should return a non-zero value. It is a good practice to make sure that every C program should return a status code, even if there is no plan to use it, since someone running the program later may decide to test it. The *return* 0 statement would have been omitted from line no 6 if void *main*() was used instead of int *main*() at line no 3. Here void is used to specify that nothing needs to be returned from the *main*() function.
7	}	This closing curly bracket shows the end of *main* function. The brackets in line no 4 and line no 7, groups statements together as *main* () function.
Gen eric	/*some text*/	These indicates Comments which may appear anywhere within a program, as long as they are placed within the delimiters /* and */. Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features. These comments will be ignored by the compiler later, therefore the comment is not part of C language grammar, it is used only for documentation purpose.

Table 2.2

2.3.3 Compiling a C program

After writing the program next step is to save the program in a file with extension .c. This program is in high level language and this language is not understood by the computer. So, the next step is to convert the high-level language program (source code) to machine language (object code). This task is performed by a software or program known as a compiler. In fact, there are four programs involved in the compilation process: pre-processor, compiler, assembler, linker. **Table 2.3** shows the brief functionality of all these entities.

Program Name	Functionality
Preprocessor	First, the C preprocessor expands all those include statements (and anything else that starts with a # and passes the result to the actual compiler. The preprocessor is not so interesting because it just replaces some short cuts used in the code with more code. The output of preprocessor is just C code; The preprocessor does not require any knowledge about the target architecture.
Compiler	The compiler effectively translates preprocessed C code into assembly code, performing various optimizations. Since a compiler generates assembly code specific to a particular architecture, the assembly output of compiler from an Intel Pentium machine can't be used on any other type of instructional machine (such as Digital Alpha machines).
Assembler	The assembly code generated by the compilation step is then passed to the assembler which translates it into machine code; the resulting file is called an object file. An object file is a binary representation of the program. The assembler gives a *memory location* to each variable and instruction. It also makes a list of all the unresolved references that presumably will be defined in other object file or libraries, *e.g.* printf file.
Linker	This is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

Table 2.3

2.3.4 Running a C Program

After compilation when run command is fired, a whole bunch of things must happen before the program is actually run. The loader reads the file and creates an address space for the process. The loader executes a jump instruction to the first instruction in the program. This generally brought the instruction into the main memory and the actual program execution starts (Refer **Figure 2.2**).

2.3.5 Integrated Development Environment

All the steps those have been discussed so far, can be executed as separate command line in a special window provided by the operating system. The alternative is to use an integrated development environment (IDE), a software package that allows to edit, compile, link, execute and many things together without leaving the environment. For example, when the compiler detects an error in a program, it can arrange for the editor to highlight the line that contain error. Though there is a variety of IDEs for C compilers, this course will use Code::Blocks (www.codeblocks.org) in all subsequent units. This is primarily because Code::Blocks offers C compilers for most of the operating systems like Windows, Macs, Linux etc.

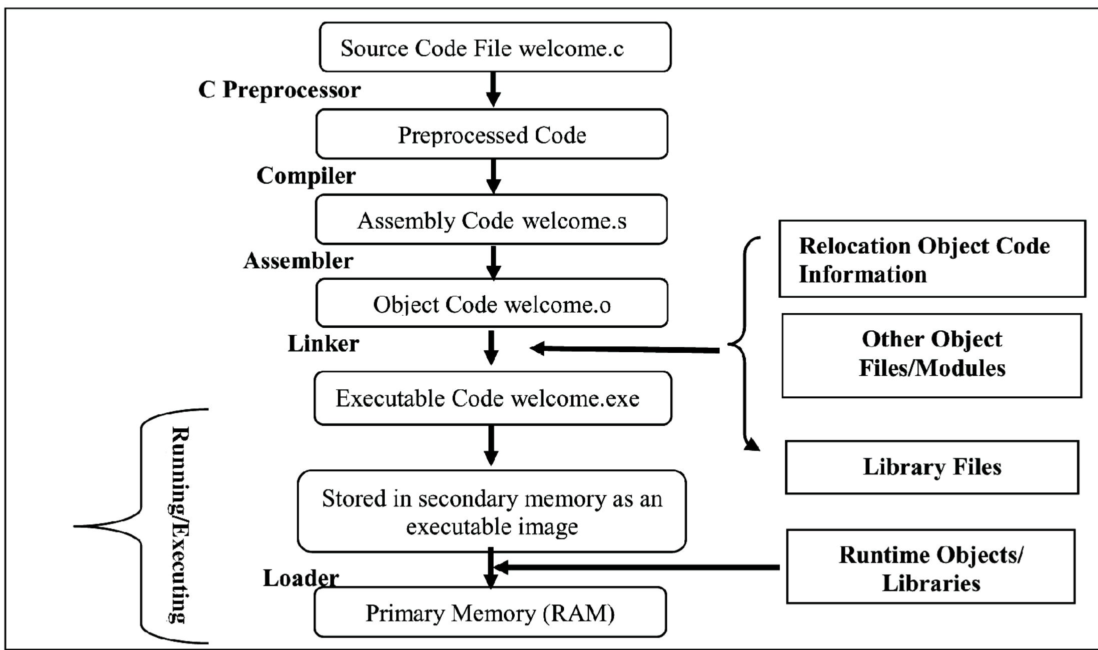


Figure 2.2

2.3.6 Installation steps for Code::Blocks IDE

Step 1 :

Go to Code::Blocks homepage by typing www.codeblocks.org in the internet browsers. To download C/C++ IDE, click download choice under the main section of the left column. **Figure 2.3** shows download option in the Code::Block homepage.

Figure 2.3

Step 2 :

There will be three options under the download link: Binaries, Source, SVN. Click on the first option Binaries. The next page represents variety of option, depending on the operating system. For example, if the operating system is windows then Figure 2.4 shows all the C compilers. Out of all the different C compilers the *codeblocks-17.12 mingw-setup.exe* will be selected which is highlighted in the Figure 2.4. To download the compiler, click on the corresponding link in *download from* column.

Figure 2.4

Step 3 :

After the download option is chosen, it takes several minutes to complete the download. Run the downloaded file codeblocks-17.12 mingw-setup.exe accepting all defaults. It again takes few minutes and get installed in the folder chosen by user.

Step 4 :

Click on the Code::Block IDE installed in the machine. The screen opens up which is shown in **Figure 2.5**. Go to the file option in the top left corner highlighted in the **Figure 2.5**. Then go to File → New → **File (Figure 2.6)**.

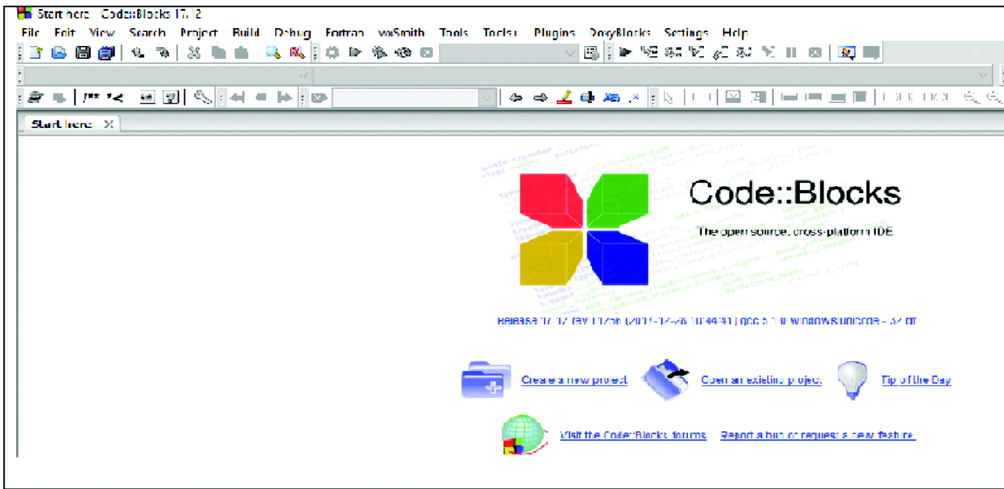


Figure 2.5

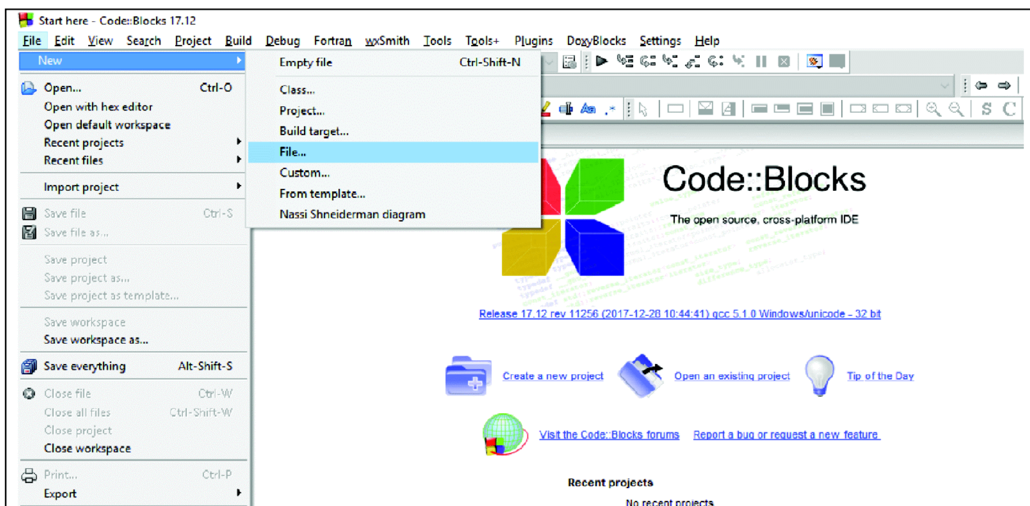


Figure 2.6

Clicking on the File option a list of source options appears. Select the option C/C++ source **(Figure 2.7)** and then click on go. The C/C++ source wizard opens up. Click on Next. Select the language as C and click on Next. Enter the file name with full path in C/C++ source wizard **(Figure 2.8)**. Then click on finish. The IDE creates welcome.c program in the path chosen in the wizard. Paste the program written in **Table 2.1**. in welcome.c file and go to File → Save option. **Figure 2.9** shows the program welcome.c in the editor.

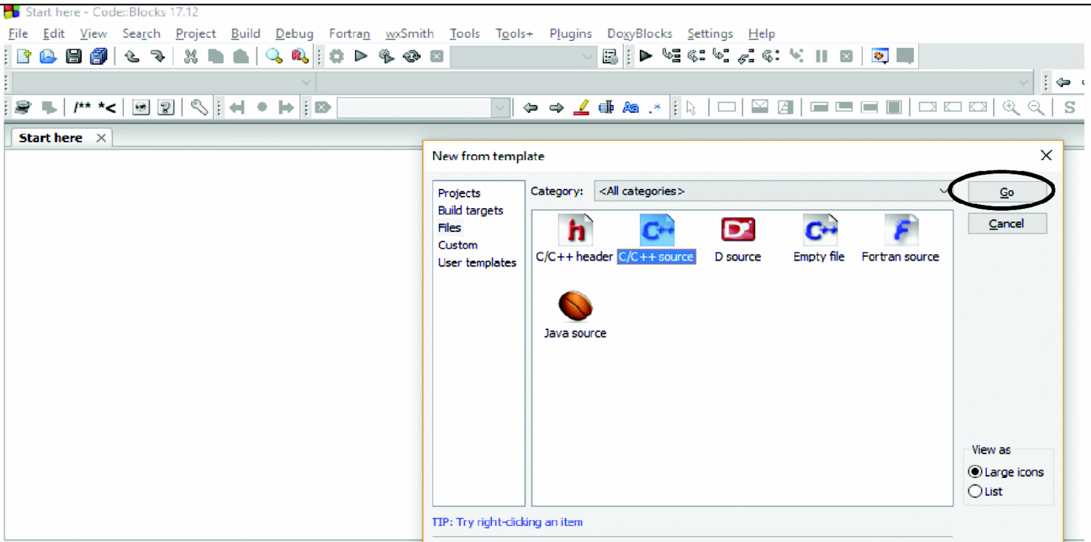


Figure 2.7

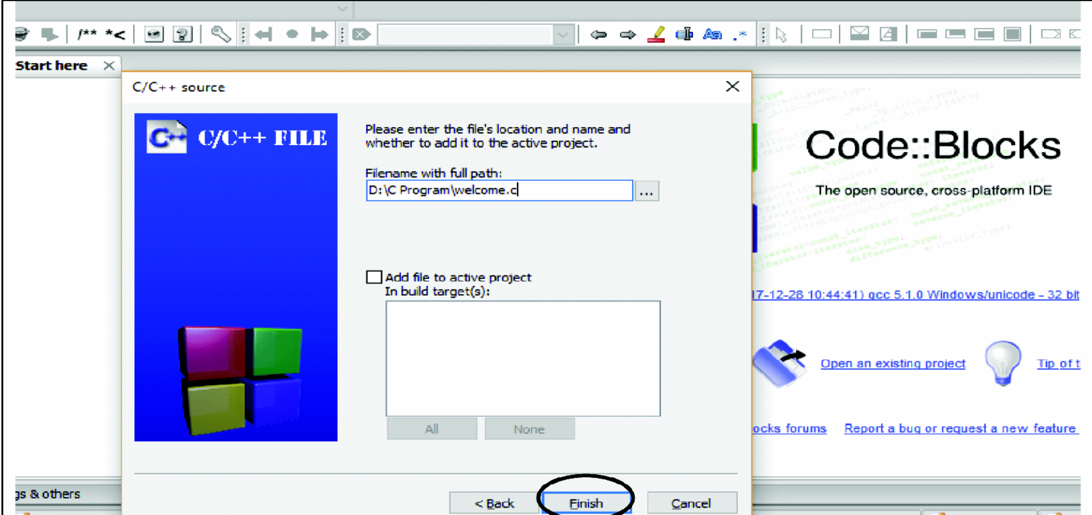


Figure 2.8

Step 5 :

Once the program is complete then it needs to be compiled and then run. The IDE gives two options Build and Run at the top ribbon (Highlighted in the **Figure 2.9**. Clicking the Build option, initiates all the steps of (Pre-processing, compiling, assembling, linking) compilation as per the sequence outlined in **Figure 2.2**.

Build
Run
Build and Run
*welcome.c - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

```
/*Program to print a message*/
#include<stdio.h>                /* header file*/
int main()                      /* main function*/
{
    printf("Welcome to C");     /* output statement*/
    return 0;
}
```

Figure 2.9

The build process produces the build log, build messages and lot of other important information about the program. **Figure 2.10** highlights the contents of the build messages which says that the welcome.c program is compiled successfully with zero errors and zero warnings.

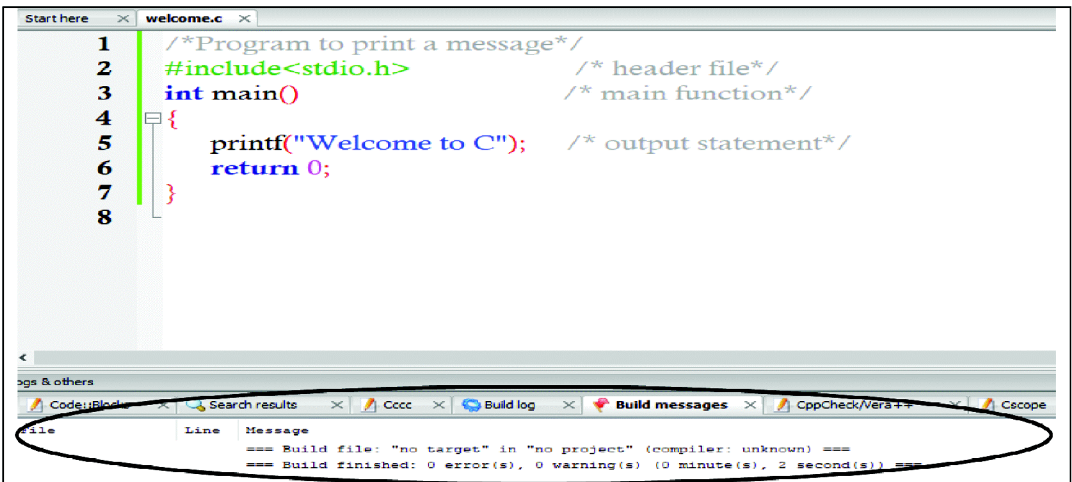


Figure 2.10

Step 6 :

At last, click on the Run option to start the execution of the program. After the successful execution of the program the desired output is displayed in the screen **(Figure 2.11)**

```
Welcome to C
Process returned 0 (0 × 0)          execution time : 0.283 s
Press any key to continue.
```

Figure 2.11

The version of Code::Blocks used in this book is 17.12, but the number will be probably even larger by the time when learners will use this. The learners need to make sure that they should select the most up-to-date version to exploit all other facilities provided by Code::Block.

2.4 Summary

Several key aspects of program and programing languages are discussed in this unit. Learners can now differentiate between high level and low level languages. They can now define what is C, features of C. Learners have seen how C is different from other High Level languages. Learners can now explain the detailed steps of a compiler and how they are connected to each other. They can now install the IDE to develop their own programs. With these basics, learners are now ready to learn the C language in detail in the following units.

2.5 References and Further Reading

1. The C Programming Language, Kernighan & Ritchie, PHI Publication, 2011.
2. Programming with C, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
3. The C Complete Reference, Fourth Editon, Herbert Schildt, Tata McGraw Hill, 2002.
4. C Programming Absolute Beginner's Guide, Third Edition, Pearson Education, Inc, 2014
5. Online reference 1- https://www.tenouk.com/Module W.html
6. Online reference 2- https://courses.cs.washington.edu/courses/cse378/97au/help/compilation.html

Unit - 3 ❑ Variables, Constants and Input/Output

Structure

3.0 Introduction
3.1 Objectives
3.2 C Character Set
3.3 Identifiers and Keywords
3.4 Data Types and Storage
- **3.4.1 Integer Types**
- **3.4.2 Floating Types**
- **3.4.3 Character Types**

3.5 Variables
- **3.5.1 Variable Type**
- **3.5.2 Variable Declarations**
- **3.5.3 Variable Initialization**

3.6 Constants
- **3.6.1 Integer Constant**
- **3.6.2 Floating Constants**
- **3.6.3 Character Constants**
- **3.6.4 String Constants**
- **3.6.5 Escape Sequence**

3.7 Data Input Output Function
- **3.7.1 The printf Function**
- **3.7.2 The scanf Function**

3.8 Summary
3.9 References and Further Reading

3.0 Introduction

This unit is concerned with the most fundamental elements which are used to construct the C statements. A statement in C is a command to be executed when the program runs. A statement is usually comprised of different elements from C character set, identifiers and keywords, data types, variables, constants etc. These

basic elements are introduced in this unit and the later unit will cover these topics in much greater detail. The last section of this unit describes two very powerful functions printf and scanf which are used in C most frequently.

3.1 Objectives

After going through this unit the learner should be able to :

- Understand what is C character set, identifier, keyword, variable and constant.
- Understand the mechanism of storing floating point data in IEEE format.
- Understand printf and scanf function for input output operation.
- Write program which takes data from user/programmer and display it according to various format.

3.2 C Character Set

C mainly uses the A to Z (uppercase letters), a to z (lowercase letters), 0 to 9 (digits), and certain special characters as building blocks to form basic program elements like constants, variables, operators, expressions, etc. C uses ASCII (American Standard Code for Information Interchange) character set, a 7-bit code capable of representing 128 characters. All these characters are listed in **Table 3.1**.

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
0	NUL	32	(blank)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	“	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	‘	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	•	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	-
31	US	63	?	95	_	127	DEL

Table 3.1

The first 32 characters and the last character are control characters. Usually, they are not displayed. However, some versions of C (some computers) support special graphics characters for these ASCII values.

C uses different combinations of these characters, such as \b, \n and \t, to represent special conditions like backspace, newline and horizontal tab, respectively. These character combinations are known as escape sequences. The escape sequences will be discussed more in Sec. 3.6.5.

3.3 Identifiers and Keywords

While writing programs in C, names are given to various program elements, such as variables, functions and arrays. These names are called identifiers. An identifier may contain letters, digits and underscores, but must begin with a letter or underscore. C is case sensitive, therefore it distinguishes between upper-case and lower-case letters in identifiers. **Table 3.2** shows some valid and invalid identifiers in C.

There are certain reserved words, called keywords, that have standard, predefined meanings in C. These keywords have special significance to C compilers and therefore can't be used as identifiers. The standard keywords are listed in **Table 3.3**.

Identifiers	Valid/Invalid	Comments
A	Valid	
"a"	Invalid	Illegal character (")
x12	Valid	
5th	Invalid	The first character cannot be a digit.
Y12	Valid	
get_next	Valid	
get-next	Invalid	Illegal character (-)
status_flg	Valid	
Status flg	Invalid	Illegal character (blank space)
_temp	Valid	
temp_	Valid	

Table 3.2

auto	*default*	*float*	*long*	*static*	*void*
break	*do*	*for*	*register*	*struct*	*while*
case	*double*	*goto*	*return*	*switch*	
char	*else*	*if*	*short*	*typedef*	
const	*enum*	*inline*	*signed*	*union*	
continue	*extern*	int	*sizeof*	*unsigned*	

Table 3.3

Check Your Progress 3.1

Which of the following are not legal identifier in C?

100_bottoles _100_bottles one_hundred_bottles Bottles_100_

Check Your Progress 3.2

Which of the following are keywords in C?

for If main printf while

3.4 Data Types and Storage

C has several basic (built-in) data types. Each of them is represented differently within the computer's memory. As a result, the range of values of these data types are also different. The important point that needs to be noted is that the memory requirement of each of these data types may vary from one compiler to another. Typically, C has following three basic data types :

- Integer Types
- Floating Types
- Character Types

3.4.1 Integer Types

The values of integer types are whole numbers. They are of two types : signed and unsigned. For the signed integer, the leftmost bit (sign bit) is 0 if the integer is positive or zero, 1 if it is negative. As the numbers in computer memory are in binary format, it can only use 0 or 1 for every digit. Therefore, the largest integer in a 16-bit machine is of the form 0111111111111111 (left most bit is 0 and remaining 15 bits are all 1's) which has the value + $(2^{15} - 1)$ = 32, 767. Similarly the largest integer in a 32-bit machine is of the form 01111111111111111111111111111111 (left most bit is 0 and remaining 31 bits are all 1s) which has the value 2,147,483,647 $(2^{31} - 1)$. For unsigned integer there is no sign bit and all the 16 bits are considered to be a part of its magnitude. Therefore, the largest unsigned integer in a 16-bit machine has the value 65,535 $(2^{16} - 1)$.

Based on the size, C has different integer types. The int type is usually 32 bits, but may be 16 bit on older CPUs. C provides long int type for integer too large to

be stored in int form. C also has short int type to store an integer in less space than normal. We can even combine different specifiers like following :

- short int
- unsigned short int
- int
- unsigned int
- long int
- unsigned long int

Other combinations are synonyms for one of these six types. The range of values represented by the above six types varies from one machine to other based on their word-length. A word is the natural unit of data and consists of number of bits processed by a computer's CPU in one go. Apart from early days' processors which typically use 16-bit word, most of the modern processors use either 32 bits or 64-bit word while processing the data. We use the term byte to specify the size of data type. The byte is a unit of digital information that commonly consists of eight bits. **Table 3.4, 3.5 and 3.6** shows the usual range of values for integer types on different machine which use word length as 16-bit, 32-bit and 64-bit respectively. From **Table 3.4** it can be easily seen that short int and int have identical range in 16-bit machine. Similarly, int and long int have identical range in 32-bit machine (**Table 3.5**). The point needs to be noted is that the ranges shown in **Table 3.4, 3.5, 3.6** aren't mandated by the C standard and may vary from one compiler to other.

Data types	**No of bytes**	**Range of values**	
short int	2	-2^{15} to $+(2^{15}-1)$	–32,768 to + 32,767
unsigned short int	2	0 to $(2^{16}-1)$	0 to 65,535
Int	2	-2^{15} to $+(2^{15}-1)$	– 32,768 to + 32,767
unsigned int	2	0 to $(2^{16}-1)$	0 to 65,535
long int	4	-2^{31} to $+(2^{31}-1)$	– 2,147,483,648 to + 2,147,483,647
unsigned long int	4	0 to $(2^{32}-1)$	0 to 4,294,967,295

Table 3.4 (Integer types on a 16-bit machine)

Data types	No of bytes	Range of values	
short int	2	-2^{15} to $+ (2^{15} - 1)$	–32,768 to + 32,767
unsigned short int	2	0 to $(2^{16} - 1)$	0 to 65,535
Int	4	-2^{31} to $+ (2^{31} - 1)$	– 2,147,483,648 to + 2,147,483,647
unsigned int	4	0 to $(2^{32} - 1)$	0 to 4,294,967,295
long int	4	$- 2^{31}$ to $+ (2^{31} - 1)$	– 2,147,483,648 to + 2,147,483,647
unsigned long int	4	0 to $(2^{32} - 1)$	0 to 4,294,967,295

Table 3.5 (Integer types on a 32-bit machine)

Data types	No of bytes	Range of values	
short int	2	-2^{15} to $+ (2^{15} - 1)$	–32,768 to + 32,767
unsigned short int	2	0 to $(2^{16} - 1)$	0 to 65,535
Int	4	-2^{31} to $+ (2^{31} - 1)$	– 2,147,483,648 to + 2,147,483,647
unsigned int	4	0 to $(2^{32} - 1)$	0 to 4,294,967,295
long int	8	$- 2^{63}$ to $+ (2^{63} - 1)$	– 9,223,372,036,854,775,808 to + 9,223,372,036,854,775,807
unsigned long int	8	0 to $(2^{64} - 1)$	0 to 18,446,744,073,709,551,615

Table 3.6 (Integer types on a 64-bit machine)

3.4.2 Floating Types

Integer data type is not sufficient for all real world applications. Sometimes it is also needed to store real numbers which have digits along with a decimal point. These numbers are stored in floating point format (decimal point "floats"). C provides three floating types, corresponding to different floating point formats :

- float Single precession floating point
- double Double precession floating point
- long double Double Extended precession floating point

The C standard doesn't state the precession provided by float, double and long double since different computers may store floating point numbers in different ways. Most modern computers follow the specifications in IEEE Standard 754, developed by Institute of Electrical and Electronics Engineers. It provides single precession (32 bit) format known as float and double precession (64 bits) known as double. Each of these formats has three parts : sign, exponent, fraction or significant precession. The no of bits reserved for exponent determines how large (or small) numbers can be, while no of bits in the fraction determines the precession. **Figure 3.1** and **Figure 3.2** shows the structure of single precession (32 bit) float and double precession (64 bit) double format respectively.

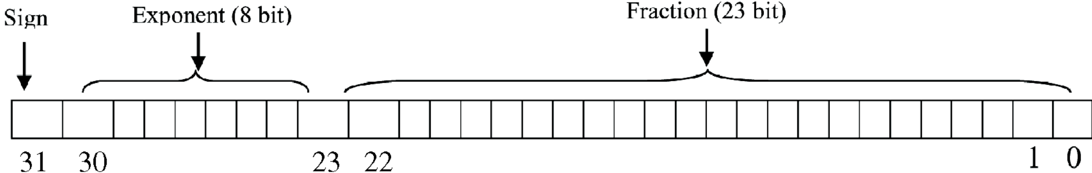


Figure 3.1 (Single precession 32 bit floating point)

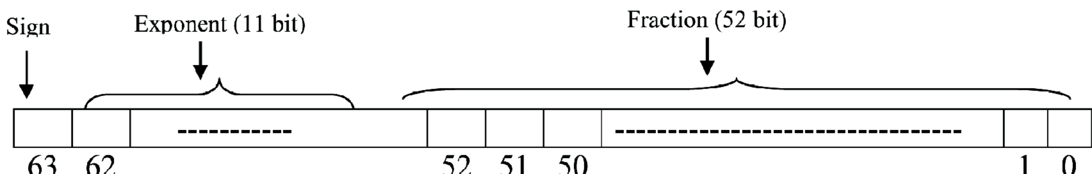


Figure 3.2 (Double precession 64 bit floating point)

Table 3.7 shows the characteristics of the floating point number when implemented according to the IEEE standard. The long double type isn't shown in the table, since its length varies from one machine to another, with 80 bits and 128 bits being the most common size.

Type	No of bits	Smallest positive value	Largest value	Precession
Float	32	1.17549×10^{-38}	3.40282×10^{38}	6 digits
double	64	2.22507×10^{-308}	1.79769×10^{308}	15 digits

Table 3.7

In IEEE standard the precession for float data type is 6 digits **(Table 3.7)**. This implies that when a decimal number with at most 6 significant digits is converted to IEEE 754 single-precision representation, and then converted back to a decimal number with the same number of digits, the final result should match the original number.

3.4.3 Character Types

The third type of basic data type is char, the character type. The values of type char can vary from one machine to another, since different machines have different underlying character set. The important point to be noted that C treats characters as small integers. For example, in ASCII, the character 'a' has the value 97, 'A' has the value 65, '0' has the value 48, and ' ' has the value 32. When characters appear in computation, C simply uses the integer value. **Figure 3.3** represents the tree diagram of the basic data types used in C.

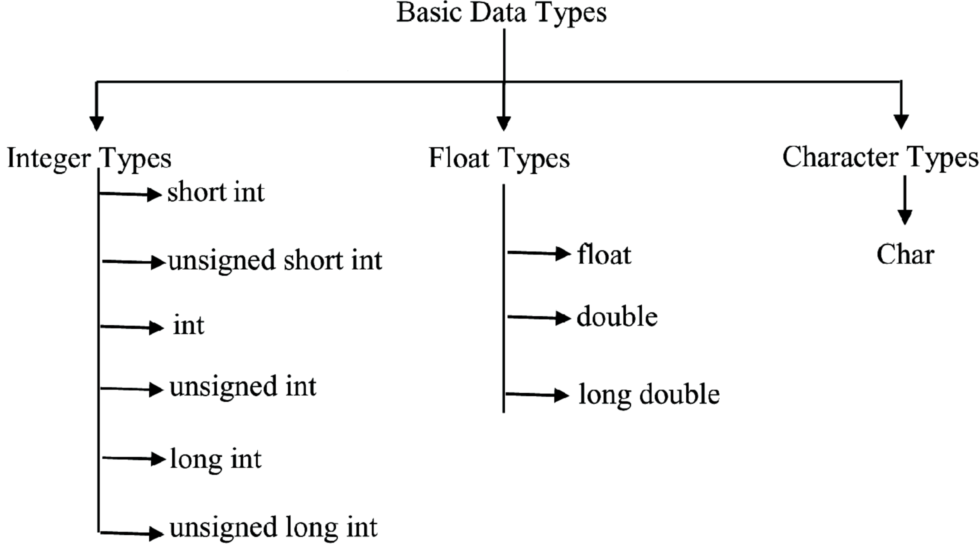


Figure 3.3 (Basic data types in C)

Check Your Progress 3.3

Which of the following are not legal type in C?

a. short unsigned int
b. short float
c. long double
d. unsigned long

3.5 Variables

A C program may perform a series of calculations to produce the desired output and thus need a way to store data temporarily during program execution. These storage locations are commonly referred as variables.

3.5.1 Variable Type

Every variable in C must have a type which specifies the type of the value the variable is capable to hold. There are variety of variable types in C. For example, int, float, char etc. Based on the type of the variable the Compiler determines how the variable is stored and what operation can be performed on it.

3.5.2 Variable Declarations

Variable must be declared before it can be used to store any value. To declare a variable two attributes are mandatory : variable type and variable name. For example, we might declare variables price and total_items as follows :

```
float price;
int total_items;
```

The first declaration states that price can store floating point number and second declaration states that total_items can store integer value. If several variables are of the same type, their declaration can be combined.

```
float price, amount, discount;
int total_items, Year_of_purchase;
```

Note that each complete declaration ends with a semicolon. The first program in **Table 2.1** (section 2.3.2) didn't include variable declarations. When main function contains declarations, these must precede statements:

```
int main()
{
        declarations;
        statements;
}
```

A variable can be given a value by means of assignment (=). For example, the statements

```
price = 10.5;
total_items = 5;
amount = 2.35;
```

assign values to price, total_items and amount. The number 10.5, 5 and 2.35 are said to be constants. Variable must be declared before assigning any value otherwise the compiler will raise an error.

See the output message in **Table 3.8**.

Line #	Code	Output message after building	Comments

1
2
3
4
5
6
7
8
9
	#include<stdio.h>
	int main()
	{
	int total_items;
	float price;
	price=10.5;
	total_items=5;
	return 0;
	}
		Message
		=== Build file: "no target" in "no project"
		=== Build finished : 0 errors (s), 0 warning (s)
			Variable declared before assigning value successfully compiled.

1
2
3
4
5
6
7
8
9
	#include<stdio.h>
	int main()
	{
	int total_items;
	price=10.5;
	total_items=5;
	float price;
	return 0;
	}
		Message
		=== Build file: "no target" in "no project"
		In function 'main' :
		errors: 'price' undeclared (first use in this
		note : each undeclared identifier is reported
		== Build failed : 1 errors (s), 0 warning (s) (
			price has been declared after assigning value in it which produces compilation error in line no 5.

Table 3.8

3.5.3 Variable Initialization

In most of the cases, C variables are not set to a default value. These variables are said to be uninitialized. One way to initialize a variable is to use assignment statement. Another easier way is to put the value of the variable in its declaration. **Table 3.9** shows both these ways.

Initialization by assignment	Initialization in declaration

#include<stdio.h>
int main()
{ int total_items;
float price;
total_items =5
price = 10.5;
return 0;
}
	#include<stdio.h>
	int main()
	{ int total_items = 5;
	float price = 10.5;
	return 0;
	}

Table 3.9

3.6 Constants

A constant is an identifier whose value cannot be changed throughout the execution of a program whereas the variable value keeps on changing. In C, there are four basic types of constants. They are :

- ➢ Integer Constant
- ➢ Floating Point Constant
- ➢ Character Constant
- ➢ String Constant

3.6.1 Integer Constant

C allows integer constants to be written in decimal (base 10), Octal (base 8) or hexadecimal (base 16). The first digit of a decimal constant is always non-zero.

Octal and Hexadecimal Numbers

An octal number is written using only the digits 0 through 7. Each position of an octal number represents a power of 8 (just as each position in a decimal number represents a power of 10). Thus the octal number 357 represents the decimal number $3 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 = 239$. The first digit must be zero for an octal no.

A hexadecimal number is written using the digits 0 through 9 and the letters A through F, which stands for 10 through 15, respectively. Each position of a hex number represents a power of 16. Thus the hex number 2D5 represents the decimal number $2 \times 16^2 + 13 \times 16^1 + 5 \times 16^0 = 725$. The hex constants always begin with 0x.

Rules to form Integer Constants

- ➢ No comma or blank space is allowed in a constant.
- ➢ It can be preceded by – (minus) sign if desired.
- ➢ The value should lie within a minimum and maximum permissible range decided by the word size of the computer.

Table 3.10 shows the examples of few valid and invalid integer constants.

The point to be noted that octal and hexadecimal are an alternative way of writing numbers. Integers are always stored as binary irrespective of the notation to express them. It can be easily switched from one notation to another at any time, and even they can be mixed. For example, in the expression 11 + 026 + 0x1FF, 11 is a decimal, 026 is an octal and 0x1FF is a hexadecimal number. The expression produce following output :

$$1 \times 10^1 + 1 \times 10^0 + 2 \times 8^1 + 6 \times 8^0 + 1 \times 16^2 + 15 \times 16^1 + 15 \times 16^0 = 544$$

The type of a decimal integer constant is normally int. However, if the value is too large to store as an int, the constant has type long int. To force the compiler to treat a constant as long integer, the letter L (or l) needs to be appended at the end of the constant. For example,

25L, 056L, 0x7FFFL

To indicate the constant as unsigned, the letter U (or u) needs to be added at the end of the constant. For example,

25U, 056U, 0x7FFFU

L and U also can be combined to indicate the constant is both unsigned and long int. For example, 0x7FFFLU.

Base	Constant	Valid/Invalid	Reason
Decimal	54	Valid	
	12,45	Invalid	Comma(,) not allowed
	1 011	Invalid	Blank space not allowed
	10-10	Invalid	Illegal character (-)
	0534	Invalid	The first digit should not be a zero.
Octal	0743	Valid	
	743	Invalid	The first digit must be a zero.
	0238	Invalid	Illegal character 8
	012.4	Invalid	Illegal character (.)
Hexadecimal	0x7FFF	Valid	
	0BEF	Invalid	x is not included after 0.
	0x5.CD	Invalid	Illegal character (.)
	0xGBC	Invalid	Illegal character G

Table 3.10

3.6.2 Floating Constants

A floating constant must contain a decimal point and/or an exponent. The exponent indicates the power of 10 by which the number is to be scaled. If an exponent is present, it must be preceded by the letter E (or e). An optional + or –

sign may appear after the letter E (or e). Rule to form a floating point constant is same as integer. **Table 3.11** shows the examples of few valid and invalid floating point constants.

Constant	Representation	Valid/Invalid	Reason
35.0	35.0	Valid	
	35.0e0	Valid	
	35.0 e0	Invalid	No blank space
	35E0	Valid	
	3.5e1	Valid	
	3.5,e1	Invalid	Comma(,) not allowed
	.35e2	Valid	
	.35e2.0	Invalid	Exponent can't be fraction
	350e-1	Valid	

Table 3.11

The floating point constants are stored as double precession numbers by default. On occasion, it may be necessary to force the compiler to store a floating constant in float or long double format. To indicate that only single precession is desired, letter F (or f) needs to be appended at the end of the constant. For example, 35.0F. Similarly, for long double the letter L (or l) is used. For example, 35.0L.

3.6.3 Character Constants

This constant is a single character enclosed in apostrophes ‘ ’ . For example, some of the character constants are shown below :

‘A’, ‘x’, ‘3’, ‘$’

‘\0’ is a null character having value zero.

3.6.4 String Constants

It consists of sequence of characters enclosed within double quotes. For example, “red” , “Blue Sea” , “41213*(I+3)”

3.6.5 Escape Sequence

A character constant is usually one character enclosed in single quotes. However, certain special characters – including new line character, can’t be written in this way,

because they are invisible (non-printing) or because they can't be entered by the keyboard. For this kind of characters C provides a special notation, the escape sequence. **Table 3.12** gives the complete set of these character escape sequences.

Name	Escape Sequence	Name	Escape Sequence
Alert (bell)	\a	Backslash	\\
Backspace	\b	Question mark	\?
Form Feed	\f	Single quote	\'
New line	\n	Double quote	\"
Carriage Return	\r	Horizontal tab	\t
Vertical tab	\v		

Table 3.12

Check Your Progress 3.4

Give the decimal value of the following integer constants.

077 0x77 0xABC

Check Your Progress 3.5

Classify the examples into Integer, floating point, Character and String constants.

'A', 0147, 0xEFH, 077.7, "A", 26.4, "EFH", '\r', abc

Check Your Progress 3.6

Which of the following are not legal constants in C? Classify each legal constant as either integer or floating point.

010E2 32.1E+5 0790 100_000 3.978e-2

3.7 Data Input Output Function

C is a function based language that uses two types of functions : Library functions and User defined functions. Library functions are inbuilt functions in C programming and the definition of the functions are present in their respective pre-processor directives **(Section 2.3.2)**. User defined function is created by the user which is out of scope of this course. There are two library functions those are used very frequently for getting input data from the user and producing the output using

input data. These functions are

- printf
- scanf

3.7.1 The printf Function

The printf function is designed to display the contents of the string, known as format string. The values are inserted at specified points of the format string. In general terms, the printf function is written as

printf(format string, arg_1, arg_2, arg_3 . . ., arg_n)

The values displayed can be constant, variable or any other expressions). The format string may contain ordinary as well as conversion specifications, which begin with % character. A conversion specification is a place holder representing a value to be filled during printing. The information that follows the % character specifies how the value is converted from its internal form (binary) to printed form. For example, the conversion specification %d means printf is to convert the binary into a string of decimal digits. The ordinary characters in format string are printed exactly as they appear in the string. The value to be printed is either constant, variable or expression mentioned in the list arg_1, arg_2, arg_3, . . ., arg_n. For example, see the program and its output in **Table 3.13**.

The program code in Table 3.13 has two integers and two float variables, both are initialized to their respective values at the time of declaration. The format string in the printf statement (line no 6), contains four conversion specifications (two %d and two %f), four escape sequences (\t) and other ordinary characters. The argument list contains four variables (two integers and two floats).

Line No	Code
1	`#include<stdio.h>`
2	`int main()`
3	`{`
4	`    int i=10,j=20;`
5	`    float a=18.2f,b=205.0f;`
6	`    printf("i=%d\tj=%d\ta=%f\tand\tb=%f ",i,j,a,b);`
7	`}`
Output	
i=10 j=20 a=18.200000 and b=205.000000	

Table 3.13

Figure 3.4 describes how the printf generates the output. In the output, the conversion specifications are replaced by the values according to the order in the argument list. \t is replaced by tab space (eight spaces together) and ordinary characters remain as it is (dotted arrow in **Figure 3.4**) in the output.

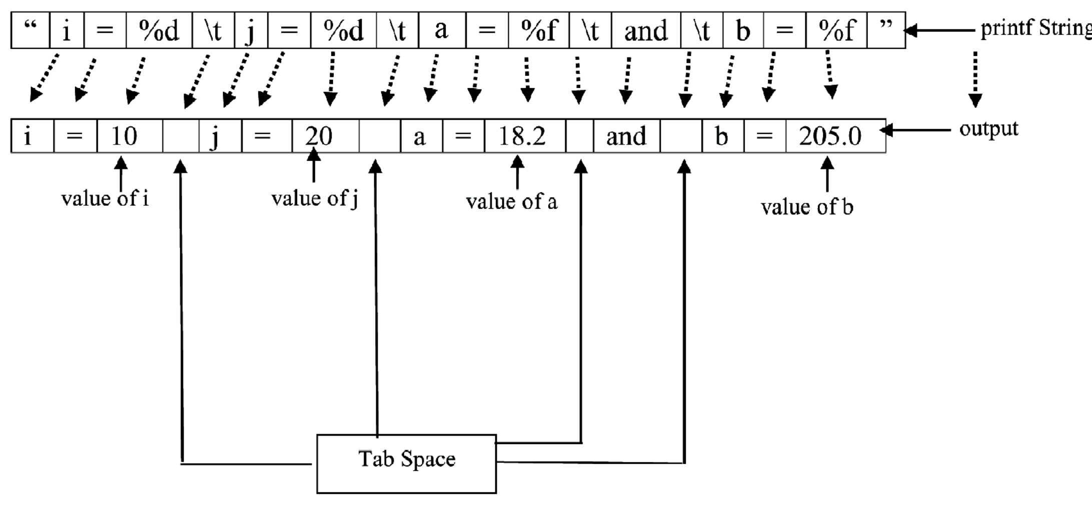


Figure 3.4

Common mistakes made by programmer while using printf function :

Case 1 :

C compiler are not required to check number of format specifications and the number of output items. Therefore, it is the responsibility of the programmer to check this numbers while writing the code. For example, suppose we add following printf statement which contain more conversion specifications than output items in the code in **Table 3.13 :**

```
printf("i=%d\tj=%d", i);
```

printf will print the value of the variable i correctly, then print a second (meaningless) integer value instead of throwing an error. **(Table 3.14)**

Case 2 :

If a printf function contain less conversion specifications than the output items, then the extra value will not be printed. **(Table 3.14)**.

Case 3 :

Compilers aren't required to check whether the type of the conversion specification is appropriate for the type of item being printed. If mismatch happens then the printf function, simply produce meaningless output.

Case	Code	Output	Comments
1	printf("i=%d\tj=%d", i);	i=10 j=6356864	j has garbage value.
2	printf("i=%d", i,j);	i=10	j value not printed.
3	printf("i=%d",a);	i=1073741824	Value of a is float and stored in the memory in IEEE format. Display the result in integer format will produce meaningless value.

Table 3.14

Conversion Specification

Conversion specification is very important to produce the output as desired. The general form of conversion specification is :

%m.pX or %-m.pX Where m and p are integer constants and X is a letter.

Here m, known as minimum field width, specifies minimum number of character to print and p, known as precession has different meaning for different choice of letter X. For example, if the conversion specification is %10.4f, then m = 10 and p = 4 and X = f. Both m and p are optional. If p is omitted, the period(.) that separates m and p is also omitted. **Table 3.15** shows commonly used letter for X known as conversion character and the corresponding meaning of precession p.

Conversion Letter(X)	Meaning
d	Data item is displayed as an integer in decimal form. p indicates the minimum number of digits to display (extra zeros are added at the beginning of the number). If p is omitted, then the value is assumed to be 1.
e	Data item is displayed as a floating-point value with an exponent. p indicates how many digits should appear after decimal point. If p is 0, the decimal point is not displayed.
f	Data item is displayed as a floating-point value without an exponent (fixed decimal format). p has the same meaning as for the letter e.
g	Data item is displayed as a floating-point value using either e-type or f-type conversion, depending on value. p indicates the maximum number of significant digit (not the digits after decimal point), Trailing zeros and trailing decimal point will not be displayed.

Table 3.15

Example 3.1

Determine the output of the C program written in **Table 3.16** with proper reason.

```
#include<stdio.h>
int main()
{
    int i=10;
    float a=18.23f;
    printf("|%d|%6d|%-6d|%6.3d|\n",i,i,i,i);
    printf("|%8.3f|%8.3e|\n",a,a);
    return 0;
}
```

Table 3.16

The output is given below and the justification is in **Table 3.17**

```
|10|    10|10    |   010|
|  18.230|1.823e+001|
```

Conversion Specification	Value of m, p and X	Output	Justification
|%d|	m = 1, p = 1, X = d	|10|	Simply print 10
|%6d|	m = + 6, p = 1, X = d	| 10|	m=+6 means the minimum field width of the output is 6 character and the output is right justified. Since the actual value is of two digits, output is preceded by four blanks.
|%-6d|	m = –6, p = 1, X = d	|10 |	m=-6 means the minimum field width of the output is 6 character and the output is left justified. Since the actual value is of two digits, output is followed by four blanks.
|%6.3d|	m = 6, p = 3, X = d	| 010|	m=+6 means the minimum field width of the output is 6 character and the output is right justified. Again p=3 means the output must display at least three digits.

Conversion Specification	Value of m, p and X	Output	Justification
			Since actual value is of two digits, output will be preceded by three blanks and one zero.
|%8.3f|	m = 8, p = 3, X = f	| 18.230|	m=+8 means the minimum field width of the output is 8 character and the output is right justified. Again p=3 means the output must display at least three digits after decimal point. Since actual value has two digits after decimal point, one zero must be appended. Now the output becomes 18.230 which has the length 6(including decimal point). Two more blanks need to be added at the beginning to make the minimum length 8.
|%8.3e|	m = 8, p = 3, X = e	**|1.823e+001|**	m=+8 means the minimum field width of the output is 8 character and the output is right justified and in exponent form. Again p=3 means the output must display at least three digits after decimal point. Here zero is not needed because the output is 10-character long.

Table 3.17

Check Your Progress 3.7

Consider two variables initialized in following way in a C program :

```
int i = 12345;
float x = 345.678;
```

What will be the output of the following statements?

1. printf ("%3d %5d %8d\n\n", i, i, i);
2. printf("%3f %10f %13f\n\n\n, x, x, x);
3. printf("%3e %13e %16e", x, x, x);

Check Your Progress 3.8

Consider a variable initialized in following way in a C program :

float x = 123.45;

What will be the output of the following statements?

1. printf("%7f %7.3f %7.lf\n\n", x, x, x);
2. printf ("%12e %12.5e %12.3@", x, x, x);

Justify with proper reason.

Check Your Progress 3.9

What output do the following calls of printf produce?

1. printf("%6d,%4d", 86,1040);
2. printf("%12.5e", 30.253);
3. printf("%.4f", 83.162);

Check Your Progress 3.10

Write printf function to display a float variable x in following formats.

1. Exponential notation; left-justified in a field of size 8; one digit after decimal point.
2. Exponential notation; right-justified in a field of size 10; six digits after decimal point.
3. Fixed decimal notation; left-justified in a field of size 8; three digits after decimal point.
4. Fixed decimal notation; right-justified in a field of size 6; no digit after decimal point.

3.7.2 The scanf Function

scanf is another useful library function which reads input according to a particular format. Like the printf, scanf also has a format string which contain both ordinary characters and conversion specifications. The conversion specifications with scanf are same as those used in printf. **Figure 3.5** shows scanf is doing reverse to what printf does. The printf function send the value of i (5) to the output device (monitor) while the scanf function accept the value given by user (assuming user types 5) and store it to the memory variable (i).

In general terms, the scanf function is written as

scanf(format string, arg_1, arg_2, arg_3, . . . arg_n)

Consider a program in **Table 3.18** where programmer is asked to enter two integer and two floating point input. Suppose, programmer enters following input line :

1∘ –20∘.3∘ – 4.0*e*3*

where ◦ and ∗ represent the white space and new line character respectively. scanf will read the line and convert its characters to numbers they represent and then assign 1, –20, .3, and –4000.0 to i, j, a, b, respectively.

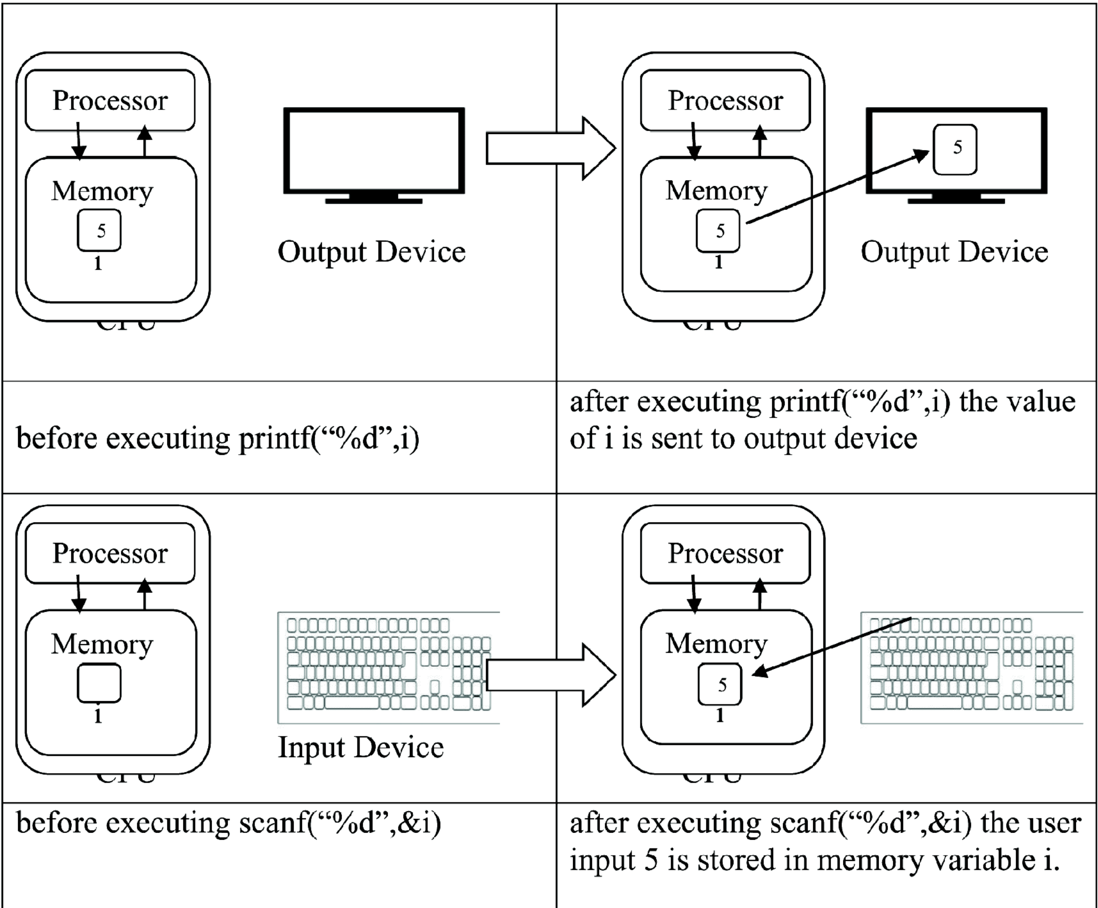


Figure 3.5

```
#include<stdio.h>
int main()
{
    int i,j;
    float a,b;
    scanf("%d%d%f%f",&i,&j,&a,&b);
    return 0;
}
```

Table 3.18

The programmer must check that the number and types of format specifications match with number and types of the variables used in the program. Another important point to be noted is that the symbol "&" needs to be placed in front of a variable while using scanf. "&" denotes the address of a variable where the input value will be stored. This is analogous to the fact that when a letter or parcel is delivered, the post office always use the address of the receiver instead of the name. Similarly, when a value is coming from input device (sender), it needs the address of the variable (receiver) instead of the name.

How scanf works

scanf is actually a pattern matching function which tries to match group of characters to conversion specifications. scanf begins reading the input string starting from left. For each conversion specification in the format string, scanf try to locate an item that is appropriate to input data, skipping white space (space, tab, new line etc.), if necessary and stop when it encounters a character that possibly can not belong to the item. If the data is read successfully then it continues to read the remaining input string otherwise return immediately without looking at the rest of the string.

Consider the same program of **Table 3.18** and assume that the programmer enters the following as input line :

```
  1
–20   .3
   –4.0e3
```

which is basically in our symbolic notation ◦ ◦1*–20 ◦ ◦ ◦.3* ◦ ◦ ◦–4.0e3* where ◦ and * represent the white space and new line character respectively. scanf will be executed in following steps **(Figure 3.6)** :

1. Skips the leading 2 spaces.
2. The first conversion specification is %d. The first non-blank character 1 in the input line can be treated as integer so scanf continue reading next character <new line>. Since <new line> can't be part of integer, it returns 1 to the variable i and puts back the <new line> character in the input string for further scanning process.
3. Next conversion specification is %d again. Now scanf repeatedly skips the white space characters (<new line>, <space>) until it reaches to non-space character minus (–). Then it reads –20 that can be treated as next integer j. After that, scanf encounters a <space> which can't be part of the integer. As a result, it returns –20 to the variable j and puts back <space> in the input string.
4. Similar to the third step variable a becomes 0.3.
5. b becomes 4000.0 and completes the scan process successfully.

Stape	Intermediate state of input string	State of memory variables
1	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[ ] [ ] [ ] [ ] i j a b
	Skips the <space characters>	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[ ] [ ] [ ] [ ] i j a b
2	Store 1 into variable i	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[1] [ ] [ ] [ ] i j a b
3	Skips the <new line character>	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[1] [ ] [ ] [ ] i j a b
	Store -20 into variable j	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[1] [-20] [ ] [ ] i j a b
4	Skips the <space characters>	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[1] [-20] [ ] [ ] i j a b
	Store .3 into variable a	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[1] [-20] [.3] [ ] i j a b
5	Skips the <new line and space characters>	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[1] [-20] [.3] [ ] i j a b
	Store -4000.0 into variable b	
	∘∘1∗–20∘∘∘.3∗∘∘∘–4.0e3∗	[1] [-20] [.3] [-4000.0] i j a b

Figure 3.6

Example 3.2

Explain how scanf of the program in **Table 3.19** will read the data for following input string 4, ∘ ∘ 5 *

```
#include<stdio.h>
int main()
{
int i,j;
scanf("%d,%d",&i,&j);
return 0;
}
```

Table 3.19

Solution :

The scanf has two conversion specification :

1. The first conversion specification(%d) : The first character of the input string is 4; since integer can begin with 4, scanf then reads the next character comma (,) which can't be the part of the integer so scanf returns 4 to the variable i and put the comma (,) back. Since the character comma (,) in input string matches with comma (,) in format string, scanf successfully continue reading the remaining input string.
2. The second conversion specification (%d) : The two space characters are skipped and 5 is read and stored in the variable j.

Therefore, the scanf successfully read the input data, which makes i=4 and j=5.

Check Your Progress 3.11

Explain how scanf of the program in **Table 3.19** will read the data for following input string

4 ∘ ∘ ∘ 5 *

The first character 4 is read and stored in i. As the input string doesn't have any matching comma (,) as in format string, the scanf stops reading the remaining input string and 5 is not stored in variable j. Therefore, j retains its old value.

Check Your Progress 3.12

Write a program that accepts a date from the user in the form of mm/dd/yyyy and then display it in the form of yyyymmdd :

Sample Output :

Enter a date (mm/dd/yyyy) : 2/17/2012

You entered date 20120217.

The program is given in **Table 3.20**.

```
#include<stdio.h>
int main()
{
    int i,j,k;
    printf("enter date mm/dd/yyyy:");
    scanf("%d/%d/%d",&i,&j,&k);
    printf("you entered date %d%.2d%.2d\n",k,i,j);
    return 0;
}
```

Table 3.20

Check Your Progress 3.13

Suppose scanf is used in following way in a program where i, k are integers and j is a float.

scanf("%d%f%d",&i,&j,&k)

If the programmer enters

10.3∘3∘5*

where ∘ and * represent the white space and new line character respectively, then what are the values of i, j and k after the scanf call? Justify the answer.

Check Your Progress 3.14

Suppose scanf is used in following way in a program where i, k are floats and j is an integer.

scanf("%f%d%f",&i,&j,&k)

If the programmer enters

12.3∘45.6∘789*

where ∘ and * represent the white space and new line character respectively, then what are the values of i, j and k after the scanf call? Justify the answer.

Check Your Progress 3.15

Write a program that formats product information entered by the user. A session with the program should look like :

Enete item number : 234
Enter unit price : 26.5
Enter purchase date (mm/dd/yyyy) : 3/8/2019

Item	Unit Price	Purchase Date
234	Rs. 26.5	03/08/2019

The item Number and date should be left justified. Unit price should be right justified. Allow Rs. amounts up to Rs. 9999.99.

Check Your Progress 3.16

Write a program that prompts the user to enter a telephone number in the form (xxx) xxx-xxxx and then display the number in the form of xxx.xxx.xxxx.

Sample Output :

Enter phone number [(xxx) xxx-xxxx] : (899) 817-6122
You entered 899.817.6122

Check Your Progress 3.17

For each of the following pairs of scanf format strings, indicate whether or not the two strings are equivalent. If they are not show how they can be distinguished?

1. "%d" versus " %d " 2. "%f" versus " %f " 3. "%f,%f" versus "%f, %f"

3.8 Summary

Several important topics of C language have been discussed in this chapter. Character set which includes alphabets, numeric characters, special characters are used as building block of identifier, keyword, variable, constants. Three basic data types have been discussed — int, char, float. Some qualifiers are used as prefixes to data types like signed, unsigned, short, and long. The constants are the fixed values and may be either integer or floating point or character or string type. The learner can now use printf and scanf function as a tool for data input/output. They can also write some small program using all these features of C language.

3.9 Reference and Further Reading

1. The C Programming Language, Kernighan & Ritchie, PHI Publication, 2011.
2. Programming with C, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
3. The C Complete Reference, Fourth Editon, Herbert Schildt, Tata McGraw Hill, 2002.
4. C Programming : A Modern Approach, Second Edition, K.N. King, W. W. Norton & Company, 2008.
5. What Every Computer Scientist Should Know About Floating-Point Arithmetic, David Goldberg (ACM Computing Surveys, March, 1991).

Unit - 4 ❑ Expressions and Operators

Structure

4.0 Introduction

4.1 Objectives

4.2 Operators

4.2.1 Arithmetic Operators

4.2.2 Assignment Operators

4.2.3 Increment and Decrement Operators

4.2.4 Relational Operators

4.2.5 Logical Operators

4.3 Operator Precedence and Associativity

4.4 Summary

4.5 References and Further Reading

4.0 Introduction

In the previous unit, variables, constants, datatypes and other building blocks of C programming were discussed and the way to declare them in C programming was also explained. In fact, the variables and constants are the simplest form of expressions. The next step is to learn more complicated expressions which is a sequence of operators and operands that does one or a combination of the following :

- Specifies the computation of a value
- Designates an object or function
- Generates side effects.

An operator performs an operation (evaluation) on one or more operands. An operand is a subexpression on which an operator acts.

This unit focuses on different types of operators available in C including the syntax and use of each operator and how they are used in C. A computer is different from calculator in a sense that it can solve logical expressions also. Therefore, apart from arithmetic operators, C also contains logical operators. Hence, logical expressions are also discussed in this unit.

4.1 Objectives

After going through this unit the learner should be able to :

- Write and evaluate arithmetic expressions;
- Express and evaluate relational expressions;
- Write and evaluate logical expressions;
- Write and compute complex expressions (containing arithmetic, relational and logical operators), and
- Check simple conditions using conditional operators.

4.2 Operators

C has several operators like arithmetic, assignment, increment, decrement, relational, logical etc.

4.2.1 Arithmetic Operators

The arithmetic operators perform addition, subtraction, multiplication and division. These operators are of two types based on the number of operands :

- Unary : need one operand
- Binary : need two operand

The binary operators are again of two types based on the nature of the operation :

- Additive : addition (+), subtraction (–)
- Multiplicative : multiplication (*), division (/), remainder (%)

The **Table 4.1** shows these different types of operators. The unary operators are used just to specify that the numeric constant is positive or negative.

Unary	**Binary**	
	Additive	Multiplicative
+ unary plus – unary minus	+ addition – subtraction	* multiplication / division % remainder

Table 4.1

For example :

```
i=+1;    /* + used as a unary operator here*/
j=–2;    /* – used as a unary operator here*/
```

The binary operators have their usual meaning except the remainder (%) operator. In C, i%j is the remainder when i is divided by j. Here the operands i and j are both integers. For example, the value of 8%6 is 2. In all other binary operators, the operand may be either integer or float or float and integer both. When integer and float operands are mixed then the result has always the type float. For example, 8+4.2f results is 12.2, and 8/4.2 gives 1.904762. Following points needs to be noted while using division (/) and remainder (%) operators.

- ➢ When both the operands of division (/) operator are integers, the / operator truncates the result by dropping the fractional part. Thus the value of 1/4 is 0 not 0.25.
- ➢ The remainder (%) operator needs two integer operands. Compiler will raise an error if both of the operands are not integers **(Table 4.2)**.

Code	Output

#include<stdio.h>
int main()
{
printf("result is %f",8.2%6);
return 0;
}
	Message
	=== Build file: "no target" in "no project" (compiler: unk...
	In function 'main':
	error: invalid operands to binary % (have 'double' and 'int')
	=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 9...

Table 4.2

- ➢ Zero can't be used as a right operand of either remainder (%) or division (/)

Example 4.1

Show that, remainder (%) operator can be implemented using other arithmetic operators.

Solution :

Consider two integers x and y. The expression x%y produces the remainder when x is divided by y. The same operation can also be achieved by x – (x/y) * y. **Table 4.3** shows this by considering x=7 and y=5.

```
#include<stdio.h>
int main()
{
    int x=7,y=5;
    printf("Remainder using operator is %d\n",x%y);
    printf("Remainder using formula is %d\n",x-(x/y)*y);
    return 0;
}
```

OUTPUT

Remainder using operator is 2
Remainder using formula is 2

Table 4.3

4.2.2 Assignment Operators

Normally we use assignmment (=) operator to store value of some constant or variable or an expression. For example :

```
i = 10;
j = i;
k = i + 10*j;
```

For any assignment statement i=j, if the type of i and j are not same then the value of j is converted to the type of i as assignment takes place. For example,

```
int i;
float j;
i = 14.8f      /* i is now 14 */
j = 14         /* j is now 14.0 */
```

Several assignment operators can be chained together. For example :

```
i = j = k = 5;
```

The assignment operator (=) is right associative meaning, if any expression is having multiple = operators, the execution order of the operators is right to left. Associativity of operators will be explained in detail in section 4.3. Therefore, the above assignment statement is equivalent to

```
i = (j = (k = 5));
```

which means 5 is assigned to k first, then to j, finally to i. One important point that needs to be noted is that the left operand of = operator can't be a constant or an expression and it must be a variable. In C, every variable has two attributes :

➢ Address of the variable
➢ Value contained in that variable.

These two parts have been shown in **Figure 4.1**. Any variable in assignment statement is used either as left operand or right operand. For example :

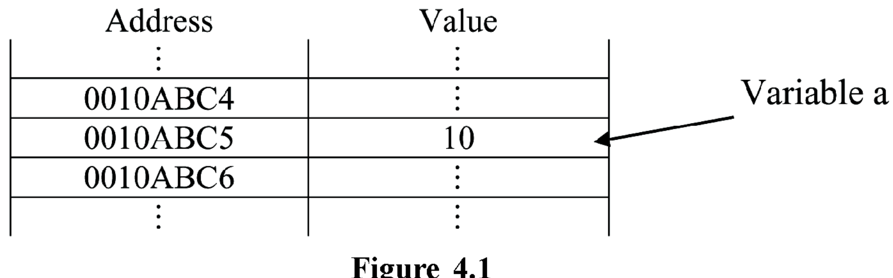


Figure 4.1

```
int a=10;
int b;
b=a;      /* variable a used as a right operand of = */
a=5;      /* variable a used as a left operand of = */
```

When variable a is used as a right operand, the value attribute of a is considered. This value is commonly known as rvalue. Therefore, after executing b=a, b takes the value 10 in the example. When variable a is used as a left operand, the address attribute of variable a is considered. Using the statement, a=5, the compiler searches the address of variable a and copy 5 as a value attribute of variable a. This address or reference is known as lvalue of the variable a. The point need to be noted that, while using assignment operator if any constant or expression (rvalue) is used as a left operand the compiler raises error (lvalue required) and stops the program execution. **Table 4.4** shows an example of this type of error.

Code	Output

#include<stdio.h>
int main()
{
int a=5;
5=a;
return 0;
}
	Message
	=== Build file: "no target" in "no project" (compiler:
	In function 'main':
	error: lvalue required as left operand of assignment
	=== Build failed: 1 error(s), 0 warning(s) (0 minute(s)

Table 4.4

Compound Assignment

Compound assignment shortens the assignment statements. For example, an assignment statement of the form i = i + 2 can be written as i += 2. The following list gives few more example :

Assignment	Compound assignment
i = i – 2	i – = 2
i = i * 2	i * = 2
i = i / 2	i / = 2
i = i % 2	i % = 2

4.2.3 Increment and Decrement Operators

In C, incrementing (adding 1) and decrementing (subtracting 1) a variable are two most common operations. The operation can be written as :

```
i = i + 1;
j = j – 1;
```

In short, these operations are also written as ++ (two consecutive + sign) and -- (two consecutive - sign). These increment and decrement operations are of two types based on the position of the sign (++/--).

- Prefix (Pre-increment/Pre-decrement) : for example, ++i and --j
- Postfix (Post-increment/Post-decrement) : for example, i++ and j – —

Before going into detail of ++ or -- operator, an important feature of C language needs to be highlighted. In Mathematics, operator can't modify the value of the operand. For example, i + j doesn't modify either i or j, it simply computes the result of adding i and j. Assignment operator in C is one of few operators which can modify its operand. This phenomenon is known as side effect of the operator. The statement i = 10 can be considered as an expression since = is an operator in C. Every expression produces a value. In this case, the value of the expression i = 10 produces the value 10 which can be ignored and then it modifies the value of i to 10 as a side effect. Ignoring the expression's value is of no loss, since the primary reason for writing the statement was to modify i.

Now at a first look, it may be assumed that both postfix and prefix increment operators do the same. In fact, they behave similarly in many places. For example,

see the program and their output in **Table 4.5**. Both expressions ++i and i++ increment the value of i from 5 to 6 after completion. Here ++i, increment i at first from 5 to 6 as a side effect and then produce the value 6. On the other hand, operator i++ first produce the value 5 (original value of i) and then increment i from 5 to 6 as a side effect. Since value of the expressions in both cases are ignored, it shows same result because of the same side effect.

Operator type	Code	Output

Prefix	#include<stdio.h>
	int main()
	{
	   int i=5;
	   ++i;
	   printf("i=%d",i);
	   return 0;
	}
		i=6

Postfix	#include<stdio.h>
	int main()
	{
	   int i=5;
	   i++;
	   printf("i=%d",i);
	   return 0;
	}
		i=6

Table 4.5

But if the operators are used little differently, then the value of i and j are different. For example, consider the output of the code in **Table 4.6**. In case 1, the first printf contains ++i and in case 2, the first printf contain i++ as arguments. Since printf function prints the value of the expression, the printf prints 6 (value of ++i) in case 1 while printf prints 5 (value of i++) in case 2. Once the first printf statement is complete i is incremented as a side effect in both the cases. Therefore, second printf prints 6 in both cases.

Case No	Operator type	Code	Output

1	Prefix	#include<stdio.h>
		int main()
		{
		int i=5;
		printf("i=%d\n",++i);
		printf("i=%d\n",i);
		return 0;
		}
			i=6
			i=6

2	Postfix	#include<stdio.h>
		int main()
		{
		int i=5;
		printf("i=%d\n",i++);
		printf("i=%d\n",i);
		return 0;
		}
			i=5
			i=6

Table 4.6

Example 4.2

Determine the output of the following code in **Table 4.7**.

Line#	Code
1	#include<stdio.h>
2	int main()
3	{
4	int i=5,j=6,k;
5	k = ++ i + j ++;
6	printf("i=%d,j=%d,k=%d\n",i,j,k);
7	return 0;
8	}

Table 4.7

To determine the output, line no 5 (containing prefix and postfix ++) needs to be analyzed. Here the first operand in the right hand side of assignment is (++i) which means it produces i+1 then increments i. The second operand is (j++) which means it produces j and then increments j. Therefore, the line can be broken into

following statements :

```
i = i + 1;
k = i + j;
j = j + 1;
```

Thus the output will be i = 6, j = 7, k = 12.

4.2.4 Relational Operators

C has various relation operators which are mainly used to compare integers and floating point numbers, with operands of mixed type allowed. The list of relational operator is given in **Table 4.8**. These operators produce 0 (false) or 1 (true) when used in expressions. For example, the value of 8 >10 is 0 and 10 >8 is 1.

Symbol	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
= =	Equal to
!=	Not equal to

Table 4.8

4.2.5 Logical Operators

Logical operators in C are used to evaluate expressions which may be true or false. C has three logical operators which are presented in the **Table 4.9**.

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td>!</td><td>Logical negation</td></tr>
<tr><td>&&</td><td>Logical and</td></tr>
<tr><td>||</td><td>Logical or</td></tr>
</table>

Table 4.9

The ! operator is unary, while ‘&&’ and ‘||’ are binary. Logical operator produces either 1 (true) or 0 (false) as their result. Normally, the operands of any logical operator are expected to be either 0 or 1. If any operand has the non-zero value, then the operator will treat the value as true (1) and any zero value as false. The logical operator behaves as per the rules listed in the truth tables in **Table 4.10**.

X	Y	x&&y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x||y
0	0	0
0	1	1
1	0	1
1	1	1

x	!x
0	1
1	0

Table 4.10

4.3 Operator Precedence and Associativity

C expressions contain more than one operator in most of the times. It may create confusion which operator should operate first. For example, consider the following expression containing two operators + and * along with three operands :

2 + 3 * 4

The above expression can be evaluated in different ways :

- Add 2 and 3, then multiply the result by 4 and finally it becomes 20 **(Figure 4.2a)**
- Multiply 3 and 4, then add 2 to the result and finally it become 14 **(Figure 4.2b)**

One way to solve this problem is to add parenthesis, writing either (2+3)*4 or 2+(3*4). Otherwise, to use operator precedence rules to resolve this potential ambiguity. For example, * operator has higher precedence than + operator, therefore, the expression 2+3*4 is evaluated as depicted in Figure 4.2b which is equivalent to 2+(3*4) and the result is 14. Operator precedence rules alone aren't enough when an expression contains two or more operators at the same precedence. In this situation, associativity of the operators is considered. An operator is left-associative if it groups from left to right. For example, binary arithmetic operators (*, /, %, + and –) are all left associative. Therefore, the expressions :

i – j – k is equivalent to (i – j) – k and
i * j / k is equivalent to (i * j) / k.

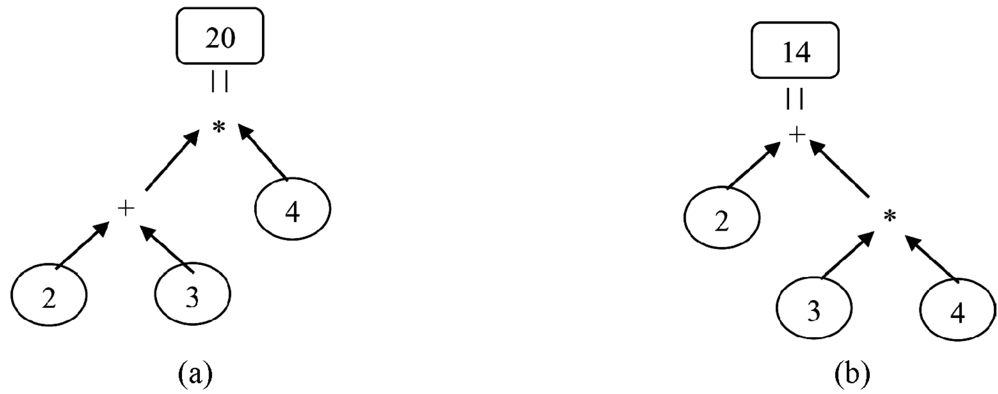


Figure 4.2

An operator is right-associative if it groups from right to left. For example, the unary arithmetic operators (+ and –), are both right–associative. Therefore, the expression

– + i is equivalent to – (+ i).

Table 4.11 gives the precedence and associativity of operators those are mostly used in C.

Precedence	Name	Symbol(s)	Associativity
1	Parenthesis	()	Left
2	increment(postfix) decrement(postfix)	++ – –	Left
3	increment(prefix) decrement(prefix) unary Plus unary Minus Logical Not	++ – – + – !	Right to left
4	Multiplicative	*, /, %	Left to right
5	Additive	+, –	Left
6	Relational	<, <=, >, >=	Left
7	Equality	= = , ! =	Left
8	logical and	&&	Left
9	logical or	| |	Left
10	Assignment	=, *=, /=, %=, +=	Right

Table 4.11

Example 4.3

Show the output produced by the following program segments. Assume i, j and k are integer variables.

I.
```
i=7, j=8, k=9;
printf("%d", (i + 10) % k / j);
```

II.
```
i=1, j=2, k=3;
i *= j *= k;
printf("%d %d %d", i, j, k);
```

III.
```
i = 7, j;
j = 6 + (i = 2.5);
printf("%d %d", i, j);
```

Solution (I) :

Replacing the variables by their values the expression (i + 10) % k / j becomes (7+10) % 9 / 8.

(7+10) % 9 / 8 = 17 % 9 / 8 [() has the highest precedence in Table 4.10)]
= (17 % 9) / 8 [% and / has same precedence and left associative, so evaluation is from left to right]
= 8 / 8 = 1

Solution (II) :

i *= j *= k expression has multiple *= operator. Since this operator is right associative **(Table 4.10)**, the evaluation of the expression starts from right. Therefore, i *= j *= k expression can be equivalently written as :

```
j *= k   ⟹   j = j * k
i *= j        i = i * j
```

we get,

j = j * k
= 2 * 3 [replacing values of j and k]
= 6

Similarly, i = i * j
=1 * 6 = 6

Therefore, the final values are i = 6, j = 6, k = 3.

Solution (III) :

The statement j = 6 + (i = 2.5); can be split into two statements as = is right associate.

```
i = 2.5;
j = 6 + i;
```

being an integer i can store integer. So the new value of i = 2 and as a result j = 6 + 2 = 8.

Example 4.4

Supply parenthesis to show how C compiler would interpret following expressions based on precedence of operator given in **Table 4.10**.

I. a * b – c * d + e II. a / b % c / d

Solution (I) :

The expression has four operators. The * operators has the highest precedence in the expression but it appears twice. Since * operator is left associative so the * which comes first from the left side is evaluated first. Then the second * is evaluated. The operands between * can be grouped as follows :

(a * b) – (c * d) + e

So the expression becomes $R_1 - R_2$ + e where R_1 = (a * b) and R_2 = (c * d).

We can see from **Table 4.10,** + and – operators have same precedence. Since they are left associative the grouping of operands is done from left to right. So, the equivalent expression is

$(R_1 - R_2)$ + e

⇒ ((a * b) – (c * d)) + e

Solution (II) :

All the operators are in same level of precedence and they are left associative. So the expression can be written as :

((a / b) % c) / d

Check Your Progress 4.1

Show the output produced by the following program segments. Assume i, j and k are integer variables.

I.
```
i=1, j=2, k=3;
printf("%d", (i + 5) % ( j + 2) / k);
```

II.
```
i=7, j=8;
i *= j + 1;
printf("%d %d", i , j);
```

III.
```
i=1, j=2, k=3;
i *= j *= k;
printf("%d %d %d", i, j, k);
```

IV.
```
i=6, j;
j = i += i;
printf("%d %d", i, j);
```

```
V. i = 2, j = 8;
   j = (i = 6) + (j = 3);
   printf("%d %d", i, j);
```

Check Your Progress 4.2

Only one of the expressions ++i and i++ is exactly same as (i += 1). Which is it? Justify your answer.

Check Your Progress 4.3

Supply parenthesis to show how C compiler would interpret following expressions based on precedence of operator given in **Table 4.10**.

```
I. – a – b + c – + d
II. a * – b / c – d
III. a = b += c++ – d + --e / f
```

Example 4.5

Show the output produced by the following program segments. Assume i, j and k are integer variables.

```
I. i = 10, j;
   j = 3 * i-- + 2;
   printf("%d %d", i, j);
```

```
II. i =10, j = 5;
    printf("%d\n", i++ - ++j);
    printf("%d %d", i, j);
```

```
III. i = 5, j=10, k=1;
     printf("%d", k>i< j);
```

```
IV. i = 2, j = 1;
    printf("%d", !!i + !j);
```

Solution (I) :

In line 2, the expression is j = 3 * i-- + 2. Let us place parenthesis based on precedence and associativity in the expression. Post increment has highest precedence. So the expression becomes :

```
   j = 3 * (i--) + 2
⇒ j = (3 * (i--)) + 2      [ * has higher precedence than + and = operators]
⇒ j = ((3 * (i--)) + 2)    [ + has higher precedence than = operator]
```

Here the post decrement operator produces the value of i first and then decrements i to i-1. Therefore, the above expression can be split into following expressions.

j = ((3 * (i--)) + 2) ⇒

```
j = (3 * i) + 2;
i = i – 1;
```

Let us evaluate the expression by placing actual value of the operand :

j = (3 * 10) + 2 = 32

i = 10 – 1 = 9

Therefore, the printf displays 9 and 32.

Solution (II) :

In line 2 printf statement has expression i++ – ++j as an argument. After placing parenthesis similar to solution (I), we get the equivalent expression as :

(i++) – (++j)

Here the post increment (i++) produces the value of i first and then increments the value of i to i+1 . The pre increment (++j) produces the value of (j+1) and then increments the value of j to j+1.

Therefore, the above expression can be split into following three consecutive expressions.

(i++) – (++j) ⇒

```
j = j + 1;
i – j;      [argument of printf statement]
i = i + 1;
```

Therefore, at first j becomes 6. Then the printf displays (10 – 6) = 4 and finally i becomes 11. The second printf displays value of i and j i.e. 11 and 6 respectively.

Solution (III) :

The argument in printf statement is the expression $k>i<j$ contains all relational operators which are left associative. So the equivalent expression is

$(k > i) < j$

⇒ $0<j$ [k = 1 and i = 5, therefore, k > i is false and statement produces 0]

⇒ 1 [j = 10, therefore, $0<j$ is true and statement produces 1]

The printf finally displays 1.

Solution (IV) :

The expression !!i + !j has logical not (!) operator and arithmetic operator +. First parenthesize the expression based on the precedence and associativity given in **Table 4.10**.

!!i + !j

⇒ (! (!i)) + (!j) [! has higher precedence and is right associative]

Since i and j both have non-zero value, they both are treated as true. (refer section 4.3.5). Therefore, the expression becomes,

(! (! TRUE)) + (! TRUE)
= (! FALSE) + FALSE
= TRUE + FALSE
= 1 + 0 [TRUE = 1 and FALSE = 0]
= 1

Therefore, the printf finally displays 1.

Check Your Progress 4.4

Write a program that asks the user to enter a two-digit number, then print the number with its digits reversed.

Sample Output :
Enter a two-digit number : **34**
The reversal is 43.

Check Your Progress 4.5

Show the output produced by the following program segments. Assume i, j and k are integer variables.

I.
```
i = 1;
printf("%d\n", i++ - 1);
printf("%d", i);
```

II.
```
i =7, j = 8;
printf("%d\n", i++ - --j);
printf("%d %d", i, j);
```

III.
```
i =7, j = 8, k=5;
printf("%d\n", i++ - j++ + --k);
printf("%d %d %d", i, j,k);
```

IV.
```
i = 5, j;
j = ++i * 3 - 2);
printf("%d %d", i,j);
```

V.
```
i = 7, j;
j = 3 * i-- + 2);
printf("%d %d", i,j);
```

VI.
```
i = 7, j;
j = 3 + --i * 2);
printf("%d %d", i,j);
```

VII.
```
i = 5, j = 0, k = -5;
printf("%d", i && j || k);
```

VIII.
```
i = 1, j = 2, k = 3;
printf("%d", i < j || k);
```

4.4 Summary

In this unit, we discussed about the different types of operators, namely arithmetic, relational, logical, increment-decrement present in C. These operators

have a pivotal role in developing complex program segments. In the following units, more usage of these operators along with few new operators will be discussed in the context of constructs like control statements, arrays etc. Since Logical operators are used further in all types of looping constructs and if/else construct (in the next unit), it is expected that the learner should understand thoroughly the usage of these operators.

4.5 References and Further Reading

1. The C Programming Language, Kernighan & Ritchie, PHI Publication, 2011.
2. Programming with C, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
3. The C Complete Reference, Fourth Editon, Herbert Schildt, Tata McGraw Hill, 2002.
4. C Programming : A Modern Approach, Second Edition, K.N. King, W. W. Norton & Company, 2008.

Unit - 5 ❑ Decision and Loop Control Statements

Structure

5.0 Introduction

5.1 Objectives

5.2 Select Statements

5.2.1 The if Statement

5.2.2 The switch Statement

5.3 Iterative Statements

5.3.1 while Loop

5.3.2 do-while Loop

5.3.3 for Loop

5.3.4 Exiting from Loop

5.4 Summary

5.5 References and Further Reading

5.0 Introduction

A program consists of different types of statements. So far, two types of statements have been encountered: the return statement **(Table 2.2)** and expression statements (unit 4). Most of the remaining statements fall into three other categories, depending on how they affect the order in which statements are executed :

- ➢ Selection statements : It allows the program to choose a particular path from set of alternatives.
- ➢ Iterative statements : It allows the program to execute a specific instruction or set of instructions repetitively.
- ➢ Jump statement : It allows the program to jump unconditionally from one place to other place.

All these different statements are the main topics of this unit.

5.1 Objectives

After going through this unit the learner will be able to :

- ➢ Work with different types of selection statements;
- ➢ Know the appropriate use of the various iterative statements in programming;

- ➢ Transfer the control from within the loops;
- ➢ Use the goto, break and continue statements in the programs; and
- ➢ Write programs using branching, looping statements

5.2 Select Statements

As introduced earlier, select statement is used when the program needs to choose a specific path from a set of paths. Two types of select statements are available :

- ➢ *if* statement
- ➢ *switch* statement

5.2.1 The if Statement

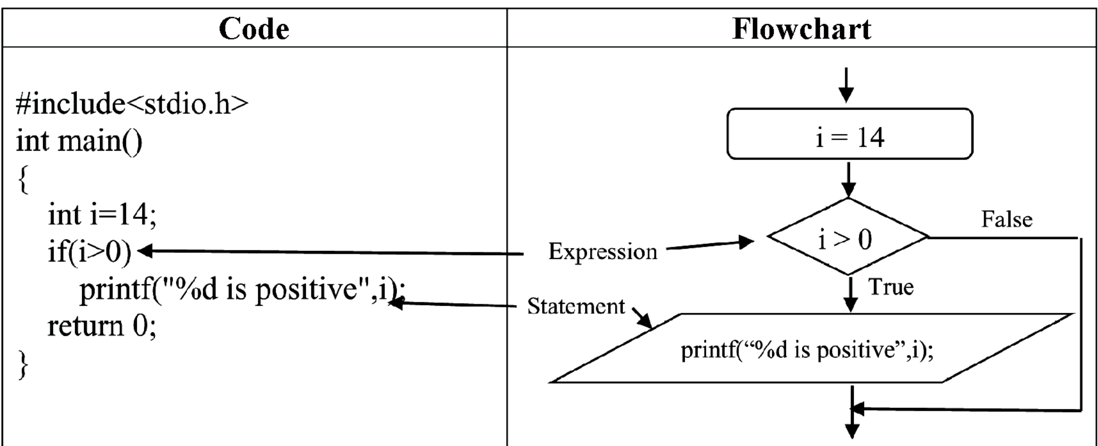
if statement has the form :

if (expression)
statement;

The following points need to be emphasized :

- ➢ Parenthesis around the *expression* are mandatory.
- ➢ When *if* statement is executed, *expression* inside the parenthesis are evaluated. If the value of the *expression* is non-zero which C interprets as true, the statement after parenthesis is executed.

For example, see the code and flowchart of a simple *if* statement in **Table 5.1**. Here the expression i > 0 produces 1 if true, 0 if false. Assigning 14 to i makes the *expression* true and as a result the *statement* after the *expression* printf("%d is positive", i) is executed. Choosing any negative value for i will make the *expression* false and therefore, printf statement will not be executed.



Code	Flowchart

#include<stdio.h>
int main()
{
int i=14;
if(i>0)
printf("%d is positive",i);
return 0;
}	

Table 5.1

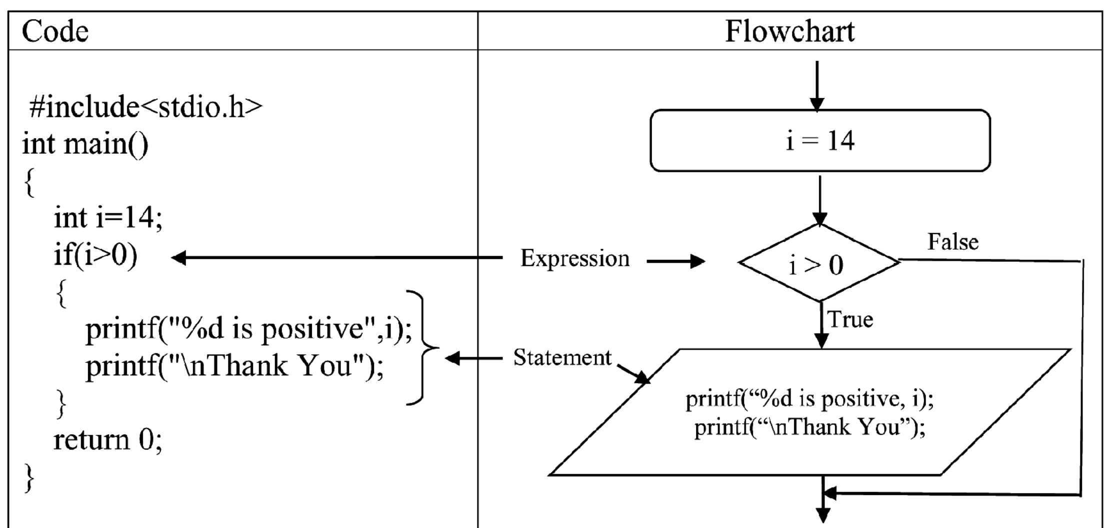
Compound Statements

if statement given in the last example is known as simple *if* statement. The statement after the *expression* has a single statement. If the statement is plural, then it is known as compound *if* statement. The form of the compound *if* is similar to simple *if* with additional braces around a group of statements which force the compiler to treat it as a single statement.

if (expression)
{ statements }

The previous example has been extended in Table to form an example of compound *if* statement.

In the example given in **Table 5.2**, two statements will be executed sequentially if the *expression* is true. The two printf statements are grouped together to form a statement block.



Code

```
#include<stdio.h>
int main()
{
   int i=14;
   if(i>0)
   {
      printf("%d is positive",i);
      printf("\nThank You");
   }
   return 0;
}
```


Table 5.2

The Else Clause

An *if* statement may have *else* clause. When *else* clause is present then the *if* statement is of the following form :

if (expression)
statements
else statements

The statements those follow *else* is executed when the *expression* in parenthesis has the value 0 or the *expression* is evaluated to be false.

Consider a program which takes an integer as input and display whether the integer is odd or even. **Table 5.3** shows the program along with a sample output. A variable t is used to store the remainder when the given integer is divided by 2. Obviously, if the expression (t==0) has the value 1 (equality condition true) the integer is even integer else (equality condition false) is odd. **Table 5.3** also shows the outputs for two different types of input. The flowchart of the *if* else statement is give in **Table 5.4**.

<table>
<tr><th>Code</th><th>Output</th></tr>
<tr><td rowspan="2"><pre>#include<stdio.h>
int main()
{
   int a,t;
   printf("Enter the integer: ");
   scanf("%d",&a);
   t=a%2;
   if(t==0)
      printf("\n%d is an even integer",a);
   else
      printf("\n%d is an odd integer",a);
return 0;
}</pre></td><td>Output 1

Enter the integer : 12
12 in an even integer</td></tr>
<tr><td>Output 2

Enter the integer : 5
5 in an odd integer</td></tr>
</table>

Table 5.3

Important point is to be noted that the *expression* used equality (t==0) operator not the assignment (t = 0). Use of assignment operator produces the value 0, which makes the expression always false and the statement following the else clause is executed.Therefore, for any input integer the program always displays the integer as odd. This has been shown in **Table 5.5**. The output suggests that for both the inputs 12 and 5 the program displays the message from the printf statement after else clause.

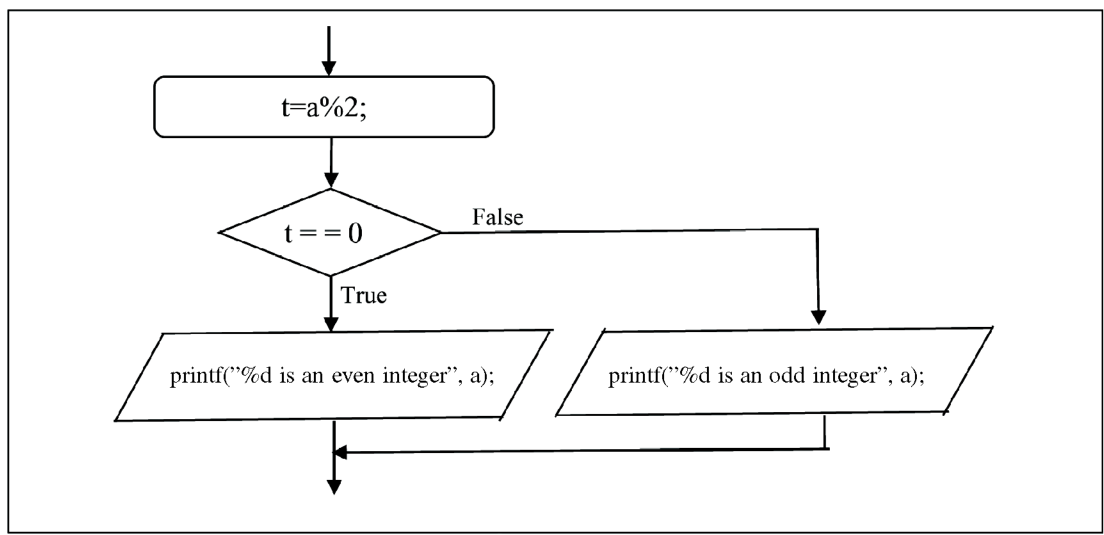


Table 5.4

Code	Output

#include<stdio.h>
int main()
{
int a,t;
printf("Enter the integer: ");
scanf("%d",&a);
t=a%2;
if(t=0)
printf("\n%d is a even integer",a);
else
printf("\n%d is an odd integer",a);
return 0;
}
	Output 1
	
	Enter the integer : 12
	12 in an odd integer
	
	Output 2
	
	Enter the integer : 5
	5 in an odd integer

Table 5.5

Nested if statement

It is quite common in C, that an if statement to be nested inside another if statement. The general form of nested if of depth 2 is shown in **Figure 5.1 :**

If statement can be nested to any depth. Consider the program, which finds the largest of the numbers a, b and c, all of which can be either integer or floating point numbers. The flowchart and the program code is shown in **Table 5.6**. This program has two inner (child) *if-else* statement (first one at line no. 9 and second at line no. 15) nested inside an outer (parent) *if* else statement (at line no. 7). The

braces are not mandatory as the *if else* statements are all simple, rather it helps the program to make more readable.

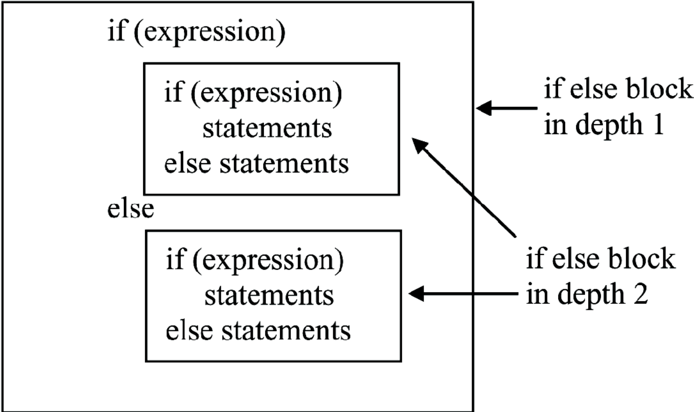


Figure 5.1

Flowchart	Line #	Code
read a, b, c a > b False True a > c False b > c False True True max = c max = b max = a max = c print max	1	#include<stdio.h>
	2	int main()
	3	{
	4	int a,b,c,max;
	5	printf("Enter the numbers: ");
	6	scanf("%d%d%d",&a,&b,&c);
	7	if(a>b)
	8	{
	9	if (a>c)
	10	max=a; if else (2)
	11	else max=c;
	12	}
	13	else Outer if else(1)
	14	{
	15	if (b>c)
	16	max=b; if else (2)
	17	else max=c;
	18	}
	19	printf("\nThe maximum number is
	20	%d",max);
	21	return 0;
	22	}

Table 5.6

Cascaded if Statement

It is often required to test a series of conditions, stopping as soon as one of them is true. A cascaded *if* is the best way in those situations. The general form of cascaded *if* is shown in **Figure 5.2 :**

Figure 5.2

```
if (expression)
statement
else if (expression)
statement
.....................
else if (expression)
statement
else statement
```

The last else statement is not mandatory. For example, let us consider a C program which will display the English word that corresponds to the grade of the students of a class. Let us also assumed that grade can be any number from 0 to 4. Any other value is treated as illegal grade. The flowchart and the C code of the program are given in **Figure 5.3** and **Table 5.7** respectively. Sample outputs for two specific choices of grade are also given in **Table 5.7**.

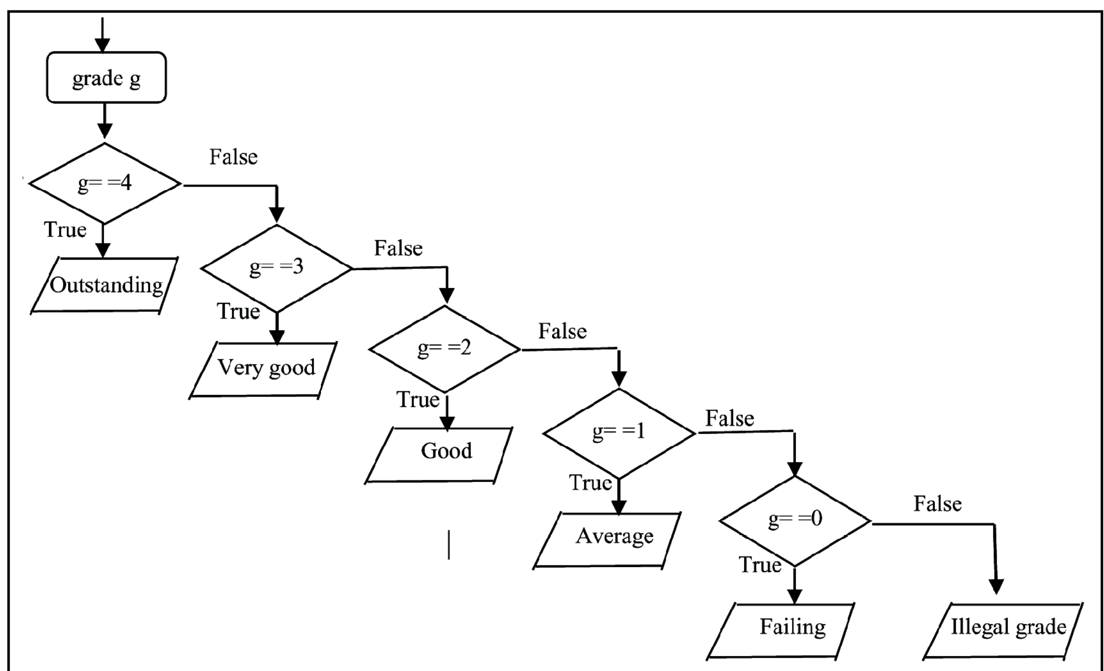


Figure 5.3

<table>
<tr><th>Code</th><th>Output</th></tr>
<tr><td rowspan="2">#include<stdio.h>
int main()
{
  int grade;
  printf("Enter the grade: ");
  scanf("%d",&grade);
  if(grade == 4)
    printf("\nPerformance is outstanding");
else if (grade == 3)
  printf("\nPerformance is very good");
else if (grade == 2)
  printf("\nPerformance is good");
else if (grade == 1)
  printf("\nPerformance is average");
else if (grade == 0)
  printf("\nPerformance is poor");
else printf("\nIllegal grade");
return 0;
}</td><td>Output 1

Enter the grade : 2
Perfromance is Good</td></tr>
<tr><td>Output 2

Enter the grade : 7
Illegal grade</td></tr>
</table>

Table 5.7

Example 5.1

Is the following *if* statement legal?

```
if (n>=1<=10)
        printf("n is in between 1 and 10\n");
```

If so, what does it do when n = 0.

Solution :

The *if* statement is legal. Let us first evaluate the expression in *if* statement. The *expression* contains two relation operator with same precedence. These operators are left associative **(Table 4.11)**. So the *expression* can be parenthesized in following way :

(n>=1<=10) $\Rightarrow$ ((n>=1)<=10)

$\Rightarrow$ (0<=10) [assuming n=0, (n>=1) is false and produces 0]

$\Rightarrow$ 1 [(0<=10) is true and produces 1]

Since C treat any non-zero value as true so the printf statement is executed and prints the message "n is in between 1 and 10". The output in **Table 5.8** shows the fact.

Code	Output

`#include<stdio.h>`
`int main()`
`{`
`  int n=0;`
`  if(n>=1<=10)`
`    printf("n is between 1 and 10");`
`  return 0;`
`}`
	n is between 1 and 10

Table 5.8

Example 5.2

Write a program that finds the largest and smallest of four integers entered by the user. (You can use maximum four if statements.)

Solution :

Assume a, b, c and d are integers. Implement following steps in C code.

1. Compare a and b and store larger integer into variable max1 and smaller one into min1.
2. Compare c and d and store larger integer into variable max2 and smaller one into min2.
3. Compare max1 and max2 and larger integer into variable largest.
4. Compare min1 and min2 and smaller integer into variable smallest.

Use four *if –else* statements for above four steps.

Check Your Progress 5.1

Is the following if statement legal?

```
if (n==1-10)
    printf("n is in between 1 and 10\n");
```

If so, what does it do when n=5.

Check Your Progress 5.2

The following if statement is unnecessarily complicated. Simplify it using one single if statement.

```
if (age >= 13)
      if (age <= 19)
            printf("Teenager\n");
      else
            printf("Not teenager\n");
else
      if (age < 13)
            printf("Not teenager\n");
```

Check Your Progress 5.3

Write a program that calculates how many digits a number contains assuming that the number has no more than four digits.

Sample output :

Enter a number : **374**
The number 374 has 3 digits

Hint : Use if statement to test the number. For example, if the number is between 0 and 9, it has one digit. If the number is between 10 to 99, it has two digits.

Check Your Progress 5.4

In a state, single residents are subject to the following income tax.

Income (in lakhs)	Amount of Taxes(% of income)
Not over 2	0
Over 2 – 3.5	5
Over 3.5 – 5	10
Over 5 – 10	20
Over 10	30

Write a program that asks the user to enter the income amount, then display the corresponding Tax.

Check Your Progress 5.5

Write a program that prompts the user to enter two dates in the form of mm/dd/yy and then indicate which date comes earlier in the calendar.

5.2.2 The switch Statement

The *switch* statement is used as an alternative way to implement cascaded *if* statement. In fact, a *switch* statement is often easier to read than cascaded *if* and often

faster than *if* statement specially when there are more than a handful number of cases. The same program given in **Table 5.7** could also be written using switch statement **(Table 5.9)**.

```
#include<stdio.h>
int main()
{
  int grade;
  printf("Enter the grade: ");
  scanf("%d",&grade);
  switch(grade) {
     case 4 : printf("\nPerformance is outstanding");
           break;
     case 3 : printf("\nPerformance is very good");
           break;
     case 2 : printf("\nPerformance is good");
           break;
     case 1 : printf("\nPerformance is average");
           break;
     case 0 : printf("\nPerformance is poor");;
           break;
  default : printf("\nIllegal  grade");
           break;
  }
  return 0;
}
```

Table 5.9

When this statement is executed, the value of the variable grade is tested against 4, 3, 2, 1, and 0. If it matches with 4, for example, the message 'Performance is outstanding' is printed, then the break statement transfers control to the statement following the *switch*. If the value of grade doesn't match any of the choices then default case applies, and message 'Illegal grade' is printed. The most common form of the *switch* statement is given in **Figure 5.4**.

```
switch (expression) {
    case constant-expression : statements
    ......................................
    case constant-expression : statements
    default : statements
}
```

Figure 5.4

Break Statement

It can be observed from the previous example that *switch* statement is a form of jump based on certain value. The word switch must be followed by an integer *expression* in parenthesis. This *expression* is known as controlling expression. Even the characters also can be controlling expression in *switch* statement as they are treated as integer in C. However, the floating point numbers and string don't qualify for becoming a controlling expression. When controlling expression is evaluated, control jumps to the *case* label matching the value of the *switch expression*. A *case* label is nothing more than a marker indicating a position in the *switch* statement. Once one of the *case* labels matches with the controlling expression, the remaining *case* labels should be ignored. This feature of the *switch* statement is achieved by the *break* statement. Without break (or some other jump statement) control will flow from one case to another. See the output of the previous program in **Table 5.10** if break is not used.

The output in **Table 5.10** shows that when the grade is chosen the value 2, the control jumps to case 2 and executes all the print statements one after another till the end of the block. Although the last case in switch statement never needs a break statement, it's a common practice to put one there anyway to guard against a missing break problem if cases should later be added.

Though forgetting *break* in *switch* statement is common error, the programmer intentionally omits the *break* statement in certain situation. For example, let us consider that we need to display the message "student failed" when the student acquires either grade as 0 or 1, and "student passed" for acquiring any of the remaining grades. In that case, omitting *break* statements is one of the possible option to solve the problem (see the code in **Table 5.11**. We can make the code even shorter as it is shown in **Table 5.12**.

<table>
<tr><th>Code</th><th>Output</th></tr>
<tr><td><pre>
#include<stdio.h>
int main()
{
   int grade;
   printf(“Enter the grade: ”);
   scanf(“%d”,&grade);
   switch(grade) {
      case 4 : printf(“\nPerformance is outstanding”);
      case 3 : printf(“\nPerformance is very good”);
      case 2 : printf(“\nPerformance is good”);
      case 1 : printf(“\nPerformance is average”);
      case 0 : printf(“\nPerformance is poor”);
      default: printf(“\nIllegal grade”);
   }
   return 0;
}
</pre></td><td>Enter the grade : 2
Performance is good
Performance is average
Performance is poor
Illegal grade</td></tr>
</table>

Table 5.10

```
#include<stdio.h>
int main()
{
   int grade;
   printf("Enter the grade: ");
   scanf("%d",&grade);
   switch(grade) {
      case 4 :
      case 3 :
      case 2 : printf("\nStudent Passed");break;
      case 1 :
      case 0 : printf("\nStudent Failed");break;
      default: printf("\nIllegal grade");break;
   }
   return 0;
}
```

Table 5.11

```
#include<stdio.h>
int main()
{
  int grade;
  printf("Enter the grade: ");
  scanf("%d",&grade);
  switch(grade) {
    case 4 : case 3 : case 2 :
        printf("\nStudent Passed");break;
    case 1 : case 0 :
        printf("\nStudent Failed");break;
    default: printf("\nIllegal grade");break;
  }
  return 0;
}
```

Table 5.12

Check Your Progress 5.6

What does the following program fragment produce? (Assume that i is an integer variable.)

```
i = 1;
    switch ( i % 3) {
    case 0 : printf("zero");
    case 1 : printf("one");
    case 2 : printf("two")
}
```

Check Your Progress 5.7

The following table shows partial list of pin codes of few districts in West Bengal.

Pin Code	District	Pin Code	District
700020	Kolkata	731303	Birbhum
700027	Kolkata	724404	Darjeeling
700046	Kolkata	734301	Darjeeling
713144	Bardhaman	741235	Nadia
713335	Bardhaman	742202	Murshidabad

Write a switch statement whose controlling expression is the variable pin code. If the value of the pin code is in the table, switch statement will print the corresponding district. Otherwise, it will display the message "District not found". Use techniques to make the switch statement as simple as possible.

Check Your Progress 5.8

Write a program that asks the user for a two-digit number, then prints the English word for the number :

Sample Output :

Enter a two-digit number : **67**
The number entered is sixty-seven

Hint : Break the number into two digits. Use one switch statement to print the word for first digit ("Twenty", "Thirty", and so forth). Use a second switch statement to print the word for second digit. Don't forget that numbers between 11 and 19 require special treatment.

5.3 Iterative Statements

An iterative statement repeatedly executes other statement/s, known as body of the iteration or loop. In C, there are mainly three types of iterative statement or loop. They are :

- while loop
- do-while loop
- for loop

5.3.1 while Loop

The *while* loop has a general form like the following :

while (expression) statement

The *expression* inside the parenthesis is known as controlling expression. The statement after the parenthesis is known as loop body. Let us understand the *while* loop using a program that asks the user to enter a number n. The program will print n lines of output, with each line containing a number between 1 and n along with its square. The program will have following sequential steps.

1. Ask user for a value of n.
2. Initialize a variable i by 1.
3. Check whether the value of i is less than or equal to the value of n. If 'Yes' then go to step 4, otherwise go to step 7.
4. Print the value of i and its square.
5. Increment the value of i.
6. Go to step 3.
7. Stop.

In summary, the program will repeatedly print a number and its square until the number exceeds n. The controlling expression (step 3) determines whether the program control will enter into or exit from the loop. The statements in step 4 and step 5 are executed repeatedly in a loop and form the body of the loop. The sequential steps mentioned above can be presented as a flowchart **(Figure 5.5)**. Variable i, known as loop control variable, needs to be initialized before entering the loop (step 2). Without that i can take any value which may give some unexpected result. The value of i also needs to be updated (step 5) inside the loop body before checking the condition in the next iteration (step 3). The C code and the output of the above program is given in **Table 5.13**.

Let us see how the program execution takes place for a specific value of n equal to 4.

n=4;	n is chosen as 4.
i=1;	i is now 1.
Is i <= n?	Yes; continue.
Print i and i*i	1 1
i++;	i is now 2.
Is i <= n?	Yes; continue.
Print i and i*i	2 4
i++;	i is now 3.
Is i < =n?	Yes; continue.
Print i and i*i	3 9
i++;	i is now 4.
Is i < =n?	Yes; continue.
Print i and i*i	4 16
i++;	i is now 5.
Is i < =n?	No; exit from the loop.

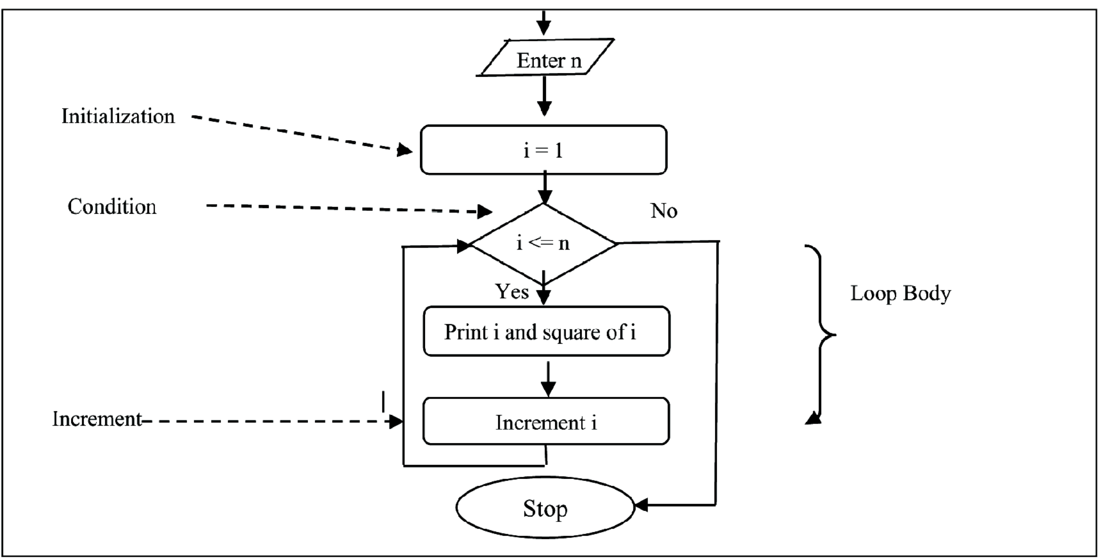


Figure 5.5

Therefore, the final output will have 4 lines as it is shown in **Table 5.13**. Notice how the loop keeps going as long as the controlling expression (i < n) is true (non-zero). When the controlling expression is false (zero), loop terminates and i is greater than or equal to n as desired. Since the controlling expression is tested before the loop body is executed, it is possible that the body isn't executed even once. For example, if the value of i initially is set to any integer greater than 4 the body will not be executed at all.

Code	Output

`#include<stdio.h>`
`int main()`
`{`
`  int i,n;`
`  printf("Enter the value of n: ");`
`  scanf("%d",&n);`
`  i=1;`
`  while(i<=n)`
`  {`
`    printf("\n%d\t%d",i,i*i);`
`    i++;`
`  }`
`  return 0;`
`}`
	Enter the value of n : 4
	1 1
	2 4
	3 9
	4 16

Table 5.13

Infinite Loop

A *while* statement will never terminate if the controlling expression always has non-zero value. For example, the code in **Table 5.14** prints the string "Hello" infinite number of times. At times, C programmers use this technique to implement infinite loop.

```
#include<stdio.h>
int main()
{
    int i=4;
    printf("Enter the value of n: ");
    while(i)
    {
        printf("Hello\n");
        i++;
    }
    return 0;
}
```

Table 5.14

Example 5.3

For a set of n numbers given by the user, write a program to calculate the sum of the numbers and print the sum as output.

Solution :

The algorithm of this program is already discussed in section 1.5.2 in Unit-1. The C-code for algorithm in **Figure 1.8** is given in **Table 5.15**.

```
#include<stdio.h>
int main()
{
    int i=1,s=0,n,a;
    printf("Enter how many numbers you want to add(n): ");
    scanf("%d",&n);
    while(i<=n)
    {
        printf("Enter the number: ");
        scanf("%d",&a);
        s=s+a;
        i++;
    }
    printf("\nThe sum is %d",s);
    return 0;
}
```

Output

```
Enter How many numbers you want to add(n) : 4
Enter the number :  3
Enter the number :  4
Enter the number :  6
Enter the number :  7
The Sum is 20
```

Table 5.15

Example 5.4

Explain the difference between the output produced by following two program fragments?

Program 1	Program 2

#include<stdio.h>
int main()
{
  int i=1;
  while(i<=6)
    printf("%d ",i++);
  return 0;
}
	#include<stdio.h>
	int main()
	{
	  int i=1;
	  while(i++<=6)
	    printf("%d ",i);
	  return 0;
	}

Solution :

The main difference between two program is the position of post increment (++) operator. In program 1, The controlling expression of *while* loop is evaluated first. Since the expression is true (non-zero) for the initial value 1, the control enters into the loop body. Because of the post increment operator, first the value 1 is printed and then i is incremented to 2. The condition is re-tested for i=2 again and keeps repeating for six times until i reaches to 7. Finally, i becomes 7 as a result the condition is violated and the loop terminates. Therefore, the final output is – 1 2 3 4 5 6.

In program 2, immediately after testing the condition value of i is incremented. Therefore, the incremented value is printed. The condition is true for 6 times and the value displayed from 2 to 7. So the final output is – 2 3 4 5 6 7.

The equivalent code for the two programs are as follows.

Program 1	Program 2

#include<stdio.h>
int main()
{
  int i=1;
  while(i<=6)
  {
    printf("%d ",i);
    i++;
  }
  return 0;
}
	#include<stdio.h>
	int main()
	{
	  int i=1;
	  while(i<=6)
	  {
	    i++;
	    printf("%d ",i);
	  }
	  return 0;
	}

Example 5.5

The following code results in an infinite number. Justify the reason.

```
int i, n =4;
i = 1;
while ( i <= n);
        i ++;
```

Solution :

The semicolon after the while loop indicates the end of the while loop. Therefore, the next statement i++ is no longer a part of loop body and variable i doesn't get incremented, so condition remains true forever. **Figure 5.6** shows difference of control-flow in a while loop based on the position of the semicolon. The Figure on the left side is example of a simple while loop with a single statement (i++) in its body. The control flows repeatedly according to the arrow diagram. On the other hand, the Figure on the right side also has a while loop but without any statement in its body (semicolon ends after controlling expression). Therefore, the i++ statement will never be executed and loop will continue for ever.

Figure 5.6

Check Your Progress 5.9

Write a C program to compute the sum of squares of n numbers using a while loop (Develop the C program using the algorithm designed in Check Your Progress 1.3).

Check Your Progress 5.10

Write a C program to compute the harmonic mean of n numbers using a while loop (Develop the C program using the algorithm designed in Check Your Progress 1.4).

Check Your Progress 5.11

Write a C program to print the following sequence of n numbers using a while loop (Develop the C program using the algorithm designed in Check Your Progress 1.5).

Check Your Progress 5.12

Write a C program to compute the factorial of n numbers using a while loop (Develop the C program using the algorithm designed in Check Your Progress 1.7).

5.3.2 do-while Loop

do-while loop is mostly similar to *while* loop with only difference — the controlling expression is tested after each execution of loop body. The general form of do statement is

```
do
{
        Statement
} while (expression);
```

When a *do* statement is executed, the loop body is executed first, then the controlling expression is evaluated. If the value of the *expression* is non-zero, the loop body is executed again and then the *expression* is evaluated again. The statement terminates if the controlling expression is evaluated to zero after the loop body has been executed. The program in **Table 5.13** can be written using *do-while* loop. The flow chart is given in **Figure 5.7**.

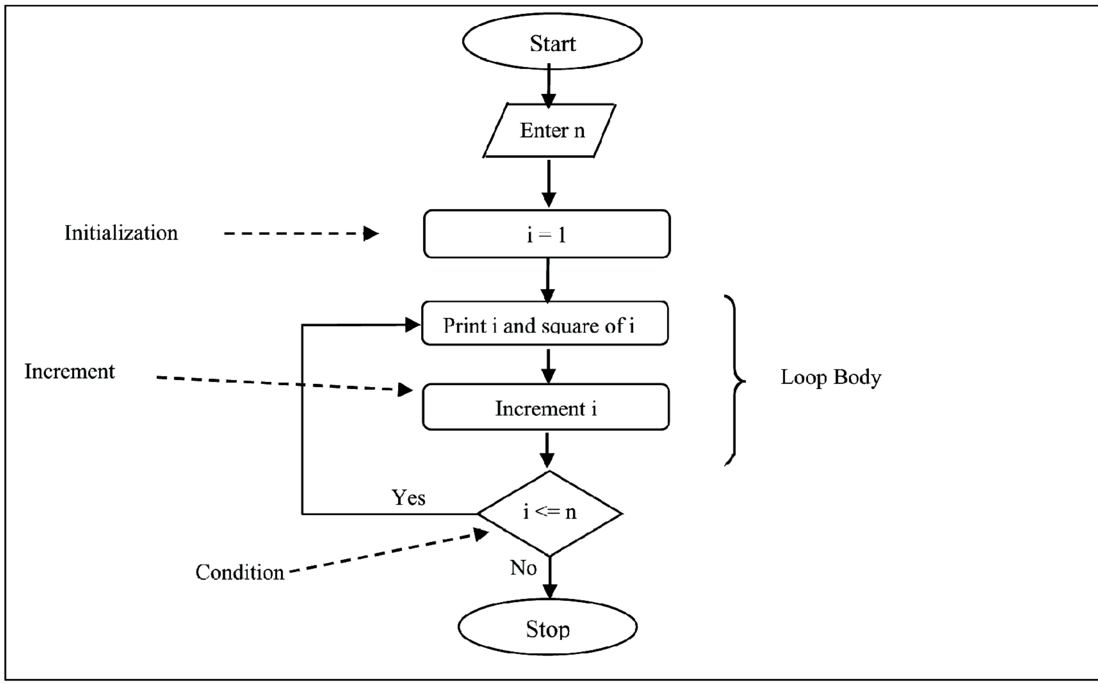


Figure 5.7

Note the difference of position of the condition in **Figure 5.5** and **Figure 5.7**. This says that the condition is checked at the end of the loop unlike while loop where condition is checked at the beginning of the loop. The C code for the same program using do-while loop and the corresponding output is given in **Table 5.16**. Even though the output remains the same for both while and do while loop implementations of the program for all values of n>0, it may not be same if n is chosen as 0. In that case the while will be false at the first time and loop body will not be executed at all. So the final output will be 0. On the other hand, in the case of do while loop the loop body will be executed at least for the first time which is not desired. The output for n equal to 0 is shown in **Table 5.17** for both while and do-while loop implementation of the given program.

Another important difference is the position of semicolon (;). In do-while loop, the semicolon (;) is present after the while statement to indicate the end of the loop whereas the same shouldn't be there in case of while loop.

```
#include<stdio.h>
int main()
{
   int i=1,s=0,n,a;
   printf("Enter how many numbers you want to add(n): ");
   scanf("%d",&n);
   do
   {
      printf("Enter the number: ");
      scanf("%d",&a);
      s=s+a;
      i++;
   } while(i<=n);
   printf("\nThe sum is %d",s);
   return 0;
}
```

Output

```
Enter How many number you want to add (n) : 4
Enter the number :  3
Enter the number :  4
Enter the number :  6
Enter the number :  7

The sum is 20
```

Table 5.16

Output (Using While loop)	Output (Using do-while loop)
Enter How many number you want to add(n) : 0 The sum is 0	Enter How many number you want to add (n) : 0 Enter the number :

Not Desired

Table 5.17

Example 5.6

Write a C program to print first n terms of Fibonacci sequence where $n > 1$ using a do-while loop (Develop the C program using the algorithm designed in section 1.5.3) considering the first two terms are 0 and 1 respectively.

Solution :

The program has been given in to **Table 5.18**. Refer the algorithm in section 1.5.3 for detailed explanation.

```
#include<stdio.h>
int main()
{
   int i=2,a=0,b=1,c,n;
   printf("Enter how many terms you want(n>1): ");
   scanf("%d",&n);
   printf("\nThe Fibonacci sequence for first %d terms is: \n",n);
   printf("\n%d\t%d",a,b);
   if (n>2)
   {  do
      {  c=a+b;
         a=b;
         b=c;
         i++;
         printf("\t%d",c);
      } while (i<n);
   }
   return 0;
}
```

Output

```
Enter how many terms you want (n>1) : 8
The Fibonacci sequence for first 8 terms is :
0    1    1    2    3    5    8    13
```

Table 5.18

Check Your Progress 5.13

The program in Example 5.6 is valid for n greater than 1. Modify the program so that it can also consider the case when value of n is greater than or equal to 1. (Hint: add the logic using select statement discussed in section 5.2.1 and print a if n is equal to 1, print a and b if n is equal to 2, execute the do-while loop in **Table 5.18** otherwise.)

Example 5.7

Write a C program to reverse the digits of an integer n using a do-while loop (Develop the C program using the algorithm designed in section 1.5.4).

The program has been given in **Table 5.19**. Refer the algorithm in section 1.5.4 for detailed explanation.

```
#include<stdio.h>
int main()
{
   int i=1,r=0,R=0,n;
   printf("Enter the number you want to reverse(n): ");
   scanf("%d",&n);
   do
   {
      r=n%10;
      n=n/10;
      R=10*R+r;
   }while (n!=0);
   printf("\nThe reversed number is %d",R);
   return 0;
}
```

Output

```
Enter the number you want to reverse(n) : 34521
The reversed number is 12543
```

Table 5.19

Check Your Progress 5.14

Write a C program to count the number of digits of an integer n using a while loop (Develop the C program using the algorithm designed in Check Your Progress 1.13).

Check Your Progress 5.15

Write a C program to compute the sum of the digits of an integer n using a while loop (Develop the C program using the algorithm designed in Check Your Progress 1.14).

Check Your Progress 5.16

Write a C program that reads in a set of *n* single digits and convert them into a single decimal integer using a while loop. For example, the algorithm should convert the set of 5 digits {2,7,4,9,3} to the integer 27493. (Develop the C program using the algorithm designed in Check Your Progress 1.15).

5.3.3 for Loop

Another way to implement repetitive statement in C is *for* loop. This loop is ideal for loops that have a counting variable, but it is versatile enough to be used for other kinds of loops as well. The *for* loop is of following form :

for (expr1; expr2; expr3)

Expression *expr1* is mainly responsible for initializing the counting variable or loop index. Expression *expr2* is controlling expression or the condition and expression *expr3* mainly modify the loop index. To understand how *for* loop works let us take the same example where a number and its square are displayed given in **Table 5.13**. The C code and its output are given in **Table 5.20**.

Code	Output

`#include<stdio.h>`
`int main()`
`{`
`   int i=1,n;`
`   printf("Enter the value of n: ");`
`   scanf("%d",&n);`
`   for(i=1;i<=n;i++)`
`      printf("\n%d\t%d",i,i*i);`
`   return 0;`
`}`
	Enter the value of n : 4
	1 1
	2 4
	3 9
	4 16

Table 5.20

Here, loop index i is initialized first and then the condition i <= n is evaluated. If the condition is true then the print statement inside the loop body is executed and then loop index i is incremented and checks the condition again. If the condition is evaluated to false(zero) the program control exits from the loop. **Figure 5.8** shows the working principle of *for* loop. *for* loop is closely related to *while* and *do-while* loop. The only difference is the relative positions of expressions which makes the syntax of *for* loop different from other loops. In fact, except in few rare cases, a *for* loop can always be replaced by equivalent *while* or *do-while* loop.

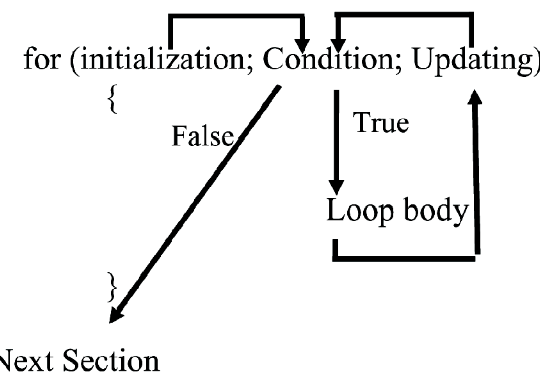


Figure 5.8

Omitting Expression in a for Loop

The expressions are not mandatory inside the parenthesis of the *for* loop. If the first expression is omitted, no initialization is performed before the loop is executed. In that case, the loop index may be initialized by a separate statement. For example,

```
i = 0;
for( ; i<=10 ; i++)
    printf("%d", i);
```

The important point to be noted is that, the semicolon between first and second expression remains. In fact, two semicolons must be present, even when the expressions are omitted.

The third expression could also be omitted. In that case, the loop index needs to be modified inside the loop body. When the first and third expressions are omitted, the resulting loop is nothing more than a while loop in disguise. **Table 5.18** compares the structure of a for loop without first and third expression vs. while loop. Finally, the second expression or the condition also can be omitted. If the condition is missing then by default it is evaluated to true, and therefore, for loop will not terminate. So for (; ;), is a valid statement in C which creates an infinite loop.

<table>
<tr><th>For Loop</th><th>While Loop</th></tr>
<tr><td><pre>
#include<stdio.h>
int main()
{
   int i,n;
   printf("Enter the value of n: ");
   scanf("%d",&n);
   i=1;
   for(;i<=n;)
   {
   printf("\n%d",i);
   i++;
   }
   return 0;
}
</pre></td><td><pre>
#include<stdio.h>
int main()
{
   int i,n;
   printf("Enter the value of n: ");
   scanf("%d",&n);
   i=1;
   while(i<=n)
   {
   printf("\n%d",i);
   i++;
   }
   return 0;
}
</pre></td></tr>
</table>

Table 5.21

5.3.4 Exiting from Loop

The conditional expression is commonly used as an exit point which is responsible for exiting the control from the loop. Now it has been observed that the loop has this kind of exit point either before (*while*, *for* loop) or after the loop body (*do-while* loop). There are certain situations, when the exit point is needed in the middle of the loop body. This additional exit point can be created using *break* statement in C. There is another C statement known as *continue* which is used very frequently to skip remaining part of loop body from its presence without jumping out of the loop. In fact, the *break* and *continue* statements are controlled jumps where the program control either jump to exit the loop or to the end of the loop body. For any kind of uncontrolled jump, C uses *goto* statement. It allows the program to jump from one statement to any other statement without any restriction.

Break Statement

The *break* statement is already used to transfer control out of *switch* statement. The *break* statement can also be used to jump out of a *while*, *do-while* and *for* loop. For example, a C program that can check whether a given number n is prime or not. A *for* loop may be used that divides n by the numbers between 2 and n-1 **(Table 5.22)**. As soon as any divisor is found the control should exit from the loop without checking the condition for remaining numbers. The loop can be terminated due to following two reason.

- **Normal Termination :** no divisor found from 2 to n-1, and the value of loop index i is incremented to n which results in violation of condition and terminates the loop. This implies n is prime number.
- **Premature Termination :** a divisor found and loop is terminated because of the break statement. In this situation, the value of loop index i < n-1. This implies n is not a prime number.

Code	Output

`#include<stdio.h>`
`int main()`
`{`
`  int i,n;`
`  printf("Enter the value of n: ");`
`  scanf("%d",&n);`
`  for(i=2;i<n;i++)`
`  {`
`  if(n%i==0)`
`    break;`
`  }`
`  if (i<n)`
`    printf("\n%d is not prime",n);`
`  else printf("\n%d is prime",n);`
`  return 0;`
`}`
	Output 1
	
	Enter the value of n : 79
	79 is prime
	
	Output 2
	
	Enter the value of n : 36
	36 is not prime

Table 5.22

After the loop has terminated, select (if-else) statement is used to determine whether termination was normal (n is prime) or premature (n is not prime). The performance of the C program in **Table 5.22** can be improved by modifying the condition of the for loop. Considering the fact that the maximum value of any factor of an integer can't be more than square root of the integer, the condition of the for loop can be replaced by $i \leq \sqrt{n}$ instead of $i < n$.

A break statement transfers control out of the innermost loop enclosing *while*, *do-while*, *for* or *switch* statement, therefore, when loops are nested, the break statement can escape one level of nesting. Consider the case of a switch statement nested inside a while loop in **Figure 5.9 :**

```
while (...)
        {    switch (...)
                 {    ...
                          break;
                      ...
                 }
        }
```

Figure 5.9

The break statement transfers control out of switch statement, but not out of while loop.

Continue Statement :

While the break statement transfers the control just past the end of the loop, continue statement transfers the control to the point before the end of the loop **(Figure 5.10)**. As a result, the loop automatically goes to the next iteration and start checking the condition again. In that way, continue eventually sends the control at the beginning of the loop. Unlike break, continue statement can't be used in switch statement.

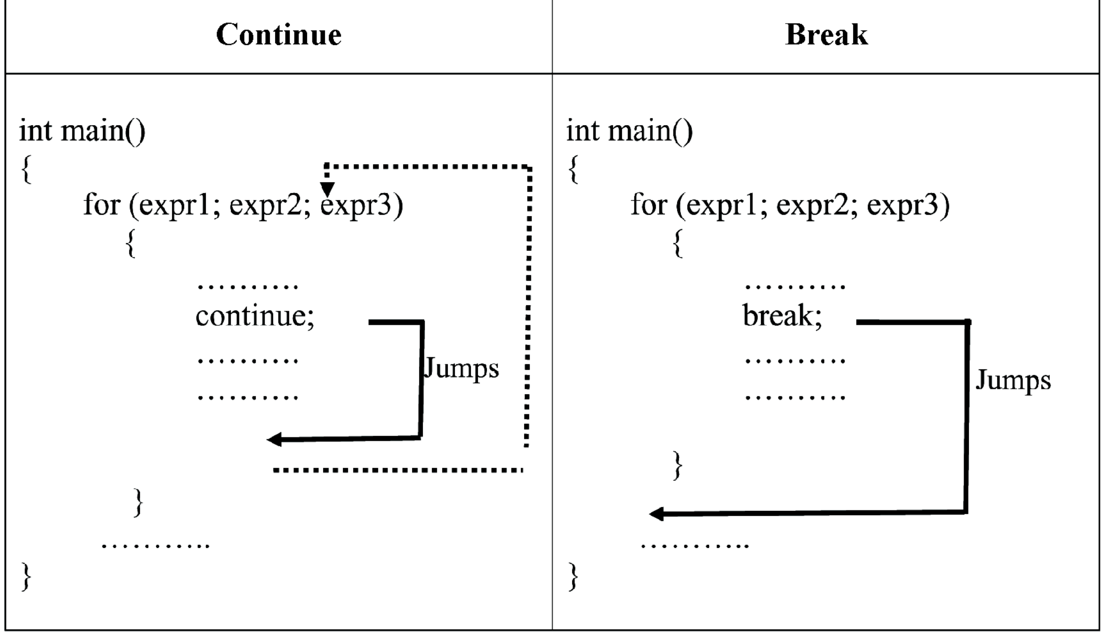


Figure 5.10

The following C program **(Table 5.23)**, that reads a series of numbers and computes their sum, illustrate a simple use of *continue*. The program takes n non-zero numbers and print their sum as output. Here, whenever the user provides a 0 the *continue* statement skips rest of the loop body but remains inside the loop.

```
#include<stdio.h>
int main()
{
   int i=1,n,x,sum=0;
   printf("Enter how many non-zero numbers you want to add(n): ");
   scanf("%d",&n);
   while(i<=n)
   {
      printf("\n Enter the number: ");
      scanf("%d",&x);
      if(x==0)
         continue;
      sum+=x;
      i++;
   }
   printf("\nSum of %d non-zero numbers is %d",n,sum);
   return 0;
}
```

Output

```
Enter how many non-zero numbers you want to add(n) : 3
Enter the number : 5
Enter the number : 0
Enter the number : –2
Enter the number : 0
Enter the number : 65

Sum of 3 non-zero numbers is 68
```

Table 5.23

goto Statement :

It has been observed that break and continue transfer control to the point past the end of loop and before the end of loop respectively. In that sense, this jump

statements are controlled or restricted. In C, there is another jump statement known as *goto* which can jump to any statement without any restriction, provided the statement has a *label*.

A *label* is an identifier placed at the beginning of a statement. For example,

```
goto L1;
  L1 : printf("Welcome");
```

Executing the statement *goto* L1, transfer the control to the statement that follows the *label* L1. The *goto* statement can be used instead of break in program given in **Table 5.22**. The modified program is given in **Table 5.24**.

Table 5.22. The modified program is given in **Table 5.24**.

```
#include<stdio.h>
int main()
{
   int i,n;
   printf("Enter the value of n: ");
   scanf("%d",&n);
   for(i=2;i<n;i++)
   {
      if(n%i==0)
      goto done;
   }
   done : if (i<n)
      printf("\n%d is not prime",n);
      else printf("\n%d is prime",n);
   return 0;
}
```

Table 5.24

Here, if a divisor is found, then the control transfers to the *label* 'done' and gives the desired result.

Example 5.8

Which one of the following statements is not equivalent to the other two (assuming that the loop bodies are the same) ?

I. for (i = 0 ; i < 5 ; i++) ....
II. for (i = 0 ; i < 5; ++i) ....
III. for (i = 0 ; i ++ < 5 ;) ....

Solution :

Statement III is different from statements I and II. Statements I and II both enter to loop body after evaluating same condition ($i < 5$). Even though the increment operators are different (post and pre), that will not impact the value of i in the next iteration. Post and pre increment operators gives same result if they are executed as a singular statement. Statement III, has different condition. Though conditional expression evaluates $i < 5$, but immediately after that it increments the value of i and enters the loop with incremented value. For example, after successful evaluation of condition for the first time the value of i is equal to 1. On the other hand, the value of i is equal to 0 while executing statement I and II for the first time. Therefore, the result is different. The output is given in **Table 5.25**.

Statement I	Statement II	Statement III

#include<stdio.h>
int main()
{
  int i;
  for(i=0;i<5;i++)
    printf("%d ",i);
  return 0;
}
	#include<stdio.h>
	int main()
	{
	  int i;
	  for(i=0;i<5;++i)
	    printf("%d ",i);
	  return 0;
	}
		#include<stdio.h>
		int main()
		{
		  int i;
		  for(i=0;i++<5;)
		    printf("%d ",i);
		  return 0;
		}

Output 0 1 2 3 4	**Output** 0 1 2 3 4	**Output** 1 2 3 4 5

Table 5.25

Check Your Progress 5.17

Which one of the following statements is not equivalent to the other two (assuming that the loop bodies are the same)?

I. while (i < 10) { ....}
II. for (; i < 10 ;) { ....}
III. do {..........} while (i < 10) ;

Check Your Progress 5.18

Show how to replace a continue statement by an equivalent goto statement in C.

Check Your Progress 5.19

Write a C program to compute the GCD (greatest common divisor) of two integers m, n using approach1 and approach2 (Euclid) discussed in section 1.6.2).

Check Your Progress 5.20

What output does the following program produce?

```
sum=0;
for (i = 0 ; i < 10 ; i++)
   {
      if (i % 2)
      continue;
      sum += i ;
   }
printf("%d",sum);
```

Example 5.9

The value of the trigonometric function cos(x) at a given x can be expressed as an infinite series using Taylor series expansion of cos(x) centered at x = 0.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Using the above formula write a C program to compute the value of cos(x) at x = 0.2 and at x = 1.6 accurate up to 4 decimal points.

Solution :

It is easy to check that the general term $t_i = (-1)^i \frac{x^{2i}}{(2i)!}$ for i = 0, 1, 2, . . . ∞.

Therefore, we can write $t_i = (-1)\frac{x^2}{(2i-1)(2i)}t_{i-1}$, assuming t_0 = 1. Now the first four terms of the series are given below in **Table 5.26**.

Value of i	t_i
0	1 (by assumption)
1	$t_1 = (-1)\frac{x^2}{(1)(2)}t_0 = (-1)\frac{x^2}{(1)(2)}1 = -\frac{x^2}{2!}$
2	$t_2 = (-1)\frac{x^2}{(3)(4)}t_1 = (-1)\frac{x^2}{(3)(4)}\left(-\frac{x^2}{2!}\right) = \frac{x^4}{4!}$
3	$t_3 = (-1)\frac{x^2}{(5)(6)}t_2 = (-1)\frac{x^2}{(5)(6)}\left(\frac{x^4}{4!}\right) = -\frac{x^6}{6!}$

Table 5.26

The C program and the output is given in the **Table 5.27**.

The program used a function fabs (floating argument) which gives absolute value of a floating point number supplied as argument. The function fabs is defined in the <math.h> header file which the program includes. The while loop terminates when the desired accuracy of 4 decimal point has been reached. The original value of cos(x) is also provided in the output. The cos(x) function is also defined in <math.h> header file. The output suggests that as and when the higher order terms in the series are added the estimated values of cos(0.2) and cos(1.6) approach to the true values of cos(0.2) and cos(1.6) respectively. Another important observation is that more number of iterations (terms) is required to achieve the desired accuracy for cos(1.6) than cos(0.2). This happens because 0.2 is closer than 1.6 with respect to center 0 of Taylor series expansion.

```
#include<stdio.h>
#include<math.h>
int main()
{
   int i=0;
   float x,t=1,sum=1;
   printf("Enter the value of x: ");
   scanf("%f",&x);
   printf("\n i\t term\t sum\t cos(x)\n"); /*Header of the output table*/
   printf("------------------------------\n");
   printf("%4d\t%6.4f\t%6.4f\t%6.4f\n",i,t,sum,cos(x));/*The 0th term displayed*/
   i++;
   while(fabs(t)>.00001)
   {
      t=(-1)*t*x*x/((2*i-1)*(2*i));
      sum=sum+t;
      printf("%4d\t%6.4f\t%6.4f\t%6.4f\n",i,t,sum,cos(x));
      i++;
   }
   printf("\n\nThe value of cos(%2.1f) is %6.4f",x,sum);
   return 0;
}
```

Output 1				Output 2			
Enter the value of x: .2				Enter the value of x: 1.6			
i	term	sum	cos(x)	i	term	sum	cos(x)
0	1.0000	1.0000	0.9801	0	1.0000	1.0000	–0.0292
1	–0.0200	0.9800	0.9801	1	–1.2800	–0.2800	–0.0292
2	0.0001	0.9801	0.9801	2	0.2731	–0.0069	–0.0292
3	–0.0000	0.9801	0.9801	3	–0.0233	–0.0302	–0.0292
				4	0.0011	–0.0292	–0.0292
				5	–0.0000	–0.0292	–0.0292
The value of cos (0.2) is 0.9801				6	0.0000	–0.0292	–0.0292
				The value of cos(1.6) is –0.292			

Table 5.27

Check Your Progress 5.21

The value of the trigonometric function log(x) at a given x can be expressed as an infinite series using Taylor series expansion of log(x) centered at x = 1 :

$$\log(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \cdots$$

Using the above formula write a C program to compute the value of log(x) at x = 1.4 accurate up to 4 decimal points. What happens when you compute log(x) at x = 2.4. Justify the result mathematically (Hint : Find out the radius of convergence of the series and check that x = 2.4 lies outside the radius of convergence. Any value beyond the radius of convergence make the series divergent and the condition will be always true which makes an infinite loop).

Check Your Progress 5.22

Write a program for e^x using Taylor expansion about x = 0 and then find out the value of e. Check that, the program can accurately (up to 4 decimal places) estimate any value of e^x where $x \in R$ (real number) as the radius of convergence is $(-\infty, \infty)$.

Hint : The formula for Taylor expansion of $f(x)$ centered at $x = a$ is given by

$$f(x) = f(a) + \frac{df}{dx}(a)\frac{(x-a)^1}{1!} + \frac{d^2 f}{dx^2}(a)\frac{(x-a)^2}{2!} + \frac{d^3 f}{dx^3}(a)\frac{(x-a)^3}{3!} + \cdots\cdots\cdots$$

5.4 Summary

A program is usually not limited to a sequence of simple instructions. During its process it may require to select a specific path based on certain condition. C provides select statement to handle those situations. Often program may repeat execution of a part of code more than once depending upon the requirements. For that purpose, C provides repetitive statements. In this unit, different select statements like *if*, *if-else*, *switch* and repetitive statements like *while*, *do-while* and *for* have been discussed with different scenarios. Different types of jumps like *break*, *continue* and *goto* statements have also been illustrated with many examples by which the control can leave a loop even if the condition for its termination is not fulfilled. All the further topics are heavily dependent on these constructs of C, and therefore it is mandatory for the learners to understand these topics completely.

5.5 References and Further Reading

1. The C Programming Language, Kernighan & Ritchie, PHI Publication, 2011
2. Programming with C, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
3. The C Complete Reference, Fourth Editon, Herbert Schildt, Tata McGraw Hill, 2002.
4. C Programming : A Modern Approach, Second Edition, K.N. King, W. W. Norton & Company, 2008.
5. Computer Science : A Structured Programming Approach Using C, Second Edition, Behrouz A. Forouzan, Richard F. Gilberg, Brooks/Cole Thomas Learning, 2001.
6. The C Primer, Leslie Hancock, Morris Krieger, McGraw Hill, 1983.

Unit - 6 ❑ Arrays

Structure

6.0 Introduction

6.1 Objectives

6.2 One Dimensional array

- **6.2.1 Array Declaration**
- **6.2.2 Array Subscripting**
- **6.2.3 Array Initialization**
- **6.2.4 Sorting the Elements of an Array in Ascending Order**

6.3 Multidimensional array

- **6.3.1 Physical View and Logical View of an Array**
- **6.3.2 Initializing Multidimensional Array**

6.4 Summary

6.5 References and Further Reading

6.0 Introduction

The basic datatypes int, char, float and double has been discussed in unit 3. These data types are known as scalar type as they are capable of holding single data item. In some situations, using scalar data type makes the coding tedious and inefficient. For example, consider a program which read 5 integers and display them in reverse order. One obvious way to solve this is by writing the program in the following way :

```
int main()
{
int a, b, c, d, e;
printf("Enter the numbers");
scanf("%d%d%d%d%d", &a, &b, &c, &d, &e);
printf("\n%d %d %d %d %d", e, d, c, b, a);
return 0;
}
```

Since the int datatype can store a single integer at a time, five different integer variables have been created. All these names of variables need to be remembered for further operations. It will be a nightmare for the programmer if the program contains more than 100 variables or so. To get rid of this issue C provides aggregate variables which can store collection of values. There are following two kinds of aggregates in C :

- Arrays
- Structures

The array can store multiple data items of homogeneous or same type. For example, marks of all 50 students of a class can be stored in an array. On the other hand, structure can store multiple data items of heterogeneous or different types. For example, Different attributes of a student like roll-no, name, address, department, marks etc. can be stored in a structure. The array will be discussed in detail in this unit. Discussion on structure is beyond the scope this syllabus. Arrays are of following two types :

- One dimensional array
- Multi-dimensional array

6.1 Objectives

After going through this unit the learner will be able to :

- Declare and use arrays of one dimension;
- Initialize arrays;
- Use subscripts to access individual array elements;
- Write programs involving arrays;
- Do searching and sorting; and
- Handle multi-dimensional arrays.

6.2 One Dimensional array

One dimensional array has a single dimension, therefore, the elements of the array are conceptually arranged one after another either in a row or a column. The one dimensional array named a with 5 elements is drawn in **Figure 6.1**.

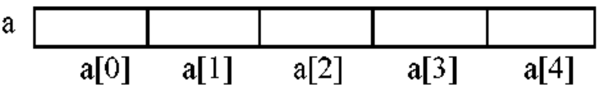


Figure 6.1

6.2.1 Array Declaration

Before using an array, it must be declared like most of the other C objects. To declare an array, following two things need to be specified.

- Type of the array elements
- Size of the array or number of elements in the array.

For example, int a[5] declares an array which can store 5 integers.

6.2.2 Array Subscripting

A particular array element can be accessed by the array name followed by an integer value in square bracket (known as subscripting or indexing the array). In C, the array elements are indexed from 0 to n-1 instead of 1 to n. For example, the elements of array a with 5 elements in **Figure 6.1** are designated by a[0], a[1], a[2], a[3], a[4]. Expressions are of the form a[i] and can be used in the same way as ordinary variables :

```
a[0] = 4;
printf("%d\n",a[5]);
++a[i];
```

The important point that needs to be noted is that it is the responsibility of the programmer to stay inside the subscript bounds (0 to n-1 for array of size n elements) while writing the program. If the subscript goes out of range, the program behavior is undefined; sometimes it may work and sometimes not. Array subscript may have only integer expression. The program fragment written in **Table 6.1** illustrate this fact. An array of size 10 is declared in line 4.

The line no 6 in the left side of **Table 6.1** shows that array index can be an integer expression. When floating expression is used in line 6 in the right side of **Table 6.1** the compiler throws error.

Line #	Code	Line #	Code
1	#include<stdio.h>	1	#include<stdio.h>
2	int main()	2	int main()
3	{	3	{
4	int a[10];	4	int a[10];
5	a[3]=5;	5	a[3]=5;
6	a[2*3]=3;	6	a[2*3.0]=3; ← Error
7		7	printf("%d\t%d",a[3],a[6]);
8	printf("%d\t%d",a[3],a[6]);	8	return 0;
9	return 0;	9	}
	}		
	Output 5 3		**Output** L.. Message === Build file: "no target" in "no proje In function 'main': 6 error: array subscript is not an integer === Build failed: 1 error(s), 0 warning(

Table 6.1

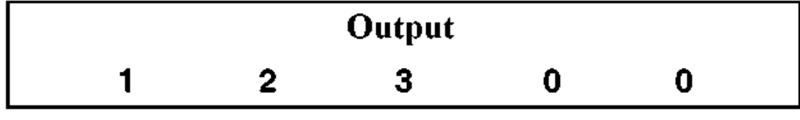
6.2.3 Array Initialization

In C, an array can be initialized at the time of declaration like any variable. The most common way of initializing an array is assigning a list of constant expressions enclosed in braces and separated by commas. For example,

int a[5] = {1, 2, 3, 4, 5};

If the initializer is shorter than the size of an array, then the remaining elements of the array are given the value 0 if the type of the array is int, float or double and null character if the type of the array is character. This is shown in the program output in **Table 6.2**.

```
#include<stdio.h>
int main()
{
    int a[5]={1,2,3};
    printf("%d\t%d\t%d\t%d\t%d",a[0],a[1],a[2],a[3],a[4]);
    return 0;
}
```



Output				
1	2	3	0	0

Table 6.2

All the elements of an array can be reset to zero using this feature.

int a[5]={0};

Sometimes it is necessary to initialize very few elements of an array and most of the other elements are defaulted to zero. In that case, C uses designated initializer.

Suppose following array in **Figure 6.2** has only two non-zero elements (5th and 11th).

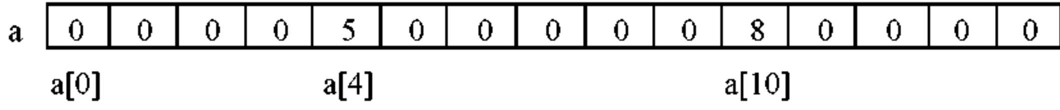


Figure 6.2

For a large array like this it is very tedious and error prone to initialize each element separately. In this situation, the designated initializer can solve the problem very easily. For example, to initialize the array by two non-zero number in 4th and 10th position the following code is sufficient :

int a[15] = {[4] =5, [10] = 8};

Each number in bracket is said to be designator. The other elements are automatically given the value zero. The designated initializer has another advantage. Initialization doesn't depend on the order in which the elements are listed. Therefore, to initialize the array in previous example, following statement also can be used :

int a[15] = {[10] = 8, [4] = 5};

An initializer may use both element by element technique and designated technique. For example, if the statement,

int a[5] = {1,3,[2] = 5 , [3] = 9,7};

is used to initialize the array then compiler executes following step by step process :

1. 1st element (a[0]) is initialized by 1; next element to be initialized is a[1].
2. 2nd element (a[1]) is initialized by 3; next element to be initialized is a[2].
3. Designator [2] initializes 3rd element (a[2]) to 5; next element to be initialized is a[3] (index next to previously initialized index).

4. Designator [3] initializes 4th element (a[3]) to 9; next element to be initialized is a[4].
5. 5th element (a[4]) is initialized by 7 and process stopped at the end of the list.

At the end, the array becomes

a	1	3	5	9	7
	a[0]	a[1]	a[2]	a[3]	a[4]

Example 6.1

Using designated initializer one can initialize an element more than once. Consider the following array declaration :

int a[5] = {1,3,4,[2] = 5, 8,[3] = 9,7};

Is the declaration legal? If so, what is the final content of the array?

Solution :

Step 1, 2, 3 :

1, 3, 4 initializes the elements a[0], a[1] and a[2] respectively in first three steps. After the third step, the next element to be initialized is a[3] or the 4th element of array.

a	1	3	4	uninitialized	uninitialized
	a[0]	a[1]	a[2]	a[3]	a[4]

Step 4 :

The designated initializer [2] force the compiler to initialize a[2] again, even though it is initialized already. Because of this re-initialization the value 4 is replaced by value 5 in a[2]. The next element to be initialized is the one following the element that was last initialized which is a[2].

a	1	3	5	uninitialized	uninitialized
	a[0]	a[1]	a[2]	a[3]	a[4]

Step 5 :

8 initializes a[3] and next element to be initialized is a[4].

a	1	3	5	8	uninitialized
	a[0]	a[1]	a[2]	a[3]	a[4]

Step 6 :

Now even though the element a [4] is supposed to be initialized, the compiler is forced to initialize a[3] because of the presence of designated initializer [3]. a[3] is now re-initialized to 9 and the next element to be initialized is a[4].

a	1	3	5	9	uninitialized
	a[0]	a[1]	a[2]	a[3]	a[4]

Step 7 :

Element a [4] is initialized to 7 and the initialization process terminates. Therefore, the final content of the array is

a	1	3	5	9	7
	a[0]	a[1]	a[2]	a[3]	a[4]

The actual result is given in **Table 6.3**.

```
#include<stdio.h>
int main()
{
   int a[5]={1,3,4,[2]=5,8,[3]=9,7};
   printf("%d\t%d\t%d\t%d\t%d",a[0],a[1],a[2],a[3],a[4]);
   return 0;
}
```

Output

```
1    3    5    9    7
```

Table 6.3

Example 6.2

Write a program that prompts the user to provide five numbers and display the numbers in reverse order.

Solution :

In this case, array can't be initialized by any of the previous initialization method as the numbers are unknown and user provides them at the runtime. Therefore, the program should have the provision to ask for the number to the user repeatedly for five times and store them one after another in the array. This could be achieved easily using any of the loop techniques discussed in the previous unit. **Table 6.4** illustrates the way of initializing array elements using while and for loop. The scanf function stores the value entered by the user in the location specified by &a[i] (ith location of the array a). Every time when the control enters into the loop

the value is stored into the array element. Another point is to be observed is that the loop index starts from 0 (not 1) and end at 4(<5) which satisfy the subscript bound mentioned earlier. Though either of the loops in **Table 6.4** can be used while inserting data into or retrieving data from the array, the for loop is more frequently used. Now to retrieve the data in reverse order the loop index starts from the highest value 4 and gradually decreases to zero as following :

```
for (i=4 ; i>=0 ; i-- )
{
printf("%d\t", a[i]);
}
```

<table>
<tr><th colspan="2">Initialization using while loop</th></tr>
<tr><th>Code</th><th>Output</th></tr>
<tr><td><pre>
#include<stdio.h>
int main()
{
   int a[5],i=0;
   while(i<5)
   {
      printf("\nEnter the element at %d: ",i);
      scanf("%d",&a[i]);
      i++;
   }
   return 0;
}
</pre></td><td>Enter the element at 0 : 4
Enter the element at 1 : 5
Enter the element at 2 : 8
Enter the element at 3 : 2
Enter the element at 4 : 3</td></tr>
</table>

<table>
<tr><th colspan="2">Initialization using for loop</th></tr>
<tr><th>Code</th><th>Output</th></tr>
<tr><td><pre>
#include<stdio.h>
int main()
{
   int a[5],i=0;
   for(i=0;i<5;i++)
   {
      printf("\nEnter the element at %d: ",i);
      scanf("%d",&a[i]);
   }
   return 0;
   }
</pre></td><td>Enter the element at 0 : 4
Enter the element at 1 : 5
Enter the element at 2 : 8
Enter the element at 3 : 2
Enter the element at 4 : 3</td></tr>
</table>

Table 6.4

The program written in Table 6.4 has a major drawback. Since the array is of fixed size 5, the user can only enter at most 5 numbers. To solve this issue, the array must be of variable length size. Unfortunately, the variable length array is only defined in C99 standard. In C99 standard, the program asking for an arbitrary number of values from user is written in **Table 6.5**.

Line #	Code
1	#include<stdio.h>
2	int main()
3	{
4	int i,n;
5	printf("Enter the no of elements: ");
6	scanf("%d",&n);
7	int a[n];
8	for(i=0;i<n;i++)
9	{
10	printf("\nEnter the element at %d: ",i);
11	scanf("%d",&a[i]);
12	}
13	return 0;
14	}

Table 6.5

The program must declare the array a[n] after the initializing variable n (line 7 of **Table 6.5**), otherwise program throws either compilation or runtime error. For C98 compiler, the method doesn't work as the compiler can't allocate memory for the array at runtime (static memory allocation).

There are two alternative ways to solve this problem in C98 compiler which the programmer often uses. They are following :

- Use a simple macro : This method doesn't solve the static memory allocation problem completely but of course gives a better solution than the program given in **Table 6.4**.
- Use system defined function for dynamic memory allocation : This method completely solves the static memory allocation problem. This method mainly uses pointers and therefore, beyond the scope of this syllabus.

Use a simple macro

The definition of simple macro has the form :

```
#define identifier replacement-list
```

A macro's *replacement-list* may include identifiers, keywords, numeric constants, character constants etc. When a program encounters a macro definition, the preprocessor (**section 2.2.3**) makes a note that identifier represents *replacement-list*; wherever identifier appears later in the program, preprocessor substitute the *replacement-list*. With this new definition, let us now analyze the program given in **Table 6.6**.

Line #	Code
1	#include<stdio.h>
2	#define n 5
3	int main()
4	{
5	int a[n],i;
6	for(i=0;i<n;i++)
7	{
8	printf("\nEnter the element at %d: ",i);
9	scanf("%d",&a[i]);
10	}
11	return 0;
12	}

Table 6.6

Before compiling the program, the preprocessor replaces the identifier n by 5 (*replacement list*) in all places inside the main. Therefore, after unfolding the macro defined in line 2 **(Table 6.6)** the program looks like :

Line #	Code
1	#include<stdio.h>
2	int main()
3	{
4	int a[5],i;
5	for(i=0;i<5;i++)
6	{
7	printf("\nEnter the element at %d: ",i);
8	scanf("%d",&a[i]);
9	}
10	return 0;
11	}

and creates an array of 5 elements entered by the user. Whenever user wants to have more numbers, they just need to change the value of n in macro definition at line no 2 of **Table 6.6**. The complete program of reversing the elements of array and the output is given in **Table 6.7**.

Code	Output

`#include<stdio.h>`
`#define n 5`
`int main()`
`{`
`  int a[n],i;`
`  for(i=0;i<n;i++)`
`  {`
`    printf("\nEnter the element at %d: ",i);`
`    scanf("%d",&a[i]);`
`  }`
`  printf("\nYou have entered\n");`
`  printf("------------------------\n");`
`  for(i=0;i<n;i++)`
`    printf("%d ",a[i]);`
`  printf("\nThe reversed numbers are\n");`
`  printf("-------------------------\n");`
`  for(i=n-1;i>=0;i--)`
`    printf("%d ",a[i]);`
`  return 0;`
`}`
	Enter the element at 0: 5
	Enter the element at 1: 6
	Enter the element at 2: 1
	Enter the element at 3: 9
	Enter the element at 4: 3
	You have entered
	------------- 	
	5 6 1 9 3
	The reversed numbers are
	------------- 	
	3 9 1 6 5

Table 6.7

Example 6.3

Write a program that prompts the user to enter an arbitrary number of integers in an array. Display the maximum of the integers.

Solution :

Entering an arbitrary number of integers is already discussed in previous example. Now to display the maximum of the given integers in an array a, following steps are needed :

1. Store the 1st element (a[0]) of the array into an integer variable named max.
2. Compare the next array element with the value of the variable max. Replace the value of max by the array element if max is less than the array element.
3. If no more array element is there to compare then stop the process and print max is the maximum value, otherwise go to step 2.

The flowchart of the above step by step process is given in **Figure 6.3**.

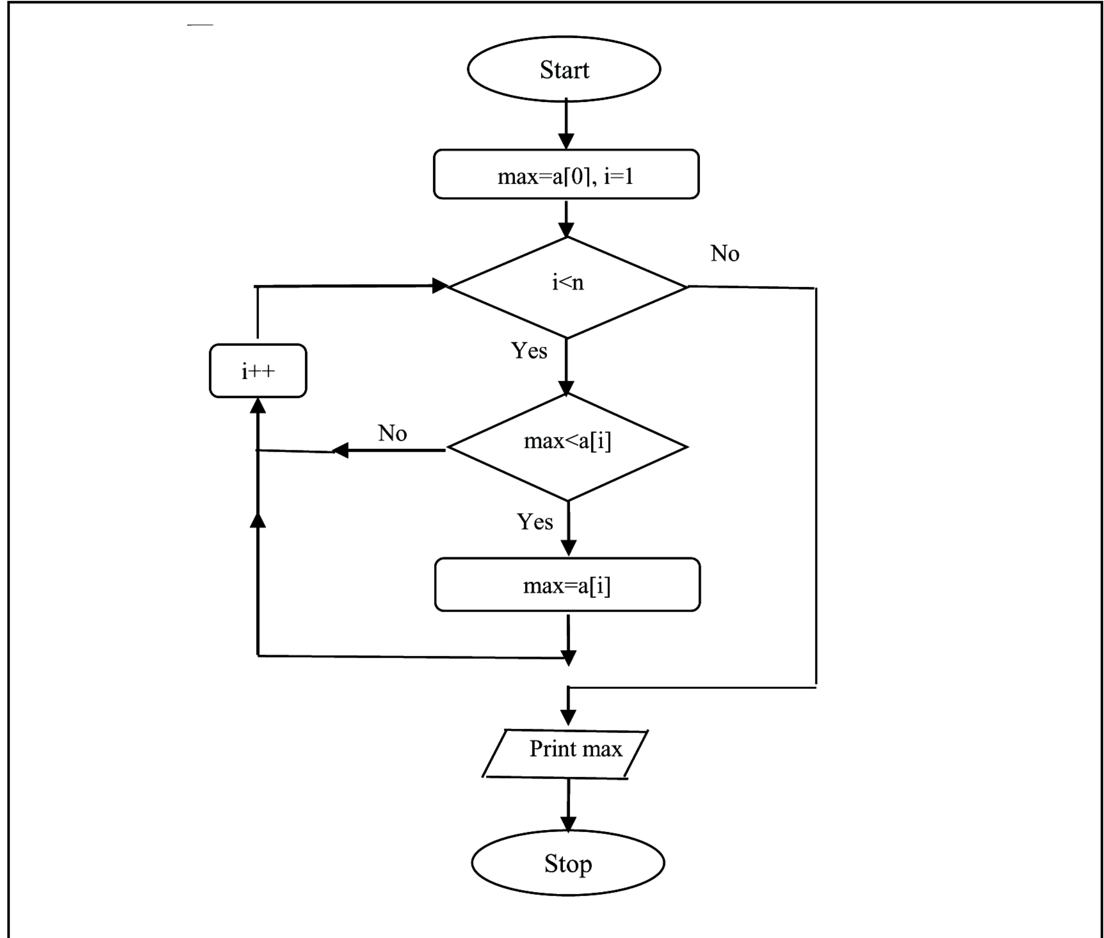


Figure 6.3

The program and its output which follows the flowchart are given in **Table 6.8**.

<table>
<tr><td>

```
#include<stdio.h>
#define n 5
int main()
{
   int a[n],i,max;
   for(i=0;i<n;i++)
   {
      printf("\nEnter the element at %d: ",i);
      scanf("%d",&a[i]);
   }
   max=a[0];
   for(i=1;i<n;i++)
      if(max<a[i])
         max=a[i];
   printf("\nThe maximum number is %d",max);
   return 0;
}
```

</td><td>

```
Enter the element at 0: 3
Enter the element at 1: 2
Enter the element at 2: –9
Enter the element at 3: 14
Enter the element at 4: 3
The maximum number is 14
```

</td></tr>
</table>

Table 6.8

Check Your Progress 6.1

Write a program that prompts user to enter n floating point numbers in an array. Display the sum of the numbers.

Check Your Progress 6.2

The Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, ...where each number is the sum of two preceding numbers. Write a program that declares an array named fib_numb of length 40 and fills the first 40 Fibonacci numbers. Hint: Fill in the first two numbers individually, then use a loop to compute the remaining numbers.

Check Your Progress 6.3

Write a program that can display the digit which is repeated (if any) in a given number.

Sample output :

Enter a number : 343834

Repeated digits : 3(3 times) 4 (2 times)

Hint : Use an array a of 10 elements where a[i] denotes the number of times the digit is repeated in the given number.

Check Your Progress 6.4

Consider two vectors $\vec{a} = \begin{bmatrix} -4 \\ 5 \\ 3 \end{bmatrix}$ and $\vec{b} = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}$. Write a program which can display the vector sum of these vectors. Hint : Use array for every vector.

6.2.4 Sorting the Elements of an Array in Ascending Order

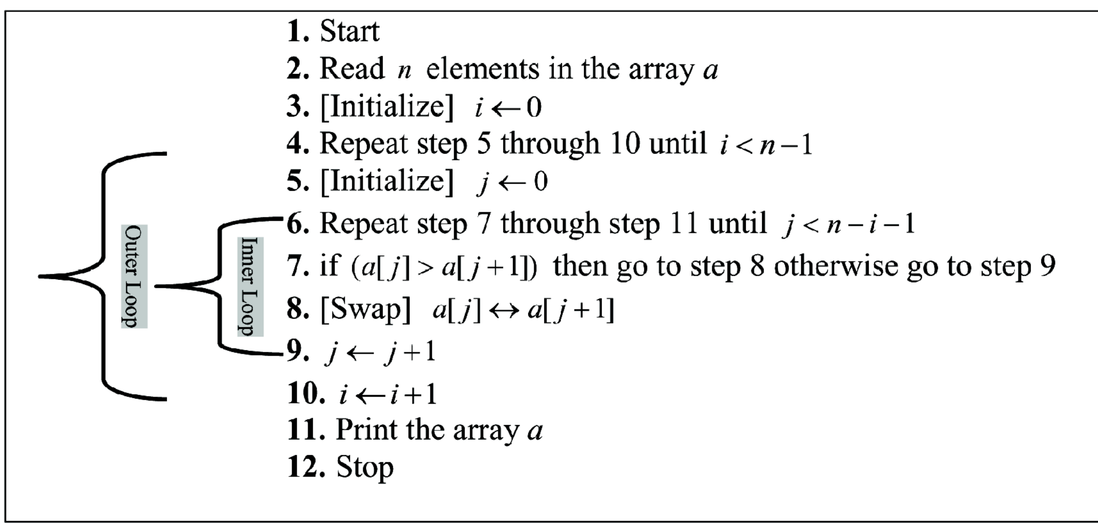
Let us consider an array *a* which contains following 6 elements :

a	*a[0]*	*a[1]*	*a[2]*	*a[3]*	*a[4]*	*a[5]*
	6	1	5	9	3	2

Several algorithms can be designed to sort these elements in assending or descending order. One of such algorithms, known as bubble sort is very popular and simple algorithm which is used to sort a given set of elements stored in an array. Bubble Sort compares all the elements one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap the elements, and then move on to compare the second and the third element, and so on.

If the array contains total n elements, then the above process will be repeated for n-1 times in the first iteration. The buble sort algorithm is given in **Table 6.9**.



1. Start
2. Read n elements in the array a
3. [Initialize] $i \leftarrow 0$
4. Repeat step 5 through 10 until $i < n-1$
5. [Initialize] $j \leftarrow 0$
6. Repeat step 7 through step 11 until $j < n-i-1$
7. if $(a[j] > a[j+1])$ then go to step 8 otherwise go to step 9
8. [Swap] $a[j] \leftrightarrow a[j+1]$
9. $j \leftarrow j+1$
10. $i \leftarrow i+1$
11. Print the array a
12. Stop

Table 6.9

Let us apply the algorithm to sort the numbers stored in array a[6]. The algorithm in **Table 6.9** has two loops. Line 4 to 10 and Line 6 to 9 form the outer and inner loop respectively. The outer loop index i is initialized to 0. For each value of *i*, the inner loop

Value		The Array Configuration
i	*j*	
0	0	⑥ ① 5 9 3 2 ⟹ 1 6 5 9 3 2 Step1: Compare *a[j]* and *a[j+1]*. Since 6>1, swap 6 and 1
0	1	1 ⑥ ⑤ 9 3 2 ⟹ 1 5 6 9 3 2 Step2: Compare *a[j]* and *a[j+1]*. Since 6>5, swap 6 and 5
0	2	1 5 ⑥ ⑨ 3 2 ⟹ 1 5 6 9 3 2 Step3: Compare *a[j]* and *a[j+1]*. Since 6<9, no swaping
0	3	1 5 6 ⑨ ③ 2 ⟹ 1 5 6 3 9 2 Step3: Compare *a[j]* and *a[j+1]*. Since 9>3, swap 9 and 3
0	4	1 5 6 3 ⑨ ② ⟹ 1 5 6 3 2 9 Step3: Compare *a[j]* and *a[j+1]*. Since 9>2, swap 9 and 2

Table 6.10

is executed for *n-i-1* times. **Table 6.10** shows the array elements while executing the inner loop for the value of outer loop index *i* = 0. The inner loop runs for 5 (*n-i-1*)

times. **Table 6.10** also shows that the largest element (9) is placed at the end ($a[5]$) of the array after completion of the inner loop. As a next step, the value of the outer loop index i is incremented and the inner loop execution starts for $i = 1$. This time the second largest element (6) of the array is placed at $a[4]$. Therefore, every complete iteration of the inner loop finally places the largest element in its appropriate position in assending order. Once the outer loop completes its execution the array is completely sorted. **Table 6.11** illustrates the algorithm for other values of outer loop index i. The C implementation and the output of the algorithm is given in **Table 6.12**.

Value of i	Array configuration while execution of the inner loop
1	(1) (5) 6 3 2 9 — No swap ⟹ 1 (5) (6) 3 2 9 — No swap ⟹ 1 5 (6) (3) 2 9 — Swap 6 and 3 ⟹ 1 5 3 (6) (2) 9 — Swap 6 and 2 ⟹ 1 5 3 2 6 9 — Inner loop ends The elements at *a[4] and a[5]* are sorted.
2	(1) (5) 3 2 6 9 — No swap ⟹ 1 (5) (3) 2 6 9 — Swap 5 and 3 ⟹ 1 3 (5) (2) 6 9 — Swap 5 and 2 ⟹ 1 3 2 5 6 9 — Inner loop ends The elements at *a[3], a[4] and a[5]* are sorted.
3	(1) (3) 2 5 6 9 — No swap ⟹ 1 (3) (2) 5 6 9 — Swap 3 and 2 ⟹ 1 2 3 5 6 9 — Inner loop ends The elements at *a[2], a[3], a[4] and a[5]* are sorted.
4	(1) (2) 3 5 6 9 — No swap ⟹ 1 2 3 5 6 9 — Inner loop end The elements at *a[1], a[2], a[3], a[4] and a[5]* are sorted. The remaining element a[0] is the lowest element in the array, located at the begining, therefore the entire array is sorted in ascending order.

Table 6.11

The algorithm in **Table 6.9** is known as bubble sort, because it bubbles up the largest element towards the last place or the highest index, just like a water bubble rises up to the water surface.

C implementation of Bubble sort algorithm

```
#include<stdio.h>
#define n 6
int main()
{
   int a[n] = {6 , 1 , 5 , 9 , 3 , 2} , i , j , temp;
   /* Display the given Array */
   printf ("\nThe given array is\n");
   printf ("-------------------------------------------\n");
   for (i=0 ; i<n ; i++)
   printf ("%d\t", a[i]);
   /* Bubble Sort Algorithm */
   for (i=0 ; i<n-1 ; i++)
   {
      for (j=0 ; j<n-i-1 ; j++)
      {
         /* Swap the elements if a[j]>a[j+1] */
         if(a[j] > a[j+1])
         {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
         }
      }
   }
   /* Display the Sorted Array */
   printf ("\n\nThe sorted array is\n");
   printf ("-------------------------------------------\n");
   for (i=0 ; i<n ; i++)
   printf ("%d\t", a[i]);
   return 0;
}
```

Output :

```
The given array is
6 _ _ _ _1 _ _ _ _5_ _ _ _ _9_ _ _ _ _3 _ _ _ _ _2
The Sorted array is
1 _ _ _ _2 _ _ _ _3_ _ _ _ _5 _ _ _ _6 _ _ _ _ _9
```

Table 6.12

Check Your Progress 6.5

Modify the code for bubble sort algorithm given in **Table 6.12** to sort the array elements in descending order. Use the modified algorithm to sort the following array of 8 elements.

5	14	67	103	11	0	27	45

Check Your Progress 6.6

Assume the following elements of array are sorted in descending order.

9	6	4	2	1	0

How many times the swapping will take place if the algorithm in **Table 6.9** is used to sort the elements in descending order.

6.3 Multidimensional array

Arrays having more than one dimension are known as multidimensional array. To understand the multidimensional array, let us consider about a car manufacturer company having four different stores across the city to sell three different brands of cars. The management of the company want to capture the sell amount to analyze the performance of individual stores. Multidimensional array can be used to design the above model. Each row of the array represents a store and each column represents a specific brand of car. **Figure 6.4** shows an array *a*[4][3] which stores the car selling information of the company.

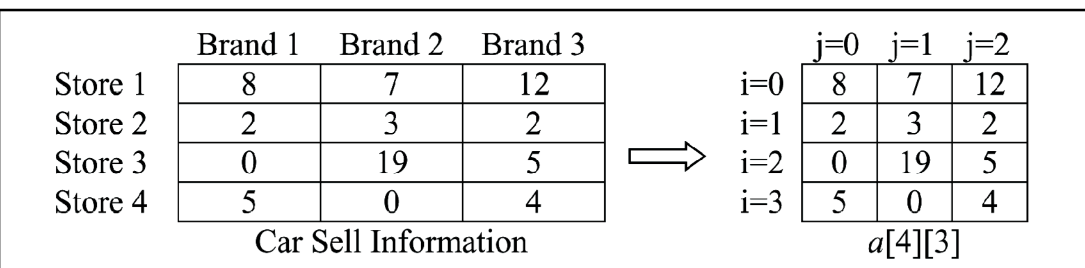


Figure 6.4

The array a is an example of two dimensional array as it has exactly two dimensions store and brand. To access the element of the array *a* in row i and column

j, one must write *a*[i][j]. If one more dimension say, year is added to the model **(Figure 6.5)** then a three dimensional array will be needed to store the information. The model in **Figure 6.5** can be represented by an three dimensional array b[4][3][3].

Most of the topics of this unit has been explained considering the number of dimensions as two, nevertheless it is easy to extend all of them in case of more dimensions.

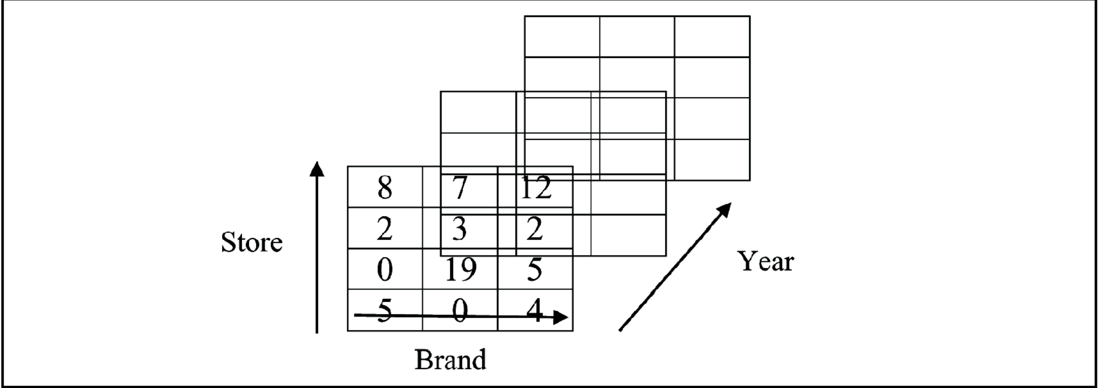


Figure 6.5

6.3.1 Physical View and Logical View of an Array

Although the two dimensional arrays are visualized as tables, that is not the way they're actually stored in computer memory. C stores arrays in row major order, with row 0 first, then row 1, and so forth. For example, **Figure 6.6** shows how array a in **Figure 6.4** is stored.

8	7	12	2	3	2	0	19	5	5	0	4
a[0][0]	*a*[0][1]	*a*[0][2]	*a*[1][0]	*a*[1][1]	*a*[1][2]	*a*[2][0]	*a*[2][1]	*a*[2][2]	*a*[3][0]	*a*[3][1]	*a*[3][2]

Figure 6.6

The array a[4][3] needs 12 (4x3) contiguous location in the memory. Now the indexing of these 12 elements defines the row and column number while inserting elements into the array locations. For example, **Table 6.13** shows the C program to insert integers into the array of size 12 and display them in the form of a 4x3 matrix.

Let us analyze the program to understand array indexing technique. At line 4, once the array is declared, 12 consecutive memory location is allocated to the array.

The program uses two loop index i and j. The outer loop index i is used for the rows and the inner loop index j is used for the columns. **Table 6.14** shows the step by step process how the array index is constructed using the C instructions given in line 6 to 13.

Program	Output

#include<stdio.h>
int main()
{
int a[4][3],i,j;
printf("Enter array elements\n");
for(i=0;i<4;i++)/*Array inserts*/
{
for(j=0;j<3;j++)
{
printf("Enter the number: ");
scanf("%d",&a[i][j]);
}
}
printf("\nThe array in matrix form\n");
printf("-----------------------\n");
for(i=0;i<4;i++) /* Array display */
{
for(j=0;j<3;j++)
{
printf("%d ",a[i][j]);
}
printf("\n");
}
return 0;
}
	Enter array elements
	Enter the number: 1
	Enter the number: 2
	Enter the number: 3
	Enter the number: 4
	Enter the number: 9
	Enter the number: 8
	Enter the number: 7
	Enter the number: 1
	Enter the number: 0
	Enter the number: 4
	Enter the number: 0
	Enter the number: 4
	
	The array is matrix form
	— — — — — — — — — —
	1 2 3
	4 9 8
	7 1 0
	4 0 4

Table 6.13

Step	Value of i	Value of j	scanf("%d",*a*[i][j]);	Index	Data
1	0	0	scanf("%d",*a*[0][0]);	*a*[0][0]	1
2		1	scanf("%d",*a*[0][1]);	*a*[0][1]	2
3		2	scanf("%d",*a*[0][2]);	*a*[0][2]	3
Inner loop ends , i incremented to 1, j is reset to 0					
4	1	0	scanf("%d",*a*[1][0]);	*a*[1][0]	4
5		1	scanf("%d",*a*[1][1]);	*a*[1][1]	9
6		2	scanf("%d",*a*[1][2]);	*a*[1][2]	8
Inner loop ends , i incremented to 2, j is reset to 0					
7	2	0	scanf("%d",*a*[2][0]);	*a*[2][0]	7
8		1	scanf("%d",*a*[2][1]);	*a*[2][1]	1
9		2	scanf("%d",*a*[2][2]);	*a*[2][2]	0
Inner loop ends , i incremented to 3, j is reset to 0					
10	3	0	scanf("%d",*a*[3][0]);	*a*[3][0]	4
11		1	scanf("%d",*a*[3][1]);	*a*[3][1]	0
12		2	scanf("%d",*a*[3][2]);	*a*[3][2]	4

Table 6.14

The important point that needs to be noted is that in each step while inserting data into the array the array index entry is created. As far as the physical construction in concerned, array is still a 12 contiguous memory locations after the insertion of data. The code actually creates the index in such way, so that the array can be displayed in the format required by the user only using the loop index variable i and *j*. For example, **Table 6.9** illustrates this fact in line 16–23 while displaying the elements. For a fixed value of i in the outer loop, the program prints the entries in *a[i][j]* for all *j* using inner loop. Once the inner loop is terminated, a new line is printed (line 22) before entering into outer loop for the next value of *i*. This new line character differentiates two rows when the array is displayed in matrix form.

6.3.2 Initializing Multidimensional Array

Initializer can be created for a two dimensional array by nesting one dimensional initializers. For example, the two dimensional array *a* in **Figure 6.4** can be initialized by the following statement :

int *a*[4][3] = {{8, 7, 12}, {2, 3, 2}, {0, 19, 5}, {5, 0, 4}};

Each inner initializer provides values for one row of the array. Initializers for higher dimensional arrays can be constructed in similar fashion. Following rules are applicable while initializing a large array :

1. If an initializer is not large enough to fill a multidimensional array, the remaining elements are given the value 0. For example, the initializer in **Table 6.15** fills only the first two rows of a; the last row will contain all 0's.

Code	Output

<pre>#include<stdio.h>
int main()
{
   int a[3][3]={{1,4,3},{0,1}},i,j;
   printf("The array is\n");
   printf("----------------\n");
   for(i=0;i<3;i++)
   {
      for(j=0;j<3;j++)
      {
         printf("%d ",a[i][j]);
      }
      printf("\n");
   }
   return 0;
}</pre>
	The array is
	— — — — — — — — — — — — —
	1 4 3
	0 1 0
	0 0 0

Table 6.15

2. If the inner list is not long enough to fill a row, the remaining elements are initialized to 0. For example, the initializer used in line 4 of the C program in **Table 6.15** has only two elements in 2nd row. The 3rd element is initialized to 0 by default.

Check Your Progress 6.7

Write a program that declares an 8x8 char array named checker_board and then uses a loop to store the following data into the array (one character per array element) :

```
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
```

Hint : The element in row i, column j, should be the letter B if i + j is an even number.

Check Your Progress 6.8

Write a program that generates an $n \times n$ identity matrix. Use for loop to initialize the matrix instead of initializer.

Sample Output :
Enter the value of n : 3
The 3x3 identity matrix is :

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

6.4 Summary

C uses arrays to describe a collection of variables with identical (homogeneous) properties. The group has a single name for all its members, with the individual member being selected by an index. This unit shows the basic purpose of using an array in the program, declaration of array and assigning values to the arrays. All elements of the arrays are stored in the consecutive memory locations. Without exception, all arrays in C are indexed from 0 up to one less than the bound given in the declaration. One important point about array declarations is that they don't permit the use of varying subscripts in C98 standard. The numbers given must be constant expressions which can be evaluated at compile time, not run time (static memory allocation). Though this limitation is not there in C99 compiler which supports varying length array. The unit also explains the techniques of declaring, initializing, accessing multi-dimensional arrays.

6.5 References and Further Reading

1. The C Programming Language, Kernighan & Ritchie, PHI Publication, 2011.
2. Programming with C, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.

3. The C Complete Reference, Fourth Editon, Herbert Schildt, Tata McGraw Hill, 2002.
4. C Programming : A Modern Approach, Second Edition, K.N. King, W. W. Norton & Company, 2008.
5. Computer Science: A Structured Programming Approach Using C, Second Edition, Behrouz A. Forouzan, Richard F. Gilberg, Brooks/Cole Thomas Learning, 2001.
6. The C Primer, Leslie Hancock, Morris Krieger, McGraw Hill, 1983.

Unit - 7 ❑ Application of C Programming : Solution of Non-Linear Equations

Structure

7.0 Introduction

7.1 Objectives

7.2 Bisection Method

- **7.2.1 Method Description**
- **7.2.2 Algorithm and Implementation Issue :**
- **7.2.3 Implementation : Bisection Method**

7.3 Regula-falsi Method or Method of false-position

- **7.3.1 Method Description**
- **7.3.2 Algorithm and Implementation Issue :**
- **7.3.3 Implementation : Regula Falsi Method**

7.4 Secant Method

- **7.4.1 Method Description**
- **7.4.2 Algorithm and implementation Issue :**
- **7.4.3 Implementation : Secant Method**

7.5 Newton Raphson Method

- **7.5.1 Method Description**
- **7.5.2 Algorithm and Implementation Issue :**
- **7.5.3 Implementation : Newton Raphson Method**

7.6 Summary

7.7 References and Further Reading

7.0 Introduction

You cannot teach a man anything; you can only help him discover it in himself. –

—Galileo

This unit is designed to show the usage of C programming in solving non-linear equations using different techniques which are taught in the undergraduate mathematics, engineering and other relevant courses. Therefore, the learners are

expected to have solid background of basic calculus. As a prerequisite, the learner needs to revisit the Calculus - CC3 taught in first semester and Numerical analysis - CC5 taught in third semester of Bachelor Degree Program in Elective Mathematics (EMT) under Netaji Subhas Open University.

The main objective of this unit is to find the roots of the equation of the form

$$f(x) = 0 \qquad \textbf{(7.1)}$$

i.e., zeros of the function $f(x)$. In most of the cases it is very difficult to obtain an exact root of the equation **(7.1)**. Therefore, it is quite natural to seek for a solution which is approximate in nature. The approximate solution may then mean either a point x^*, for which equation **(7.1)** is approximately satisfied, i.e., for which $|f(x^*)|$ is small, or a point x^* which is close to a solution of **(7.1)**. Moreover, an approximate solution obtained on a computer will almost always be in error due to round off or instability or due to the particular arithmetic used. Indeed, there may be many approximate solutions which are equally valid even though the required solution is unique. This is definitely a drawback of using computer while solving non-linear equations, nevertheless, the computational techniques have become indispensable for solving complex numerical problems in modern days because of its speed and memory capacity. A number of iterative methods to find out the approximate solution of the equation 7.1 will be discussed in this unit. These methods have the following basic strategy :

Step 1 : Guess an initial solution

Step 2 : If the solution achieve the desired accuracy then stop the process, otherwise go to next step.

Step 3 : Improve the solution and go to step 2.

The above strategy is iterative as it expects the solution to converge after finite no of iterations. In this unit, the following iterative methods will be discussed in detail :

- Bisection method
- Regula falsi or False-position method
- Secant Method
- Newton Rapson Method

7.1 Objectives

After going through this topic, the learner should able to

- Explain the difference between closed domain (bracketing methods) and open domain methods (non-bracketing methods)

- Use parameterized macro
- Explain how the internal halving (bisection) method works
- Write C program for bisection method
- Explain how the false position (regula falsi) and secant methods work
- Write C program for regular falsi and secant methods
- Explain how Newton Raphson method works
- Write C program for Newton Raphson method
- Visualize different methods and theorem with the help of graphs and data.

7.2 Bisection Method

The first technique used here is known as bisection method which is based on intermediate value theorem.

7.2.1 Method Description

Suppose f is a continuous function defined on the interval $[a, b]$, with $f(a)$ and $f(b)$ of opposite sign. The intermediate value theorem implies that a number p exists in $[a, b]$, with $f(p) = 0$. Even though the procedure will work for the equation having more than one root, for simplicity let us assume the root is unique in the interval $[a, b]$. The method calls for a repeated halving (or bisecting) of subintervals of $[a, b]$, and, at each step, locating the half containing p. Even though the bisection method can't find the root if $f(a)$ and $f(b)$ of same sign but that doesnot necessarily mean that the equation has no root. In those cases, the equation may have even number of roots **(Figure 7.1)**.

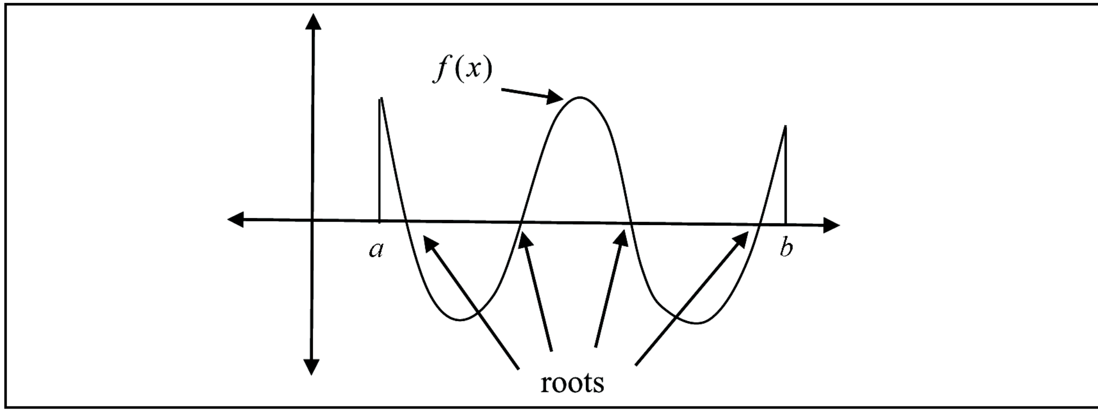


Figure 7.1

Let us illustrate this with an example of a simple polynomial equation

$$f(x) = x^3 - x - 3 \tag{7.2}$$

with an initial interval $[a_1, b_1]$ where $a_1 = 1$ and $b_1 = 3$.

It can be seen that, $f(1) = -3 < 0 < f(3) = 21$. Since $f(x)$ is continuous, $f(x)$ must vanishes in the interval [1, 3]. Now the interval [1, 3] will be divided into two equal parts. To do that, let us find out the midpoint of the interval $p_1 = 2$. So the new intervals, those need to be investigated for the desired root, are $[a_1, p_1] \equiv [1, 2]$ and $[p_1, b_1] \equiv [2, 3]$. Now according to the method the interval needs to be chosen in such a way that the values at two end points are of opposite signs. Clearly, the next interval [1, 2] needs to be investigated since $f(1) = -3$ and $f(2) = 3$ are of opposite signs. Therefore, the new value of left end point is $a_2 = 1$ and new value of right end point $b_2 = 2$ in the next iteration. Now the same interval halving process is to be continued until the condition $f(p_1) \approx 0$ is satisfied. The value of $f(p_1)$ is decreasing at every iteration and finally becomes 0. Mathematically speaking, the sequence $f(p_1), f(p_2), f(p_3), \ldots\ldots\ldots\ldots, f(p_n)$ converges to limit 0 as $n \to \infty$. In that case, p_n is the desired root. **Figure 7.2** illustrates this process up to third iteration using the graph of the function $f(x) = x^3 - x - 3$.

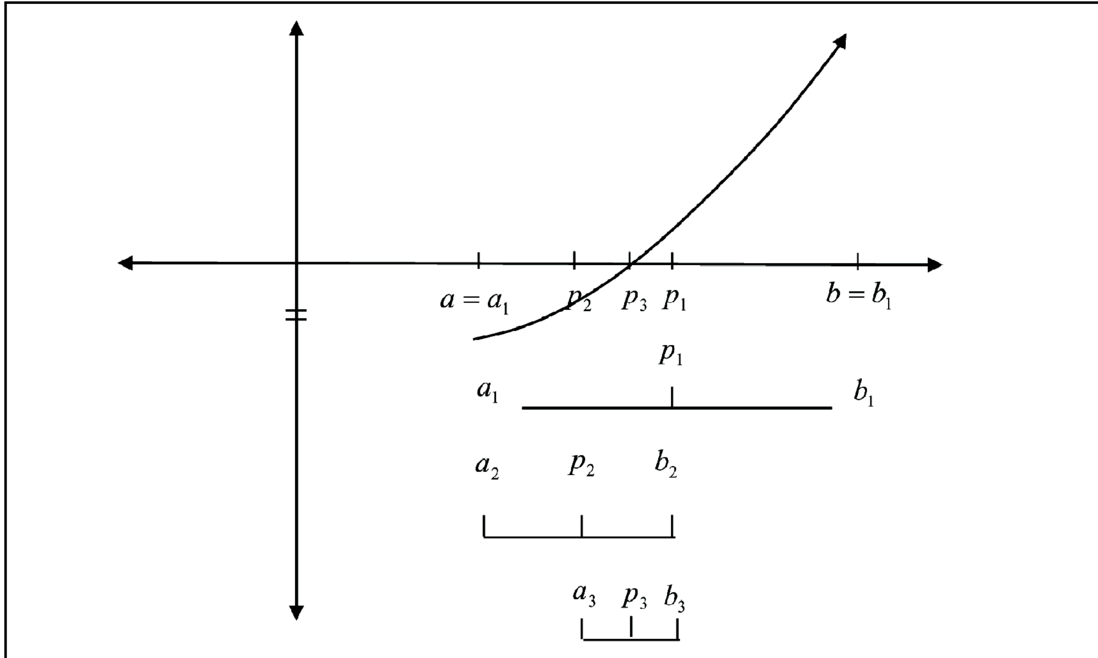


Figure 7.2

Here the algorithm terminates when the value $f(x)$ is close to zero (not exactly zero) even when the solution exist. This is because of the limited precession of the floating point numbers while representing them in binary format. For example, the number 0.1 can't be represented exactly in binary (similarly as fraction $^1/_3$ can't be represented exactly in decimal).Therefore, floating points are normally rounded off to the nearest value while arithmetic operations are performed on them. This generates a certain amount of error which may finally be quite large and produce different result than the actual one. Because of this, instead of comparing the values of two floating point numbers for equality, it is better checking whether their difference is within some error bound or not. Therefore, the condition $f(p_i) \approx 0$ needs to be written as $-.001 <= f(p_i) <= 0.001$ assuming the error bound or accuracy is up to 2 decimal places in this case.

7.2.2 Algorithm and Implementation Issue :

Algorithm 7.1 : Bisection Method

Input : Function $f(x)$, end points of the interval a, b ; accuracy or tolerance t.

Output : Approximate solution p or message of failure.

Steps	*Description*
1.	*Read a, b and t*
2.	*if f (a) and f (b) have same sign, print "root cannot be found" and go to step 8.*
3.	*if* $\|f(a)\| < t$ *then* $p = a$ *and go to step 8.*
4.	*if* $\|f(b)\| < t$ *then* $p = b$ *and go to step 8.*
5.	$p = a + (b - a)/2.$
6.	*if f (a) and f (p) have opposite sign then* $b = p$, *otherwise* $a = p$.
7.	*if* $\|f(p)\| > t$ *then go to step 5.*
8.	*Print the root p.*
9.	*Stop.*

Other stopping criteria can be applied in Step 6 of **Algorithm 7.1** or in any of the iterative techniques in this chapter. For example, we can select a tolerance $t > 0$ and generate $p_1, p_2, p_3, \ldots\ldots\ldots\ldots p_n$ until the following condition is met :

$$|p_n - p_{n-1}| < t \tag{7.3}$$

The flow chart of the algorithm is given in **Table 7.1**. Now to convert the above algorithm into C program following problems may arise :

1. How to provide a function like $f(x) = x^3 - x - 3$ in C program?
2. How to determine the absolute value of a number (integer or floating point)? Absolute value is required in several places of the algorithm 7.1.
3. At the time of halving the interval, the formula used is $p = a + (b - a) / 2$ (Step no 4) instead of $p = (a + b) / 2$. Is there any benefit of using this in the context of C?

To solve problem 1, there are several options in C that can be implanted. Among them functions are one of the best. A user defined function for $f(x) = x^3 - x - 3$ can be designed. Since user defined functions are out of the scope of this syllabus an alternate way can be applied which is known as parameterized macro. As of now we used simple macro in previous unit. The macro can also be parameterized.

Parameterized Macro

The definition of a parameterized macro (also known as function-like macro) has the form #define *identifier* (x_1, x_2, ...x_n) *replacement list*

where x_1, x_1, ...x_n are the parameters of the macro. The parameters may appear as many times as desired in the *replacement list*. The important point that needs to be noted that there must be no space between the *identifier* (macro name) and left parenthesis. If space is left the processor will assume it a simple macro and will treat (x_1, x_1, ...x_n) as a part of *replacement list*. Therefore, $f(x) = x^3 - x - 3$ can be written using a parameterized macro as the root of the function will be not found.

#define $f(x)$ $x^* x^* x - x - 3$

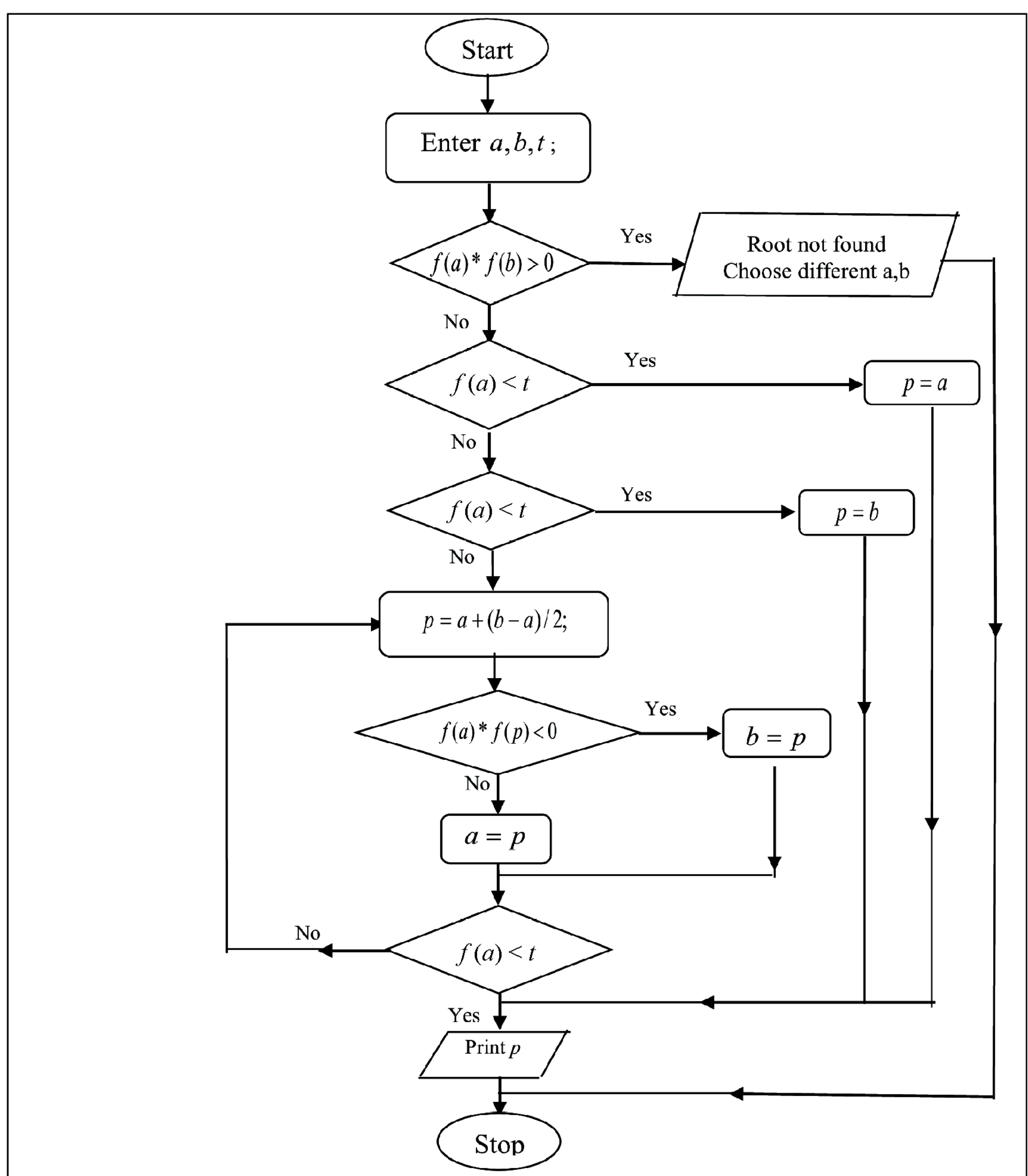


Table 7.1

Example 7.1

What is the output of the program in **Table 7.2**?

Line no	Code
1	#include<stdio.h>
2	#define print(n) printf("The value is %d",n)
3	int main()
4	{
5	int i=1;
6	print(i);
7	return 0;
8	}

Table 7.2

Solution :

The program uses a parameterized macro print(n) in line 2 and invoke the macro from the program using print(i) statement in line no 6. The preprocessor first replace the macro print(i) by the replacement list printf (“The value is %d”,i) and then the code is compiled. Therefore, the output becomes “The value is 1”.

Example 7.2

What is the value of j and k after executing the following code?

```
#include<stdio.h>
#define f(x) x+2
int main()
{
   int i=3,j,k;
   j=f(i);
   k=f(i)*5;
   return 0;
}
```

Solution :

```
j = f(i)
  = i+2 [definition of macro f(x)=x+2]
  = 3+2 [i=3]
  = 5
k = f(i)*5
  = i+2*5 = 3+2*5 = 13
```

It may appear that the value of k = 25. But, actually the macro f(i) is not evaluated at first, rather replaces its definition by i+2. After this replacement the evaluation starts. Therefore, the value of k finally becomes 13. To get the value of k is 25, the macro needs to be redefined as #define f(x) (x+2), then k = f(i)*5 =(i+2)*5=(3+2)*5=25.

Example 7.3

Suppose a macro is defined in the following way to represent $f(x) = x^3 - x - 3$:

```
#define f(x) x*x*x-x-3
```

What is the value of f(1)*f(2)? What is the result if the macro definition is changed to :

```
#define f(x) (x*x*x-x-3)
```

To solve the problem 2 in **Table 7.1** a new library function is needed. C has several library functions to determine the absolute value based on the type of the number for which absolute value is to be determined. Two commonly used functions are :

- abs(argument)
- fabs(argument)

The abs() function is used when the argument is an integer and fabs() function is used when the argument is floating point number. In most of the cases in numerical analysis, the fabs() function is used as it can handle both float and integer. Any program which use fabs() function, need to add the header file <math.h> as a preprocessor directive as the fabs() function is defined under <math.h> file.

The formula $p = a + (b - a) / 2$ (Step no 4 in **Algorithm 7.1**) is used instead of $p = (a + b) / 2$ to reduce the round off error. For example, consider a machine which can handle arithmetic calculation no more than 4 digits. Let $a = 56.35$ and $b = 68.27$ are two numbers. To determine the average of these two numbers using the formula $(a + b) /2$, first the addition takes place. The result of addition $(a + b)$ will produce a number 124.62 which is rounded to124.6, the nearest 4-digit number. Then after dividing this result by 2, the average is determined as 62.3. On the other hand, the formula $a + (b - a) /2$. produce the average as 62.31. The later result is more accurate since no round off error has been produced by the arithmetic operation $a + (b - a) /2$.

Another important issue of the **Algorithm 7.1** is to stop the algorithm when both $f(a)$ and $f(b)$ are of same sign (Step 1). One way to solve this is to use return

statement in main function. Another way is to call the exit function, which belongs to <stdlib.h> header file. Argument passed to exit has the same meaning as main's return value: both indicate the program status at termination. To indicate normal termination 0 is passed.

exit(0); /*Normal termination*/

In the bisection method, the exit function is used to terminate the program when $f(a)$ and $f(b)$ are of same sign (Step 1).

When using a computer to generate approximations, it is good practice to set an upper bound on the number of iterations. This eliminates the possibility of entering an infinite loop, a situation that can arise when the sequence diverges (and also when the program is incorrectly coded). Therefore, the do-while loop of the program contains an additional condition on maximum number of iterations.

The C implementation of bisection method has been given in **Table 7.3**. The macro defined in this program uses a C library function pow() which is defined in the math.h header file. The pow() function takes two arguments (base value and power value) and, returns the power raised to the base number. For Example, x^y in mathematics can be used as $pow(x, y)$ in C programming. Therefore, the given function $x^3 - x - 3$ can be defined by $pow(x, 3) - x - 3$ using a parameterized macro in C program. Choosing $a = 1$, $b = 3$, $t = 0.01$, $n = 20$ and $f(x) = x^3 - x - 3$, the above program produces the output which is given in **Table 7.4**. The output shows that the root is 1.67 which takes 7 iterations to determine the desired root accurate up to 2 decimal point. The output numbers in **Table 7.4** have only 3 digits after decimal point. Changing the format of floating point more digits can be accommodated (refer **Section 3.7.1**).

7.2.3 Implementation : Bisection Method

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define f(x) (pow(x,3)-x-3) /* Function f(x) */
int main()
{
   float a,b,p,t;
```

```
    int i=0,n;
    printf("Enter the value of a,b,eps: ");
    scanf("%f%f%f",&a,&b,&t);
    printf("\nHow many maximum no of iterations you need? ");
    scanf("%d",&n);
    if (f(a)*f(b)>0) /*Check the initial interval chosen properly*/
    { printf("Root can't be found using bisection method");
       exit(0);
    }
 if(fabs(f(a))<t) /*Checking for root at the left end of initial interval*/
          p=a;
    else if (fabs(f(b))<t) /*Checking for root at the right end of initial interval*/
          p=b;
    else
    { printf("\nIter\t a\t b\t p\t f(a)\t f(b)\t f(p)\n");
       printf("--------------------------------------------------------\n");
       do /*Iterative method to find the root*/
       { i++;
          p=a+(b-a)/2;
          printf("%d\t%6.3f\t%6.3f\t%6.3f\t%6.3f\t%6.3f\t%6.3f\n",i,a,b,p,f(a),
          f(b),f(p));
          if (f(a)*f(p)<0)
             b=p;
          else a=p;
       } while (fabs(f(p))>t && i<n);
    }
    if(i==n)
    {
       printf("\nExceeds maximim number of iterations");
       exit(0);
    }
    printf("\n\nThe desired root is %6.2f",p);
    return 0;
 }
```

Table 7.3

Enter the value of a,b, eps: 1 3 .01

How many maximum no of iteration you need? 20

Iter	a	b	p	f(a)	f(b)	f(p)
1	1.000	3.000	2.000	–3.000	21.000	3.000
2	1.000	2.000	1.500	–3.000	3.000	–1.125
3	1.500	2.000	1.750	–1.125	3.000	0.609
4	1.500	1.750	1.625	–1.125	0.609	–0.334
5	1.625	1.750	1.688	–0.334	0.609	0.118
6	1.625	1.688	1.656	–0.334	0.118	–0.113
7	1.656	1.688	1.672	–0.113	0.118	0.001

The desired root is 1.67

Table 7.4

Check Your Progress 7.1

Use the program in **Table 7.5** to find an approximate value of x accurate up to 5 decimal point in [0.5, 1.5] which satisfy the equation $e^x - 2\cos(e^x - 2)$. Hint : e^x and $\cos(x)$ are written as exp(x) and cos(x) in <math.h>.

Check Your Progress 7.2

Find an approximation to $\sqrt{3}$ correct up to 4 decimal point using the Bisection Algorithm. [Hint : Consider $f(x) = x^2 - 3$.]

Example 7.4

Use the program in **Table 7.3** to find an approximate solution (accurate up to 2 decimal point) of the equation $(x - 1)^{15} = 0$.

Solution :

The approximate solution must be very close to 1 as the polynomial has factor $(x - 1)$. Let us run the program in **Table 7.5** just by changing the macro definition of the function. Let us also choose the value of $a = -2$ and $b = 3$ which satisfy the initial criteria of bisection method. The output is shown in **Table 7.5**.

Surprisingly, the root produced in **Table 7.5** is 0.5 which is far away from the actual root 1. The algorithm is stopped because at $p = 0.5$ the value satisfies the stopping criteria $f(p) = (0.5 - 1)^{15} = (-0.5)^{15} = .000030517 \approx 0$.

```
Enter the value of a, b, eps : -2 3 .01
How many maximum no of iterations you need? 20
Iter   a        b       p       f(a)             f(b)        f(p)
------------------------------------------------------------------
1      -2.000   3.000   0.500   -14348907.000    32768.000   -0.000
The desired root is   0.50
```

Table 7.5

To analyze the problem let us look into the graph of the function $f(x) = (x - 1)^{15}$ in **Figure 7.2**. It shows that during the entire interval [0.5, 1.5] the value of the

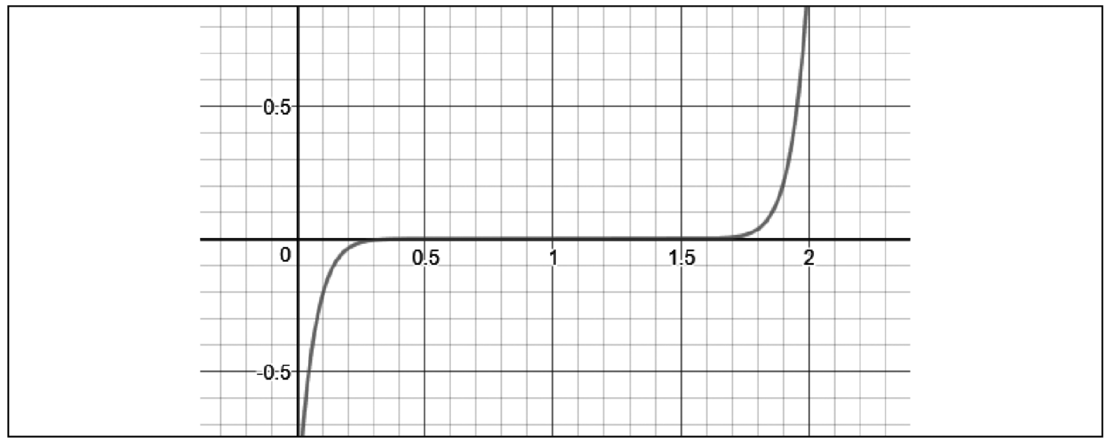


Figure 7.3

function $f(x) = (x - 1)^{15}$ has the value very close to 0. Therefore, in this case the stopping criteria $|f(p)| \leq t$ will not work. Instead of that, the stopping criteria $|p_n - p_{n-1}| < t$ (given in equation 7.3) may work better here. For this, every time after entering into the do-while loop the current value of p needs to be compared with previous value of p. Therefore, two variables p_0 and p_1 will be created to denote old value and current value of p in the program. The modified program is given in **Table 7.6**.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define f(x) pow(x-1,15) /* Function f(x) */
int main()
```

```
{
    float a,b,p0,p1,t;
    int i=0,n;
    printf("Enter the value of a,b,eps: ");
    scanf("%f%f%f",&a,&b,&t);
    printf("\nHow many maximum no of iterations you need? ");
    scanf("%d",&n);
    if (f(a)*f(b)>0) /*Check the initial interval chosen properly*/
    { printf("Root can't be found using bisection method");
       exit(0);
    }
    if(fabs(f(a))<t) /*Checking for root at the left end of initial interval*/
       p1=a;
    else if (fabs(f(b))<t) /*Checking for root at the right end of initial interval*/
          p1=b;
    else
    { printf("\nIter a\t b\t p\t f(a)\t f(b)\t f(p)\n");
       printf("-----------------------------------------------------------------\n");
       p1=b; /* p is initialized to right end of the interval*/
       do /*Iterative method to find the root*/
       { i++;
       p0=p1; /*p0 contains the previous value of p*/
       p1=a+(b-a)/2; /* New value of p */
printf("%d\t%6.3f\t%6.3f\t%6.3f\t%12.3f\t%12.3f\t%6.3f\n",i,a,b,p1,f(a),f(b),f(p1));
       if (f(a)*f(p1)<0)
          b=p1;
       else a=p1;
       } while (fabs((p1-p0)/p1)>t && i<n);
    }
    if(i==n)
    {
       printf("\nExceeds maximim number of iterations");
       exit(0);
    }
    printf("\n\nThe desired root is %6.3f",p1);
    return 0;
}
```

Table 7.6

Now this modified program produces more accurate result as given in the **Table** 7.7.

Enter the value of a, b, eps: –2 3 .01

How many maximum no of iterations you need? 20

Iter	a	b	p	f(a)	f(b)	f(p)
1	–2.000	3.000	0.500	–14348907.000	32768.000	–0.000
2	0.500	3.000	1.750	–0.000	32768.000	0.013
3	0.500	1.750	1.125	–0.000	0.013	0.000
4	0.500	1.125	0.813	–0.000	0.000	–0.000
5	0.813	1.125	0.969	–0.000	0.000	–0.000
6	0.969	1.125	1.047	–0.000	0.000	0.000
7	0.969	1.047	1.008	–0.000	0.000	0.000
8	0.969	1.008	0.988	–0.000	0.000	–0.000
9	0.988	1.008	0.998	–0.000	0.000	–0.000

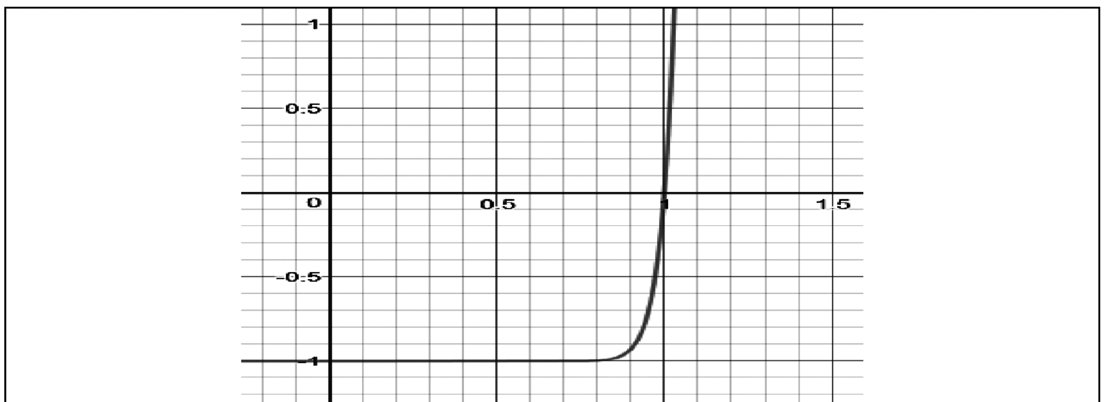
The desired root is 0.998

Table 7.7

Therefore, we can easily conclude that the stopping criteria should include both $|p_n - p_{n-1}| < t$ and $|f(p_n)| < t$ together to take care of all type of functions.

Check Your Progress 7.3

Use the programs in **Table 7.3** and **Table 7.6** to find an approximate solution (accurate up to 2 decimal point) of the equation $x^{25} - 1 = 0$ separately with the initial conditions $a = -4$, $b = 4$ and $n = 20$. Which program gives more accurate result? Explain graphically. The graph for the function is given below.



Check Your Progress 7.4

Modify the program of bisection method by including both stopping criteria $|p_n - p_{n-1}| < t$ and $|f(p_n)| < t$ where p_n, $f(p_n)$, t have their usual meaning.

Even though the Bisection method is conceptually simpler, it has significant drawbacks. It is relatively slow to converge. However, the method has the important property that it always converges to a solution, and for that reason it is often used as a starter for the more efficient methods which will be discussed later in this chapter.

7.3 Regula-falsi Method or Method of false-position

7.3.1 Method Description

The bisection method presented in the last section was based on the procedure which continuously divides the initial interval until it reaches to the approximated root. This method never relies on the nature of the function. As a result, it uses same interval halving method irrespective of the nature of the function. Let us take an example, $f(x) = x^3 - x - 15 = 0$ to investigate how the process changes if function is assumed to be approximately linear in the local region of interest. Starting with initial interval [1, 3] we can write,

$$f(1) = -15 \leq 0 \leq f(3) = 9$$

Since $|f(3)|$ is closer to zero than $|f(1)|$, the root is likely to be closer to 3 than to 1 (assuming $f(x)$ is linear in the interval). Hence instead of the midpoint, or average value 2 (average of 1 and 3), the weighted average (w) can be considered. More weight should be given to 3 than 1 to push the solution towards 3. Therefore, weighted average of 1 and 3 can be calculated by the following formula

$$w = \frac{|f(3)| * 1 + |f(1)| * 3}{|f(3)| + |f(1)|} \tag{7.4}$$

Now since $f(1)$ and $f(3)$ have opposite sign, the formula can be simplified as

$$w = \frac{f(3) * 1 - f(1) * 3}{f(3) - f(1)} \tag{7.5}$$

This gives, for the above example,

$$w = \frac{9 * + 1 + 15 * 3}{9 + 15} = 2.25$$

and $f(w) = -5.859$. Therefore, the root now lies in the interval [2.25, 3]. This process now continues until the desired accuracy is reached. The point to be noticed is that after iteration 1, the bisection method gives the midpoint value as 2 while this method gives 2.25 which is closer to the actual root 2.6. The formula of weighted average $w = \frac{f(b)*a - f(a)*b}{f(b) - f(a)}$ gives the point at which the straight line(known as secant line) through the points $\{a, f(a)\}$and $\{b, f(b)\}$ intersects the x-axis. This method is known as regula falsi or false-position method. **Figure 7.4** shows the method in comparison to the Bisection method.

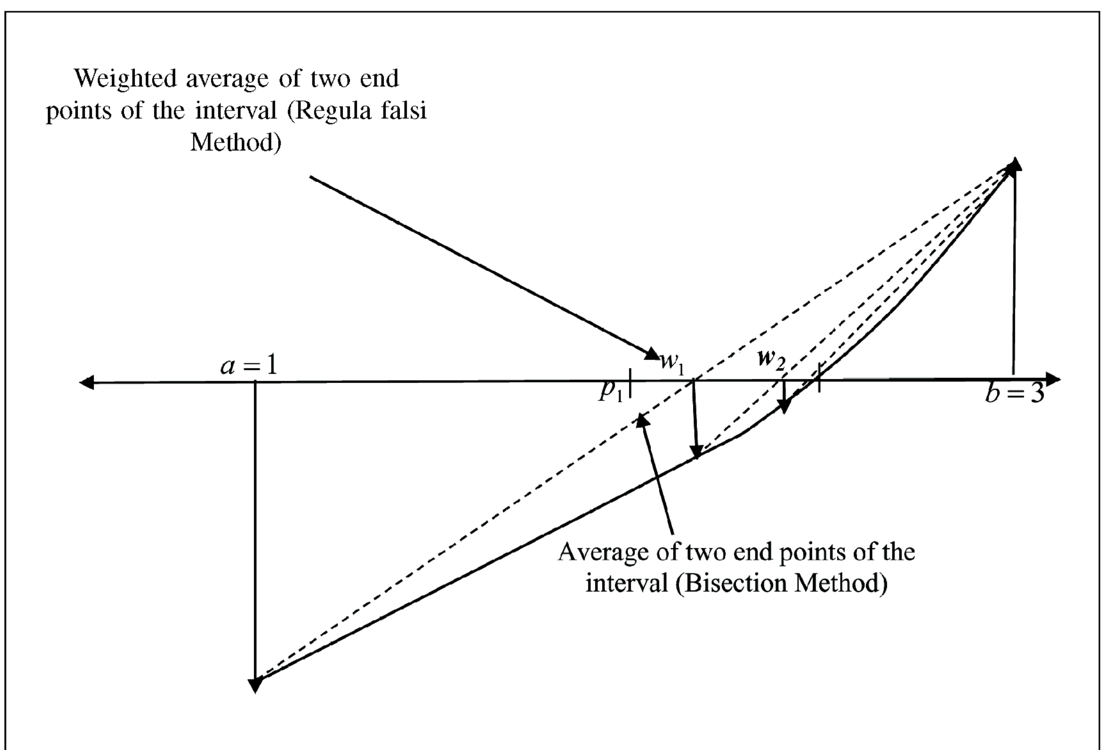


Figure 7.4

7.3.2 Algorithm and Implementation Issue :

Algorithm 7.2 : Regula falsi Method

Input : Function $f(x)$, end points of the interval a, b ; accuracy or tolerance t, maximum number of iterations n.

Output : Approximate solution w or message of failure.

Steps	*Description*
1.	*Read a, b, t and n and set i=0.*
2.	*if f (a) and f (b) have same sign, print "root cannot be found" and go to step 11.*
3.	*if* $\lvert f(a)\rvert \le t$ *then* $w = a$ *and go to step 10.*
4.	*if* $\lvert f(b)\rvert \le t$ *then* $w = b$ *and go to step 10.*
5.	*Increment i*
6.	$w = (f(b)*a - f(a)*b)/(f(b) - f(a))$.
7.	*if f (a) and f (w) have opposite sign then* $b = w$, *otherwise* $a = w$.
8.	*if* $\lvert f(w), > t$ *and* $i < n$ *then go to step 5.*
9.	*if* $(i = n)$ *then print message "Exceeds maximum number of iterations" and go to step 11.*
10.	*Print the root w.*
11.	*Stop.*

The flowchart is almost same as bisection method with only change in the formula where weighted average replaces the average of two end points.

7.3.3 Implementation : Regula Falsi Method

The C implementation along with an output of regula falsi method has been given in **Table 7.8**.

```
include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define f(x) (pow(x,3)-x-15) /* Function f(x) */
int main()
{
  float a,b,w,t;
  int i=0,n;
  printf("Enter the value of a,b,eps: ");
  scanf("%f%f%f",&a,&b,&t);
  printf("\nHow many maximum no of iterations you need? ");
  scanf("%d",&n);
  if (f(a)*f(b)>0) /*Check the initial interval chosen properly*/
  { printf("Root can't be found using bisection method");
     exit(0);
  }
```

```
    if(fabs(f(a))<t) /*Checking for root at the left end of initial interval*/
       w=a;
    else if (fabs(f(b))<t) /*Checking for root at the right end of initial interval*/
         w=b;
    else
    { printf("\nIter\t a\t b\t w\t f(a)\t f(b)\t f(w)\n");
       printf("-----------------------------------------------------------\n");
       do /*Iterative method to find the root*/
       { i++;
         w=(f(b)*a-f(a)*b)/(f(b)-f(a));
         printf("%d\t%6.3f\t%6.3f\t%6.3f\t%6.3f\t%6.3f\t%6.3f\n",i,a,b,w,f(a),
         f(b),f(w));
         if (f(a)*f(w)<0)
           b=w;
         else a=w;
       } while (fabs(f(w))>t && i<n);
    }
  if (i==n)
    {
       printf("\nExceeds maximim number of iterations");
       exit(0);
    }
    printf("\n\nThe desired root is %6.2f",w);
    return 0;
  }
```

Enter the value of a,b, eps : 1 3 .01

How many maximum no of iterations you need? 20

Iter	a	b	w	f(a)	f(b)	f(w)
1	1.000	3.000	2.250	–15.000	9.000	–5.859
2	2.250	3.000	2.546	–5.859	9.000	–1.047
3	2.546	3.000	2.593	–1.047	9.000	–0.157
4	2.593	3.000	2.600	–1.157	9.000	–0.023
5	2.600	3.000	2.601	–0.023	9.000	–0.003

The desired root is 2.60

Table 7.8

Now while bisection method **(Table 7.3)** is used to determine the root of the function $f(x) = x^3 - x - 15$, it produces the same result but takes 3 more iterations to converge. **The Table 7.9** shows this result. This observation is of no surprise if the nature of function is taken into consideration. **Figure 7.4** shows the near linear nature of function $f(x) = x^3 - x - 15$ in the interval [1, 3].

Enter the value of a,b, eps : 1 3 .01

How many maximum no of iterations you need? 20

Iter	a	b	p	f(a)	f(b)	f(p)
1	1.000	3.000	2.000	–15.000	9.000	–9.000
2	2.000	3.000	2.500	–9.000	9.000	–1.875
3	2.500	3.000	2.750	–1.875	9.000	3.047
4	2.500	2.750	2.625	–1.875	3.047	0.463
5	2.500	2.625	2.563	–1.875	0.463	–0.736
6	2.563	2.625	2.594	–0.736	0.463	–0.144
7	2.594	2.625	2.609	–0.144	0.463	0.157
8	2.594	2.609	2.602	–0.144	0.157	0.006

The desired root is 2.60

Table 7.9

Example 7.5

Determine the root of the equation $f(x) = \frac{1}{x} - 0.4$ using

1. Bisection method
2. Regula falsi method

Which of the above method gives better result? Justify your answer. You can choose initial interval [0.4, 4] and accuracy up to 2 decimal point.

Solution :

By executing the program written in **Table 7.4** the following output for bisection method has been generated.

```
Enter the value of a,b, eps : .4  4 .01
How many maximum no of iteration you need? 20
Iter     a        b        p        f(a)     f(b)     f(p)
-----------------------------------------------------------
1      0.400    4.000    2.200    2.100   -0.150    0.055
2      2.200    4.000    3.100    0.055   -0.150   -0.077
3      2.200    3.100    2.650    0.055   -0.077   -0.023
4      2.200    2.650    2.425    0.055   -0.023    0.012
5      2.425    2.650    2.538    0.012   -0.023   -0.006
The desired root is    2.54
```

This shows the approximated value of the root is 2.54 and the method takes 5 iterations to converge. Again regula falsi method **(Table 7.8)** produce following output.

```
Enter the value of a,b, eps : .4 4 .01
How many maximum no of iterations you need? 20
Iter     a        b        w        f(a)     f(b)     f(w)
-----------------------------------------------------------
1      0.400    4.000    3.760    2.100   -0.150   -0.134
2      0.400    3.760    3.558    2.100   -0.134   -0.119
3      0.400    3.558    3.389    2.100   -0.119   -0.105
4      0.400    3.389    3.247    2.100   -0.105   -0.092
5      0.400    3.247    3.127    2.100   -0.092   -0.080
6      0.400    3.127    3.027    2.100   -0.080   -0.070
7      0.400    3.027    2.943    2.100   -0.070   -0.060
8      0.400    2.943    2.872    2.100   -0.060   -0.052
9      0.400    2.872    2.812    2.100   -0.052   -0.044
10     0.400    2.812    2.762    2.100   -0.044   -0.038
11     0.400    2.762    2.720    2.100   -0.038   -0.032
12     0.400    2.720    2.685    2.100   -0.032   -0.028
13     0.400    2.685    2.655    2.100   -0.028   -0.023
14     0.400    2.655    2.631    2.100   -0.023   -0.020
15     0.400    2.631    2.610    2.100   -0.020   -0.017
16     0.400    2.610    2.592    2.100   -0.017   -0.014
17     0.400    2.592    2.577    2.100   -0.014   -0.012
18     0.400    2.577    2.565    2.100   -0.012   -0.010
19     0.400    2.565    2.555    2.100   -0.010   -0.009
The desired root is    2.55
```

Even though regula falsi method produce the value of the root 2.55, almost same as bisection method, the total no of iterations 19 is much higher with respect to 5 iterations of bisection method. Therefore, it can be easily concluded that bisection method is much better than regula falsi method for this example.

This is because of the fact that the given function shows very less or no linearity in the chosen interval **(Figure 7.5)**.

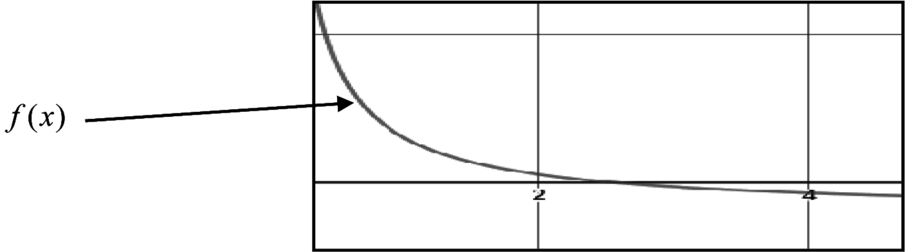


Figure 7.5

7.4 Secant Method

The bisection and regula falsi methods are known as bracketing methods since they always converge to a root which is inside the chosen interval. Both these methods need an additional constraint that the values of the function at the end point of interval are opposite in sign. Relaxing this criterion from the regula falsi method, a new method was developed which has faster rate of convergence with respect to regula falsi method. Unfortunately, this method loses the guaranteed convergence characteristic as the bracketing constraint is removed from the method.

7.4.1 Method Description

The formula of weighted average $w = \dfrac{f(b) * a - f(a) * b}{f(b) - f(a)}$ in regula falsi method gives the point at which secant line through the points $(a, f(a))$, $(b, f(b))$ intersects the x-axis. In the next iteration, the method uses the same formula to determine the intersecting point between x-axis and the secant line either through $(a, f(a))$, $(w, f(w))$ or through $(w, f(w))$, $(b, f(b))$ based on the sign of $f(w)$. The secant method always considers secant line passing through two most recently used points, i.e. $(b, f(b))$, $(w, f(w))$ in this case. More generically, the intersecting point of secant line and the x-axis can be found from following equation :

$$x_{i+1} = \frac{f(x_i) * x_{i-1} - f(x_{i-1}) * x_i}{f(x_i) - f(x_{i-1})} \quad \text{for i,.....,1,2,3,.......n} \qquad \textbf{(7.5)}$$

This process now continues until the desired accuracy is reached. **Figure 7.6** shows the secant method for first 3 iterations.

7.4.2 Algorithm and implementation Issue :

Algorithm 7.3 : Secant Method

Input : Function $f(x)$, two arbitrary points a, b ; accuracy or tolerance t , maximum number of iterations n.

Output : Approximate root x if method converges to the root x or message of failure.

Steps	*Description*
1.	*Read a, b, t and n and set* $i=0$
2.	$x_0 = a,\ x_1 = b$
3.	*if* $\lvert f(x_0)\rvert \le t$ *then* $x = x_0$ *and go to step 11.*
4.	*if* $\lvert f(x_1)\rvert \le t$ *then* $x = x_1$ *and go to step 11.*
5.	*Increment i*
6.	$x = (f(x_1)*x_0 - f(x_0)*x_1)/(f(x_1) - f(x_0))$.
7.	$x_0 = x_1$
8.	$x_1 = x$
9.	*if* $\lvert f(x)\rvert > t$ *and* $i < n$ *then go to step 5.*
10.	*if* $(i = n)$ *then print message "Exceeds maximum number of iterations" and go to step 12.*
11.	*Print the root x.*
12.	*Stop.*

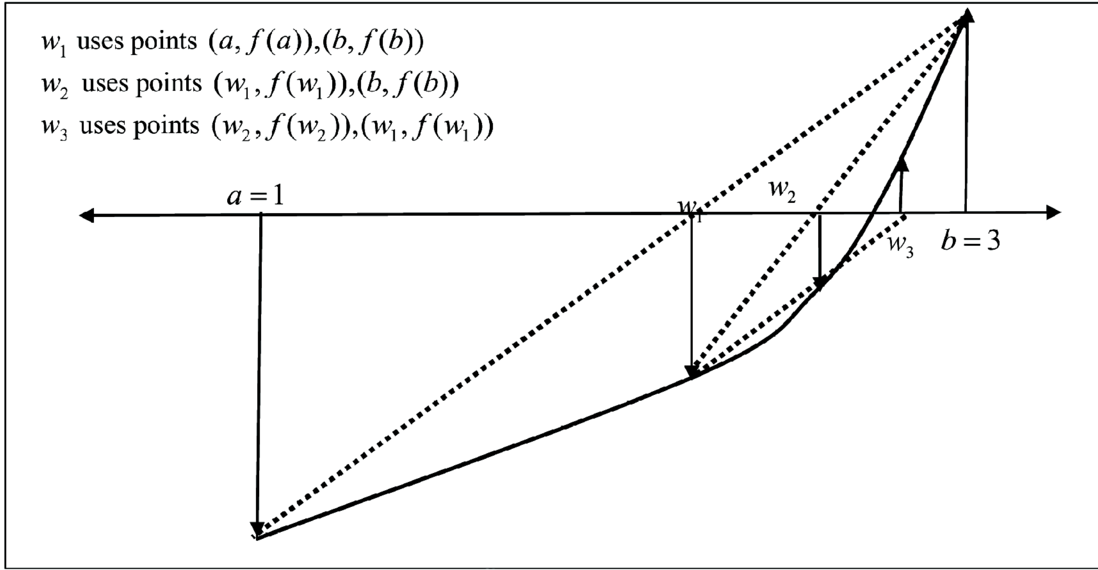


Figure 7.6

Check Your Progress 7.5

Draw the flow chart of the algorithm for secant method.

7.4.3 Implementation : Secant Method

The C implementation of secant method has been given in **Table 7.10**.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define f(x) (pow(x,3)-x-15) /* Function f(x) */
int main()
{
   float x0,x1,x2,t;
   int i=0,n;
   printf("Enter the value of a,b,eps: ");
   scanf("%f%f%f",&x0,&x1,&t);
   printf("\nHow many maximum no of iterations you need? ");
   scanf("%d",&n);
   if(fabs(f(x0))<t) /*Checking for root at the left end of initial interval*/
      x2=x0;
   else if (fabs(f(x1))<t) /*Checking for root at the right end of initial interval*/
         x2=x1;
   else
   { printf("\nIter x0\t x1\t x2\t f(x0)\t f(x1)\t f(x2)\n");
   printf("-----------------------------------------------------------\n");
   do /*Iterative method to find the root*/
      { i++;
      x2=(f(x1)*x0-f(x0)*x1)/(f(x1)-f(x0));
printf("%d\t%6.3f\t%6.3f\t%6.3f\t%6.3f\t%6.3f\t%6.3f\n",i,x0,x1,x2,f(x0),f(x1),f(x2));
      x0=x1;
      x1=x2;
      } while (fabs(f(x2))>t && i<n);
   }
if (i==n)
   {
      printf("\nExceeds maximum number of iterations");
      exit(0);
   }
   printf("\n\nThe desired root is %6.2f",x2);
   return 0;
}
```

Table 7.10

The equation $f(x) = (x)^3 - x - 15 = 0$ is solved using the program in **Table 7.10** using initial value $a = 1$, $b = 3$ and accuracy $t = .01$. The output of the program is given in **Table 7.11**.

```
Enter the value of x0, x1, esp : 1 3 .01
How many maximum no of iterations you need? 20
Iter    xo      x1      x2      f(x0)     f(x1)     f(x2)
----------------------------------------------------------
1       1.000   3.000   2.250   –15.000    9.000    –5.859
2       3.000   2.250   2.546     9.000   –5.859    –1.047
3       2.250   2.546   2.610    –5.859   –1.047     0.172
4       2.546   2.610   2.601    –1.047    0.172    –0.004
The desired root is   2.60
```

Table 7.11

The output clearly shows that secant method takes only 4 iterations to achieve the desired accuracy 0.01 which is better than bisection as well as regula falsi method. The point to be noted that the initial values a and b can be chosen arbitrarily without any constraint which was not the case in any of the previous two methods. For example, bisection and regular falsi methods can't be used to solve the equation $f(x) = x^3 - x - 15 = 0$ with initial interval [3, 4] since $f(3)$, $f(4)$ both are positive. On the other hand, the secant method can successfully produce the output for same set of initial values. The result shows in **Table 7.12**.

The output of bisection method	**The output of regula falsi method**
Enter the value of a, b, eps : 3 4 .01 How many maximum no of iterations you need? 20 Root can't be found using bisection method	Enter the value of a, b, eps : 3 4 .01 How many maximum no of iterations you need? 20 Root can't be found using regula falsi method

The output of secant method

```
Enter the value of x0, x1, eps : 3 4 .01
How many maximum no of iterations you need? 20
Iter    x0      x1      x2      f(x0)     f(x1)     f(x2)
----------------------------------------------------------
1       3.000   4.000   2.750     9.000   45.000     3.047
2       4.000   2.750   2.659    45.000    3.047     1.145
3       2.750   2.659   2.605     3.047    1.145     0.064
4       2.659   2.605   2.601     1.145    0.064     0.001
The desired root is   2.60
```

Table 7.12

Check Your Progress 7.6

The polynomial $f(x) = x^3 - 2x - 1$ has a zero between 1 and 2. Using the secant method (Algorithm 7.3), find this zero correct to three significant figures.

Check Your Progress 7.7

The polynomial $f(x) = x^2 - 4$ has a zero at $x = 2$. Check that, the root can be found out using secant method when the initial values are $a = 1$, $b = 4$ and $t = 0.1$. Why the same method doesn't work if the initial values are $a = -3$, $b = 3$ and $t = .01$? Modify the program written in **Table 7.10** so that the program can generate proper message when this type of situations arises.

7.5 Newton Raphson Method

Newton's method (sometimes called the Newton-Raphson method) for solving nonlinear equations is one of the most well-known and powerful procedures in all of numerical analysis. It always converges if the initial approximation is sufficiently close to the root, and it converges faster than the methods discussed as of now. Its only disadvantage is that the derivative $f'(x)$ of the nonlinear function $f(x)$ must be evaluated.

7.5.1 Method Description

Let x_0 be a good estimate of the root r and let $r = x_0 + h$ where the number h measures how far the estimate x_0 is from the truth. Since r is the root of the equation, we can conclude,

$$0 = f(r) = f(x_0 + h) \tag{7.6}$$

$$= f(x_0) + hf'(x_0) + \frac{h^2}{2!} f''(x_0) + \cdots\cdots \quad \text{[Using Taylor's series]} \tag{7.7}$$

Assuming h is small and using linear approximation (tangent line), the higher order terms can be ignored from equation 7.7. Therefore, we get,

$$0 \approx f(x_0) + hf'(x_0) \tag{7.8}$$

It follows that,

$$h \approx -\frac{f(x_0)}{f'(x_0)}$$

The new estimate x_1 of r is therefore given by

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

The next estimate x_2 is obtained from x_1 in exactly the same way as x_1 was obtained from x_0 :

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

Continue in this way. If x_n is the current estimate, then the next estimate x_{n+1} is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \textbf{(7.9)}$$

The geometric interpretation of the Newton-Raphson method has been given in **Figure 7.7.** The curve $y = f(x)$ meets the x-axis at r. Let x_0 be the current estimate of r. The tangent line to $y = f(x)$ at the point $(x_0, f(x_0))$ has equation

$$y = f(x_0) + f'(x_0)(x - x_0) \qquad \textbf{(7.10)}$$

If the tangent line intersects the x-axis at x_1, then we can write from equation **(7.10)**,

$$0 = f(x_0) + f'(x_0)(x_1 - x_0)$$

$$\Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \qquad \textbf{(7.11)}$$

Equation **(7.11)** is same as equation **(7.9)** for $n = 0$. Therfore, x_1 can be thought of as next estimate of r in Newton-Raphson method. Similarly, the tangent line drawn at $(x_1, f(x_1))$ intersects the x-axis at x_2, giving a new estimate x_2. Repeating this process the actual root r can be obtained theoretically.

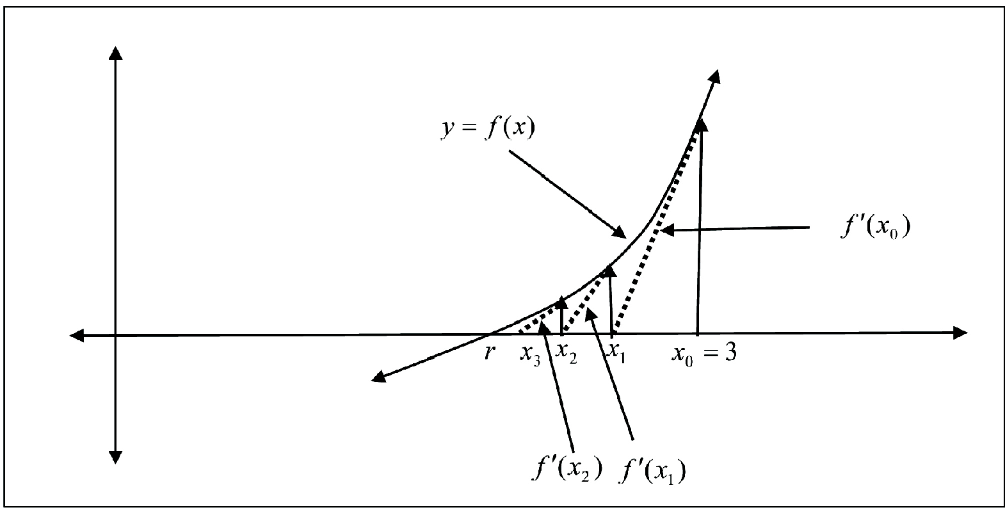


Figure 7.7

7.5.2 Algorithm and Implementation Issue :

Algorithm 7.4 : Newton Raphson Method

Input : Function $f(x)$, an arbitrary point x_0; accuracy or tolerance, maximum number of iterations n.

Output : Approximate root x if method converges to the root x.

Steps	*Description*
1.	*Read x_0, t, n and set $i = 0$*
2.	*if $\|f(x_0)\| \le t$ then $x = x_0$ and go to step 8.*
3.	*Increment i*
4.	$x = x_0 - \dfrac{f(x_o)}{f'(x_0)}$
5.	$x_0 = x$
6.	*if $\|f(x)\| > t$ and $i < n$ then go to step 3.*
7.	*if $(i = n)$ then print message "Exceeds maximum number of iterations" and go to step 9.*
8.	*Print the root x .*
9.	*Stop.*

Check Your Progress 7.2

Draw the flow chart of the algorithm for Newton Raphson method.

7.5.3 Implementation : Newton Raphson Method

The C implementation of Newton-Raphson method has been given in **Table 7.13**.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define f(x) (pow(x,3)-x-15) /* Function f(x) */
#define df(x) (3*pow(x,2)-1) /* Derivative of f(x) */
int main()
{
   float x0,x,t;
   int i=0,n;
   printf("Enter the value of x0,eps: ");
   scanf("%f%f",&x0,&t);
   printf("\nHow many maximum no of iteration you need? ");
```

```
    scanf("%d",&n);
    if(fabs(f(x0))<t) /*Checking for root at the initial point*/
      x=x0;
    else
    { printf("\nIter\t x\t f(x)\n");
      printf("-------------------------------------------------------\n");
      do /*Iterative method to find the root*/
      {
        i++;
        x=x0-f(x0)/df(x0);
        printf("%d\t%6.3f\t%6.3f\n",i,x,f(x));
      x0=x;
      } while (fabs(f(x))>t && i<n);
    }
  if (i==n)
    {
    printf("\nExceeds maximim number of iterations");
    exit(0);
    }
    printf("\n\nThe desired root is %6.2f",x);
    return 0;
  }
```

Table 7.13

Newton's method requires the value of the derivative $f'(x)$ in addition to the value the function $f(x)$. Therefore, one more macro to calculate $f'(x)$ has been added in the program in **Table 7.13**. When the function $f(x)$ is an algebraic function or a transcendental function, $f'(x)$ can be determined analytically. However, when the function $f(x)$ is a general nonlinear relationship between an input x and an output $f(x)$, $f'(x)$ cannot be determined analytically. In that case, $f'(x)$ can be estimated numerically by evaluating $f(x)$ at x_i and $x_i + \varepsilon\,(> 0)$, and approximating $f'(x)$ as

$$\frac{f(x+\varepsilon)-f(x)}{\varepsilon} \qquad \textbf{(7.12)}$$

This procedure doubles the number of function evaluations at each iteration. However, it eliminates the evaluation of $f'(x)$ at each iteration. If ε is small, round-off errors are introduced, and if ε is too large, the convergence rate is decreased.

The equation $f(x) = x^3 - x - 15 = 0$ is solved using initial value $x_0 = 3$ and accuracy $t = .01$. The output of the program is given in **Table 7.14**.

```
Enter the value of x0, esp : 3 .01
How many maximum no of iterations you need? 20
        Iter       x        f(x)
        ------------------------------------
        1        2.654     1.037
        2        2.602     0.021
        3        2.601     0.000
The desired root is   2.60
```

Table 7.14

The output clearly shows that Newton-Raphson method takes only 3 iterations to achieve the desired accuracy 0.01 which is better than all the methods discussed previously.

The accuracy and speed of the Newton-Raphson method mainly depends on two assumtions. Among them, the first one is tangent line approximation. To understand this, let us consider a particle travelling in a straight line, and let $f(x)$ be its position at time x. Then $f'(x)$ is the velocity at time x. If the acceleration of the particle were always 0, then the change in position from time x_0 to time $x_0 + h$ would be $hf'(x_0)$. So the position at time $x_0 + h$ would be $f(x_0) + hf'(x_0)$. It can be noted that this is the tangent line approximation, which can be thought of as the zero-acceleration approximation. If the velocity varies in the time from x_0 to $x_0 + h$, that is, if the acceleration is not 0, then in general the tangent line approximation will not correctly predict the displacement at time $x_0 + h$. And the bigger the acceleration, the bigger the error. It can be shown that if the function f is twice differentiable then the error in the tangent line approximation is $\frac{1}{2}h^2 f''(c)$ for some c between x_0 and $x_0 + h$. In particular, if $|f''(x)|$ is large between x_0 and $x_0 + h$, then the error in the tangent line approximation is large. Thus we can expect large second derivatives to be bad for the Newton-Rapshon Method (see the example given in **Example 7.6**).

The second assumption in this method is to start with a intial value which is close to the actual root. The result can be very bad if the value of x_0 is quite far from r. We have seen that $h = -\frac{f(x_0)}{f'(x_0)}$. Therefore, the value of $f'(x)$ close to 0, makes the value of h large enough which may produce bad result. Thus we can expect first

derivatives close to 0 to be bad for the Newton's Method (see the example given in **Example 7.7**).

Example 7.6

A devotee of Newton-Raphson used the method to solve the equation $3x^{\frac{1}{3}} = 0$, using the initial estimate $x_0 = 0.1$. Calculate the next 10 Newton Method estimates.

Solution :

Here $f'(x) = x^{-\frac{2}{3}}$ and $f''(x) = -\frac{2}{3}x^{-\frac{5}{3}} \Rightarrow |f''(x)| = \frac{2}{3}x^{-\frac{5}{3}}$. It is easy to observe that the equation $3x^{\frac{1}{3}} = 0$ has the root at $x = 0$. The graph of the function $f(x)$ shows that the slope of the tangent line at the point x, increases when x slides slowly along the curve $f(x)$ **(Figure 7.8)** from the intilal point $x_0 = 0.1$ to the root $r = 0$. The $f''(x)$ values for different values of x from the intilal point x_0 to the root r are given in **Table 7.15**. Now let us apply the Newton-Raphson algorithm to find out the next 10 estimates of the root r starting from $x_0 = 0.1$. The C program for Newton Raphson method given in **Table 7.13** uses macro definitions for $f(x)$ and $f'(x)$ which contains the pow (base, power) function defined in math.h library. This function can't generate the result when the base is negative and the power is a fraction. To avoid that we can deduce simpler expression for x_{n+1} in the following manner :

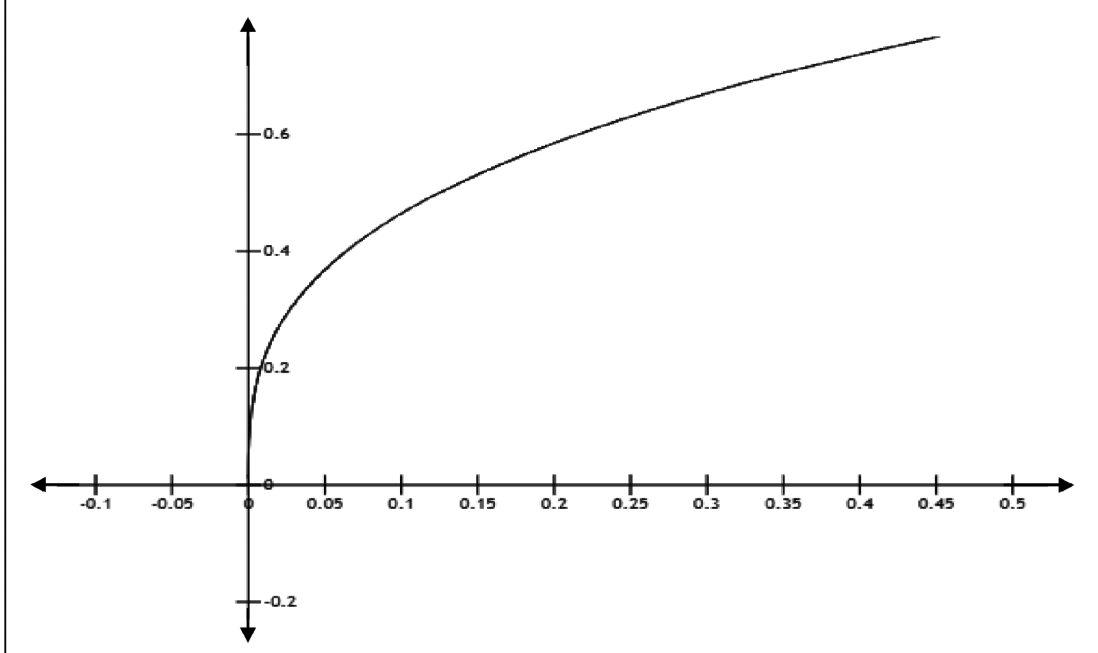


Figure 7.8

We know from equiation 7.9,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{3x_n^{\frac{1}{3}}}{x_n^{-\frac{2}{3}}}$$

$$\Rightarrow x_{n+1} = x_n - 3x_n = -2x_n \qquad \textbf{(7.13)}$$

x	$f''(x)$
0.1	30.63449
0.05	97.25843
0.01	1421.927
0.0001	3063449

Table 7.15

Using the expression given in equation **(7.13)**, the program given in **Table 7.13** can be modified to find next ten estimates of the root for the equation $3x^{\frac{1}{3}} = 0$. The modified C program and the corresponding output are given in **Table 7.16** and **Table 7.17** respectively.

The output of **Table 7.17** shows that the new estimates are going further from the root r = 0. In fact, if we start with any non-zero estimate, the Newton-Raphson estimates oscillate more and more wildly. The example shows that the Newton-Raphson method may be highly susceptible to produce the erroneous result if $f''(x)$ is large between the intial value of x and the actual root r.

```
#include<stdio.h>
int main()
{
   float x0,x;
   int i=0,n;
   printf("Enter the value of x0:");
   scanf("%f",&x0);
   printf("\nHow many maximum no of iterations you need? ");
   scanf("%d",&n);
   printf("\nIter\t x\n");
   printf("------------------------\n");
   do
      { i++;
         x=-2*x0;
         printf("%d\t%6.3f\n",i,x);
         x0=x;
   } while (i<=n);
   return 0;
}
```

Table 7.16

```
Enter the value of x0 : 0.1
How many maximum no of iterations you need? 10
      Iter           x
    ----------------------
       1          -0.200
       2           0.400
       3          -0.800
       4           1.600
       5          -3.200
       6           6.400
       7         -12.800
       8          25.600
       9         -51.200
      10         102.400
      11        -204.800
```

Table 7.17

Example 7.7

A devotee of Newton-Raphson used the method to solve the equation $x^{100} = 0$, using the initial estimate $x_0 = 0.1$. Calculate the next 10 Newton Method estimates.

Solution :

Here $f'(x) = 100x^{99}$ and observe that the value of $f'(x)$ is very close to 0 from intial value $x_0 = 0.1$ to $r = 0$. The Newton Raphson-Method iteration is

$$x_{n+1} = X_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^{100}}{100x_n^{99}}$$

$$\Rightarrow x_{n+1} = \frac{99}{100}x_n \qquad \textbf{(7.14)}$$

Using the iteration equation **(7.14)** now we can run the program given in **Table 7.16**. The output in **Table 7.18** shows that the convergence rate is very slow and after 10th iteration the value of $x = .09$ which needs lot more iterations to reach the original root r. This slow convergence is due to the fact that the value of first derivative is close to 0 for the function $f(x)$.

```
Enter the value of x0 : 0.1
How many maximum no of iterations you need? 10
      Iter        x
    ----------------------
      1        0.099
      2        0.098
      3        0.097
      4        0.096
      5        0.095
      6        0.094
      7        0.093
      8        0.092
      9        0.091
      10       0.090
      11       0.090
```

Table 7.18

Example 7.4

Use the Newton-Raphson method, with 3 as starting point, to find a fraction that is within 10^{-4} of $\sqrt{10}$.

Solution :

The value of the $\sqrt{10}$ can also be thought of as the root of the equation $f(x) = x^2 - 10 = 0$. Now $f'(x) = 2x$. Modify the program written in **Table 7.13** using following macros for $f(x)$ and $f'(x)$.

```
#define f(x) (pow(x,2)-10)
#define f(x) (2*x)
```

Assuming $x_0 = 3$, and $t = .00001$, the above program produces the following output :

```
Enter the value of x0, esp : 3 .00001
How many maximum no of iterations you need? 20
      Iter      x        f(x)
    ------------------------------------
      1       3.167     0.028
      2       3.162     0.000
      3       3.162     0.000
The desired root is    3.16
```

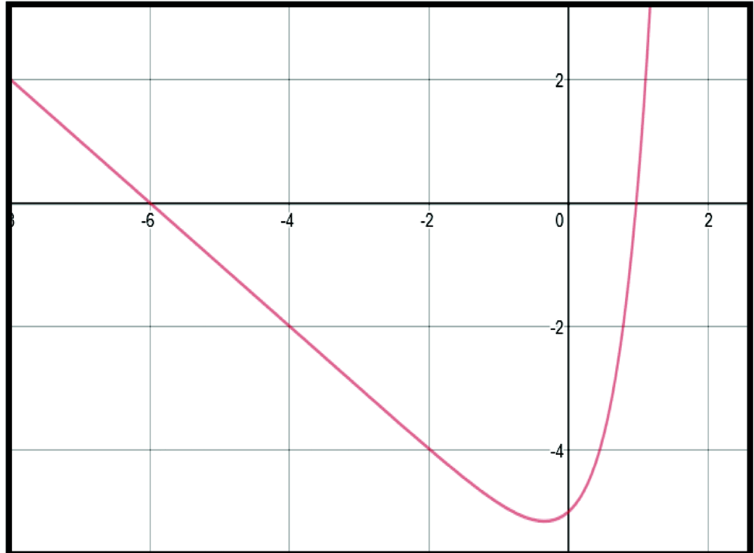
It can be noted that after replacing $f(x)$ and $f'(x)$ in equation 7.8, we get

$$x_{n+1} = x_n - \frac{(x_n^2 - 10)}{2x_n} = \frac{2x_n^2 - (x_n^2 - 10)}{2x_n} = \frac{x_n^2 + 10}{2x_n} = \frac{1}{2}\left(x_n + \frac{10}{x_n}\right)$$

The above formula is used in **Section 1.6.1** while finding the square root of a number using Babylonian method.

Check Your Progress 7.3

Find a solution of $e^{2x} = x + 6$, correct to 3 decimal places; use the Newton-Raphson Method. The graph of the function $e^{2x} - x - 6 = 0$ is given below.



Check Your Progress 7.4

It costs a firm Rs. $C(q)$ to produce q grams per day of a certain chemical, where $C(q) = 1000 + 2q + 3q^{\frac{2}{3}}$. The firm can sell any amount of the chemical at Rs 200 a gram. Find the break-even point of the firm, that is, how much it should produce per day in order to have neither a profit nor a loss. Use the Newton Method and give the answer to the nearest gram.

Check Your Progress 7.5

A loan of Rs. A is repaid by making n equal monthly payments of Rs. M, starting a month after the loan is made. It can be shown that if the monthly interest rate is r, then

$$Ar = M\left(1 - \frac{1}{(1+r)^n}\right)$$

A car loan of Rs. 600000 was repaid in 60 monthly payments of Rs. 15000. Use the Newton Method to find the monthly interest rate corrected up to 3 decimal points.

Check Your Progress 7.6

The function $f(x) = x - 0.2 \sin x - 0.5$ has exactly one zero between 0.5 and 1.0, since $f(0.5)f(1.0) < 0$, while $f'(x)$ does not vanish on [0.5, 1]. Locate the zero corrected up to 3 decimal point using **Algorithms 7.3.2, 7.4.2, 7.5.2** and **7.6.2**.

7.6 Summary

Several methods to solve non-linear equation on single variable have been discussed in this chapter. First two closed domain (need two points where the sign of the function is opposite) methods for finding the roots of a nonlinear equation are presented in this unit : interval halving (bisection) and false position (regula falsi). Both of these methods are guaranteed to converge because they keep the root bracketed within a continually shrinking closed interval. Both methods are quite robust, but converge slowly. The C implementation of these methods using macros are also discussed in this unit.

The next two methods discussed here are secant method and Newton-Raphson method. These methods have a higher-order convergence rate (1.62 for the secant method and 2.0 for Newton's method). These methods converge rapidly in the vicinity of a root. When the derivative $f'(x)$ is difficult to determine or time consuming to evaluate, the secant method is more efficient. In extremely sensitive problems, these two methods may misbehave and require some bracketing technique.

7.7 References and Further Reading

1. Elementary Numerical Analysis – An algorithmic Approach, Third Edition, S.D. Conte, Carl de Boor, Tata McGraw-Hill, 2012.
2. Numerical Recipes in C, Second Edition, H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Tata McGraw Hill, 2003.
3. Numerical Methods for Engineers and Scientists, First Editon, Joe D. Hoffman, Tata McGraw Hill, 1992.

Unit - 8 ❑ Application of C Programming : Solution of System of Linear Equations by Direct Methods : Gauss Elimination and Gauss Jordan Elimination

Structure

8.0 Introduction

8.1 Objectives

8.2 System of Linear Equations

8.3 Matrix Notation

8.4 Solving Linear System by Direct Method

- **8.4.1 The Row Echelon Form**
- **8.4.2 Gauss Elimination**
- **8.4.3 Reduced Row Echelon Form**
- **8.4.4 Gauss-Jordan Elimination**

8.5 Summary

8.6 References and Further Reading

8.7 Hints and Solution

8.0 Introduction

Considering the inconceivable complexity of processes even in a simple cell, it is little short of a miracle that the simplest possible model - namely, a linear equation between two variables - actually applies in quite a general number of cases.

—Ludwig von Bertalanffy

In the last unit, different algorithms of solving non-linear equations along with their C-implementations were discussed. C programming is also very useful for another very important type of problems viz. to find out the solution of system of linear equations. There are two fundamentally different approaches for solving systems of linear algebraic equations :

- ➢ Direct elimination methods
- ➢ Iterative methods

Direct elimination methods are systematic procedures based on algebraic elimination, which obtain the solution in a fixed number of operations. Examples of direct elimination methods are Gauss elimination, Gauss-Jordan elimination, the matrix inverse method, and LU decomposition method. We will discuss two important forms of matrix namely row-echelon form and reduced row echelon form in this unit. We will also discuss C implantations of Gauss elimination and Gauss-Jordan elimination method with the help of above two forms in this unit. The remaining direct methods will be discussed in **Unit 9**. The iterative methods will be discussed in **Unit 10**. As a prerequisite, the learner needs to revisit the Algebra – CC1 taught in first semester of Bachelor Degree Program in Elective Mathematics (EMT) under Netaji Subhas Open University. The C programs of this unit primarily uses the array, therefore the learners are expected to have complete understanding of C array and their processing mentioned in **Unit 6**.

8.1 Objectives

After going through this topic, the learner should be able to

- Describe the general structure of a system of linear algebraic equations.
- Determine different types of solution for a system of linear algebraic equations using C program :
  (a) a unique solution, (b) no solution, (c) an infinite number of solutions.
- Perform elementary matrix algebra with C programming language.
- Convert a system of linear algebraic equations into row echelon form.
- Solve a system of linear algebraic equations by Gauss elimination using C program.
- Convert a system of linear algebraic equations into reduced row echelon form.
- Solve a system of linear algebraic equations by Gauss-Jordan elimination using C program.
- Determine whether a set of vectors are linearly independent using C program.

8.2 System of Linear Equations :

A linear equation in the variables $x_1, x_2, \ldots, x_n$ is an equation that can be written in the form

$$a_1x_1 + a_2x_2 + \ldots, + a_nx_n = b \qquad \textbf{(8.1)}$$

where b and the **coefficients** $a_1, a_2, \ldots\ldots\ldots, a_n$ are real or complex numbers, usually known in advance. The subscript n may be any positive integer. The equations

$$2x_1 + 3x_2 - 5 = x_3 \quad \text{and} \quad x_1 = 3\left(x_2 - \sqrt{5}\right) + x_3$$

are both linear as they can be rearranged algebraically as in equation (1). The equations

$$2x_1 + 3x_2 - 5 = x_1x_3 \quad \text{and} \quad x_1 = 3\left(\sqrt{x_2} - 5\right) + x_3$$

are not linear because of the presence of x_1x_3 in the first equation and $\sqrt{x_2}$ in the second.

A **system of linear equations** (or **linear system**) is a collection of one or more linear equations involving the same variables—say, $x_1, x_2, \ldots x_n$.

A **solution** of the system is a list $(s_1, s_2, \ldots s_n)$ of numbers that makes each equation a true statement when the values $s_1, s_2, \ldots s_n$ are substituted for $x_1, x_2, \ldots x_n$, respectively. The set of all possible solutions is called the **solution set** of the linear system. Two linear systems are **equivalent** if they have the same solution set. That is each solution of the first system is also the solution of the second and vice versa.

8.3 Matrix Notation

The essential information of a linear system can be recorded compactly in a matrix. The system

$$\begin{aligned} x_1 - 2x_2 + x_3 &= 0 \\ 2x_2 - 8x_3 &= 8 \\ -4x_1 + 5x_2 + 9x_3 &= -9 \end{aligned} \quad \text{can be written as} \quad \begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -8 \\ -4 & 5 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 8 \\ -9 \end{bmatrix}$$

which is of the form $\boldsymbol{Ax} = \boldsymbol{b}$ where the square matrix $\boldsymbol{A} = \begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -8 \\ -4 & 5 & 9 \end{bmatrix}$ is known as *coefficient matrix*, $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ and $\boldsymbol{b} = \begin{bmatrix} 0 \\ 8 \\ -9 \end{bmatrix}$ are column vectors whose elements are the variables and right hand side values of the linear equations respectively. We can also define the augmented matrix of the linear system by the following matrix :

$$\begin{bmatrix} 1 & -2 & 1 & 0 \\ 0 & 2 & -8 & 8 \\ -4 & 5 & 9 & -9 \end{bmatrix} \quad \textbf{(8.2)}$$

8.4 Solving Linear System by Direct Method

The basic strategy to solve a linear system is to replace one system with an equivalent system (i.e., one with the same solution set) which is easier to solve. Use x_1 term in the first equation of a system to eliminate the x_1 terms in other equations. Then use the x_2 term in the second equation to eliminate the x_2 terms in other equations, and so on, until a very simple equivalent system is obtained finally. Following three basic operations are used to simplify a linear system :

Basic Operations to simplify linear system
• Replace one equation by the sum of itself and multiple of another equation. • Interchange two equations. • Multiply all the terms in an equation by a non-zero constant.

Table 8.1

Considering the augmented matrix of the same linear system, row operations corresponding to the three basic operations mentioned in **Table 8.1** can be applied. These row operations are known as elementary row operations and transforms one matrix to another row equivalent matrix.

Elementary row operations corresponding to basic operations listed in **Table 8.1**
• (Replacement) Replace one row by the sum of itself and multiple of another row. • (Interchange)Interchange two rows. • (Scaling) Multiply all the entries in a row by a non-zero constant.

Table 8.2

Therefore, the typical method which we have learned in high school to solve linear equations can be formalised by using matrix and elementary row operations. The following **Table 8.3** illustrate this fact for matrix in equation 8.2.

Steps	Linear Equations	Augmented Matrix
0 (Initial State of the system)	$x_1 - 2x_2 + x_3 = 0$ ----(1) $2x_2 - 8x_3 = 8$ ----(2) $-4x_1 + 5x_2 + 9x_3 = -9$ ----(3)	$\begin{bmatrix} 1 & -2 & 1 & 0 \\ 0 & 2 & -8 & 8 \\ -4 & 5 & 9 & -9 \end{bmatrix}$
1	[new eq(3)] = [eq(3)] + 4[eq(1)]	$R_3 = R_3 + 4R_1$
	$x_1 - 2x_2 + x_3 = 0$ ----(1) $2x_2 - 8x_3 = 8$ ----(2) $-3x_2 + 13x_3 = -9$ ----(3)	$\begin{bmatrix} 1 & -2 & 1 & 0 \\ 0 & 2 & -8 & 8 \\ 0 & -3 & 13 & -9 \end{bmatrix}$
2	[new eq(3)]= [eq(3)] + $\frac{3}{2}$ [eq(2)]	$R_3 = R_3 + \frac{3}{2}R_2$
	$x_1 - 2x_2 + x_3 = 0$ ----(1) $2x_2 - 8x_3 = 8$ ----(2) $x_3 = 3$ ----(3)	$\begin{bmatrix} 1 & -2 & 1 & 0 \\ 0 & 2 & -8 & 8 \\ 0 & 0 & 1 & 3 \end{bmatrix}$ **Row Echelon Form**
3	[new eq(2)] = $\frac{1}{2}$[eq(2)]	$R_2 = \frac{1}{2}R_2$
	$x_1 - 2x_2 + x_3 = 0$ ----(1) $x_2 - 4x_3 = 4$ ----(2) $x_3 = 3$ ----(3)	$\begin{bmatrix} 1 & -2 & 1 & 0 \\ 0 & 1 & -4 & 4 \\ 0 & 0 & 1 & 3 \end{bmatrix}$ **The Diagonal Elements are 1**
4	[new eq(2)] = [eq(2)] + 4[eq(3)], [new eq(1)] = [eq(1)] – [eq(3)]	$R_2 = R_2 + 4R_3$, $R_1 = R_1 - R_3$
	$x_1 - 2x_2 = -3$ ----(1) $x_2 = 16$ ----(2) $x_3 = 3$ ----(3)	$\begin{bmatrix} 1 & -2 & 0 & -3 \\ 0 & 1 & 0 & 16 \\ 0 & 0 & 1 & 3 \end{bmatrix}$
5 **(Final Solution)**	[new eq(1)]= [eq(1)] + 2 [eq(3)]	$R_1 = R_1 + 2R_2$
	$x_1 = 29$ ----(1) $x_2 = 16$ ----(2) $x_3 = 3$ ----(3)	$\begin{bmatrix} 1 & 0 & 0 & 29 \\ 0 & 1 & 0 & 16 \\ 0 & 0 & 1 & 3 \end{bmatrix}$ **Reduced Row Echelon Form**

Table 8.3

8.4.1 The Row Echelon Form

The matrix after step-2 of **Table 8.3** is in ***echelon form*** (or ***row echelon form***). It has following three properties.

1. All nonzero rows are above any row(s) of all zeros (if any).
2. Each ***leading entry*** (leftmost nonzero entry) of a row is in a column to the right of the leading entry of the row above it.
3. All entries in a column below the leading entry are zero.

Table 8.4

The matrix formed after the first two steps of the **Table 8.3** is in row echelon form and the matrix is further transformed into reduced row echelon form at the end of the entire process.

Algorithm 8.1 : Convert a rectangular matrix into row echelon form.

Input : A (m x n) matrix $A(m, n)$. The entry in ith row and jth column of matrix A is represented by $a[i][j]$. The column number of leading entry is kept in a variable *lead*.

Output : The row echelon form of the given matrix $A(m, n)$.

```
Steps   Description
 1.     i=0, lead=0
 2.     while (i<m-1 and lead<n)
 3.       |  if a[i][lead] ≠ 0 then go to step 4, otherwise
          |  find (the smallest) number p>i such that a[p][lead] ≠ 0
          |  and swap row i and row p and go to step 2.
          |     If no such p exists, then go to step 9.
 4.       |     for k=i+1, i+2, . . ., m-1 do:
 5.       |       |  t=a[k][lead]/a[i][lead]
 6.       |       |  for j=lead, lead+1, . . ., n-1 do:
 7.       |       |__________|__a[k][j] = a[k][j]–t*a[i][lead]
 8.       |  i=i+1
 9.       |__lead=lead+1 and go to step 2.
10. Print the matrix A and stop.
```

Let us illustrate the above algorithm with the help of the following 4×5 matrix **(8.3a)**.

$$a[0][0] \rightarrow \begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ -4 & -5 & 3 & -8 & 1 \\ 2 & -5 & -4 & 1 & 8 \\ -6 & 0 & 7 & -3 & 1 \end{bmatrix} \qquad \textbf{(8.3a)}$$

Initial value of variable i = 0 and lead = 0 (step-1 of **Algorithm-8.1**). This means the leading entry of 1st row is a[0][0]. The value of m = 4 and n = 5 for the matrix in (8.3a). Now $a[0][0] = 2 \neq 0$ and therefore, the control goes to step 4 as per the **Algorithm-8.1**. Now we create zero below a[0][0] by applying appropriate row operations (step 4–7). As an example let us analyse how zero is created in a[1][0]. The value of k = 1(step 4), then t = a[1][0]/a[0][0] = (-4)/2 = -2(step 5). Now apply the row operation $R_2 = R_2 - (-2)^* R_1$ to transform R_2 (step 6 to step 7). The same process will be applied to create zeros in a[2][0], a[3][0] by incrementing the loop index k in step 4. After completing the above steps the matrix looks like **(8.3b)**. The value of variable i and *lead* are incremented and becomes 1(step 8 to step 9). Then control goes to initial while loop. The leading entry of second row becomes a[1][1].

$$a[1][1] \rightarrow \begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & -9 & -3 & -4 & 10 \\ 0 & 12 & 4 & 12 & -5 \end{bmatrix} \qquad \textbf{(8.3b)}$$

Now after applying similar steps the zeros are created below a[1][1] and the new matrix looks like **(8.3c)**. The leading entry of the third row becomes a[2][2] and control goes to initial while loop.

$$a[2][2] \rightarrow \begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 4 & 7 \end{bmatrix} \qquad \textbf{(8.3c)}$$

Since the value of leading entry a[2][2] = 0 the Algorithm **8.1** searches a row p>2 for which $a[p][2] \neq 0$ as per the step 3. The next entry a[3][2] = 0, therefore no

such p exists and the control goes to step 9. Now the *lead* variable is incremented and the leading entry of third row becomes $a[2][3] = 2 \neq 0$ without changing any other element in the matrix **(8.3d)**. After that the control goes to initial while loop again.

$$a[2][3] \rightarrow \begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 4 & 7 \end{bmatrix} \qquad \textbf{(8.3d)}$$

The algorithm proceeds similarly and create zeros below the leading entry $a[2][3]$. The final matrix in row echelon form looks like **(8.3e)**.

$$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix} \qquad \textbf{(8.3e)}$$

The C-program for *row echelon form* is given in **Table 8.5 :**

```
#include<stdio.h>
#include<math.h>
void main()
{
   float a[20][20]={{2,4,-1,5,-2},{-4,-5,3,-8,1},{2,-5,-4,1,8},{-6,0,7,-3,1}};
   float temp[20],t,e =.0001;
   int m, n, i, j, k, p, lead=0;
   printf("Enter the number of rows and columns: ");
   scanf("%d%d",&m,&n);

   /* Display the given matrix*/

   printf("\n\nThe given matrix is\n");
   printf("------------------------------------------\n");
   for (i=0; i<m; i++)
{
   for (j=0; j<n; j++)
   { printf("%6.2f\t", a[i][j]);
   }
```

```
        printf("\n");
    }
  /*Main logic of row echelon form*/
    i=0;    /*i is pointing to first row*/
    while(i<m-1 && lead<n)
    {
        if (fabs(a[i][lead])>e) /*Check if the entry is nonzero*/
        {
            /*Apply row operation to create zeros below leading entry*/
            for(k=i+1; k<m; k++)
                {
                    t=a[k][lead]/a[i][lead];
                    for(j=lead; j<n; j++)
                        a[k][j]=a[k][j]-t*a[i][j];
                }
            i++;
        lead++;
    }
    else
    {
        /*Search a row below current row such that the entry in the same column
is non-zero */
        for(p=i+1; p<m; p++)
        {
            if (fabs(a[p][i])>e) /*Check if the entry is nonzero*/
                break;
        }
        /*if non-zero entry found swap the rows*/
        if(p<m)
            {
                for(j=0; j<n; j++)
                {
                    temp[j]=a[i][j];
                    a[i][j]=a[p][j];
                    a[p][j]=temp[j];
                }
            }
        else
```

```
        /*if all entry below the leading entry is zero then go to search the next
column*/
        lead++;
      }
    }
  printf("\n\nThe echelon matrix is\n");
      printf("------------------------------------------\n");
    /* Display the matrix in row echelon form*/
    for (i=0; i<m; i++)
    {
      for (j=0; j<n; j++)
      {
        printf("%6.2f\t", a[i][j]);
      }
      printf("\n");
    }
  }
```

Table 8.5

It is important to note that in the above C program **(Table 8.5)**, the expression fabs(a[i][lead])>e (-e< $a[i][lead]$<e) is used instead of testing the equality $a[i][lead]$==0 where e (0.0001) is defined as a small quantity. Due to round of errors, most floating point numbers end up being slighltly imprecise. As long as this imprecision stays small, it can easily be ignored. However, it also means that numbers expected to be equal often differs slightly, and a simple equality test fails. Therefore, in such scenarios it is better to check the differences of the numbers within some error bound.

Theorem 8.1: Existence and Uniqueness Theorem :

A linear system is consistent if and only if an echelon form of augmented matrix has no row of the form

$$[0 \quad 0 \quad 0 \ldots\ldots\ldots\ldots\ldots \; b] \text{ where } b \neq 0. \tag{8.4}$$

If a linear system is consistent, then the solution set contains either

i. a unique solution, or
ii. infinitely many solutions.

The row of the form in **(8.4)** implies an equation of the form

$0\,x_1 + 0\,x_2 + 0\,x_3 + \ldots + 0\,x_n = b$ where $b \neq 0$, obtained after it has been reduced by elementary row operations. Since $0 = b$ where $b \neq 0$ is invalid, we say the system is inconsistent.

Example 8.1

Determine if the following system is consistent without completely solving the system :

$$x_1 + 3x_3 = 2$$
$$x_2 - 3x_4 = 3$$
$$-2x_2 + 3x_3 + 2x_4 = 1$$
$$3x_1 + 7x_4 = -5$$

Solution :

The augmented matrix of the given system of equations is

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

Execute the program in **Table 8.5** with data given by above matrix. The output is :

```
Enter the no of rows and columns :
4   5
The given matrix is
-----------------------------------------
1.00  0.00   3.00   0.00   2.00
0.00  1.00   0.00  -3.00   3.00
0.00 -2.00   3.00   2.00   1.00
3.00  0.00   0.00   7.00  -5.00
The echelon matrix is
-----------------------------------------
1.00  0.00   3.00   0.00   2.00
0.00  1.00   0.00  -3.00   3.00
0.00  0.00   3.00  -4.00   7.00
0.00  0.00   0.00  -5.00  10.00
Enter the no of rows and columns : 4   5
```

We can say that the echelon matrix does not have any row of the form [0 0 0 0 b] with b nonzero. Therefore, using **Theorem 8.1**, we can conclude that the system is consistent.

8.4.2 Gauss Elimination

Algorithm 8.1 is generic in the sense that it could be applicable to both square and non-square matrices. The row echelon form of any square matrix is a upper (right) triangular matrix i.e. all entries $a[i][j] = 0$ for $i > j$. Which means that the algorithm eliminates the coefficients $a[i][j]$ of matrix A for $i > j$. It can be easily seen in the example 8.1 where the intial coefficient matrix A is a square matrix.

$$\begin{bmatrix} x_1 & +0 & +3x_3 & +0 & =2 \\ 0 & +x_2 & +0 & -3x_4 & =3 \\ 0 & -2x_2 & +3x_3 & +2x_4 & =1 \\ 3x_1 & +0 & +0 & +7x_4 & =5 \end{bmatrix} \xrightarrow{\text{Algorithm 8.1}} \begin{bmatrix} x_1 & +0 & +3x_3 & +0 & =2 \\ 0 & +x_2 & +0 & 3x_4 & =3 \\ 0 & +0 & +3x_3 & -4x_4 & =7 \\ 0 & +0 & +0 & -5x_4 & =10 \end{bmatrix}$$

Therefore, if we need to solve the system of linear equations of the form $\boldsymbol{Ax} = \boldsymbol{b}$ where A is a square matrix then the **Algorithm 8.1** produces equivalent system of linear equations of the following form.

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots\ldots & + & a_{1(n-1)}x_{n-1} & + & a_{1n}x_n & = b_1 \\ & & a_{12}x_2 & + & \ldots\ldots & + & a_{2(n-1)}x_{n-1} & + & a_{2n}x_n & = b_2 \\ & & \ldots\ldots & \ldots\ldots & \ldots\ldots & & \ldots\ldots & \ldots\ldots & \ldots\ldots & \ldots\ldots \\ & & & & & & a_{(n-1)(n-1)}x_{n-1} & + & a_{(n-1)n}x_n & = b_{n-1} \\ & & & & & & & & a_{nn}x_n & = b_n \end{array} \tag{8.5}$$

The solution of the above system of linear equations can easily be found if the diagonal elements $a[i][i]$ are non-zero for all i in $0, 1, 2, \ldots, (n-1)$ of the upper triangular matrix A. From the last equation of the linear system given in **(8.5)**, we find,

$$x_n = \frac{b_n}{a_{nn}} \qquad [\text{since } a_{nn} \neq 0]$$

Now once we know x_n, from the second last equation given in **(8.5)**, we can find,

$$x_{n-1} = \frac{b_{n-1} - a_{(n-1)n}x_n}{a_{(n-1)(n-1)}} \qquad [\text{since } a_{(n-1)(n-1)} \neq 0]$$

With x_n and x_{n-1} now determined, the third last equation

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

contains only one true unknown, namely, x_{n-2}. Once again, we can solve for x_{n-2},

$$x_{n-2} = \frac{b_{n-2} - a_{(n-2)(n-1)}x_{n-1} - a_{(n-2)n}x_n}{a_{(n-2)(n-2)}} \qquad [\text{since } a_{(n-2)(n-2)} \neq 0]$$

In general, with x_{k+1}, x_{k+2}, ..., x_n already computed, the kth equation can be uniquely solved for x_k, since $a_{kk} \neq 0$ to give

$$x_k = \frac{b_k - \sum_{j=k+1}^{n} a_{kj} x_j}{a_{kk}} \tag{8.6}$$

This process of determining the solution of system of linear equations given in **(8.5)** is called back substitution.

Now applying back substitution method in the echelon matrix obtained in **example 8.1** we get the solution,

$$x_4 = \frac{b_4}{a_{44}} = \frac{10}{-5} = -2$$

$$x_3 = \frac{b_3 - a_{34}x_4}{a_{33}} = \frac{7-(-4)(-2)}{3} = -\frac{1}{3}$$

$$x_2 = \frac{b_2 - a_{23}x_3 - a_{24}x_4}{a_{22}} = \frac{3-(0)(-1/3)-(-3)(-2)}{1} = -3$$

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4}{a_{11}} = \frac{2-(0)(-3)-(3)(-1/3)-(0)(-2)}{1} = 3$$

Therefore, the final solution of the system of linear equations is $x_1 = 3$, $x_2 = -3$, $x_3 = -1/3$, $x_4 = -2$.

The following procedure in **Table 8.6** outlines the Gauss Elimination process :

Steps	***Description***
1.	*If the coefficient matrix A(**m** × **n**) is a square matrix i.e, **m** = **n** then go to next Step, otherwise unique solution is not possible and stop the process.*
2.	*Write the augmented matrix of the system of linear equations.*
3.	*Use the row reduction method **(algorithm 8.1)** to obtain an equivalent augmented matrix in row echelon form.*
4.	*If the diagonal entries of reduced coefficient matrix is non-zero then go to next step otherwise either system is inconsistent or unique solution is not possible and stop the process.*
5.	*Apply back-substitution method.*
6.	*Write the solution of the given system of linear equations.*

Table 8.6

8.4.3 Reduced Row Echelon Form

If a matrix in echelon form (follows property in **Table 8.4**) satisfies the following additional conditions of **Table 8.7**, then it is in reduced echelon form (or reduced row echelon form).

1. The leading entry in each nonzero row is 1.
2. Each leading 1 is the only nonzero entry in the corresponding column.

Table 8.7

Pivot Position :

When row operation on a matrix produce an echelon form, further row operations to obtain the reduced row echelon form don't change the position of the leading entries. Since the reduced echelon form is unique, the leading entries are always in the same positions in any echelon form obtained from a given matrix. These leading entries correspond to the leading 1's in the reduced echelon form.

Algorithm 8.2 : Convert a matrix in row echelon form to reduced row echelon form.

Input : A (m × n) matrix $A(m, n)$ in row echelon form . The entry in ith row and jth column of matrix A is represented by $a[i][j]$.

Output : The reduced row echelon form of the given matrix $A(m, n)$.

Steps	*Description*
1.	*Create an array pivotcol[m] to store the column index of the pivot element of every row. The array was initially reset to 0.*
2.	*Divide every element of a row by non-zero pivot element of that row to produce 1 in the pivot element. skip if pivot element is 0.*
3.	*for i=m-1, m – 2, . . ., 1 do:*
	if pivot element of row i = 0 then go to step 3.
	for k=i-1, i-2, ..., 0 do:
	t=a[k][pivotcol[i]]
	for j=n-1, n-2, . . ., 0 do:
	*a[k][j] = a[k][j]–t*a[i][j]*
4.	*Print the matrix A and stop.*

Let us illustrate the above algorithm with the help of the matrix **(8.3e)** which is in row echelon form and produced by **Algorithm 1.1.** In step-1, an array pivotcol[m] has to be reset to 0. The initial configurations are following.

$$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix} \qquad \begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline 0 \\ \hline 0 \\ \hline \end{array} \qquad \textbf{(8.7a)}$$

Row Echelon Matrix Pivotcol array

Now for every row, the leading entry (first non-zero entry in a row) has been identified. The column index of the leading entry has been inserted to pivotcol array. As an example, the leading entry of 3rd row is 2 which is in the 4th column, therefore the 3rd element (pivotcol[2]) of pivotcol array is 3. Therefore the pivot entry in 3rd row is $a[2][3]$ More generically, we can say that if the leading entry of ith row is in j^{th} column, then pivotcol[i]=j and pivot element in row i is $a[i][pivotcol[i]]$. **(8.7b)** shows the pivotcol array after placing the pivot element for all rows.

$$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix} \qquad \begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline 3 \\ \hline 4 \\ \hline \end{array} \qquad \textbf{(8.7b)}$$

Row Echelon Matrix Pivotcol array

In step2, the elements of every row have been divided by the corresponding pivot element of that row so that the pivot elements become 1**(8.7c)**.

$$\begin{bmatrix} 1 & 2 & -0.5 & 2.5 & -1 \\ 0 & 1 & 0.33 & 0.67 & -1 \\ 0 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \textbf{(8.7c)}$$

In step3, we apply elementary row operations to create zeros above the pivot element. The row operation starts from pivot element of the last row and continues in the upward direction up to the pivot element of the second row which is shown in the following Figure. (from **8.7d-8.7f**).

$$\begin{bmatrix} 1 & 2 & -0.5 & 2.5 & 0 \\ 0 & 1 & 0.33 & 0.67 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(8.7d)} \qquad \begin{bmatrix} 1 & 2 & -0.5 & 0 & 0 \\ 0 & 1 & 0.33 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(8.7e)}$$

$$\begin{bmatrix} 1 & 0 & -1.17 & 0 & 0 \\ 0 & 1 & 0.33 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(8.7f)}$$

The final matrix in **(8.7f)** is in reduced echelon form.

The C-program to convert a matrix of row echelon form into *reduced row echelon form* is given in **Table 8.8**.

```
#include<stdio.h>
#include<math.h>
void main()
{
   float a[20][20]={{3,-7,8,-5,8,9},{0,3,-6,6,4,-5},{0,0,0,0,.67,2.67}};
   float temp[20], pivot, t, e =.0001;
   int m,n,i,j,k,p,pivotcol[20]={0},lead=0;
   printf("Enter the number of rows and columns");
   scanf("%d%d",&m,&n);
   /* Display the given matrix*/
   printf("\n\nThe given matrix is\n");
   printf("----------------------------------------\n");
   for (i=0; i<m; i++)
   { for (j=0; j<n; j++)
      {
         printf("%6.2f\t", a[i][j]);
      }
      printf("\n");
   }
   for(i=0;i<m;i++)
   /* Insert the column index of non-zero leading entry of ith row in pivotcol[i]*/
```

```
    for(j=i;j<n;j++)
    {  if(fabs(a[i][j])>e)
       {
          pivotcol[i]=j;
          break;
       }
       if(j==n)
          break;
    }
    /*Divide each row by corresponding non-zero pivot element to make pivot
element 1,skip if pivot element is 0.*/
  for(i=0;i<m;i++)
  {
    pivot=a[i][pivotcol[i]];
    if (fabs(pivot)<e)
       continue;
    else
    {
       for(j=0;j<n;j++)
       a[i][j]=a[i][j]/pivot;
    }
  }
  /*Apply row operation to create zeros above the pivot element*/
  for(i=m-1;i>=1;i--)
  {
    if(fabs(a[i][pivotcol[i]])<e)
    { continue; /*ignore row operation if pivot element of the row is zero*/
    }
       else
       { for(k=i-1;k>=0;k--)
          {
             t=a[k][pivotcol[i]];
             for(j=n-1;j>=pivotcol[i];j--)
                a[k][j]=a[k][j]-t*a[i][j];
          }
       }
```

```
    }
      printf("\n\nThe reduced echelon matrix is\n");
      printf("----------------------------------------\n");
        /* Display the matrix in reduced row echelon form*/
    for (i=0; i<m; i++)
    { for (j=0; j<n; j++)
      {
        printf("%6.2f\t", a[i][j]);
      }
      printf("\n");
    }
}
```

Table 8.8

The following procedure in **Table 8.9** outlines how to find and describe all solutions of the linear system.

Steps	*Description*
1.	*Write the augmented matrix of the system.*
2.	*Use the row reduction method* ***(algorithm 8.1)*** *to obtain an equivalent augmented matrix in row echelon form.*
3.	*Decide whether the system is consistent using (Check if any row of the row echelon matrix has the form mentioned in* ***(theorem 8.1)****. If there is no solution stop, otherwise, go to next step.*
4.	*Continue row reduction to obtain reduced row echelon form* ***(algorithm 8.2)****.*
5.	*Write the system of equations corresponding to the matrix obtained in step 4.*
6.	*Rewrite each nonzero equation from step 4 so that its one basic variable is expressed in terms of any non-basic(free) variables appearing in the equation.*

Table 8.9

C Program for the procedure mentioned in **Table-8.9** is given in **Table 8.10**.

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
void main()
{
   float a[20][20]={{0,1,2,1,0,0},{1,0,3,0,1,0},{4,-3,8,0,0,1}},temp[20];
   float t, pivot, e =0.0001;
   int m,n,i,j,k,p,pivotcol[20]={0},lead=0;
   char flg_inconsitent='n',flg_manysol='n';
   printf("Enter the number of rows and columns: ");
   scanf("%d%d",&m,&n);

   /* Display the given matrix*/

   printf("\n\nThe given matrix is\n");
   printf("-----------------------------------------\n");
   for (i=0; i<m; i++)
{
   for (j=0; j<n; j++)
   {
      printf("%6.2f\t", a[i][j]);
   }
   printf("\n");
   }
/*Main logic of row echelon form*/
   i=0; /*i is pointing to first row*/
   while(i<m-1 && lead<n)
   { if (fabs(a[i][lead])>e) /*Check if the entry is nonzero*/
      {
         /*Apply row operation to create zeros below leading entry*/
         for(k=i+1; k<m; k++)
            {
               t=a[k][lead]/a[i][lead];
               for(j=lead; j<n; j++)
                  a[k][j]=a[k][j]-t*a[i][j];
            }
```

```
            i++;
            lead++;
        }
        else
        {
            /*Search a row below current row such that the entry in the same
column is non-zero */
            for(p=i+1; p<m; p++)
            { if (fabs(a[p][i])>e) /*Check if the entry is nonzero*/
                    break;
            }
            /*if non-zero entry found swap the rows*/
            if(p<m)
              { for(j=0; j<n; j++)
                 { temp[j]=a[i][j];
                   a[i][j]=a[p][j];
                   a[p][j]=temp[j];
                 }
              }
            else
              /*if all entry below the leading entry is zero then go to search the
next column*/
              lead++;
        }
    }
  printf("\n\nThe echelon matrix is\n");
            printf("------------------------------------------\n");
        /* Display the matrix in row echelon form*/

    for (i=0; i<m; i++)
    { for (j=0; j<n; j++)
        {
            printf("%6.2f\t", a[i][j]);
        }
        printf("\n");
    }
```

```
  /*search any row of the form [0 0 0 . . . 0 b] where b is nonzero to check for
consistency*/
     for(i=m-1;i>=0;i--)
     {
        for(j=n-2;j>=0;j--)
        {
           if (fabs(a[i][j])>e)
              break;
        }
        if(j==-1 && fabs(a[i][n-1])>e)
        {
           printf("The system is inconsistent");
           exit(0);
        }
     }
  /*Main Logic for reduced echelon form*/
  for(i=0;i<m;i++)
        /* Insert the column index of non-zero leading entry of ith row in pivotcol[i]*/
        for(j=i;j<n;j++)
        {
           if(fabs(a[i][j])>e)
           {
              pivotcol[i]=j;
              break;
           }
              if(j==n)
              break;
           }
           /*Divide each row by corresponding non-zero pivot element to make
pivot element 1,skip if pivot element is 0.*/
     for(i=0;i<m;i++)
     {
        pivot=a[i][pivotcol[i]];
        if (fabs(pivot)<e)
           continue;
        else
        {
```

```
        for(j=0;j<n;j++)
            a[i][j]=a[i][j]/pivot;
        }
    }
/*Apply row operation to create zeros above the pivot element*/
for(i=m-1;i>=1;i--)
{
    if(fabs(a[i][pivotcol[i]])<e)
    {
        continue; /*ignore row operation if pivot element of the row is zero*/
    }
    else
    {
        for(k=i-1;k>=0;k--)
            {
                t=a[k][pivotcol[i]];
                for(j=n-1;j>=pivotcol[i];j--)
                    a[k][j]=a[k][j]-t*a[i][j];
            }
        }
    }
        printf("\n\nThe reduced echelon matrix is\n");
        printf("----------------------------------------\n");

    /* Display the matrix in reduced row echelon form*/
    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            printf("%6.2f\t", a[i][j]);
        }
        printf("\n");
    }
}
```

Table 8.10

8.4.4 Gauss-Jordan Elimination.

The procedure outlined in the **Table 8.9** is a variant of what is commonly known as Gauss Jordan Elimination method.

It can be noted that, while generating the echelon matrix we have chosen non-zero leading entries (pivot elements) to avoid multiplication/division by zero. Whenever any zero element is found as leading entry, the row has been swapped by next row where the corresponding column entry is non-zero. This method is a simpler version of strategy what is known as partial pivoting. In partial pivoting, before each elimination step the rows need to be rearranged in such way so that the leading entry (pivot element) takes the largest magnitude of all the values of the column. This technique reduces the round off error which is originated due to repeatitive row-operations on the matrix. There are many better techniques for pivoting to reduce the round off error further, but we have not considered them so that the code remains simple for the learner at undergraduate level.

Theorem 8.2

Let A be an $m \times n$ matrix, with column $\vec{a_1}, \vec{a_2}, \cdots \vec{a_n}$ and $\vec{b}$ in R^m. The matrix equation

$$Ax = b$$

has the solution set as the vector equation

$$x_1 \vec{a_1} + x_2 \vec{a_2} + \cdots + x_n \vec{a_n} = \vec{b}$$

which, in turn, has the same solution set as the system of linear equations which has augmented matrix of the form

$$[\vec{a_1} \quad \vec{a_2} \quad \cdots \quad \vec{a_n} \quad \vec{b}]$$

Example 8.2

Find the general solution of the linear system given below. Explain the geometric representation of the solution.

$$\begin{aligned} 3x_2 - 6x_3 + 6x_4 + 4x_5 &= -5 \\ 3x_1 - 7x_2 + 8x_3 - 5x_4 + 8x_5 &= 9 \\ 3x_1 - 9x_2 + 12x_3 - 9x_4 + 6x_5 &= 15 \end{aligned}$$

Solution :

Execute the program written is **(Table 8.10)** with the following :

Input : Given augmented matrix.

Output :

```
Enter the no of rows and columns : 3  6
The given matrix is
-------------------------------------------------
0.00   3.00  -6.00   6.00   4.00  -5.00
3.00  -7.00   8.00  -5.00   8.00   9.00
3.00  -9.00  12.00  -9.00   6.00  15.00
The echelon matrix is
-------------------------------------------------
3.00  -7.00   8.00  -5.00   8.00   9.00
0.00   3.00  -6.00   6.00   4.00  -5.00
0.00   0.00  -0.00   0.00   0.67   2.67
The reduced echelon matrix is
-------------------------------------------------
1.00   0.00  -2.00   3.00   0.00 -24.00
0.00   1.00  -2.00   2.00   0.00  -7.00
0.00   0.00  -0.00   0.00   1.00   4.00
```

Write the system of equations from the reduced echelon matrix.

$$
\begin{aligned}
x_1 - \quad\; 2x_3 + 3x_4 \qquad &= -24 \\
x_2 - 2x_3 + 2x_4 \qquad &= -7 \\
x_5 &= 4
\end{aligned}
$$

The pivot columns of the matrix are 1, 2, 5, so the basic variables are x_1, x_2, x_5 and the remaining variables x_3, x_4 are non-basic(free). Solving the basic variables, we obtain the general solution :

$$
\begin{aligned}
x_1 &= -24 + 2x_3 - 3x_4 \\
x_2 &= -7 + 2x_3 - 2x_4 \\
x_5 &= 4
\end{aligned}
$$

The final solution in terms of vector equation is therefore,

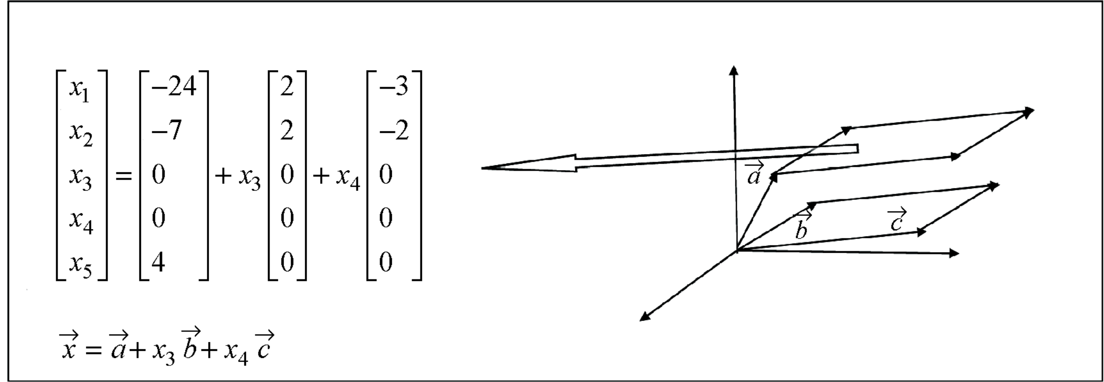


Figure 8.1

Geometrically $x_3\vec{b}$ expresses all the vectors obtained by stretching (or shrinking or reflecting) the vector $\vec{b}$ by a scalar quantity x_3. Similarly, $x_4\vec{c}$ expresses all the vectors obtained by stretching(or shrinking or reflecting) the vector $\vec{c}$ by a scalar quantity x_4. Therefore, by parallelogram law of addition of vectors, we can say the quantity $x_3\vec{b}+x_4\vec{c}$ is the diagonal passing through the point of contact of two adjacent sides $x_3\vec{b}$, $x_4\vec{c}$ of the parallelogram. Since both the scalar quantity x_3 and x_4 are arbitrary real number, $x_3\vec{b}+x_4\vec{c}$ represents the entire plane spanned by vectors $\vec{b}$ and $\vec{c}$. After translating this plane by another vector $\vec{a}$ we get the new plane $\vec{a}+x_3\vec{b}+x_4\vec{c}$ which is the final solution set of the linear system. Therefore, The geometric representation of the set of solutions is a plane passing through the vector $\vec{a}$ and parallel to the plane spanned by vectors $\vec{b}$ and $\vec{c}$ (**Figure 8.1**). Here all the vectors are usually treated as a position vectors. Therefore, if a be a point in the space then $\vec{a}$ is staright line segment from an arbitrary origin O to the point a.

Example 8.3

Consider a linear system of the form $Ax = b$ where A is $n \times n$ matrix, x is $1 \times n$ matrix b is $1 \times n$ matrix. What is the no of arithmetic operations needed to convert the augmented matrix into row echelon form using **Algorithm** 8.1?

Solution :

To determine the no of arithmetic operations to convert a matrix to row echelon form, we define a new unit called floating point operation (flop) as follows : 1 flop = 1 addition/subtraction + 1 multiplication/division

Step1. The $n \times (n + 1)$ augmented matrix of the linear system is

$$
\overbrace{\begin{matrix}
a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} & b_1 \\
\end{matrix}}^{n+1}
$$

$$
n\text{-}1 \left\{ \begin{matrix}
a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} & b_2 \\
a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} & b_3 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} & b_n
\end{matrix} \right.
$$

Figure 8.2

Step 2. Now we will convert the matrix into row echelon form.

We define $m_{i1} = \frac{a_{i1}}{a_{11}}$ for $i = 2,3,.....n$ **(8.8)**

We apply $R_i = R_i - m_{i1}R_1$ for i = 2, 3,.....n **(8.9)**

Equation (8.8) needs *(n-1)* division and equation (8.7) needs *(n-1)(n+1)* flops **(Figure 8.2)**. In the next step equation (8.6) needs *n-2* division and equation (8.7) needs *(n-2)n* flops **(Figure 8.3)**.

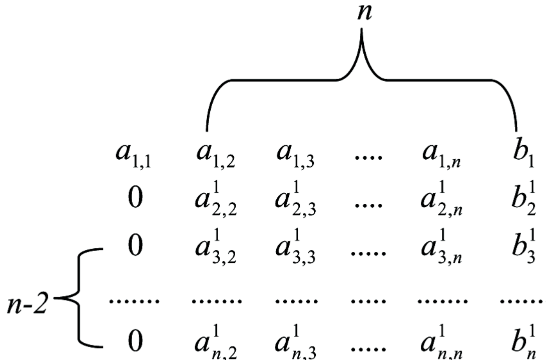


Figure 8.3

After second iteration the matrix will look like the following **(Figure 8.4)**.

$$\begin{array}{cccccc} a_{1,1} & a_{1,2} & a_{1,3} & .... & a_{1,n} & b_1 \\ 0 & a^1_{2,2} & a^1_{2,3} & .... & a^1_{2,n} & b^1_2 \\ 0 & 0 & a^2_{3,3} & ..... & a^2_{3,n} & b^2_3 \\ .... & .... & ...... & ..... & .... & .... \\ 0 & 0 & a^2_{n,3} & ..... & a^2_{n,n} & b^2_n \end{array}$$

Figure 8.4

Therefore, if the total no of operation is *T(n)* after the completion of **Algorithm 8.1**, then

$T(n) = [(n-1)(n+1)+(n-2)(n) + \ldots +1.3]$ flops $+ [(n-1) + (n-2) + \ldots +1]$ *divisions*

$= T_1$ *flops* $+ T_2$ *divisions*

Now $T_1 = (n-1)(n+1) + (n-2)(n) + (n-3)(n-1) + \ldots +1.3$

$= (n-1)(n-1+2) + (n-2)(n-2+2) + (n-3)(n-3+2) \ldots +1.(1+2)$

$= (n-1)^2 + 2(n-1) + (n-2)^2 + 2(n-2) + (n-3)^2 + 2(n-3) \ldots +1^2 + 2.1$

$= [(n-1)^2 + (n-2)^2 + (n-3)^2 + \ldots 1^2] + 2[(n-1) + (n-2) + (n-3) \ldots +1]$

$$= \left[\frac{(n-1)n(2n-1)}{6}\right] + 2\left[\frac{(n-1)n}{2}\right] = n(n-1)\left[\frac{2n-1}{6}+1\right] = \frac{n(n-1)(2n+5)}{6}$$

$$= \frac{1}{6}(2n^3 + 3n^2 - 5n) = \left(\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n\right)$$

$$T_2 = (n-1) + (n-2) + (n-3) + \ldots + 1 = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2 - n)$$

Therefore, $T(n) = \left(\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n\right) flops + \left(\frac{1}{2}n^2 - \frac{1}{2}n\right) divisions$

When n becomes very large ($n \to \infty$) the largest power of n will dominate the other terms. Ignoring the lower power of n and other constants we can write $T(n) = O(n^3)$. O (big 'O') notation is used to measure asymptotic ($n \to \infty$) upper bound of time complexity of any algorithm. In summary, we can say that when $n \to \infty$ total no of flops is proportional to n^3 in **Algorithm 8.1**.

Check Your Progress 8.1

Write a C program for back substitution method used in Gauss Elimination. Find out the solution of the system of linear equations given in **example 8.1** using the program.

Check Your Progress 8.2

Find the general solution of the linear system given below (Using the C program in **Table 8.10**).

$$\begin{array}{rrrrl} x_1 & -2x_2 & -x_3 & +3x_4 & = 0 \\ -2x_1 & +4x_2 & +5x_3 & -5x_4 & = 3 \\ 3x_1 & -6x_2 & -6x_3 & +8x_4 & = 2 \end{array}$$

Check Your Progress 8.3

An interpolating polynomial for the data (x_i, y_i) for $i = 0,1,2,\ldots,n$ is a polynomial of degree n whose graph passes through every data point. One method for finding the interpolating polynomial is to solve a system of linear equations for the $(n + 1)$ polynomial coefficients, obtained by satisfying the $(n + 1)$ polynomials with $(n + 1)$ data points. Find the interpolating polynomial $f(x) = a_0 + a_1x + a_2x^2$ for the data (1,12), (2,15), (3,16) (Using the C program in **Table 8.10**). That is, find the value of a_1, a_1, a_2.

Check Your Progress 8.4

Do the three lines $x_1 - 4x_2 = 1$, $2x_1 - x_2 = -3$ and $-x_1 - 3x_2 = 4$ have a common point of intersection. Explain (Using the C program in **Table 8.10**).

Check Your Progress 8.5

Consider a given matrix $A(n, n+1)$ in row echelon form. What is the no of arithmetic operations needed to convert the given row echelon matrix into reduced row echelon form using **Algorithm 8.2** ?

Linear Independence :

An indexed set of vectors $\{v_1, v_2, ...v_p\}$ in R^n is said to be ***linearly independent*** if the vector equation $x_1v_1 + x_2v_2 + ... + x_pv_p = 0$ has only the trivial solution. The set $\{v_1, v_2, ...v_p\}$ is said to be ***linearly dependent*** if there exists weights $x_1, x_2, ..., x_p$ not all zero, such that

$$x_1v_1 + x_2v_2 + ... + x_pv_p = 0$$

Check Your Progress 8.6

Let $v_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, v_2 = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, v_3 = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$. Determine if the set is linearly independent

(Using the C program in **Table 8.10**).

Check Your Progress 8.7

Given two matrices $A = \langle a_{ij} \rangle$ and $B = \langle b_{ij} \rangle$, write a C program to check if the matrices are equal.

Check Your Progress 8.8

Given two (m x n) matrices $A = \langle a_{ij} \rangle$ and $B = \langle b_{ij} \rangle$, write a C program to add A and B.

Matrix Transpose :

Given $m \times n$ matrix A, the transpose of A is the $n \times m$ matrix denoted by A^T where columns are formed from the corresponding rows of A.

Check Your Progress 8.9

Write a C program to find the transpose of a given matrix $A = \begin{bmatrix} -5 & 2 \\ 1 & -3 \\ 0 & 4 \end{bmatrix}$

8.5 Summary

Two direct methods for solving systems of linear algebraic equations are presented in this chapter, namely Gauss Elimination method and Gauss Jordan Elimination method. Two important matrix forms, row-echelon and reduced row-echelon forms have been also discussed with the help of C program in this chapter. How these forms help us to find linear dependent/independent vectors is also illustrated at the end of the chapter.

8.6 References and Further Reading

1. Elementary Numerical Analysis – An algorithmic Approach, Third Edition, S.D. Conte, Carl de Boor, Tata McGraw-Hill, 2005
2. Linear Algebra and Its Application, David C. Lay, Pearson, 2007
3. Numerical Recipes in C, Second Edition, H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Tata McGraw Hill, 2003.
4. Numerical Methods for Engineers and Scientists, Second Editon, Joe D. Hoffman, CRC Press, 2001.

8.7 Hints and Solution

Solution 8.2 : Enter the number of rows and columns : 3 5 The echelon matrix is -- 1.00 -2.00 -1.00 3.00 0.00 0.00 0.00 3.00 1.00 3.00 0.00 0.00 0.00 0.00 5.00 The system is inconsistent	**Solution 8.3 :** The polynomial is $f(x) = 7 + 6x - x^2$
Solution 8.4 :Solution 8.5 : The common point is $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1.86 \\ -0.71 \end{bmatrix}$	$T(n) = O(n^2)$

Solution 8.6 :	
Enter the number of rows and columns : 3 4 The given matrix is --- 1.00 4.00 2.00 0.00 2.00 5.00 1.00 0.00 3.00 6.00 0.00 0.00	The echelon matrix is ---------------------------------- 1.00 4.00 2.00 0.00 0.00 -3.00 -3.00 0.00 0.00 0.00 0.00 0.00
The reduced echelon matrix is -- 1.00 0.00 -2.00 0.00 -0.00 1.00 1.00 -0.00 0.00 0.00 0.00 0.00	

The reduced matrix shows that x_3 is free variable. The reduced system could be written as

$$x_1 - 2x_3 = 0 \Rightarrow x_1 = 2x_3$$
$$x_2 + x_3 = 0 \Rightarrow x_2 = -x_3$$
$$0 = 0 \Rightarrow x_3 = x_3$$

The solution set is $x = x_3 \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}$ which contains a nonzero solution when $x_3 \neq 0$.

Therefore, the set $\{v_1, v_2, v_3\}$ is not linearly independent.

Unit - 9 ❑ Application of C Programming : Solution of System of Linear Equations by Direct Methods : Matrix Inverse and LU Decomposition

Structure

9.0 Introduction

9.1 Objectives

9.2 Matrix Multiplication

9.3 Matrix Inverse

9.4 The Matrix Inverse Method

9.5 LU Decomposition

9.6 Summary

9.7 References and Further Reading

9.8 Hints and Solution

9.0 Introduction

In the last unit, Gauss Elimination and Gauss-Jordan Elimination methods for solving system of linear equations along with their C-implementation was discussed. In this chapter, two more direct methods for solving system of linear equations will be discussed, namely Matrix Inverse Method and LU Decomposition Method. As a prerequisite, the learner needs to have complete understanding of the algorithm (**Algorithm 8.1** and **8.2**) and corresponding C implementations mentioned in **Unit 8**.

9.1 Objectives

After going through this topic, the learner should be able to

- Understand the concept of matrix multiplication.
- Multiply two matrices using C program.
- Understand the concept of matrix inverse.
- Determine inverse of a matrix using C program.

- Solve a system of linear algebraic equations by Matrix Inverse Method using C program.
- Understand the purpose of matrix factorization/decomposition.
- Solve a system of linear algebraic equations by LU decomposition method using C program.

9.2 Matrix Multiplication :

The process of multiplying matrices is called *matrix multiplication* and is defined, in general, as follows : Let $A = \langle a_{ij} \rangle$ be an $m \times n$ matrix, $B = \langle b_{ij} \rangle$ an $p \times r$ matrix; then the matrix $C = \langle c_{ij} \rangle$ *is the* (matrix) **product of** A **with** B or $C = A\ B$, provided $n = p$ and C is of order $m \times r$ and the entry in row i and column j of C is the sum of the products of corresponding entries from row i of A and column j of B. If c_{ij} denotes the (i, j)-entry of AB, then $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} \ldots + a_{in}b_{nj}$ (see **Figure 9.1**).

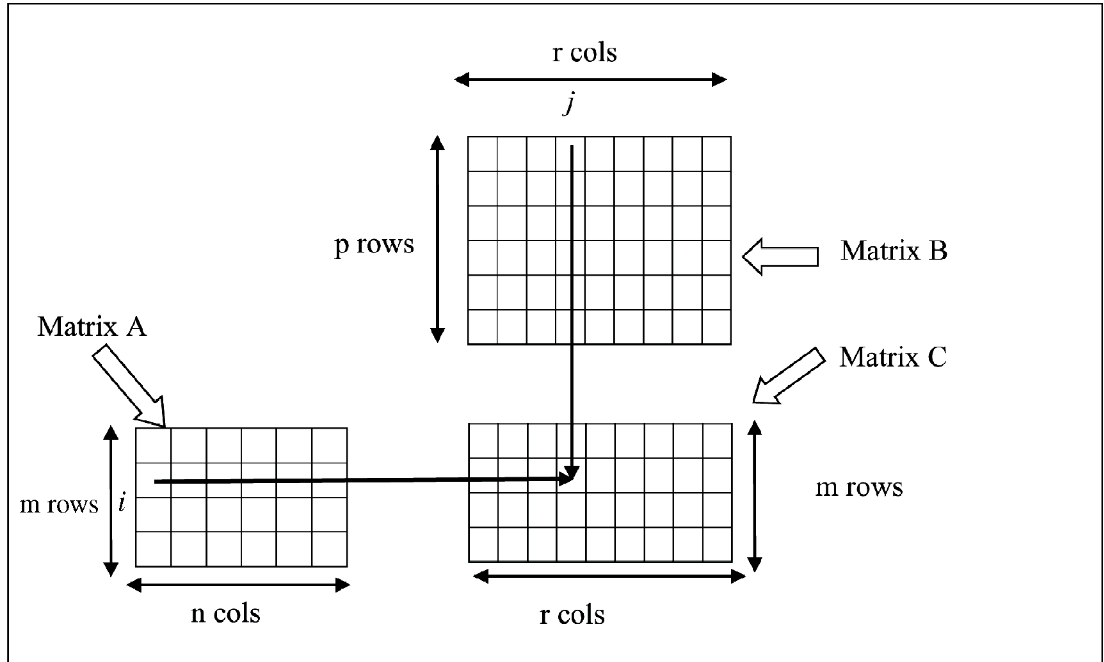


Figure 9.1

Algorithm 9.1 : Matrix Multiplication
Input **:** Matrix A (m, n) and B (p, r)
Output **:** Matrix C (m, r).

Steps	*Description*
1.	*If* $(n \neq p)$ *then stop, otherwise go to step 2.*
2.	*for i=1, 2, ………, m do:*
	for j=1, 2, …………, r do:
	c[i][j] = 0
	for k=1, 2, ……, n do:
	*c[i][j] =c[i][j] + a[i][k]*b[k][j]*
3.	*Print the matrix C and stop.*

When a matrix B multiplies a vector $\mathbf{x}$, it transforms $\mathbf{x}$ into the vector $B\mathbf{x}$. If this vector is then multiplied in turn by a matrix A, the resulting vector is $A(B\mathbf{x})$ (See **Figure 9.2**).

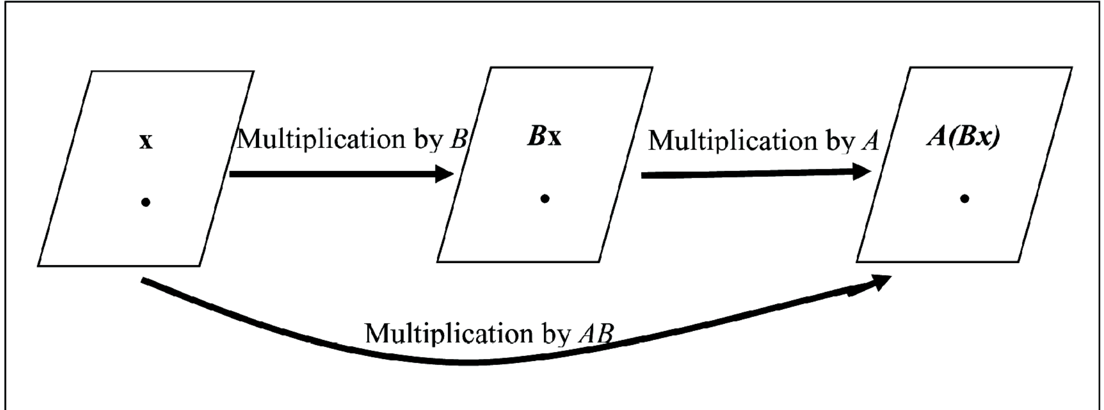


Figure 9.2

Check Your Progress 9.1

Write a C program to multiply two matrices A and B (**Algorithm 9.1**). Using the program compute AB where $A = \begin{bmatrix} 2 & 3 \\ 1 & -5 \end{bmatrix}$ and $B = \begin{bmatrix} 4 & 3 & 6 \\ 1 & -2 & 3 \end{bmatrix}$.

Check Your Progress 9.2

Using the program for multiplying two matrices compute AB, where

$$A = \begin{bmatrix} 2 & -5 & 0 \\ -1 & 3 & -4 \\ 6 & -8 & -7 \\ -3 & 0 & 9 \end{bmatrix} \text{ and } B = \begin{bmatrix} 4 & -6 \\ 7 & 1 \\ 3 & 2 \end{bmatrix}.$$

9.3 Matrix Inverse :

An $n \times n$ matrix A is said to be **invertible** if there is an $n \times n$ matrix C such that

$$CA = I \text{ and } AC = I$$

where $I = I_n$, the $n \times n$ identity matrix. In this case, C is an **inverse** of A **(Figure 9.3)**. In fact C is uniquely determined by A and denoted by A^{-1}. So that

$$A^{-1}A = I \text{ and } AC^{-1} = I$$

If A is an invertible $n \times n$ matrix, then for each **b** in R^n, the equation $A\mathbf{x} = \mathbf{b}$ has the unique solution $\mathbf{x} = A^{-1}\mathbf{b}$.

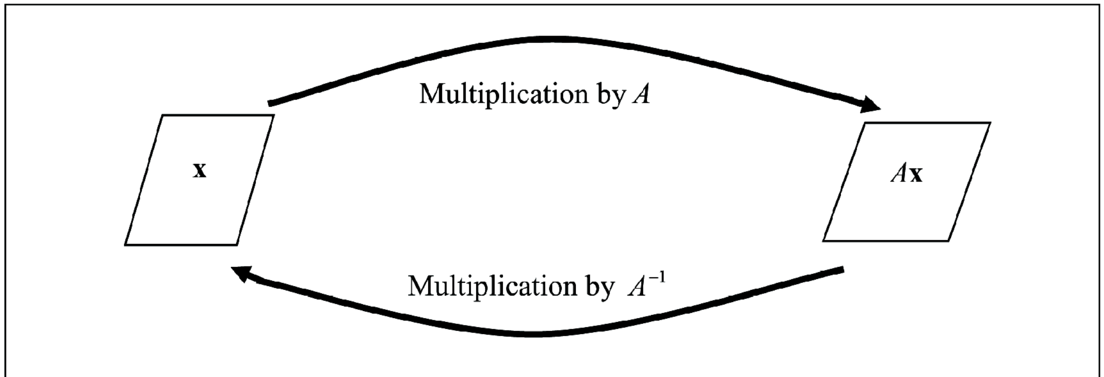


Figure 9.3

Theorem 9.1

An $n \times n$ matrix A is invertible if and only if A is row equivalent to I_n, and in this case the sequence of elementary row operations that reduces A to I_n also transforms I_n to A^{-1}.

If A is invertible, then by **Theorem 9.1** A is row equivalent to I_n ($A \sim I_n$). Then, since each step of the row reduction A corresponds to left multiplication by an elementary matrix, there exists elementary matrices $E_1, E_2, \ldots\ldots\ldots, E_p$ such that

$$A \sim E_1A \sim E_2(E_1A) \sim E_3(E_2E_1A) \sim \ldots \sim E_p(E_{p-1} \ldots\ldots E_1\mathrm{A}) = I_n$$

That is, $E_p(E_{p-1} \ldots\ldots E_1A) = I_n$ **(9.1)**

Since the product $E_1E_2 \ldots\ldots E_p$ of invertible matrix is invertible, **equation 9.1** leads to

$$(E_pE_{p-1}..........E_1)^{-1}(E_pE_{p-1}..........E_1)A = (E_pE_{p-1}..........E_1)^{-1}I_n$$

$$\Rightarrow A = (E_pE_{p-1}.............E_1)^{-1}$$

$$\Rightarrow A^{-1} = [(E_pE_{p-1}..............E_1)^{-1}]^{-1} = E_pE_{p-1}...........E_1$$

Then $A^{-1} = E_pE_{p-1} \ldots E_1I_n$, which says that A^{-1} results from applying E_1, E_2, ..., E_p successively to I_n. This is the same sequence in **equation (9.1)** that produces A to I_n.

The following procedure **(Table 9.1)** outlines how to find A^{-1} :

Steps	***Description***
1.	*Place A and I side by side to form augmented matrix [A I].*
2.	*Use the row reduction method of **Algorithm 8.1** and **Algorithm 8.2** together (step 1 to 4 of **Table-8.9**) to convert matrix [A I] into reduced row echelon form.*
3.	*If row reduction process in step 2 converts the augmented matrix [A I] to [I A^{-1}] then A is invertible and stop, otherwise A is not invertible.*

Table 9.1

Example 9.1

Find the inverse of the matrix $A = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 3 \\ 4 & -3 & 8 \end{bmatrix}$ if exists.

Solution :

The augmented matrix *[A I]* is $\begin{bmatrix} 0 & 1 & 2 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 1 & 0 \\ 4 & -3 & 8 & 0 & 0 & 1 \end{bmatrix}$

After executing the program of Table 8.9 (steps 1 to 4 of **Table 8.9**) with the above augmented matrix as input we get the reduced echelon matrix as following

```
----------------------------------------
1.00 0.00 0.00 -4.50  7.00 -1.50
0.00 1.00 0.00 -2.00  4.00 -1.00
0.00 0.00 1.00  1.50 -2.00  0.50
```

The first 3 columns forms I_3. Therefore the remaining 3 columns forms A^{-1}.

$$\text{So } A^{-1} = \begin{bmatrix} -4.5 & 7 & -1.5 \\ -2 & 4 & -1 \\ 1.5 & -2 & .5 \end{bmatrix}.$$

Check Your Progress 9.3

Find the inverse of the matrix $A = \begin{bmatrix} 1 & -2 & 1 \\ 4 & -7 & 3 \\ -2 & 6 & -4 \end{bmatrix}$ if it exists. Also validate using matrix multiplication algorithm that $AA^{-1} = I_3$.

Check Your Progress 9.4

Find the inverse of the matrix $A = \begin{bmatrix} 1 & -2 & 1 \\ -1 & 5 & 6 \\ 5 & -4 & 5 \end{bmatrix}$ if it exists.

9.4 The Matrix Inverse Method

Systems of linear algebraic equations having equal number of variables and equating can be solved using the matrix inverse, A^{-1}. Consider the general system of linear algebraic equations :

$$Ax = b \tag{9.2}$$

Multiplying the **equation 9.2** by A^{-1} yields

$$A^{-1}Ax = A^{-1} b$$

$$\Rightarrow x = A^{-1} b \tag{9.3}$$

Thus, when the matrix inverse A^{-1} of the coefficient matrix A is known, the solution vector x is simply the product of the matrix inverse A^{-1} and the right-hand-side vector b. Not all matrices have inverses. Singular matrices, that is, matrices whose determinant is zero, do not have inverses and therefore, they are not invertible. The corresponding system of equations does not have a unique solution. The following procedure **(Table 9.2)** outlines the Matrix Inverse Method to find out the solution of systems of linear equations.

Steps	*Description*
1.	*Use the procedure in **Table 9.1** to find the inverse A^{-1} of the the coefficient matrix A of the system of linear equations.*
2.	*Multiply the matrix A^{-1} and the right hand side vector b of the system of linear equations. Refer **Algorithm 9.1** and **Check your progress 9.1**.*
3.	*The result of the multiplication in step 2 gives the solution vector x of the system of linear equations.*

Table 9.2

Example 9.2

Consider the following system of linear equations given in **section 8.4** in **Unit 8**.

$$\begin{aligned} x_1 - 2x_2 + x_3 &= 0 \\ 2x_2 - 8x_3 &= 8 \\ -4x_1 + 5x_2 + 9x_3 &= -9 \end{aligned}$$

Find the solution of the above linear system.

Solution :

Here the coefficient matrix $A = \begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -8 \\ -4 & 5 & 9 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \\ 8 \\ -9 \end{bmatrix}$.

Apply the step 1 of **Table 9.1.**

$$[A \quad I] = \begin{bmatrix} 1 & -2 & 1 & 1 & 0 & 0 \\ 0 & 2 & -8 & 0 & 1 & 0 \\ -4 & 5 & 9 & 0 & 0 & 1 \end{bmatrix}$$

Apply the step 2 of **Table 9.1** to get the reduced row echelon matrix.

```
Enter the no of rows and columns : 3  6
The given matrix is

----------------------------------------
 1.00 -2.00  1.00  1.00  0.00  0.00
 0.00  2.00 -8.00  0.00  1.00  0.00
-4.00  5.00  9.00  0.00  0.00  1.00

The echelon matrix is
```

```
------------------------------------------
1.00  -2.00   1.00   1.00   0.00   0.00
0.00   2.00  -8.00   0.00   1.00   0.00
0.00   0.00   1.00   4.00   1.50   1.00
```

The reduced echelon matrix is

```
------------------------------------------
1.00   0.00   0.00  29.00  11.50   7.00
0.00   1.00   0.00  16.00   6.50   4.00
0.00   0.00   1.00   4.00   1.50   1.00
```

Therefore, $A^{-1} = \begin{bmatrix} 29 & 11.5 & 7 \\ 16 & 6.5 & 4 \\ 4 & 1.5 & 1 \end{bmatrix}$

Apply the step 3 of **Table 9.1** to multiply A^{-1} and b using the **Algorithm 9.1**.

$$\begin{bmatrix} 29 & 11.5 & 7 \\ 16 & 6.5 & 4 \\ 4 & 1.5 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 8 \\ -9 \end{bmatrix} = \begin{bmatrix} 29 \\ 16 \\ 3 \end{bmatrix}$$

Therefore, the solution $x_1 = 29$, $x_2 = 16$ and $x_3 = 3$.

Note :

In practical work A^{-1} is seldom computed, unless the entries of A^{-1} are needed. Computing both A^{-1} and $A^{-1}\,b$ takes about three times as many arithmetic operations as solving $Ax = b$ by row reduction, and row reduction may be more accurate.

Check Your Progress 9.5

Solve the following system using Matrix Inverse Method :

$$\begin{aligned} 8x_1 + 6x_2 &= 2 \\ 5x_1 + 4x_2 &= -1 \end{aligned}$$

9.5 LU Decomposition

LU factorization, is motivated by the fairly common industrial and business problem of solving several sets of equations, all with the same coefficient matrix :

$$Ax = b_1, \qquad Ax = b_2, \qquad Ax = b_3, \ldots, Ax = b_n \tag{9.4}$$

that is, for solving the equation $Ax = b$ with different values of b for the same A. Note that in row reduction method left-hand side A and the right-hand side b are modified within the same loop and there is no way to save the steps taken during the elimination process. If the equation has to be solved for different values of b, the elimination step has to be done all over again.

To avoid this, one could solve the first **equation 9.4** by row reduction and obtain LU factorization/decomposition of A at the same time. Thereafter, the remaining equations are solved with LU factorization.

Assume that A is an $m \times n$ matrix that can be row reduced to echelon form without row interchanges. Then A can be written in the form $A = LU$, where L is an $m \times m$ lower triangular matrix with 1's on the diagonal and U is $m \times n$ echelon form of A. For instance, see the example in **Figure 9.4**. Such a factorization is called *LU* factorization of A. The matrix L is invertible and is called unit lower triangular matrix.

$$A = \underset{L}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ * & 1 & 0 & 0 \\ * & * & 1 & 0 \\ * & * & * & 1 \end{bmatrix}} \underset{U}{\begin{bmatrix} \bullet & * & * & * & * \\ 0 & \bullet & * & * & * \\ 0 & 0 & 0 & \bullet & * \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}$$

Figure 9.4

Symbol (*) denotes any real number and symbol (•) denotes leading entry of a row. When $A = LU$, the equation $A\mathbf{x} = \boldsymbol{b}$ can be written as $L(U\mathbf{x}) = \boldsymbol{b}$. Writing $\boldsymbol{y}$ for $U\mathbf{x}$, one can find $\boldsymbol{x}$ by solving the following pair of equations :

$$\begin{aligned} Ly &= b \\ Ux &= y \end{aligned} \qquad \textbf{(9.5)}$$

Each equation is easy to solve because L and U are triangular **(Figure 9.4)**.

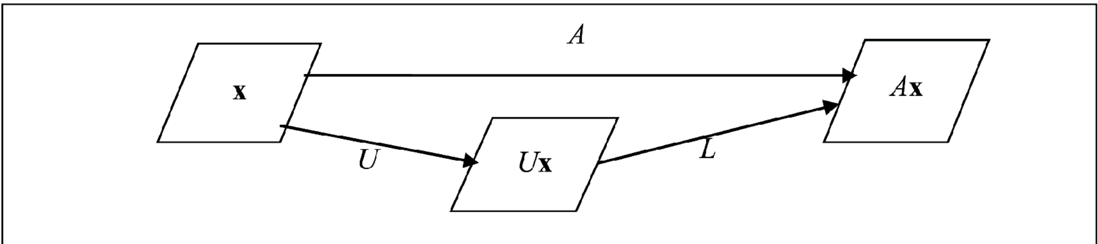


Figure 9.5

LU Decomposition Algorithm :

Suppose A can be reduced to an echelon form U using only row replacement that add multiple of one row to another row below it. In this case there exists unit lower triangular matrices $E_1, E_2, \ldots\ldots, E_p$ such that

$$E_p \ldots\ldots E_2 E_1 A = U \tag{9.6}$$

Then $$A = (E_p \ldots\ldots E_2 E_1)^{-1} U$$

where $$L = (E_p \ldots\ldots E_2 E_1)^{-1} \tag{9.7}$$

It can be shown that product and inverse of unit triangular matrices are also unit triangular. Thus L is unit lower triangular.

Note that the row operation in **equation 9.6**, which reduce A to U, also reduces L to I, because $E_p \ldots\ldots E_2 E_1 L = (E_p \ldots\ldots E_2 E_1)(E_p \ldots\ldots E_2 E_1)^{-1} = I$. This observation is the key to constructing L. The following procedure **(Table 9.3)** outlines the LU decomposition algorithm.

Steps	***Description***
1.	*Reduce A to an echelon form U by a sequence of row replacement operations, if possible*
2.	*Place entries in L such that the same sequence of row operations reduce L to I.*

Table 9.3

Let us find a LU factorization of matrix **(8.3a)**

$$A = \begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ -4 & -5 & 3 & -8 & 1 \\ 2 & -5 & -4 & 1 & 8 \\ -6 & 0 & 7 & -3 & 1 \end{bmatrix}$$

Since A has four rows, L should be 4×4. The first column of L is the first column of A divided by the top pivot entry.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 1 & & 1 & 0 \\ -3 & & & 1 \end{bmatrix} \tag{9.8}$$

Compare the first columns of A and L. The row operations that create zeros in the first column of A will also create zeros in the first column of L. The rest of L can be determined by same correspondence of row operations.

Iteration(i)	A_i	L_i
0	$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ -4 & -5 & 3 & -8 & 1 \\ 2 & -5 & -4 & 1 & 8 \\ -6 & 0 & 7 & -3 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
1	$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & -9 & -3 & -4 & 10 \\ 0 & 12 & 4 & 12 & -5 \end{bmatrix}$	$\begin{bmatrix} 2 & 0 & 0 & 0 \\ -4 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ -6 & 0 & 0 & 1 \end{bmatrix}$
2	$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 4 & 7 \end{bmatrix}$	$\begin{bmatrix} 2 & 0 & 0 & 0 \\ -4 & 3 & 0 & 0 \\ 2 & -9 & 1 & 0 \\ -6 & 12 & 0 & 1 \end{bmatrix}$
3	$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}$	$\begin{bmatrix} 2 & 0 & 0 & 0 \\ -4 & 3 & 0 & 0 \\ 2 & -9 & 2 & 0 \\ -6 & 12 & 4 & 1 \end{bmatrix}$
4	$\begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}$	$\begin{bmatrix} 2 & 0 & 0 & 0 \\ -4 & 3 & 0 & 0 \\ 2 & -9 & 2 & 0 \\ -6 & 12 & 4 & 5 \end{bmatrix}$
Final Step	$U = \begin{bmatrix} 2 & 4 & -1 & 5 & -2 \\ 0 & 3 & 1 & 2 & -3 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}$	$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 1 & -3 & 1 & 0 \\ -3 & 4 & 2 & 1 \end{bmatrix}$

Table 9.4

C Program for *LU* Decomposition is given in **Table 9.5**.

```
#include<stdio.h>
#include<math.h>
void main()
{
   float a[20][20]={{1,-2,-4,-3},{2,-7,-7,-6},{-1,2,6,4},{-4,-1,9,8}};
   float temp[20],t,t1,c[20][20]={},e=.0001;
   int m, n, i, j, k, p,r, lead=0,q;
   printf("Enter the number of rows and columns: ");
   scanf("%d%d",&m,&n);
   /* Display the given matrix*/
   printf("\n\nThe given matrix is\n");
   printf("----------------------------------------\n");
   for (i=0; i<m; i++)
   {
      for (j=0; j<n; j++)
      {
         printf("%6.2f\t", a[i][j]);
      }
      printf("\n");
   }
/* Insert one in all diagonal entries to build the Lower Triangular Matrix */
   for(r=0;r<m;r++)
   {
      c[r][r]=1;
   }
/*Main logic of row echelon form*/
   i=0; /*i is pointing to first row*/
   p=0; /*p is pointing to first row of Lower Triangular matrix*/
   q=0; /*q is pointing to first column of Lower Triangular matrix*/
   while(i<m-1 && lead<n)
   {
      if (fabs(a[i][lead])>e) /*Check if the entry is nonzero*/
      { /*Insert the entire column of leading entry of a matrix into Lower
         Triangular matrix c*/
```

```
        for (r=i;r<m;r++)
            {
                c[r][q]=a[r][lead];
            }
        /*Apply row operation to create zeros below leading entry*/
        for(k=i+1; k<m; k++)
            {
                t=a[k][lead]/a[i][lead];
                for(j=lead; j<n; j++)
                a[k][j]=a[k][j]-t*a[i][j];
            }
        i++;
        lead++;
        q++;
    }
    else
    {
        /*Search a row below current row such that the entry in the same
        column is non-zero */
        for(p=i+1; p<m; p++)
        {
            if (fabs(a[p][i])>e) /*Check if the entry is nonzero*/
                break;
        }
        /*if non-zero entry found swap the rows*/
        if(p<m)
        {
            for(j=0; j<n; j++)
            {
                temp[j]=a[i][j];
                a[i][j]=a[p][j];
                a[p][j]=temp[j];
            }
        }
    else
        /*if all entry below the leading entry is zero then go to search
```

```
        the next column*/
        lead++;
      }
    }
/*Divide each entry of Lower Triangular matrix by diagonal element*/
   for(j=0;j<m;j++)
     {
        t1=c[j][j];
        for(i=j;i<m;i++)
        c[i][j]=c[i][j]/t1;
     }
     printf("\n\nThe Upper Triangular Matrix is\n");
     printf("------------------------------------------\n");
   /* Display the Upper Triangular matrix */
   for (i=0; i<m; i++)
     {
        for (j=0; j<n; j++)
        {
          printf("%6.2f\t", a[i][j]);
        }
        printf("\n");
     }
        printf("\n\nThe Lower Triangular Matrix is\n");
        printf("------------------------------------------\n");
   /* Display the Lower Triangular matrix */
   for (i=0; i<m; i++)
   {
     for (j=0; j<m; j++)
     {
        printf("%6.2f\t", c[i][j]);
     }
     printf("\n");
   }
}
```

Table 9.5

Example 9.3

Find a LU decomposition of $A = \begin{bmatrix} 2 & -4 & -2 & 3 \\ 6 & -9 & -5 & 8 \\ 2 & -7 & -3 & 9 \\ 4 & -2 & -2 & -1 \\ -6 & 3 & 3 & 4 \end{bmatrix}$ using the program in **9.5**.

Solution :

Execute the program of **Table 9.5** considering the input as A.

Enter the number of rows and columns : 5 4

```
                 The given matrix is
             ------------------------------------------
                  2.00  -4.00  -2.00  3.00
                  6.00  -9.00  -5.00  8.00
                  2.00  -7.00  -3.00  9.00
                  4.00  -2.00  -2.00 -1.00
                 -6.00   3.00   3.00  4.00

The Upper Triangular Matrix is          The Lower Triangular Matrix is
------------------------------------------  ------------------------------------------
  2.00  -4.00  -2.00   3.00              1.00   0.00   0.00   0.00   0.00
  0.00   3.00   1.00  -1.00              3.00   1.00   0.00   0.00   0.00
  0.00   0.00   0.00   5.00              1.00  -1.00   1.00   0.00   0.00
  0.00   0.00   0.00   0.00              2.00   2.00  -1.00   1.00   0.00
  0.00   0.00   0.00   0.00             -3.00  -3.00   2.00   0.00   1.00
```

Solving the system of linear equations using L and U :

After decomposing the matrix, A into L and U, the next step is to solve $Ly = b$ and $Ux = y$ to get the unique solution of a system of linear equations $Ax = b$. The generic form of $Ly = b$ is as follows :

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ a_{10} & 1 & 0 & 0 & \cdots & 0 \\ a_{20} & a_{21} & 1 & 0 & \cdots & 0 \\ a_{30} & a_{31} & a_{32} & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{(m-1)0} & a_{(m-1)1} & a_{(m-1)2} & a_{(m-1)3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \cdots \\ y_{m-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \cdots \\ b_{m-1} \end{bmatrix} \quad \textbf{(9.9)}$$

$$\Rightarrow \begin{array}{lcllll} y_0 & = & b_0 & & & \\ y_1 & = & b_1 & - a_{10}y_0 & & \\ y_2 & = & b_2 & - a_{20}y_0 & - a_{21}y_1 & \\ y_3 & = & b_3 & - a_{30}y_0 & - a_{31}y_1 & - a_{32}y_2 \\ .... & .... & .... & .... & .... & .... \\ y_{m-1} & = & b_{m-1} & - a_{(m-1)0}y_0 & .... & - a_{(m-1)(m-2)}y_{m-1} \end{array} \qquad (9.10)$$

$$\Rightarrow \begin{array}{lcl} y_0 & = & b_0 \\ y_i & = & b_i - \sum_{j=0}^{j<i} a_{ij}y_j \quad \text{where} \quad i = 1, 2, ...m-1 \end{array} \qquad (9.11)$$

The algorithm to solve $Ly = b$ is outlined in **Algorithm 9.2**.

Algorithm 9.2 : Solve $Ly = b$ where L is Unit Lower Triangular matrix generated by LU decomposition.

Input : A (m x m) Unit Lower Triangular matrix $L(m, m)$. A (m x 1) matrix $b(m, 1)$ which is the right hand side vector of $Ax = b$.

Output : A (m x 1) matrix $y(m, 1)$ which is the solution of $Ly = b$.

Steps	***Description***
1.	*[y][0]= b[0]*
2.	*for i=1, 2, ………, m-1 do:*
3.	*s=0*
4.	*for j=0, 1,…………., i-1 do:*
5.	*s = s + a[i][j]*y[j]*
6.	*y[i] = b[i] – s*
7.	*Print Matrix y and stop.*

Now to get the final solution we need to solve the equation $Ux = y$. For unique solution scenario the generic form of $Ux = y$ is as follows :

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & .... & a_{0(m-1)} \\ 0 & a_{11} & a_{12} & a_{13} & .... & a_{1(m-1)} \\ 0 & 0 & a_{22} & a_{23} & .... & a_{2(m-1)} \\ 0 & 0 & 0 & a_{33} & .... & a_{3(m-1)} \\ .... & .... & .... & .... & .... & .... \\ 0 & 0 & 0 & 0 & .... & a_{(m-1)(m-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ .... \\ x_{m-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ .... \\ y_{m-1} \end{bmatrix} \qquad (9.12)$$

$$
\begin{aligned}
x_0 &= (y_0 - a_{01}x_1 \quad \ldots \quad - a_{0(m-1)}x_{m-1})/a_{00} \\
x_1 &= (y_1 - a_{12}x_2 \quad \ldots \quad - a_{1(m-1)}x_1)/a_{11} \\
\Rightarrow \ldots &= \ldots \quad \ldots \quad \ldots \qquad (9.13) \\
x_{m-3} &= (y_{m-3} - a_{(m-3)(m-1)}x_{m-1} - a_{(m-3)(m-2)}x_{m-2})/a_{(m-3)(m-3)} \\
x_{m-2} &= (y_{m-2} - a_{(m-2)(m-1)}x_{m-1})/a_{(m-2)(m-2)} \\
x_{m-1} &= y_{m-1}/a_{(m-1)(m-1)}
\end{aligned}
$$

$$
\begin{aligned}
x_{m-1} &= y_{m-1}/a_{(m-1)(m-1)} \\
\Rightarrow x_i &= y_i - \sum_{j=m-1}^{j>i} a_{ij}x_j \quad \text{where } i = m-2,\ m-3, \ldots, 2, 1, 0 \qquad (9.14)
\end{aligned}
$$

The algorithm to solve $Ux = y$ is outlined in **algorithm 9.3**

Algorithm 9.3 : Solve $Ux = y$ where U is Upper Triangular matrix generated by LU decomposition.

Input : A (m x m) Upper Triangular matrix $U(m, m)$. A (m x 1) matrix $y(m, 1)$ which is the solution of the system $Ly = b$.

Output : A (m x 1) matrix $x(m, 1)$ which is the solution of $Ux = y$.

Steps	***Description***
1.	$x[m-1] = \dfrac{y[m-1]}{a[m-1][m-1]}$
2.	*for i=m-2, m-3 ,.....,2,1,0 do:*
3.	*s=0*
4.	*for j=m-1, m-2 ,.....,i+1 do:*
5.	*s = s + a[i][j]* x[j]*
6.	*x[i] = (y[i] –s) / a[i][i]*
7.	*Print Matrix x and stop.*

Check Your Progress 9.6

Write C programs for **Algorithm 9.2** and **Algorithm 9.3**. Append this program code with the program in **Table 9.5**.

Example 9.4

Solve the following set of linear equations by LU decomposition Method.

$$\begin{array}{rcrcrcr} x_1 & - & 2x_2 & + & x_3 & = & 0 \\ & & 2x_2 & - & 8x_3 & = & 8 \\ -4x_1 & + & 5x_2 & + & 9x_3 & = & -9 \end{array}$$

Solution :

```
Enter the number of rows and columns : 3  3
The given coefficient matrix is
-----------------------------------------
1.00  -2.00   1.00
0.00   2.00  -8.00
-4.00  5.00   9.00

The Upper Triangular Matrix is              The Lower Triangular Matrix is
-----------------------------------------   -----------------------------------------
1.00  -2.00   1.00                           1.00   0.00   0.00
0.00   2.00  -8.00                           0.00   1.00   0.00
0.00   0.00   1.00                          -4.00  -1.50   1.00

The y Matrix is                             The x Matrix or final solution is
-----------------------------------------   -----------------------------------------
y[0] = 0.00                                 x[0] = 29.00
y[1] = 8.00                                 x[1] = 16.00
y[2] = 3.00                                 x[2] = 3.00
```

Check Your Progress 9.7

Solve the following set of linear equations by LU decomposition Method :

$$\begin{array}{rcrcrcrcr} 3x_1 & - & 7x_2 & - & 2x_3 & + & 2x_4 & = & -9 \\ -3x_1 & + & 5x_2 & + & x_3 & & & = & 5 \\ 6x_1 & - & 4x_2 & & & - & 5x_4 & = & 7 \\ -9x_1 & + & 5x_2 & - & 5x_3 & + & 12x_4 & = & 11 \end{array}$$

Check Your Progress 9.8

A set of linear equations is given below :

$$\begin{array}{rcrcrcrcr} x_1 & - & 2x_2 & - & 4x_3 & - & 3x_4 & = & 1 \\ 2x_1 & - & 7x_2 & - & 7x_3 & - & 6x_4 & = & 7 \\ -x_1 & + & 2x_2 & + & 6x_3 & + & 4x_4 & = & 0 \\ -4x_1 & - & x_2 & + & 9x_3 & + & 8x_4 & = & 3 \end{array}$$

Solve the above equation using C programming by following methods and compare the results.

(a) LU decomposition algorithm. **(C Program in Table 9.5, Algorithm 9.2, 9.3)**

(b) Row reduction algorithm **(Algorithm 8.1, 8.2)**

Check Your Progress 9.9

Consider a given invertible matrix A (n, n) is decomposed into L (*Unit Lower Triangular*) and U (*Upper Triangular*) matrices. To solve the linear system $Ax = b$, **Algorithm 9.2** and **9.3** are used. What is the number of arithmetic operations needed to implement these two algorithms? Express the number in terms of O (big 'O') notation.

Comparison between LU Decomposition algorithm and Row Reduction (Gaussian Elimination) algorithm :

To compare the LU decomposition and Row Reduction, let us consider the problem of solving n sets of linear equations with the same coefficient matrix A but different right hand sides $\boldsymbol{b}$. The problem can be expressed as

Solve $Ax_i = b_i$ where $i = 1, 2, 3, \ldots, n$ and $x_i = (x_{i0}, x_{i1}, \ldots, x_{i(n-1)})^T$,

$$b_i = (b_{i0}, b_{i1}, \ldots, b_{i(n-1)})^T$$

Now LU Decomposition method applies following steps to solve the above system :

1. Decompose A into L and U :
   Known as forward elimination method. This method is similar to the row reduction method to convert a matrix into echelon form **(Algorithm 8.1)**. Only inserting the entries of L are the additional step which doesn't add any new flops. Therefore, the total number of flops is approximately proportional to n^3 (Refer **Example 8.3**) when n is very large. Since each set of linear equations have the same coefficient matrix A, this decomposition method needs to be executed only for once.

2. Solve the set $Ly_i = b_i$:
   Known as forward substitution method. The total number of flops is approximately proportional to n^2 (Refer **Check Your Progress 9.9**) when n is very large. Since each set of equations have different right hand side, this forward substitution needs to be executed n times for n set of equations. Therefore, total number of flops is approximately proportional to $n \times n^2 = n^3$.

3. Solve the set $Ux_i = y_i$:

   Known as back substitution method and here also the total number of flops is approximately proportional to n^2 (Refer **Check Your Progress 9.9**) when n is very large. By the same reasoning this step also needs to be executed for n times and total no of flops is also proportional to $n \times n^2 = n^3$.

Therefore, the total number of flops required to solve n set of systems using *LU* Decomposition method is proportional to $n^3 + n^3 + n^3 = 3n^3$. When n becomes very large we can avoid the proportionality constants and we can write total number of flops is = $O(n^3)$

Now *Row Reduction (Gauss Elimination)* method applies following steps to solve the above system :

1. Convert the augmented matrix into *Row Echelon form* :

   Known as forward elimination method. In this method the total number of flops is approximately proportional to n^3 (Refer **Example 8.3**) when n is very large. Since for each set of linear equations this method needs to be executed, the total number of flops for entire set is approximately proportional to $n \times n^3 = n^4$.

2. *Convert the Row Echelon form into Reduced Row Echelon form* :

   Known as backward substitution method. The total number of flops is approximately proportional to n^2 (Refer **Check Your Progress 8.4**) when n is very large. Since for each set of linear equations this method needs to be executed, the total number of flops for entire set is proportional to $n \times n^2 = n^3$.

Therefore, the total number of flops required to solve n set of system is approximately proportional to $n^4 + n^3$. When n becomes very large we can avoid the lower power of n and other proportionality constants and we can write total number of flops is = $O(n^4)$.

Therefore, we can conclude *LU* decomposition is superior than *Row Reduction* algorithm if we consider set of system of linear equations with same coefficient matrix. For scenario of a particular system of linear equation there is not much difference in the performance and both of the algorithm has number of flops = $O(n^3)$.

9.6 Summary

The two direct methods for solving systems of linear algebraic equations are presented in this chapter namely, The Matrix Method and LU Decomposition Method. Some general guidelines for selecting a method for solving systems of linear algebraic equations are given below :

- Row reduction and reduced row reduction forms are fundamental to all the direct methods those are considered here.
- Direct elimination methods are preferred for small systems (n ~< 50 to 100) and systems with few zeros (non-sparse systems). Gauss elimination or Gauss Jordan Elimination is the method of choice, as the Matrix Method needs more arithmetic operations than the elimination methods.
- LU factorization methods are the methods of choice when more than one b vector is considered with the same coefficient matrix.

9.7 References and Further Reading

1. Elementary Numerical Analysis – An algorithmic Approach, Third Edition, S.D. Conte, Carl de Boor, Tata McGraw-Hill, 2005
2. Linear Algebra and Its Application, David C. Lay, Pearson, 2007
3. Numerical Recipes in C, Second Edition, H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Tata McGraw Hill, 2003.
4. Numerical Methods for Engineers and Scientists, Second Editon, Joe D. Hoffman, CRC Press, 2001.

9.8 Hints and Solutions

Solution 9.1 :

$$\begin{bmatrix} 11 & 0 & 21 \\ -1 & 13 & -9 \end{bmatrix}$$

Solution 9.2 :

$$\begin{bmatrix} -27 & -17 \\ 5 & 1 \\ -53 & -58 \\ 15 & 36 \end{bmatrix}$$

Solution 9.3 :

The reduced echelon matrix is

```
-----------------------------------------
 1.00  0.00  0.00 -0.50  0.70  0.65
 0.00  1.00  0.00 -0.50  0.30  0.35
-0.00 -0.00  1.00 -0.50  0.10 -0.05
```

$$A^{-1} = \begin{bmatrix} -0.5 & 0.7 & 0.65 \\ -0.5 & 0.3 & 0.35 \\ -0.5 & 0.1 & -0.05 \end{bmatrix}$$

Solution 9.4 :

A is not an invertible matrix.

Solution 9.7 :

The x Matrix or final solution is

----

x[0]= 3.00, x[1]= 4.00, x[2]= -6.00, x[3]= -1.00

Solution 9.8 :

x[0]= -2.00, x[1]= -1.00, x[2]= 2.00, x[3]= -3.00

Solution 9.9 :

Algorithm 9.2

For each execution of outer loop, the inner loop will run for i times (j from 0 to i-1). The outer loop i runs from 1 to $n-1$

$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$ [ignoring the lower order term of n and other constants]

Algorithm 9.3

For each execution of outer loop, the inner loop will run for n-1-i times (j from n-1 to i+1). The outer loop i runs from n-2 to 0

$\sum_{i=n-2}^{0} (n-1-i) = \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$ [ignoring the lower order term of n and other constants]

Unit - 10 ❑ Application of C Programming : Solution of System of Linear Equations by Iterative Methods : Jacobi and Gauss-Siedel Method

Structure

10.0 Introduction
10.1 Objectives
10.2 Jacobi Method
10.3 The Gauss-Seidel Method
10.4 Summary
10.5 References and Further Reading
10.6 Hints and Solutions

10.0 Introduction

In last two units (**Unit 8 and 9**), different direct methods for solving systems of linear equations along with their C-implementations were discussed. All nonsingular systems of linear algebraic equations have a solution and theoritically the solution can always be obtained by some direct method mentioned earlier. However, the presence of roundoff errors while performing large number of arithmetic operations, is a major pitfall in the application of direct methods. Round-off errors are inherent as exact infinite precision numbers are approximated by finite precision numbers in computer and other digital calculators. The effects of round-off can be reduced by a procedure known as iterative methods, which is presented in this unit. All the iterative methods obtain the solution asymptotically by an iterative procedure in the following approach :

1. A trial solution is assumed.
2. The trial solution is substituted into the system of equations to determine the mismatch, or error, in the trial solution.
3. An improved solution is obtained from the mismatch data.

Examples of iterative methods are Jacobi iteration, Gauss-Seidel iteration. We will also discuss their respective C implantations in this unit.

10.1 Objectives

After going through this topic, the learner should be able to

- Describe the general structure of an iterative procedure for solving the system of linear algebraic equations.
- Understand the differences between direct methods and iterative methods.
- Solve a system of linear algebraic equations by Jacobi Method using C program.
- Solve a system of linear algebraic equations by Gauss-Seidel Method using C program.
- Understand the concept of Strictly Diagonally Dominant Matrix.

10.2 Jacobi Method

The first iterative technique is called the Jacobi method and this method makes two assumptions :

(1) that the given system has a unique solution and

(2) that the coefficient matrix A has no zeros on its main diagonal.

If any of the diagonal entries are zero, then rows or columns must be interchanged to obtain a coefficient matrix that has nonzero entries on the main diagonal.

Let us consider a system of linear equations to illustrate the principle of Jacobi Method.

$$\begin{array}{llllll} a_{11}x_1 & +a_{12}x_2 & +a_{13}x_3 & +\ldots & +a_{1n}x_n & =b_1 \\ a_{21}x_1 & +a_{22}x_2 & +a_{23}x_3 & +\ldots & +a_{2n}x_n & =b_2 \\ a_{31}x_1 & +a_{32}x_2 & +a_{33}x_3 & +\ldots & +a_{3n}x_n & =b_3 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{n1}x_1 & +a_{n2}x_2 & +a_{n3}x_3 & +\ldots & +a_{nn}x_n & =b_n \end{array} \tag{10.1}$$

To begin the Jacobi method, solve the first equation for x_1, the second equation for x_2 and so on, as follows :

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n}{a_{11}}$$

$$x_2 = \frac{b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n}{a_{22}}$$

$$\Rightarrow \quad x_3 = \frac{b_3 - a_{31}x_1 - a_{32}x_2 - \cdots - a_{3n}x_n}{a_{33}} \tag{10.2}$$

....

$$x_n = \frac{b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{(n-1)(n-1)}x_{n-1}}{a_{nn}}$$

Therefore, the general form of solution is

$$x_i = \frac{b_i - \sum_{j=1, i \neq j}^{n} a_{ij}x_j}{a_{ii}} \tag{10.3}$$

Then make an initial approximation of the solution (x_1, x_2, x_3, . . ., x_n) and substitute these values of x_i into the right-hand side of the rewritten equations to obtain the first approximation. In the same way, the second approximation is performed by substituting the first approximation's x-values into the right-hand side of the rewritten equations. By repeated iterations, a sequence of approximations will be formed that converges to the actual solution if certain criterion is satisfied.

Algorithm 10.1 : Solve $Ax = b$ where A is Coefficient matrix of the linear system and b is the right hand side of the linear system.

Input : A (n x n) is a square matrix with nonzero diagonal elements. b is a (n x 1) matrix.

Output : x is a (n x k) matrix where each element x_{ij} is the value of unknown variable x_i in j th iteration.

Steps	Description
1.	*k = 0*
2.	*for i=0, 1 , . . ., n-1 do:*
3.	*x[i][k] = 0*
4.	*Increment k*
5.	*for i=0, 1 , . . ., n-1 do:*
6.	*s = 0*
7.	*for j=0, 1 , . . ., n-1 do:*
8.	*if (i ≠ j) Do step 9.*
9.	*s = s + a[i][j]* x[j][k – 1]*
10.	*x[i][k] = b[i] – s) / a[i][i]*
11.	*go to step 4 if |x[i][k] – x[i][k – 1]| for at least one i > e (the desired accuracy)*
12.	*Print Matrix x*
13.	*Print k th column of matrix x as final solution and stop.*

Let us illustrate the **Algorithm 10.1** of Jacobi Method using the following example.

$$\begin{aligned} 5x_1 - 2x_2 + 3x_3 &= -1 \\ -3x_1 + 9x_2 + x_3 &= 2 \\ 2x_1 - x_2 - 7x_3 &= 3 \end{aligned} \qquad \textbf{(10.4)}$$

Start with initial solution $x_1 = 0$, $x_2 = 0$, $x_3 = 0$ (step 1 to step 3)

Calculate the next better approximation of the solution using following equations (step 4 to step 10).

$$\begin{aligned} x_1 &= -\frac{1}{5} + \frac{2}{5}x_2 - \frac{3}{5}x_3 \\ x_2 &= \frac{2}{9} + \frac{3}{9}x_1 - \frac{1}{9}x_3 \\ x_3 &= -\frac{3}{7} + \frac{2}{7}x_1 - \frac{1}{7}x_2 \end{aligned} \qquad \textbf{(10.5)}$$

So the first approximation is

$$\begin{aligned} x_1 &= -\frac{1}{5} + \frac{2}{5}(0) - \frac{3}{5}(0) = -0.200 \\ x_2 &= \frac{2}{9} + \frac{3}{9}(0) - \frac{1}{9}(0) = 0.222 \\ x_3 &= -\frac{3}{7} + \frac{2}{7}(0) - \frac{1}{7}(0) = -0.429 \end{aligned}$$

Continue the iterations until two successive approximations are identical when rounded to three significant digits. In this example, the absolute difference for $x_1 = (0.200 - 0) = 0.2$, for $x_2 = (0.222 - 0) = 0.222$ and for $x_3 = (0.429 - 0) = 0.429$ are all greater than .001(desired accuracy), so next approximate solution will be calculated. (step 11). In this way finally the following sequence of approximations, shown in **Table 10.1**, is obtained.

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4	itr=5	itr=6	itr=7
x[0]	0	-0.2	0.146	0.192	0.181	0.185	0.186	0.186
x[1]	0	0.222	0.203	0.328	0.332	0.329	0.331	0.331
x[2]	0	-0.429	-0.517	-0.416	-0.421	-0.424	-0.423	-0.423

Table 10.1

Therefore, the solution in iteration-6 and iteration - 7 are identical and the final solution is

x[0]= 0.186
x[1]= 0.331
x[2]=-0.423

The c implementation of the **Algorithm 10.1** is given in **Table 10.2**.

C Program for Jacobi Method

```
#include<stdio.h>
#include<math.h>
void main()
{
   /*a[i][j] is coefficient of x[j] in ith equation , b[i] is the rhs of ith equation
   x[i][j] is the value of ith unknown variable x[i] in jth iteration*/
   float a[20][20]={{5,-2,3},{-3,9,1},{2,-1,-7}},b[20]={-1,2,3},x[20][20],s,t, e
   =.001;
   int n,i,j,k=0,sol_flag;/*sol_flag =1 means solution reaches to the desired
   accuracy*/
   printf("Enter the number of equations: ");
   scanf("%d",&n);
   /* Display the coefficient matrix*/
```

```
        printf("\n\nThe given coefficient matrix is\n");
        printf("-----------------------------------------\n");
        for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
        {
            printf("%6.2f\t", a[i][j]);
        }
        printf("\n");
    }
/* Insert zeros as initial solution */
    for(i=0;i<n;i++)
        {
            x[i][k]=0;
        }
        /*Find out the approximate value of x[i] until it reaches the desired accuracy*/
        do
        {
            k++;
            /*Use the value of x[j] found in previous iteration to find new x[j]*/
            for(i=0;i<n;i++)
            {
                s=0;
                for(j=0;j<n;j++)
                {
                    if(i!=j)
                        s=s+a[i][j]*x[j][k-1];
                }
                x[i][k]=(b[i]-s)/a[i][i];
            }
        sol_flag=1;/*Initially assume that accuracy is reached*/
        for(i=0;i<n;i++)
        {
            t=fabs(x[i][k]-x[i][k-1]);
            /*Check the difference of x[i] in two consecutive iterations. If it is greater
            than the given accuracy then make sol_flg=0*/
```

```
        if(t>e)
        {
            sol_flag=0;
            break;
          }
        }
    }
    while( sol_flag==0 );
    printf("-------------------------------------------\n");
/* Display the sequence of solutions */
printf("\t");
    for(i=0;i<=k;i++)
        printf("itr=%d ",i);
    printf(" \n------------------------------------------------------------------------------\n");
    for (i=0; i<n; i++)
          {
            printf("x[%d] ", i);
            for(j=0;j<=k;j++)
            printf("%6.3f ", x[i][j]);
                printf("\n");
          }
    printf("\nThe x Matrix or final solution is\n");
    printf(" \n--------------------------------------------------\n");
    /* Display the x matrix */
    for(i=0;i<n;i++)
    printf("x[%d]=%6.3f\n",i,x[i][k]);
}
```

Table 10.2

Check Your Progress 10.1

Solve the following linear system by Jacobi Method rounding up the solution to three significant digits :

$$\begin{aligned} 10x_1 + 2x_2 - x_3 &= 7 \\ x_1 + 8x_2 + 3x_3 &= -4 \\ -2x_1 - x_2 + 10x_3 &= 9 \end{aligned}$$

10.3 The Gauss-Seidel Method

A modification of the Jacobi method is called the Gauss-Seidel method, which often requires fewer iterations to produce the same degree of accuracy. With the Jacobi method, the values of x_i obtained in the nth approximation remain unchanged until entire of the (n+1)th approximation has been calculated. With the Gauss Seidel method, on the other hand, the new value of each x_i is used as soon as it is known. That is, once we have determined x_1 from the first equation, its value is then used in the second equation to obtain the new x_2. Similarly, the new x_1 and x_2 are used in the third equation to obtain the new x_3 and so on. Therefore, to calculate the unknown variable x_i we use all x_j of current iteration when $i > j$ and all x_j of previous iteration when $i < j$. The **Algorithm 10.2** shows the detailed steps of Gauss-Seidel method.

Algorithm 10.2 : Solve $Ax = b$ where A is Coefficient matrix of the linear system and b is the right hand side of the linear system.

Input : A (n x n) is a square matrix with nonzero diagonal elements. b is a (n x 1) matrix.

Output : x is a (n x k) matrix where each element x_{ij} is the value of unknown variable x_i in the j th iteration.

Steps	*Description*
1.	*k = 0*
2.	*for i=0, 1 ,....., n-1 do:*
3.	*x[i][k] = 0*
4.	*Increment k*
5.	*for i=0, 1 ,....., n-1 do:*
6.	*s = 0*
7.	*for j=0, 1 , . . ., n-1 do:*
8.	*if (i > j) Do step 9*
9.	*s = s + a[i][j]*x[j][k]*
10.	*if (i < j) Do step 11*
11.	*s = s + a[i][j]* x[j][k – 1]*
12.	*x[i][k] = (b[i] – s) / a[i][i]*
13.	*go to step 4 if |x[i][k] – x[i][k – 1]| for at least one i >e(the desired accuracy)*
14.	*Print Matrix x*
15.	*Print k th column of matrix x as final solution and stop.*

Let us find the solution of the linear system mentioned in **(10.4)** using Gauss-Seidel method.

$$\begin{array}{rcrcrcr} 5x_1 & - & 2x_2 & + & 3x_3 & = & -1 \\ -3x_1 & + & 9x_2 & + & x_3 & = & 2 \\ 2x_1 & - & x_2 & - & 7x_3 & = & 3 \end{array}$$

The sequence of approximations found by **Algorithm 10.2** is shown in **Table 10.3**.

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4	itr=5
x[0]	0	-0.2	0.167	0.191	0.186	0.186
x[1]	0	0.156	0.334	0.333	0.331	0.331
x[2]	0	-0.508	-0.429	-0.422	-0.423	-0.423

Table 10.3

Therefore, the solution in iteration – 4 and iteration – 5 are identical and the final solution is

x[0] = 0.186
x[1] = 0.331
x[2] = -0.423

It can be noted that the Jacobi method also gave the same result but took more no of iterations. This shows faster convergence of Gauss-Seidel method over Jacobi method. But with the advent of parallel processor Jacobi method is becoming popular again. This is clearly because of the fact that all new components of the vector x in the Jacobi method are calculated from all the old component values of x within an iterative cycle. All component updates can therefore be carried out in parallel with synchronization being required only between iterative cycles. In contrast, the Gauss-Seidel method use new values within an iterative cycle in a systematic manner that demands a strictly sequential evaluation of the components. **Table 10.4** shows the C-implementation of Gauss-Seidel method.

C Program for Gauss-Seidel Method

```
#include<stdio.h>
#include<math.h>
void main()
{ /*a[i][j] is coefficient of x[j] in ith equation, b[i] is the rhs of ith equation
  x[i][j] is the value of ith unknown variable in jth iteration*/
```

```
    float a[20][20]={{5,-2,3},{-3,9,1},{2,-1,-7}},x[20][20], e =.001;
    float b[20]={-1,2,3},s,t;
    int n,i,j,k=0,sol_flag;/*sol_flag =1 means solution reaches to the desired accuracy*/
    printf("Enter the number of equations: ");
    scanf("%d",&n);

    /* Display the given matrix*/

    printf("\n\nThe given coefficient matrix is\n");
    printf("----------------------------------------\n");
    for (i=0; i<n; i++)
  {
        for (j=0; j<n; j++)
        {
            printf("%6.2f\t", a[i][j]);
        }
        printf("\n");
  }
 /* Insert zeros as initial solution */
    for(i=0;i<n;i++)
        {
            x[i][k]=0;
        }
        /*Find out the approximate value of x[i] until it reaches the desired accuracy*/
        do
        {
            k++;
            for(i=0;i<n;i++)
            {
                s=0;
                for(j=0;j<n;j++)
                {
                    if(i>j)
                        s=s+a[i][j]*x[j][k];
                    else if(i<j)
                            s=s+a[i][j]*x[j][k-1];
                }
```

```
            x[i][k]=(b[i]-s)/a[i][i];
         }
         sol_flag=1;
         for(i=0;i<n;i++)
         {
            t=fabs(x[i][k]-x[i][k-1]);
            if(t>e)
            {
               sol_flag=0;
               break;
            }
         }
      }
      while( sol_flag==0 );
      printf("----------------------------------------\n");
   /* Display the x matrix */
   printf("\t");
   for(i=0;i<=k;i++)
      printf("itr=%d ",i);
   printf(" \n----------------------------------------------------------------------\n");
   for (i=0; i<n; i++)
      {
         printf("x[%d] ", i);
         for(j=0;j<=k;j++)
         printf("%6.3f ", x[i][j]);
                  printf("\n");
      }
   printf("\nThe x Matrix or final solution is\n");
   printf(" \n----------------------------------------------------\n");
   /* Display the x matrix */
   for(i=0;i<n;i++)
   printf("x[%d]=%6.3f\n",i,x[i][k]);
}
```

Table 10.4

Check Your Progress 10.2

Solve the following linear system by Gauss-Seidel Method rounding up the solution to three significant digits.

$$
\begin{aligned}
10x_1 &+ 2x_2 - x_3 = 7 \\
x_1 &+ 8x_2 + 3x_3 = -4 \\
-2x_1 &- x_2 + 10x_3 = 9
\end{aligned}
$$

Neither of the iterative methods presented in this section always converges. That is, it is possible to apply the Jacobi method or the Gauss-Seidel method to a system of linear equations and obtain a divergent sequence of approximations. In such cases, it is said that the method diverges.

Example 10.1

Solve the following system of equations using (a) Jacobi Method and (b) Gauss-Seidel Method

$$
\begin{aligned}
x_1 - 5x_2 &= -4 \\
7x_1 - x_2 &= 6
\end{aligned}
$$

Solution

Part (a) :

Execute the program (Jacobi Algorithm) in **Table 10.2** on the matrix and consider only the first 6 iterations (**Table 10.5**).

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4	itr=5	itr=6
x[0]	0	-4	-34	-174	-1224	-6124	-42874
x[1]	0	-6	-34	-244	-1224	-8574	-42874

Table 10.5

For this particular system of linear equations, it can be determined that the actual solution is x[0] = 1 and x[1] = 1 and it is clear from **Table 20** that the approximations given by the Jacobi method become progressively worse instead of better, and the method diverges.

Part (b) :

Execute the program (Gauss Siedel Algorithm) in **Table 10.4** on the matrix and consider only the first 6 iterations (**Table 10.6**).

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4	itr=5	itr=6
x[0]	0	-4	-174	-6124	-214374	-7503124	-262609360
x[1]	0	-34	-1224	-42874	-1500624	-52521872	-1838265472

Table 10.6

The problem of divergence is not resolved by using the Gauss-Seidel method rather than the Jacobi method. In fact, for this particular system the Gauss-Seidel method diverges more rapidly, as shown in **Table 10.6**.

Strictly Diagonally Dominant Matrix

Let us now look at a special type of coefficient matrix A, called a strictly diagonally dominant matrix, for which it is guaranteed that both methods will converge.

A (n x n) matrix A is strictly diagonally dominant if the absolute value of each entry on the main diagonal is greater than the sum of the absolute values of the other entries in the same row. That is,

$$\begin{aligned} |a_{11}| &> |a_{12}| + |a_{13}| + \dots + |a_{1n}| \\ |a_{22}| &> |a_{21}| + |a_{23}| + \dots + |a_{2n}| \\ &\dots\dots \\ |a_{nn}| &> |a_{n1}| + |a_{n2}| + \dots + |a_{n(n-1)}| \end{aligned} \quad \textbf{(10.6)}$$

Consider the following linear system of **Example 10.1.** The given coefficient matrix is

$$\begin{bmatrix} 1 & -5 \\ 7 & -1 \end{bmatrix} \quad \textbf{(10.7)}$$

Clearly the above coefficient matrix is not strictly diagonally dominant.

Theorem 10.1

If A is strictly diagonally dominant, then the system of linear equations given by $A\boldsymbol{x} = \boldsymbol{b}$ has a unique solution to which the Jacobi method and the Gauss-Seidel method will converge for any initial approximation.

In **Example 10.1** the coefficient matrix A can be converted to strictly diagonally dominant matrix by interchanging the rows. After this interchange, convergence is assured. After interchanging row 1 and row 2 ($R_1 \leftrightarrow R_2$) of A in (10.7), the strictly diagonally dominant matrix is formed **(10.8)**.

$$\begin{bmatrix} 7 & -1 \\ 1 & -5 \end{bmatrix} \quad \textbf{(10.8)}$$

Now after running Jacobi method on the matrix in **(10.8)** the following sequence of solutions has been found **(Table 10.7)** :

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4	itr=5
x[0]	0	0.857	0.971	0.996	0.999	1
x[1]	0	0.8	0.971	0.994	0.999	1

Table 10.7

Again after running Gauss Seidel method on the matrix in **(10.8)** the following sequence of solutions has been found **(Table 10.8)** :

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4
x[0]	0	0.857	0.996	1	1
x[1]	0	0.971	0.999	1	1

Table 10.8

Now both the methods gives the final solution as x[0] = 1 and x[1] = 1.

It can be noted that strict diagonal dominance **(Theorem 10.1)** is a sufficient but not necessary condition for convergence of the Jacobi or Gauss-Seidel methods. For instance, the coefficient matrix of the system

$$\begin{array}{rcrcr} -4x_1 & + & 5x_2 & = & 1 \\ x_1 & + & 2x_2 & = & 3 \end{array}$$

is not a strictly diagonally dominant matrix, and yet both methods converge to the solution x[0] = 1 and x[1] = 1 with an initial approximation of x[0] = 0 and x[1] = 0.

Check Your Progress 10.3

Show that the Gauss-Seidel method diverges for the given system using the initial approximation $(x_1, x_2, x_3, \ldots, x_n) = (0,0,0, \ldots, 0)$:

$$\begin{array}{rcrcrcr} x_1 & + & 3x_2 & - & x_3 & = & 5 \\ 3x_1 & - & x_2 & & & = & 5 \\ & & x_2 & + & 2x_3 & = & 1 \end{array}$$

Check Your Progress 10.4

Show that the Gauss-Seidel method diverges for the given system using the initial approximation $(x_1, x_2, x_3, \ldots, x_n) = (0,0,0, \ldots, 0)$:

$$\begin{array}{rcrcrcr} 2x_1 & - & 3x_2 & & & = & -7 \\ x_1 & + & 3x_2 & - & 10x_3 & = & 9 \\ 3x_1 & & & + & x_3 & = & 13 \end{array}$$

Check Your Progress 10.5

Interchange the rows of the system of linear equations in **Check Your Progress 10.3** to obtain a system with a strictly diagonally dominant coefficient matrix. Then apply the Gauss-Seidel method to approximate the solution to two significant digits.

Check Your Progress 10.6

Interchange the rows of the system of linear equations in **Check Your Progress 10.4** to obtain a system with a strictly diagonally dominant coefficient matrix. Then apply the Gauss-Seidel method to approximate the solution to two significant digits.

Check Your Progress 10.7

Solve the system

$$\begin{aligned} 10x_1 - 2x_2 - x_3 - x_4 &= 3 \\ -2x_1 + 10x_2 - x_3 - x_4 &= 15 \\ -x_1 - x_2 + 10x_3 - 2x_4 &= 27 \\ -x_1 - x_2 + 2x_3 + 10x_4 &= -9 \end{aligned}$$

Using Jacobi, Gauss-Seidel method and compare the results.

10.4 Summary

The iterative methods for solving systems of linear algebraic equations are presented in this chapter. Some general guidelines for selecting an iterative method for solving systems of linear algebraic equations are given below :

- For large systems that are not diagonally dominant, the round-off errors can be large.
- Iterative methods are preferred for large, sparse matrices that are diagonally dominant.

10.5 References and Further Reading

1. Elementary Numerical Analysis – An algorithmic Approach, Third Edition, S.D. Conte, Carl de Boor, Tata McGraw-Hill, 2005
2. Linear Algebra and Its Application, David C. Lay, Pearson, 2007

3. Numerical Recipes in C, Second Edition, H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Tata McGraw Hill, 2003.

4. Numerical Methods for Engineers and Scientists, Second Editon, Joe D. Hoffman, CRC Press, 2001.

10.6 Hints and Solutions

Solution 10.1 :

x[0] = 1, x[1] = -1, x[2] = 1 and number of iterations = 8.

Solution 10.2 :

x[0] = 1, x[1] = -1, x[2] = 1 and number of iterations = 7.

Solution 10.3 :

Solution Diverges. The first five approximations are

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4	itr=5
x[0]	0	5	-29.5	332.75	-3470.875	36467.188
x[1]	0	10	-93.5	993.25	-10417.63	109396.563
x[2]	0	-4.5	47.25	-496.125	5209.313	-54697.781

Solution 10.5 :

Solution converges to x[0] = 2, x[1] = 1, x[2] = 0 in five iterations.

x[i]	itr=0	itr=1	itr=2	itr=3	itr=4	itr=5
x[0]	0	1.67	2.04	1.99	2	2
x[1]	0	1.11	0.97	1.01	1	1
x[2]	0	-0.06	0.02	0	0	0

Unit - 11 ❑ Application of C Programming : Interpolation

Structure

11.0 Introduction
11.1 Objectives
11.2 Interpolation
11.2.1 Direct Fit Polynomial
11.2.2 Lagrange Interpolation Method
11.2.3 Newton's Divided-Difference Method
11.2.4 Newton's Forward-Difference method
11.3 Summary
11.4 References and Further Reading

11.0 Introduction

In many problems in science and engineering, the data being considered are known only at a set of discrete points, not as a continuous function. For example, the continuous function

$y = f(x)$ may be known at n discrete values of x :
$y_i = f(x_i)$ $(i = 1, 2,.......n)$

In many applications, the values of the discrete data at the specific points are not all that is needed. Values of the function at points other than the known discrete points may be needed (i.e., interpolation). The derivative, integral of the function may be required. Thus, processes of interpolation, differentiation, and integration of a set of discrete data are of interest. These processes are illustrated in **Figure 11.1**.

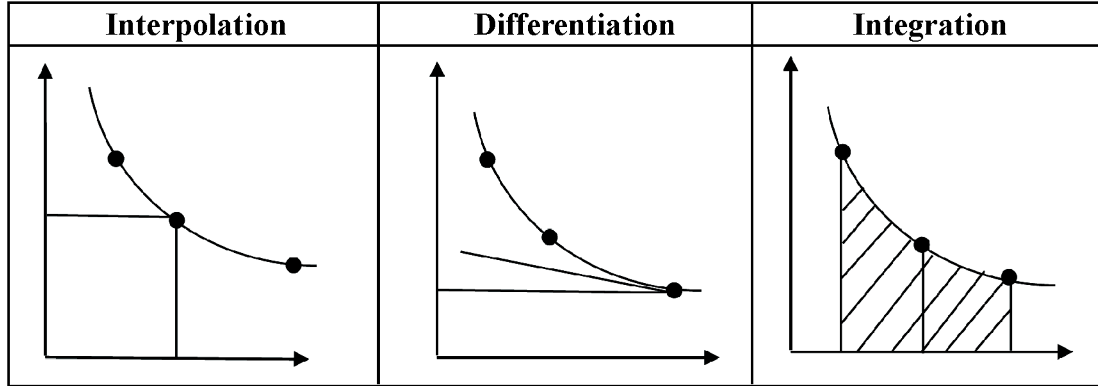


Figure 11.1

These processes are performed by fitting an approximating function to the set of discrete data and performing the desired process on the approximating function.

Many types of approximating functions exist. In fact, any type of analytical function can be used as an approximating function. Three of the more common approximating functions are :

- Polynomials
- Trigonometric functions
- Exponential functions

Approximating functions should have the following properties :

- The approximating function should be easy to determine
- It should be easy to evaluate.
- It should be easy to differentiate.
- It should be easy to integrate.

Polynomials satisfy all four of these properties. There are two fundamentally different ways to fit a polynomial to a set of discrete data :

- Exact fits
- Approximate fits

An exact fit yields a polynomial that passes exactly through all of the discrete points. This type of fit is useful for small sets of smooth data. Exact polynomial fit methods and corresponding C implementations are discussed in **Sections 11.3.1 to 11.3.4.** An approximate fit yields a polynomial that passes through the set of data in the best manner possible, without being required to pass exactly through any of the data points. We have not discussed approximate polynomial fits here as it is out of scope of the syllabus.

11.1 Objectives

After going through this topic, the learner should able to

- List the uses of functional approximation
- List the required properties of an approximating function
- Fit a direct fit polynomial of any degree to a set of tabular data
- Explain the concept underlying Lagrange polynomials
- Fit a Lagrange polynomial of any degree to a set of tabular data

- ➢ Write a C program for direct fit and Lagrange method
- ➢ Define a divided difference and construct a divided difference table
- ➢ Write a C program to display the divided difference table
- ➢ Write a C program for divided difference method
- ➢ Apply the Newton forward-difference polynomial
- ➢ Write a C program for Newton forward-difference method

11.2 Interpolation

Let us consider following data set of $(x, f(x)$ pairs given in **Table 11.1** generated

x	$f(x)$
3.35	0.298507
3.4	0.294118
3.5	0.285714
3.44	?

Table 11.1

by the function $f(x) = \frac{1}{x}$ which is unknown initially. From the given data the unknown value of $f(x)$ at $x = 3.44$ needs to be determined. To do that, first the approximate function that satisfy the first three points needs to be determined. A polynomial can be constructed which is suitable for approximating function $f(x)$ because of the following two theorems :

Theorem 11.1 : Weierstrass polynomial approximation theorem

If $f(x)$ is a continuous function in the closed interval $[a, b]$, then for every $\varepsilon > 0$ there exists a polynomial $P_n(x)$, where the value of n depends on the value of ε, such that for all $x \in [a, b]$,

$$|P_n(x) - f(x)| < \varepsilon$$

Consequently, any continuous function can be approximated to any accuracy by a polynomial of high enough degree.

Theorem 11.2 Uniqueness theorem of polynomials

Given $n + 1$ discrete data points $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$,.......,$(x_n, f(x_n))$, there exists a unique polynomial P_n of degree at most n such that

$$P_n(x_i) = f(x_i) \text{ for all } i \text{ such that } 0 \le i \le n$$

11.2.1 Direct Fit Polynomial

First let us consider a completely general procedure for fitting a polynomial to a set of data. Given $n + 1$ sets of data $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$,.......,$(x_n, f(x_n))$, let the unique n^{th} degree polynomial $P_n(x)$ that passes exactly through the $n + 1$ points be

$$P_n(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n \quad \textbf{(11.1)}$$

For simplicity of notation, let $f(x_i) = f_i$. Substituting each data point into **(11.1)** yields $n + 1$ equations :

$$\begin{aligned} a_0 + a_1x_0 + a_2x_0^2 + \dots a_nx_0^n &= f_0 \\ a_0 + a_1x_1 + a_2x_1^2 + \dots a_nx_1^n &= f_1 \\ &\text{-----------------} \\ a_0 + a_1x_n + a_2x_n^2 + \dots a_nx_n^n &= f_n \end{aligned} \quad \textbf{(11.2)}$$

There are $n + 1$ linear equations containing the $n + 1$ unknown coefficients a_0 to a_n. The Equations given in **(11.2)** can be solved for a_0 to a_n by the method described in **section 8.5.3** (Reduced Row Echelon Form or Gauss Jordon elimination). The resulting polynomial is the unique n^{th} - degree polynomial that passes exactly through the $n + 1$ data points. The coefficient matrix is known as Vandermonde Matrix (V) given in **(11.3)**.

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \cdots & & \cdots & \cdots & \cdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \quad \textbf{(11.3)}$$

It can be noticed that $\det(V) \neq 0$ (determinant of V) if x_i are distinct which leads to the uniqueness of the polynomial.

Let us now determine the polynomial that fit the data set given in **Table 11.1**. The polynomial in this case would be of degree 2 (quadratic) as the number of points considered is 3 (x_0 = 3.35, x_1 = 3.4, x_2 = 3.5). Let the interpolating polynomial be of the form $P_2(x) = a_0 + a_1x + a_2x^2$, the augmented matrix obtained by augmentaing the functional values as a column to the corresponding Vandermonde matrix is as follows :

$$\begin{bmatrix} 1 & 3.35 & (3.35)^2 & 0.298507 \\ 1 & 3.40 & (3.40)^2 & 0.294118 \\ 1 & 3.50 & (3.50)^2 & 0.285714 \end{bmatrix} \quad \textbf{(11.4)}$$

The program given in **Table 8.10** (Row reduced echelon form) is executed for the above data. The accuracy up to 5 decimal places is considered. The output is given in **Table 11.2**.

```
Enter the number of rows and columns : 3 4

The given matrix is
----------------------------
1.00    3.35    11.22    0.30
1.00    3.40    11.56    0.29
1.00    3.50    12.25    0.29

The echelon matrix is
----------------------------
1.00    3.35    11.22    0.30
0.00    0.05     0.34   -0.00
0.00    0.00     0.02    0.00

The reduced echelon matrix is
----------------------------
1.000000    0.000000    0.000000     0.876534
0.000000    1.000000    0.000000    -0.256064
0.000000    0.000000    1.000000     0.024931
```

Table 11.2

Table 11.2 shows the value of a_0 = 0.876534, a_1 = –0.256064, a_2 = 0.024931

Therefore, the interpolating polynomial for the data set of **Table 11.1** is $P_2(x) = 0.876534 - 0.256064x + 0.024931x^2$

Now the value of the polynomial at x = 3.44 is

P_2 (3.44) = 0.876534 – 0.256064 × (3.44) + 0.024931 × $(3.44)^2$ = 0.290697. Therefore, the absolute error due to interpolation is

$$|P_2(3.44) - f(3.44)| = \left|P_2(3.44) - \frac{1}{3.44}\right| = |0.290697 - 0.290698| = 0.000001. \quad \textbf{(11.5)}$$

If the polynomial is chosen of degree 1(linear) and the number of points considered is 2 ($x_0 = 3.35$, $x_1 = 3.5$) then by the same process the polynomial $P_1(x)$ can be determined. In that case,

$P_1(x) = 0.584217 - 0.085287x$

$P_1(3.44) = 0.584217 - 0.085287 \times (3.44) = 0.29083$

Therefore, the absolute error due to interpolation is

$$|P_1(3.44) - f(3.44)| = \left|P_1(3.44) - \frac{1}{3.44}\right| = |\,0.29083 - 0.290698\,| = 0.000132.$$

The error is larger than the error in equation **(11.5)** when the quadratic polynomial is used. The advantages of higher-degree interpolation are therefore obvious.

The main advantage of direct fit polynomials is that the explicit form of the approximating function is obtained, and interpolation at several values of x can be accomplished simply by evaluating the polynomial at each value of x. A second advantage is that the data can be unequally spaced.

The main disadvantage of direct fit polynomials is that each time the degree of the polynomial is changed, all of the work required to fit the new polynomial must be redone. The results obtained from fitting other degree polynomials is of no help in fitting the next polynomial. One approach for deciding when polynomial interpolation is accurate enough is to interpolate with successively higher-degree polynomials until the change in the result is within an acceptable range. This procedure is quite laborious using direct fit polynomials.

11.2.2 Lagrange Interpolation Method

The direct fit polynomial presented in **Section 11.3.1**, while quite straightforward in principle, has several disadvantages. It requires a considerable amount of effort to solve the system of equations for the coefficients. For a high-degree polynomial (n greater than about 4), the system of equations can be ill-conditioned, which causes large change in the solution vector (a_i) due to very small changes in the elements of the Vandermonde matrix. This happens when the linear system is sensitive to round-off errors. In that case, a simpler, more direct procedure is desired. One such procedure is the Lagrange polynomial, which can be fit to unequally spaced data or equally spaced data.

Consider the following data set of $(x, f(x))$ pairs given in **Table 11.3** :

x	$f(x)$
x_0	f_0
x_1	f_1
x_2	f_2
x_3	f_3
x_4	f_4

Table 11.3

The linear Lagrange polynomial $P_1(x)$ which passes through two points (x_0, f_0) and (x_1, f_1) is given by,

$$P_1(x) = \frac{(x-x_1)}{(x_0-x_1)}f_0 + \frac{(x-x_0)}{(x_1-x_0)}f_1 \qquad \textbf{(11.6)}$$

Equation **11.6** can be written as,

$$P_1(x) = l_0(x)f_0 + l_1(x)f_1 \qquad \textbf{(11.7)}$$

where $l_0(x) = \frac{(x-x_1)}{(x_0-x_1)}$ and $l_1(x) = \frac{(x-x_0)}{(x_1-x_0)}$, both being linear function known as cardinal function.

At $x = x_0$,

$P_1(x_0) = l_0(x_0)f_0 + l_1(x_0)f_1$

$$\Rightarrow P_1(x_0) = f_0 \quad [\because l_0(x_0) = \frac{(x_0-x_1)}{(x_0-x_1)} = 1 \text{ and } l_1(x_0) = \frac{(x_0-x_0)}{(x_1-x_0)} = 0] \qquad \textbf{(11.8)}$$

At $x = x_1$,

$P_1(x_1) = l_0(x_1)f_0 + l_1(x_1)f_1$

$$\Rightarrow P_1(x_1) = f_1 \quad [\because l_0(x_1) = \frac{(x_1-x_1)}{(x_0-x_1)} = 0 \text{ and } l_1(x_1) = \frac{(x_1-x_0)}{(x_1-x_0)} = 1] \qquad \textbf{(11.9)}$$

The equation **11.8** and **11.9** verifies that $P_1(x)$ passes through the points (x_0, f_0) and (x_1, f_1).

Similarly, Given three points, (x_0, f_0), (x_1, f_1) and (x_2, f_2), the quadratic Lagrange polynomial $P_2(x)$ which passes through the three points is given by,

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f_2 \quad \textbf{(11.10)}$$

Equation **11.10** can be written as,

$$P_2(x) = l_0(x) f_0 + l_1(x) f_1 + l_2(x) f_2 \quad \textbf{(11.11)}$$

where $l_0(x) = \dfrac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}, l_1(x) = \dfrac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}, l_2(x) = \dfrac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$

At $x = x_0$,

$$P_2(x_0) = l_0(x_0) f_0 + l_1(x_0) f_1 + l_2(x_0) f_2$$

$$\Rightarrow P_2(x_0) = f_0 \quad [\because l_0(x_0) = \frac{(x_0-x_1)(x_0-x_2)}{(x_0-x_1)(x_0-x_2)} = 1,\ l_1(x_0) = l_2(x_0) = 0] \quad \textbf{(11.11)}$$

At $x = x_1$,

$$P_2(x_1) = l_0(x_1) f_0 + l_1(x_1) f_1 + l_2(x_1) f_2$$

$$\Rightarrow P_2(x_1) = f_1 \quad [\because l_1(x_1) = \frac{(x_1-x_0)(x_1-x_2)}{(x_1-x_0)(x_1-x_2)} = 1,\ l_0(x_1) = l_2(x_1) = 0] \quad \textbf{(11.12)}$$

At $x = x_2$,

$$P_2(x_2) = l_0(x_2) f_0 + l_1(x_2) f_1 + l_2(x_2) f_2$$

$$\Rightarrow P_2(x_2) = f_2 \qquad [\because l_0(x_2) = l_1(x_2) = 0,\ l_2(x_2) = 1] \quad \textbf{(11.13)}$$

The equation **11.11**, **11.12** and **11.13** verifies that $P_2(x)$ passes through the points (x_0, f_0), (x_1, f_1) and (x_2, f_2).

Therefore, Lagrange polynomial passing through $(n + 1)$ data points (x_0, f_0), (x_1, f_1), . . ., (x_n, f_n) has the general form :

$$P_n(x) = l_0(x) f_0 + l_1(x) f_1 + \ldots\ldots\ldots\ldots + l_n(x) f_n$$

$$= \sum_{i=0}^{n} l_i(x) f_i \quad \text{where } l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \left(\frac{x - x_j}{x_i - x_j} \right) \quad \textbf{(11.14)}$$

with the property $l_i(x_j) = 1$, when $i = j$

$= 0$, when $i \neq j$

Algorithm 11.1 : Lagrange Interpolation Method

Input : Array $x[0: n - 1]$ and $y[0: n - 1]$ to store n points (pairs of (x_i, y_i)). Array $L[0: n - 1]$ to store the cardinal functions $(l_0, l_1, \ldots, l_{n-1})$. A value of $x = a$ (say) within x[0] to $x[n - 1]$ where value of the function $f(x)$ or y is unknown.

Output : The value of the function at $x = a$.

Steps	*Description*	
1.	*for i=0,1, . . ., n-1*	*/* Set the values of L[0: n – 1] to 1*/*
2.	| *L[i]=1*	
	for i=0,1, . . ., n-1	*/*Find the cardinal function*/*
3.	| *for j=0,1, . . ., n-1*	
4.	| | *if (i ≠ j)*	
5.	| | | *L[i]=L[i]*(a-x[j])/(x[i]-x[j])*	
6.	*b=0*	
7.	*for i=0,1, . . ., n-1*	*/*Find the value of the function at x = a */*
8.	*b=b + L[i]*y[i]*	
9.	*Print the value b and stop*	

The C program for Lagrange method is given in **Table 11.4**.

```
#include<stdio.h>
#include<math.h>
#define k 20
int main()
{
   float x[k],y[k],a,b=0,L[k];
   int i,j,n;
   printf("How many points: ");
   scanf("%d",&n);
   printf("Enter the value of a:");
   scanf("%f",&a);
   for(i=0;i<n;i++) /* Insert the (x,y) pairs*/
   {
      printf("\nEnter the x and y coordinates: ") ;
      scanf("%f%f",&x[i],&y[i]);
   }
```

```
for(i=0;i<n;i++) /* Set the values of L[0 : n–1] to 1*/
{
  L[i]=1;
}
for(i=0;i<n;i++) /*Find the cardinal function*/
{
  for(j=0;j<n;j++)
  {
    if(i!=j)
    L[i]= L[i]*(a-x[j])/(x[i]-x[j]);
  }
}
for(i=0;i<n;i++) /*Find the value of the function at x=a */
  b=b+L[i]*y[i];
printf("\nValue of cardinal Function\n");
printf("-------------------------------\n\n");
for(i=0;i<n;i++) /*Find the value of the cardinal functions*/
  printf("L[%d]=%f\t",i,L[i]);
printf("\n\nThe value of y is %f",b);
return 0;
}
```

Table 11.4

The output for the data given in **Table 11.1** is given in **Table 11.5**.

```
How many points : 3
Enter the value of a: 3.44
Enter the x and y coordinates : 3.35      0.298507
Enter the x and y coordinates : 3.40      0.294118
Enter the x and y coordinates : 3.50      0.285714
Value of cardinal Function
— — — — — — — — — — — — — — — — — — — —
L[0] =–0.319998    L[1]=1.079998     L[2]=0.240000
The value of y is 0.290697
```

Table 11.5

The absolute error due to interpolation is

$$|P_2(3.44) - f(3.44)| = \left|P_2(3.44) - \frac{1}{3.44}\right| = |0.290697 - 0.290698| = 0.000001.$$

Now let us add one more point in the data given in **Table 11.1**.

x	$f(x)$
3.35	0.298507
3.4	0.294118
3.5	0.285714
3.6	0.277778
3.44	?

Table 11.6

After executing the program given in **Table 11.4** on the data set of **Table 11.6,** following output is produced **(Table 11.7)** :

```
How many points : 4
Enter the value of a: 3.44
Enter the x and y coordinates : 3.35    0.298507
Enter the x and y coordinates : 3.40    0.294118
Enter the x and y coordinates : 3.50    0.285714
Enter the x and y coordinates : 3.60    0.277778
Value of cardinal Function _ _ _ _ _ _ _

L[0] =-0.204799   L[1]=0.863998   L[2]=0.384000   L[3]= -0.043200
The value of y is 0.290698
```

Table 11.7

The output in Table 11.7 shows one more cardinal functions and the value of the function is more accurate. The absolute error due to Lagrange interpolation is

$$|P_2(3.44) - f(3.44)| = \left|P_2(3.44) - \frac{1}{3.44}\right| = |0.290698 - 0.290698| = 0.000000.$$

The results are summarized below, where the results of linear, quadratic, and cubic interpolation, and the errors, *error* (3.44) = |P(3.44) – 0.290698|, are tabulated in **(Table 11.8)**

n	P_n (3.44)	Type of the polynomial	Error
1	0.290607	Linear	0.000091
2	0.290697	Quadratic	0.000001
3	0.290698	Cubic	0.000000

Table 11.8

The main advantage of the Lagrange polynomial is that the data may be unequally spaced. There are several disadvantages. All of the work must be redone for each degree polynomial. All the work must be redone for each value of x.

11.2.3 Newton's Divided-Difference Method

It can be observed in the last section that the Lagrange method is not flexible when additional data points are considered. All the cardinal function calculation depends on all the points and the unknown value of x. Therefore, all the cardinal functions calculated to find the interpolating polynomial using n points needs to be re-calculated if one more point has been added. To avoid this, Newton's divided difference method can be used to find the interpolating polynomial. The main idea of the method is as follows :

Suppose, the polynomial $P_n(x)$ interpolates $(n + 1)$ points. If one more point is added, the new interpolating polynomial $P_{n+1}(x)$ can be found using the polynomial $P_n(x)$ and one more additional term for the new point. The method is iterative in nature. To illustrate the method let us consider the (x_i, f_i) pairs given in **Table 11.3**.

Start with the iteration assuming, $n = 0$ and $P_0(x)$ passes through the point (x_0, f_0). Therefore,

$$P_0(x) = f_0 \tag{11.15}$$

From equation **11.15**, we get $P_0(x_0) = f_0$ which verifies that $P_0(x)$ passes through the point (x_0, f_0).

For the next iteration $n = 1$, $P_1(x)$ passes through the points (x_0, f_0) and (x_1, f_1) and can be written in the following form :

$$P_1(x) = P_0(x) + a_1(x - x_0) \tag{11.16}$$

Now using equation **11.16**, we can write

$$P_1(x_0) = P_0(x_0) + a_1(x_0 - x_0) = P_0(x_0) = f_0$$

Which verifies $P_1(x)$ passes through (x_0, f_0). Now, since $P_1(x)$ also passes through (x_1, f_1), we can write,

$$P_1(x_1) = f_1$$

$$\Rightarrow P_0(x_1) + a_1(x_1 - x_0) = f_1$$

$$\Rightarrow f_0 + a_1(x_1 - x_0) = f_1 \qquad [\because P_0(x) = f_0 \text{ from } \mathbf{11.15}]$$

$$\Rightarrow a_1 = \frac{(f_1 - f_0)}{(x_1 - x_0)}$$

a_1 is known as divided difference and it is defined as the ratio of the difference in the function values at two points $(f_1 - f_0)$ divided by the difference in the values of the corresponding independent variable $(x_1 - x_0)$. Thus, the first divided difference in the ith iteration is defined as

$$f[x_i, x_{i+1}] = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} = f_i^{(1)} \text{ where } i = 0, 1, 2, \ldots n - 1 \text{ and } a_1 = f[x_0, x_1]$$

Similarly for $n = 2$, $P_2(x)$ passes through the points (x_0, f_0), (x_1, f_1) and (x_2, f_2) which can be written in the following form :

$$P_2(x) = P_1(x) + a_2(x - x_0)(x - x_1) \qquad \textbf{(11.17)}$$

It can be verified that $P_2(x)$ satisfies the points (x_0, f_0), (x_1, f_1). Now, since $P_2(x)$ also passes through (x_2, f_2), we can write,

$$P_2(x_2) = f_2$$

$$\Rightarrow P_1(x_2) + a_2(x_2 - x_0)(x_2 - x_1) = f_2$$

$$\Rightarrow a_2 = \frac{(f_2 - P_1(x_2))}{(x_2 - x_0)(x_2 - x_1)} \qquad \textbf{(11.18)}$$

From equation **11.18**, using simple algebra, it can be shown that,

$$a_2 = \frac{\frac{f_2 - f_1}{x_2 - x_1} - \frac{f_1 - f_0}{x_1 - x_0}}{(x_2 - x_0)} = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = f[x_0, x_1, x_2]$$

In General, the second divided difference is defined as :

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i} = f_i^{(2)}$$

The divided difference table **(Table 11.9)** can be formed from **Table 11.3**.

Now the polynomial becomes,

$$P_n(x) = f_0^0 + (x - x_0)f_0^{(1)} + (x - x_0)(x - x_1)f_0^{(2)} + \ldots + (x - x_0)(x - x_1)\ldots(x - x_{n-1})f_0^{(n)}$$

Implementation of divided difference table in C :

To show the difference table in C, let us first introduce two integer variables i and j as row index and column index of the table respectively. A two dimensional array $M[n][n + 1]$ can be used to store the data of the difference table. Now the first two columns of array M will have the (x_i, f_i) pairs. The first divided difference will be stored in the cell $M[1][2]$ (See the **Table 11.9**). Also it can be observed from the **Table 11.9** that higher order divided differences, those are required to determine the

x	$f(x)$	$f_i^{(1)}$	$f_i^{(2)}$	$f_i^{(3)}$	$f_i^{(4)}$
x_0	f_0				
x_1	f_1	$\frac{f_1 - f_0}{x_1 - x_0} = f_0^{(1)}$			
x_2	f_2	$\frac{f_2 - f_1}{x_2 - x_1} = f_1^{(1)}$	$\frac{f_1^{(1)} - f_0^{(1)}}{x_2 - x_0} = f_0^{(2)}$		
x_3	f_3	$\frac{f_3 - f_2}{x_3 - x_2} = f_2^{(1)}$	$\frac{f_2^{(1)} - f_1^{(1)}}{x_3 - x_1} = f_1^{(2)}$	$\frac{f_1^{(2)} - f_0^{(2)}}{x_3 - x_0} = f_0^{(3)}$	
x_4	f_4	$\frac{f_4 - f_3}{x_4 - x_3} = f_3^{(1)}$	$\frac{f_3^{(1)} - f_2^{(1)}}{x_4 - x_2} = f_2^{(2)}$	$\frac{f_2^{(2)} - f_1^{(2)}}{x_4 - x_1} = f_1^{(3)}$	$\frac{f_1^{(3)} - f_0^{(3)}}{x_4 - x_0} = f_1^{(4)}$

Table 11.9

polynomial, can only be present in the cell $M[2][3], M[3][4], M[4][5], \ldots, M[n][n + 1]$. In all these cases $i = j - 1$ holds. Therefore, as a first step the column index j will be set at 2 and row index i at $j - 1$. Then all the divided differences can be determined by the following equation :

$$M_{ij} = \frac{M_{[i][j-1]} - M_{[i-1][j-1]}}{M_{[i][0]} - M_{[i-j+1][0]}} \qquad \textbf{(11.19)}$$

The numerator of the right hand side of equation **11.19** can easily be understood from the arrow (→) given in **Table 11.9**. In case of denominator, since the divided difference formula has only the x_i values, the entries of denominator will always be from column 0. After creating all the entries of the difference table the $(i+1)^{th}$ term of the interpolating polynomial will be generated using the following formula :

$$t_{i+1} = \left(\prod_{j=0}^{i-1}(x - x_j)\right) M[i][i+1] \quad \text{for } i = 1,2,\ldots\ldots,n \qquad \textbf{(11.20)}$$

The final formula for the interpolating polynomial is therefore

$$P_n(x) = f_0^0 + \sum_1^n t_{i+1} = M_{01} + \sum_1^n t_{i+1} \qquad \textbf{(11.21)}$$

Algorithm 11.2 : Divided difference Method

Input : Matrix $M[n][n+1]$ to store the difference table. The first two columns of matrix M store the (x_i, f_i) pairs. An unknown value of $x = a$ (say) within, x_0 to x_n where value of the function $f(x)$ at $x = a$ is unknown.

Output : The value of the function at the non-tabular point $x = a$.

Steps	*Description*
1.	*t=1,b*
2.	*for (j=2,3, . . ., n) /*Create the values of divided differences*/*
3.	*for(i=j-1,j, . . ., n-1)*
4.	*M[i][j]=(M[i][j-1]-M[i-1][j-1])/(M[i][0]-M[i-j+1][0])* */*eqn 11.19*/*
5.	*Print M.*
6.	*b=M[0][1]*
7.	*for(i=1,2,.....n)*
8.	*t=t*(a-M[i-1][0])*M[i][i+1]*
9.	*b=b+t*
10.	*Print b and stop.*

The C program for divided difference method is given in **Table 11.10**.

```
#include<stdio.h>
#include<math.h>
#define k 20
int main()
{
   float b=0,t,M[k][k],a;
   int i,j,n;
   printf("How many points: ");
   scanf("%d",&n);
   printf("Enter the value of a:");
   scanf("%f",&a);
   for(i=0;i<n;i++) /*Insert the (x,y) pairs*/
   {
      printf("\nEnter the x and y coordinates: ") ;
      scanf("%f%f",&M[i][0],&M[i][1]);
   }
   for(i=0;i<n;i++) /*Reset the difference column to 0*/
      for(j=2;j<=n;j++)
         M[i][j]=0;
   for(j=2;j<=n;j++) /*Determination of difference table*/
      for(i=j-1;i<n;i++)
         M[i][j]=(M[i][j-1]-M[i-1][j-1])/(M[i][0]-M[i-j+1][0]);
   printf("\nThe difference table is\n");
   printf("---------------\n");
   for(i=0;i<n;i++) /*Display the difference table*/
   {
      for(j=0;j<=n;j++)
         printf("%f\t",M[i][j]);
      printf("\n");
   }
   b=M[0][1];
   for(i=1;i<=n;i++) /*Determine the final value of y from polynomial*/
   {
```

```
        t=t*(a-M[i-1][0])*M[i][i+1];
        b=b+t;
    }
    printf("\n\nThe value of y is %f",b);
    return 0;
}
```

Table 11.10

The output for the data given in **Table 11.6** is given in **Table 11.11**.

```
How many points : 4
Enter the value of a: 3.44
Enter the x and y coordinates : 3.35    0.298507
Enter the x and y coordinates : 3.40    0.294118
Enter the x and y coordinates : 3.50    0.285714
Enter the x and y coordinates : 3.60    0.277778
The difference table is
— — — — — — — — — — — — — —
3.350000    0.298507     0.000000    0.000000     0.000000
3,400000    0.294118    –0.087780    0.000000     0.000000
3.500000    0.285714    –0.084040    0.024934     0.000000
3.600000    0.277778    –0.079360    0.023399    –0.006138
The value of y is 0.290599
```

Table 11.11

The absolute error due to interpolation is

$$|P_2(3.44) - f(3.44)| = \left|P_2(3.44) - \frac{1}{3.44}\right| = |0.290599 - 0.290698| = 0.000099.$$

It is quite obvious that if additional points are to be added in the data set, extra columns appear in the divided difference table but the existing entries of the difference table still remain useful which drastically reduce the computation.

Check Your Progress 11.1

Table 11.12 is for the function $f(x) = \frac{2}{x} + x^2$.

x	$f(x)$	x	$f(x)$	x	$f(x)$
0.4	5.1600	1.4	3.3886	2.4	6.5933
0.6	3.6933	1.6	3.8100	2.6	7.5292
0.8	3.1400	1.8	4.3511	2.8	8.5543
1.0	3.0000	2.0	5.0000		
1.2	3.1067	2.2	5.7491		

Table 11.12

Consider the data in the range $0.4 < x < 1.2$ in **Table 11.12**. Using Lagrange interpolation method calculate

a. P_2 (0.9) using the first three points,
b. P_2 (0.9) using the last three points,
c. P_3 (0.9) using the first four points,
d. P_3 (0.9) using last four points, and
e. P_4 (0.9) using all five data points.

Check Your Progress 11.2

Repeat Check Your Progress 11.1 in the range $0.4 < x < 1.2$ for $x = 1.5$.

Check Your Progress 11.3

The constant pressure specific heat C_p and enthalpy h of low pressure air are tabulated in **Table 11.13**.

T	$C_p(T)$	$h(T)$	T	$C_p(T)$	$h(T)$
1000	1.1410	1047.25	1400	1.1982	1515.79
1100	1.1573	1162.17	1500	1.2095	1636.19
1200	1.1722	1278.66	1600	1.2197	1757.66
1300	1.1858	1396.58			

Table 11.13

Using divided difference method, calculate

a. C_p (1180) using two points,
b. C_p (1120) using three points,
c. C_p (1480) using two points, and
d. C_p (1480) using three points.

Check Your Progress 11.4

Repeat Check Your Progress 11.3 for $h(T)$ instead of $C_p(T)$.

Check Your Progress 11.5

The data in **Table 11.12** can be fit by the expression $f(x) = \frac{a}{x} + bx^2$, Construct the divided difference Table and determine a and b.

11.2.4 The Newton Forward-Difference method

Constructing approximating polynomials to tabular data is considerably simpler when the values of the independent variable are equally spaced. Implementation of polynomial fitting for equally spaced data is best accomplished in terms of differences. Consequently, the concept of differences, difference tables, and difference polynomials are used to determine the interpolating polynomial.

A difference table is an arrangement of a set of data, (x_i, f_i), in a table with the x values in monotonic ascending order, with additional columns composed of the differences of the numbers in the preceding columns. To understand the difference table, let us consider the data for the function $f(x) = \frac{1}{x}$ where the x values are equally spaced in **Table 11.14**.

x	$f(x)$	$\Delta f(x)$	$\Delta^2 f(x)$	$\Delta^3 f(x)$	$\Delta^4 f(x)$
3.1	$0.322581 = f_0$				
3.2	$0.312500 = f_1$	$f_1 - f_0 = \Delta f_0$			
3.3	$0.303030 = f_2$	$f_2 - f_1 = \Delta f_1$	$\Delta f_1 - \Delta f_0 = \Delta^2 f_0$		
3.4	$0.294118 = f_3$	$f_3 - f_2 = \Delta f_2$	$\Delta f_2 - \Delta f_1 = \Delta^2 f_1$		
3.5	$0.285714 = f_4$	$f_4 - f_3 = \Delta f_3$	$\Delta f_3 - \Delta f_2 = \Delta^2 f_2$		
3.6	$0.277778 = f_5$	$f_5 - f_4 = \Delta f_4$	$\Delta f_4 - \Delta f_3 = \Delta^2 f_3$		

Table 11.14

Note that in **Table 11.14**, the forward difference operator Δ is defined as $\Delta f_i = f_{i+1} - f_i$

From equation **11.15**, we can write, the polynomial of degree 0 as $P_0(x) = f_0$.

From equation **11.16**, we can write, the polynomial of degree 1 as

$$P_1(x) = f_0 + a_1(x - x_0)$$

$$\Rightarrow P_1(x) = f_0 + \frac{(f_1 - f_0)}{(x_1 - x_0)}(x - x_0) \qquad \textbf{(11.22)}$$

Assuming $s = \frac{x - x_0}{h}$ and h = distance between two consecutive x_i.

Replacing the value of s, h and Δf_0 in equation **11.22**, we get

$$P_1(x) = f_0 + \frac{\Delta f_0}{h}(sh)$$

$$\Rightarrow P_1(x) = f_0 + s\Delta f_0 \qquad \textbf{(11.23)}$$

From equation **(11.17)**, we can write, the polynomial of degree 2 as

$$P_2(x) = P_1(x) + a_2(x - x_0)(x - x_1)$$

$$\text{where } a_2 = \frac{\frac{f_2 - f_1}{x_2 - x_1} - \frac{f_1 - f_0}{x_1 - x_0}}{(x_2 - x_0)} = \frac{\frac{\Delta f_1}{h} - \frac{\Delta f_0}{h}}{2h} = \frac{\Delta^2 f_0}{2h^2}$$

Therefore, $$P_2(x) = f_0 + s\Delta f_0 + \frac{\Delta^2 f_0}{2h^2}(sh)(sh - h)$$

$$\Rightarrow P_2(x) = f_0 + s\Delta f_0 + \frac{\Delta^2 f_0}{2}(s)(s - 1) \qquad \textbf{(11.24)}$$

After generalizing the result from equation **11.23** and **11.24**, we can write the following :

Given $n + 1$ data points, (x_i, f_i), one form of the unique n^{th} - degree polynomial that passes through the $n + 1$ points is given by,

$$P_n(x) = f_0 + s\Delta f_0 + \frac{s(s-1)}{2!}\Delta^2 f_0 + \frac{s(s-1)(s-2)}{3!}\Delta^3 f_0 + \ldots\ldots\ldots\ldots\ldots\ldots$$

$$+ \frac{s(s-1)(s-2)\ldots\ldots[s-(n-1)]}{n!}\Delta^n f_0 \qquad \textbf{(11.25)}$$

where s is the interpolating variable such that

$s = \frac{x - x_0}{h}$ and h = distance between two consecutive x_i.

Check Your Progress 11.6

Write a C program that can produce the forward difference table for the data given in **Table 11.14**. Modify the above program so that it can determine the value of $P_5(3.44)$ using Newton forward-difference formula given in equation **11.25**.

Check Your Progress 11.7

Solve the problems in **Check Your Progress 11.1** using Newton forward-difference method using the program for divided difference method written in the last problem.

11.3 Summary

Procedures for developing approximating polynomials for discrete data are presented in this unit. For small sets of smooth data, exact fits are desirable. The direct fit polynomial, the Lagrange polynomial, and the divided difference polynomial are explained with relevant C programs for non-equally spaced data. For equally spaced data, polynomials based on differences are recommended. The Newton forward-difference polynomial is simple to fit and evaluate. This polynomial is used extensively to develop procedures for numerical integration in the next unit (Unit 12).

11.4 References and Further Reading

1. Elementary Numerical Analysis – An algorithmic Approach, Third Edition, S.D. Conte, Carl de Boor, Tata McGraw-Hill, 2005.
2. Numerical Recipes in C, Second Edition, H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Tata McGraw Hill, 2003.
3. Numerical Methods for Engineers and Scientists, Second Editon, Joe D. Hoffman, CRC Press, 2001.

Unit - 12 ❑ Application of C Programming : Integration

Structure

12.0 Introduction

12.1 Objectives

12.2 Integration

12.3 Newton-Cotes Formulas

12.3.1 Rectangular Rule

12.3.2 Trapezoidal Rule

12.3.3 Simpson's 1/3 Rule

12.3.4 Simpson's 3/8 Rule

12.4 Summary

12.5 References and Further Reading

12.0 Introduction

In the last unit (**Unit 11**) interpolation within a set of discrete data points was discussed. In this unit integration of a set of tabular data will be presented, which is another important topic in numerical analysis that is used in mathematical and engineering applications. This unit will show different methods and their C implementations related to integration.

12.1 Objectives

After going through this topic, the learner should able to

- ➢ Describe the general features of numerical integration.
- ➢ Describe the procedure for numerical integration using Newton forward difference polynomials
- ➢ Write a C program for rectangular method to integrate a function.
- ➢ Write a C program for trapezoid rule.
- ➢ Write a C program for Simpson's 1/3 and 1/8 rule
- ➢ Describe the relative advantage and disadvantages of Simpson's 1/3 rule and Simpson's 3/8 rule

12.2 Integration

Evaluation of integrals is one of the most important problems in mathematics and other relevant disciplines. The problem of numerical integration, or numerical quadrature, is that of estimating the number

$$I(f) = \int_a^b f(x)dx \tag{12.1}$$

This problem arises when the integration cannot be carried out exactly or when $f(x)$ is known only at a finite number of points. In that case, numerical integration (quadrature) formulas can be developed by fitting approximating functions (polynomials) to discrete data and integrating the approximating function :

$$I(f) = \int_a^b f(x)dx \cong \int_a^b P_n(x)dx \tag{12.2}$$

The process is illustrated in **Figure 12.1**.

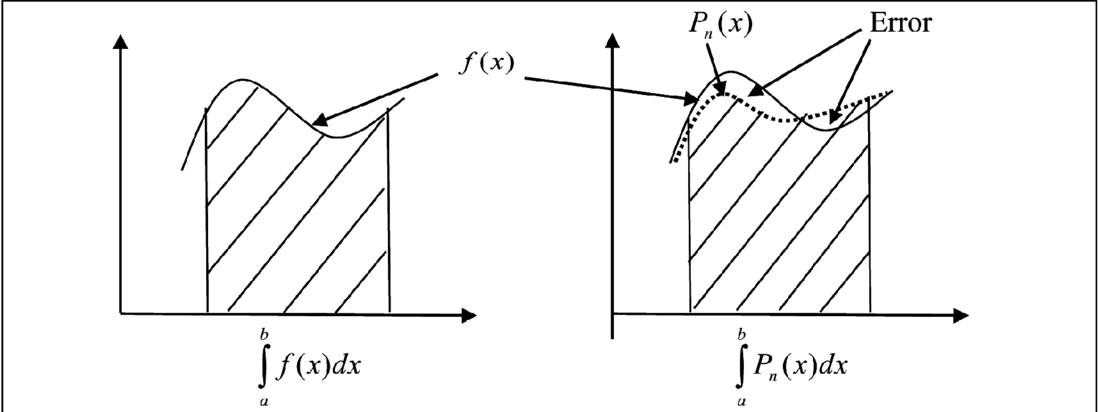


Figure 12.1

12.3 Newton-Cotes Formulas

Several methods have been discussed in **Section 11.3** to fit an interpolating polynomial for both unequally spaced data and equally spaced data. Any of these methods can be chosen to find the polynomial and integrate it for the given range. When the function to be integrated is known at equally spaced points, the Newton forward-difference polynomial presented in **Section 11.3.4** can be fit to the discrete data with much less effort. The resulting formulas are called Newton-Cotes formulas. Thus, rewriting the equation **(12.2)** using the interpolating polynomial, we get,

$$I(f) \cong \int_a^b P_n(x)dx \tag{12.3}$$

where $P_n(x)$ is the Newton forward-difference polynomial given in equation **11.25 (Unit 11)**.

$$P_n(x) = f_0 + s\Delta f_0 + \frac{s(s-1)}{2!}\Delta^2 f_0 + \frac{s(s-1)(s-2)}{3!}\Delta^3 f_0 + \cdots\cdots\cdots$$

$$+\frac{s(s-1)(s-2)\cdots\cdots[s-(n-1)]}{n!}\Delta^n f_0 \tag{12.4}$$

where s is the interpolating variable such that

$$s = \frac{x - x_0}{h} \tag{12.5}$$

and h = distance between two consecutive x_i.

Equation **(12.3)** requires that the approximating polynomial be an explicit function of x, whereas equation **(12.4)** is implicit in x. Let us transform the equation 12.3 so that equation **(12.4)** can be used directly. Thus from equation **(12.5)**, we get,

$$x = x_0 + hs$$
$$dx = hds$$

Using the above result, we can write equation **(12.3)** as

$$I(f) \cong \int_a^b P_n(x)dx = h\int_{s(a)}^{s(b)} P_n(x_0 + hs)ds \tag{12.6}$$

The limits of integration, $x = a$ and $x = b$, are expressed in terms of the interpolating parameter s by choosing $x = a$ as the base point of the polynomial, so that $x = a$ corresponds to $s = \frac{a-a}{h} = 0$ and $x = b$ corresponds to $s = \frac{b-a}{h}$. Introducing these results into eqaution **(12.6)** yields

$$I(f) \cong h\int_0^s P_n(x_0 + hs)ds \tag{12.7}$$

Each choice of the degree n of the interpolating polynomial P_n yields a different Newton-Cotes formula. **Table 12.1** lists the more common formulas.

n	Formula
0	Rectangular rule
1	Trapezoidal rule
2	Simpson's $\frac{1}{3}$ rule
3	Simpson's $\frac{3}{8}$ rule

Table 12.1

Now let us define few more terminology before using Newton-Cotes formula. The distance between the lower and upper limits of integration is called the range of integration. The distance between any two data points is called an increment **(Figure 12.2)**. A constant (0-degree polynomial) requires a single data point to obtain a fit. A linear polynomial requires one increment and two data points to obtain a fit. A quadratic polynomial requires two increments and three data points to obtain a fit. And so on for higher-degree polynomials. The group of increments required to fit a polynomial is called an interval. A linear polynomial requires an interval consisting of only one increment. A quadratic polynomial requires an interval containing two increments. And so on. The total range of integration can consist of one or more intervals. Each interval consists of one or more increments, depending on the degree of the approximating polynomial. Therefore, the upper limit of integration (s) in equation **(12.7)** is the total no of increments in between $x = a$ and $x = b$.

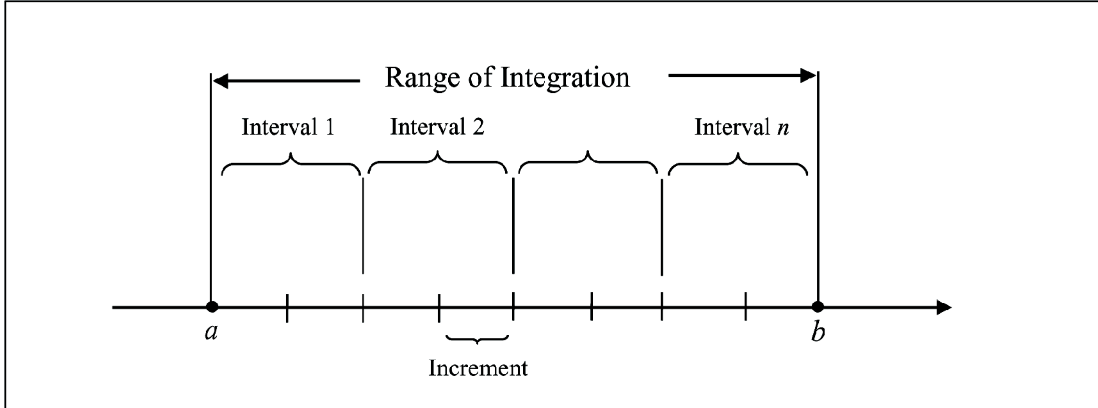


Figure 12.2

12.3.1 Rectangular Rule

In this rule, the value of n in the interpolating polynomial P_n is considered to be 0. Therefore, it becomes 0 degree polynomial which is a straight line parallel to x axis (**Figure 12.3a**). If n = 0, then from equation **(12.4)** we can write

$$P_0(x) = f_i \text{ (constant)}.$$

Now if the value of $\int_{x_i}^{x_{i+1}} P_0(x)dx$ is assumed to be ΔI_i then using equation **(12.7)** we can write,

$$\Delta I_i = \int_{x_i}^{x_{i+1}} P_0(x)dx = h\int_0^s f_i dx \text{ where } h = x_{i+1} - x_i \quad \textbf{(12.8)}$$

Now the upperlimit of the integration, $s = \frac{x_{i+1} - x_i}{h} = \frac{h}{h} = 1$. So the equation **(12.8)** becomes,

$$\Delta I_i = h\int_0^1 f_i ds = hf_i \quad \textbf{(12.9)}$$

It is also evident from the geometry of the **Figure 12.3a** that the quantity hf_i is the area of the rectangle. Here $h = x_{i+1} - x_i$ is the breadth and f_i is the length of the rectangle. The composite rectangular rule is obtained by applying equation **(12.9)** over all the intervals of interest (**Figure 12.3b**). Thus,

$$I = \sum_{i=0}^{n-1} \Delta I_i = h(f_0 + f_1 + f_2 + \cdots\cdots + f_{n-1}) \quad \textbf{(12.10)}$$

Note that, since the left most ordinates (f_0) is considered as length of the first rectangle and total no of rectangles is n, therefore the index starts from 0 and ends at $n - 1$.

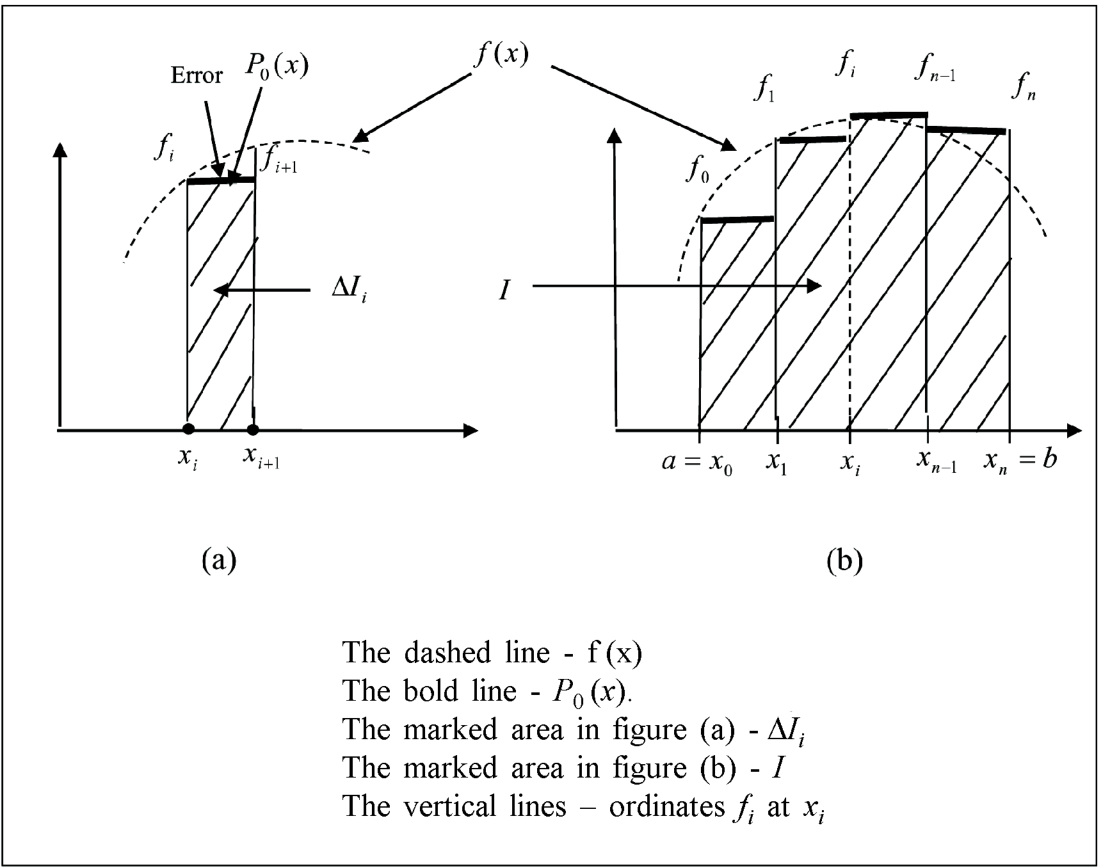


Figure 12.3

C program implementation of composite rectangular rule

To store the values of f_i an array of appropriate length needs to be created first. If the given data set has $n + 1$ points then total no of interval is n. The value of h can be calculated using following formula :

$$h = \frac{b - a}{n} \tag{12.11}$$

The C program for the rectangular method is given in the **Table 12.2**.

```
/*Program for rectangular Method*/
#include<stdio.h>
#include<math.h>
#define f(x) 1/(x)
```

```
#define k 80
int main()
{
   float a,b,h,y[k],s=0,val;
   int i,m,n;
   printf("How many points: ");
   scanf("%d",&n);
   printf("Enter the lower and upper limit of integration: ");
   scanf("%f%f",&a,&b);
   m=n-1; /*Number of intervals*/
   h=(b-a)/m;
   printf("\nh=%.2f and Number of Intervals=%d\n\n",h,m);
   for(i=0;i<n;i++) /*Insert f[i]*/
   {
      y[i]=f(a+i*h);
      printf("f[%.2f]=%f\n",a+i*h,y[i]);
   }
   for(i=0;i<m;i++)
      s=s+y[i];
   val=h*s;
   printf("\n\nThe value of the integration is %f",val);
   return 0;
}
```

Table 12.2

Consider the data set, generated from the function $f(x) = \frac{1}{x}$ where the values of x_i's are equally spaced, in **Table 12.3**.

x_i	f_i	x_i	f_i
3.1	0.322581	3.5	0.285714
3.2	0.312500	3.6	0.277778
3.3	0.303030	3.7	0.270270
3.4	0.294118		

Table 12.3

A parametrized macro for $f(x)=\frac{1}{x}$ is defined in the program in **Table 12.2**. The output of the rectangular method is given in **Table 12.4**.

```
How many points : 7
Enter the lower and upper limit of integration : 3.1  3.7
h = 0.10 and Number of Intervals = 6
f[3.10] = 0.322581
f[3.20] = 0.312500
f[3.30] = 0.303030
f[3.40] = 0.294118
f[3.50] = 0.285714
f[3.60] = 0.277778
f[3.70] = 0.270270

The value of the integration is 0.179572
```

Table 12.4

The approximate value of the integration is found to be 0.179572. Let us check the true value of integration analytically.

$$\int_{3.1}^{3.7} \frac{1}{x}dx = [\ln(x)]_{3.1}^{3.7} = \ln(3.7) - \ln(3.1) = 0.176930$$

The absolute error of integration is |0.179572 – 0.176930| = 0.002642. Let us execute the same program after reducing value of h half of its previous value. Therefore, the new value of h = 0.05. Assuming both the limit of integration remains same, the required value of n (number of intervals) can be found from the equation **(12.11)** as follows :

$$h=\frac{b-a}{n}$$

$$\Rightarrow n=\frac{b-a}{h}=\frac{0.6}{0.05}=12$$

Therefore, the no of points needed for h = 0.05 is $n + 1 = 13$. The output of the program in **Table 12.2** after reducing the value of h is given in **Table 12.5**.

```
How many points : 13
Enter the lower and upper limit of integration : 3.1  3.7
h = 0.05 and number of Intervals = 12
f`[3.10] = 0.322581
f`[3.15] = 0.317460
f`[3.20] = 0.312500
f`[3.25] = 0.307692
f`[3.30] = 0.303030
f`[3.35] = 0.298507
f`[3.40] = 0.294118
f`[3.45] = 0.289855
f`[3.50] = 0.285714
f`[3.55] = 0.281690
f`[3.60] = 0.277778
f`[3.65] = 0.273973
f`[3.70] = 0.270270

The value of the integration is 0.178245
```

Table 12.5

The approximate value of the integration is found to be 0.178245 in this case.

The absolute error of integration is $|0.178245 - 0.176930| = 0.001315$. This result shows that decreasing the value of h will increase the accuracy of the integration process by reducing the error. **Figure 12.4** shows how decreasing the value of h (adding more intervals) reduces the integration error.

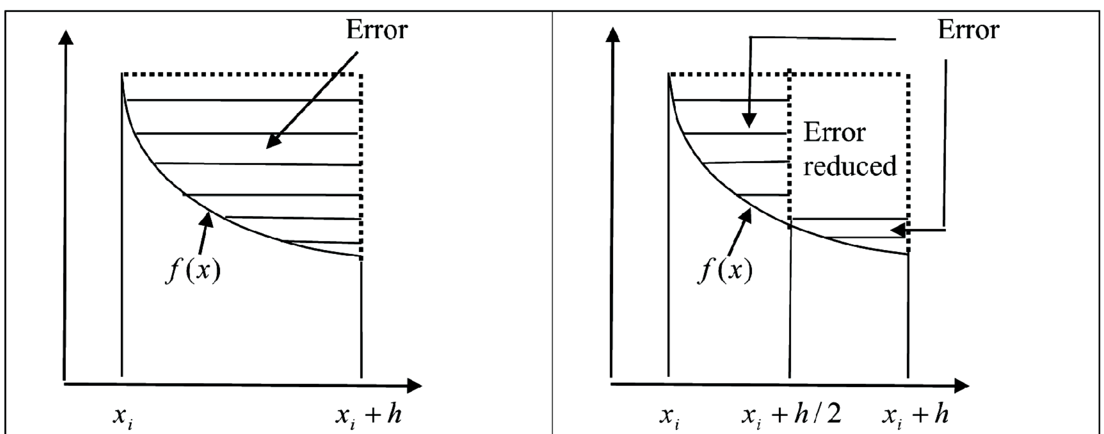


Figure 12.4

Table 12.6 shows the values of absolute error for different choice of h for integration by rectangular rule.

Value of h	Interval number	Number of points	Absolute error
0.20	3	4	0.005335
0.10	6	7	0.002642
0.05	12	13	0.001315
0.025	24	25	0.000656
0.01	70	71	0.000225

Table 12.6

The relationship between h and the absolute error is given in **Figure 12.5**. The graph shows that the relationship is approximately linear. Therefore, the upper bound of absolute error is not more than a constant multiple of h while using composite rectangular method. We can write absolute error as $O(h)$.

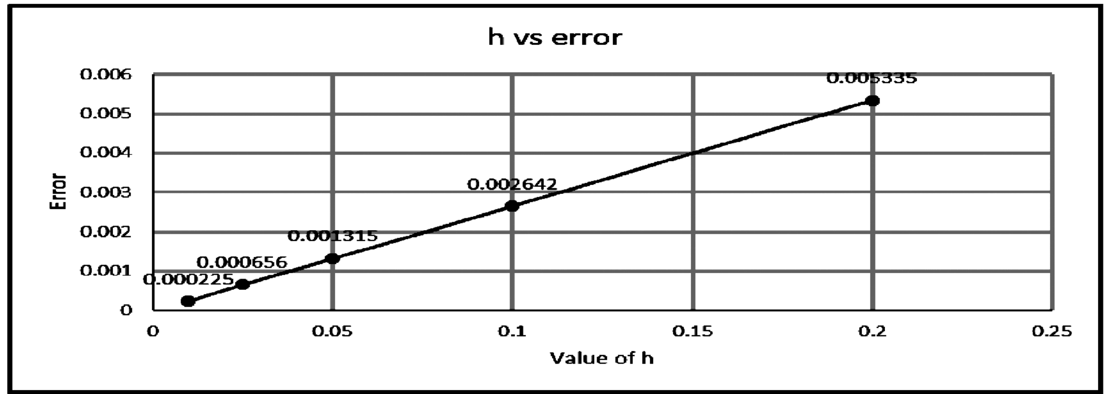


Figure 12.5

12.3.2 Trapezoidal Rule

The trapezoid rule for a single interval is obtained by fitting a first-degree polynomial $P_1(x)$ to two discrete points $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$ as illustrated in **Figure 12.6**. If $n = 1$, then from equation **(12.4)** we can write,

$$P_1(x) = f_i + s\Delta f_i \quad \text{where} = \Delta f = f_{i+1} - f_i \qquad \textbf{(12.12)}$$

Now if the value of $\int_{x_i}^{x_{i+1}} P_1(x)dx$ is assumed to be ΔI_i then using equation **(12.7)** we can write,

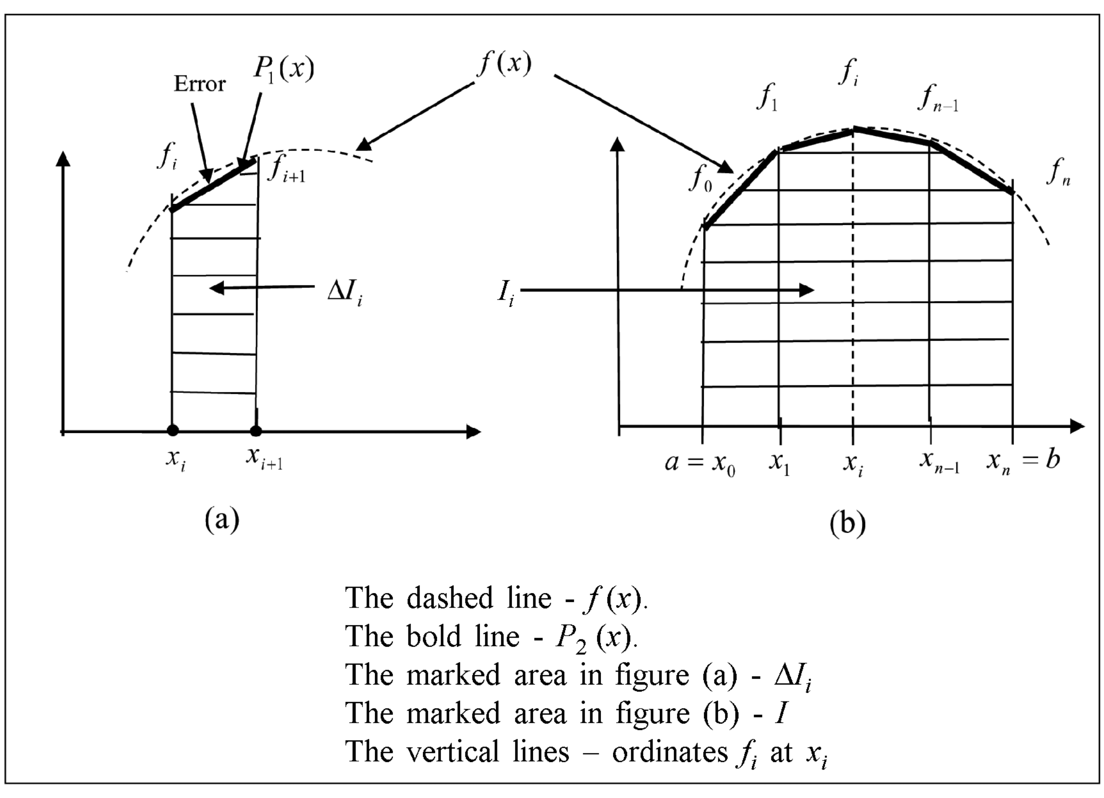
$$\Delta I_i = \int_{x_i}^{x_{i+1}} P_1(x)dx = h\int_0^s (f_i + s\Delta f_i)ds \quad \text{where } h = x_{i+1} - x_i \tag{12.13}$$

Now the upperlimit of the integration, $s = \frac{x_{i+1} - x_i}{h} = \frac{h}{h} = 1$. So the equation **(12.13)** becomes,

$$\Delta I_i = h\int_0^1 (f_i + s\Delta f_i)ds = h\left[sf_i + \frac{s^2}{2}\Delta f_i\right]_0^1 = h\left(f_i + \frac{1}{2}\Delta f_i\right)$$

$$= \frac{h}{2}(2f_i + f_{i+1} - f_i)$$

$$= \frac{h}{2}(f_i + f_{i+1}) \tag{12.14}$$



The dashed line - $f(x)$.
The bold line - $P_2(x)$.
The marked area in figure (a) - ΔI_i
The marked area in figure (b) - I
The vertical lines – ordinates f_i at x_i

Figure 12.6

Thus,

$$I = \sum_{i=0}^{n-1} \Delta I_i = \frac{h}{2}(f_0 + f_1) + \frac{h}{2}(f_1 + f_2) + \cdots + \frac{h}{2}(f_{n-1} + f_n)$$

$$= \frac{h}{2}[f_0 + 2(f_1 + f_2 + \cdots + f_{n-1}) + f_n] \quad \textbf{(12.15)}$$

The C program for the composite trapezoidal method is given in the **Table 12.7**.

```
/*Program for trapezoidal Method*/
#include<stdio.h>
#include<math.h>
#define f(x) 1/(x)
#define k 80
int main()
{
   float a,b,h,y[k],s=0,val;
   int i,m,n;
   printf("How many points: ");
   scanf("%d",&n);
   printf("Enter the lower and upper limit of integration: ");
   scanf("%f%f",&a,&b);
   m=n-1; /*number of intervals*/
   h=(b-a)/m;
   printf("\nh=%.2f and number of Intervals=%d\n\n",h,m);
   for(i=0;i<n;i++) /*Insert f[i]*/
   {
      y[i]=f(a+i*h);
      printf("f[%.2f]=%f\n",a+i*h,y[i]);
   }
   for(i=1;i<n-1;i++)
      s=s+y[i];
   val=(h/2)*(y[0]+2*s+y[n-1]);
   printf("\n\nThe value of the integration is %f",val);
   return 0;
}
```

Table 12.7

Consider the same data set generated from the function $f(x)=\frac{1}{x}$ in **Table 12.3**. The output of the composite trapezoidal rule after executing the C program on the given data set is shown in **Table 12.8**.

```
How many points : 7
Enter the lower and upper limit of integration : 3.1  3.7
h = 0.10 and number of Intervals = 6
f`[3.10] = 0.322581
f`[3.20] = 0.312500
f`[3.30] = 0.303030
f`[3.40] = 0.294118
f`[3.50] = 0.285714
f`[3.60] = 0.277778
f`[3.70] = 0.270270

The value of the integration is 0.176957
```

Table 12.8

The approximate value of the integration is found to be 0.176957. The actual value of integration is already found as 0.176930. Therefore, the absolute error of integration is $|0.176957 - 0.176930| = 0.000027$. It can be observed that the error is significantly lesser in trapezoidal rule than rectangular rule. This result can be easily explained from the graph of the function $f(x)=\frac{1}{x}$ in **Figure 12.7**. The graph is nearly linear in the range $3.7 \geq x \geq 3.1$. As a result, a linear polynomial(Trapezoidal) better approximate the function than a constant (Rectangular).

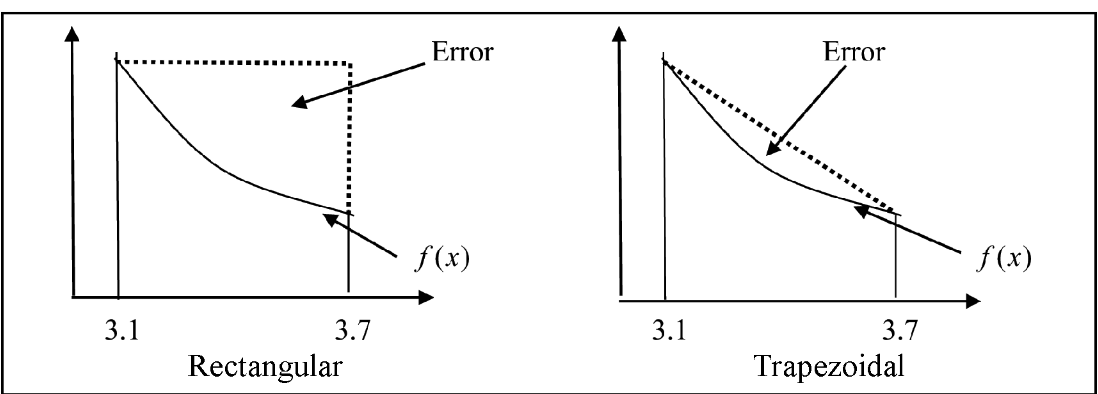


Figure 12.7

Now absolute error of this method also depends on the value of h. **Table 12.9** shows values of absolute error for different choice of h.

Value of *h*	**Interval number**	**Number of points**	**Absolute error**
0.20	3	4	0.000104
0.15	4	5	0.000059
0.10	6	7	0.000027
0.05	12	13	0.000007
0.025	24	25	0.000002

Table 12.9

The relationship between h and the absolute error is given in **Figure 12.8**. The graph clearly shows that the relationship is quadratic. Therefore, the upper bound of absolute error is not more than a constant multiple of h^2 while using composite trapezoidal method. Mathematically we can write absolute error is $O(h^2)$.

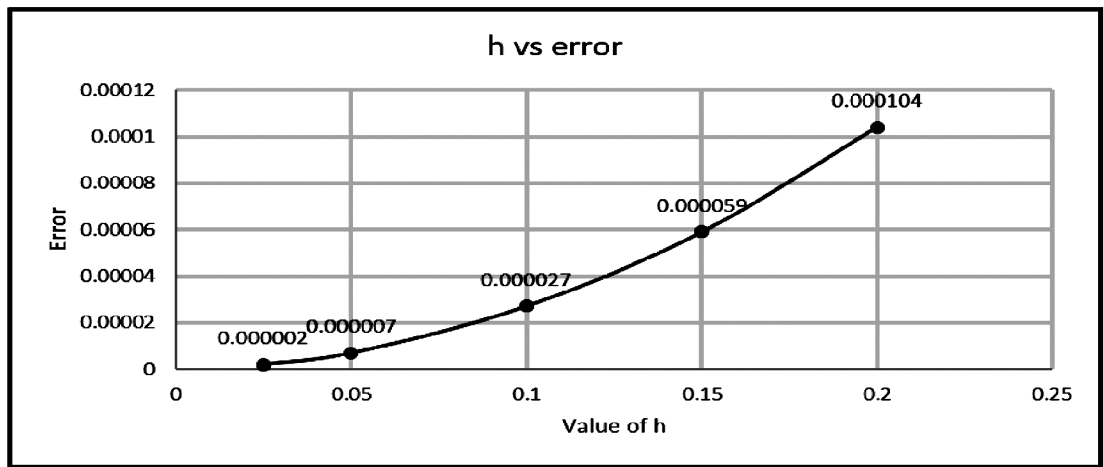


Figure 12.8

12.3.3 Simpson's 1/3 Rule

Simpson's 1/3 rule is obtained by fitting a second-degree polynomial to three equally spaced discrete points, as illustrated in **Figure 12.9a**. The number of intervals is 2. Therefore,

$$\Delta I_i = h\int_0^2 (f_i + s\Delta f_i + \frac{s(s-1)}{2}\Delta^2 f_i)ds = h\left[sf_i + \frac{s^2}{2}\Delta f_i + \frac{s^3}{6}\Delta^2 f_i - \frac{s^2}{4}\Delta^2 f_i\right]_0^2$$

$$= h\left(2f_i + 2\Delta f_i + \frac{1}{3}\Delta^2 f_i\right)$$

$$= h\left[2f_i + 2(f_{i+1} - f_i) + \frac{1}{3}(f_{i+2} - 2f_{i+1} + f_i)\right]$$

$$= h\left(\frac{4}{3}f_{i+1} + \frac{1}{3}f_{i+2} + \frac{1}{3}f_i\right)$$

$$= \frac{h}{3}\left(f_i + 4f_{i+1} + f_{i+2}\right) \qquad \textbf{(12.16)}$$

The composite Simpson's 1/3 rule is obtained by applying equation **(12.16)** over all the intervals of interest **(Figure 12.9b)**. Note that the total number of intervals must be a multiple of two.

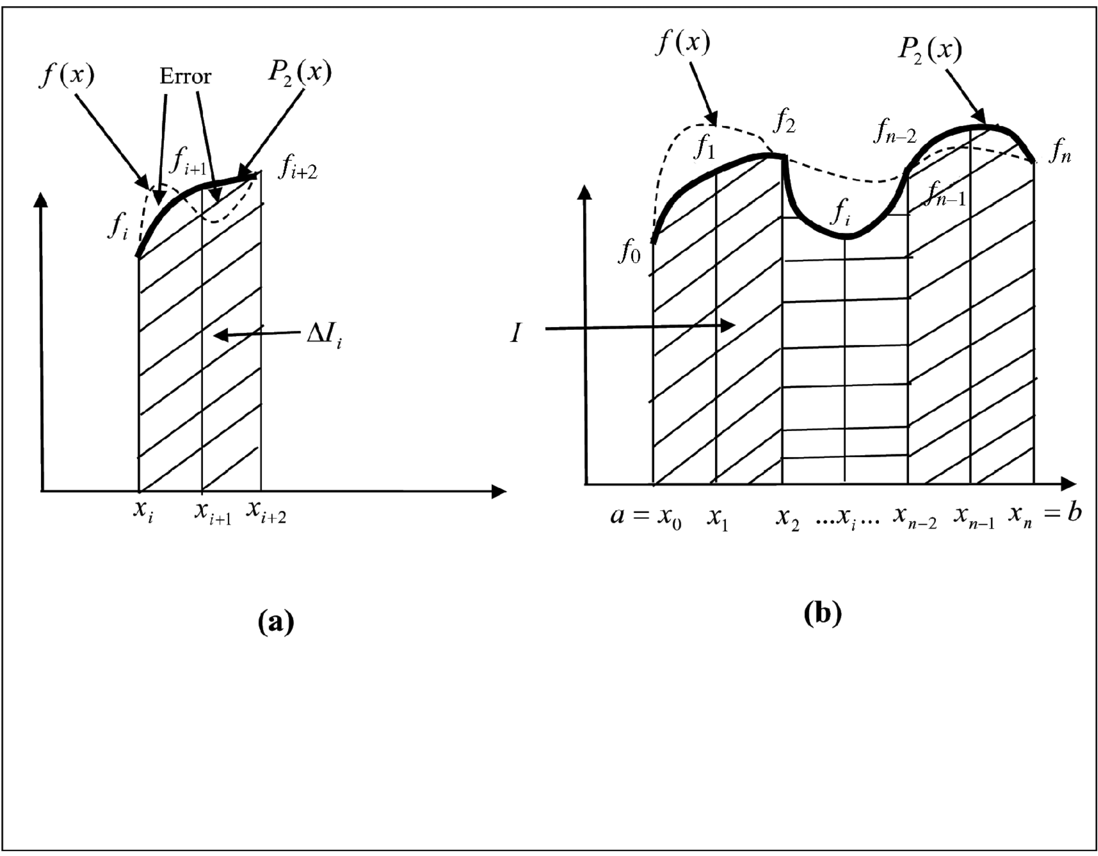


Figure 12.9

Thus,

$$I = \frac{h}{3}(f_0 + 4f_1 + f_2) + \frac{h}{3}(f_2 + 4f_3 + f_4) + \cdots + \frac{h}{3}(f_{n-2} + f_{n-1} + f_n)$$

$$= \frac{h}{3}\left[f_0 + 4(f_1 + f_3 + \cdots f_{n-1}) + 2(f_2 + f_4 + \cdots + f_{n-2}) + f_n\right] \quad \textbf{(12.17)}$$

$$= \frac{h}{3}\left[f_0 + 4f_{odd} + 2f_{even} + f_n\right]$$

The C program for the composite Simpson's 1/3 rule is given in the **Table 12.10**.

```
/*Program for Simpson's 1/3 Method*/
#include<stdio.h>
#include<math.h>
#define f(x) 1/(x)
#define k 80
int main()
{
   float a,b,h,y[k],s=0,val;
   int i,m,n;
   printf("How many points: ");
   scanf("%d",&n);
   printf("Enter the lower and upper limit of integration: ");
   scanf("%f%f",&a,&b);
   m=n-1; /*Number of intervals*/
   h=(b-a)/m;
   printf("\nh=%.2f and number of Intervals=%d\n\n",h,m);
   for(i=0;i<n;i++) /*Insert f[i]*/
   {
      y[i]=f(a+i*h);
      printf("f[%.2f]=%f\n",a+i*h,y[i]);
   }
   for(i=1;i<n-1;i++)
   {
      if (i%2==1)
         s=s+4*y[i];
      else
         s=s+2*y[i];
   }
   val=(h/3)*(y[0]+s+y[n-1]);
```

```
    printf("\n\nThe value of the integration is %f",val);
    return 0;
}
```

Table 12.10

Consider the same data set generated from the function $f(x)=\frac{1}{x}$ in **Table 12.3**.

The output of the composite Simpson's $\frac{1}{3}$ rule after executing the C program on the given data set is shown in **Table 12.11**.

```
How many points : 7
Enter the lower and upper limit of integration : 3.1  3.7
h = 0.10 and number of Intervals = 6
f`[3.10] = 0.322581
f`[3.20] = 0.312500
f`[3.30] = 0.303030
f`[3.40] = 0.294118
f`[3.50] = 0.285714
f`[3.60] = 0.277778
f`[3.70] = 0.270270
The value of the integration is 0.176931
```

Table 12.11

The approximate value of the integration is found to be 0.176931. The actual value of integration is already found as 176930. Therefore, the absolute error of integration is $|0.176931 - 0.176930| = 0.000001$. It can be shown that the absolute error for composite Simpson's $\frac{1}{3}$ method is of the order h^4 ($O(h^4)$).

12.3.4 Simpson's 3/8 Rule

Simpson's 3/8 rule is obtained by fitting a third-degree polynomial to four equally spaced discrete points. Therefore, in this case the number of interval is 3 so the upper limit of integration is 3. Therefore,

$$\Delta I_i = h\int_0^3\left(f_i + s\Delta f_i + \frac{s(s-1)}{2}\Delta^2 f_i + \frac{s(s-1)(s-2)}{3!}\Delta^3 f_i\right)ds \quad \textbf{(12.18)}$$

Now replacing the Δ, Δ^2, Δ^3 and integrating the equation **(12.18)** the final result for Simpson's 3/8 rule is

$$\Delta I_i = \frac{3}{8}h(f_i + 3f_{i+1} + 3f_{i+2} + f_{i+3}) \quad \textbf{(12.19)}$$

The composite Simpson's 3/8 rule for equally spaced points is obtained by applying equation **(12.19)** over the entire range of integration. Note that the total number of intervals must be a multiple of three. Thus,

$$I = \frac{3}{8}h(f_0 + 3f_1 + 3f_2 + f_3) + \frac{3}{8}h(f_3 + 3f_4 + 3f_5 + f_6) + \cdots\cdots$$

$$+ \frac{3}{8}h(f_{n-3} + 3f_{n-2} + 3f_{n-1} + f_n)$$

$$\Rightarrow I = \frac{3}{8}h(f_0 + 3f_1 + 3f_2 + 2f_3 + 3f_4 + 3f_5 + 2f_6 + \cdots 3f_{n-1} + f_n) \qquad \textbf{(12.20)}$$

Simpson's 1/3 rule and Simpson's 3/8 rule have the same order of error, $O(h^4)$. Then question may arise why Simpson's 3/8 rule is needed. One reason is that Simpson's 1/3 rule can be used only when the number of intervals is even number. If the total number of intervals is odd then three intervals can be evaluated by the 3/8 rule, and the remaining even number of increments can be evaluated by the 1/3 rule.

Check Your Progress 12.1

The following integrals are used to illustrate numerical integration methods. All of these integrals have exact solutions, which should be used for error analysis in the numerical problem.

A. $\int_0^5 (3x^2 + 2)dx$

B. $\int_0^{10} (5x^4 + 4x^3 + 2x + 3)dx$

C. $\int_0^{\Pi} (5 + \sin(x))dx$

D. $\int_{0.1}^{1} e^x dx$

i. Evaluate integrals (A) and (B) by the trapezoidal rule for 1, 2, 4 intervals. Compute the errors and ratio of the errors.
ii. Evaluate integrals (C) by the trapezoidal rule for 1, 2, 4 intervals. Compute the errors and ratio of the errors.
iii. Evaluate integrals (D) by the trapezoidal rule for 1, 2, 4 intervals. Compute the errors and ratio of the errors.

Check Your Progress 12.2

Write the C program for Simpson's 3/8 rule. Evaluate the integral (D).

Check Your Progress 12.3

Consider the function $f(x)$ tabulated in **Table 12.12**. Evaluate the integral $\int_{1.2}^{2.8} f(x)dx$ using the trapezoidal rule with 1, 2, 4 and 8 intervals. The exact value is 8.43593. Compute the errors and ratios of the errors.

x	$f(x)$	x	$f(x)$	x	$f(x)$
0.4	6.0900	1.4	6.9686	2.2	4.4782
0.6	7.1400	1.6	6.5025	2.4	3.6150
0.8	7.4800	1.8	5.9267	2.6	2.6631
1.0	7.5000	2.0	5.2500	2.8	1.6243
1.2	7.3100				

Table 12.12

Check Your Progress 12.4

Consider the **Table 12.12** again. Evaluate the integral $\int_{1.2}^{2.8} f(x)dx$ using the Simpson's 1/3 rule with 3 intervals. Is this value correct? If not, then evaluate the integral using the strategy mentioned at the end of **Section 12.3.4**.

12.4 Summary

Procedures for developing numerical integration for discrete data are presented in this unit. These procedures are based on fitting approximating polynomials to the data and integrating the approximating polynomials. The Newton-Cotes formulas, which are based on Newton foward-difference polynomials, give simple integration formulas for equally spaced data. Rectangular, trapezoidal and Simpson's $\frac{1}{3}$ and $\frac{1}{8}$ methods are discussed in detail. The formulas for composite integration methods are also derived and explained in this unit. Of all the methods considered, it is likely that Simpson's rules are efficient as these methods produce less error.

12.5 References and Further Reading

1. Elementary Numerical Analysis – An algorithmic Approach, Third Edition, S.D. Conte, Carl de Boor, Tata McGraw-Hill, 2005
2. Numerical Recipes in C, Second Edition, H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Tata McGraw Hill, 2003.
3. Numerical Methods for Engineers and Scientists, Second Editon, Joe D. Hoffman, CRC Press, 2001.