CHAPTER 1

Introduction to Numerical Computing

1.1 INTRODUCTION

Numerical computations play an indispensable role in solving real life mathematical, physical and engineering problems. They have been in use for centuries even before digital computers appeared on the scene. Great mathematicians like Gauss, Newton, Lagrange, Fourier and many others in the eighteenth and nineteenth centuries developed numerical techniques which are still widely used. The advent of digital computers has, however, enhanced the speed and accuracy of numerical computations.

What is numerical computing? It is important to understand the answer to this fundamental question before we proceed further. *Numerical computing* is an approach for solving complex mathematical problems using only simple arithmetic operations. The approach involves formulation of mathematical models of physical situations that can be solved with arithmetic operations. It requires development, analysis and use of algorithms.

Numerical computations invariably involve a large number of arithmetic calculations and, therefore, require fast and efficient computing devices. The microelectronics revolution and the subsequent development of high power, low cost personal computers have had a profound impact on the application of numerical computing methods to solve scientific problems.

The traditional numerical computing methods usually deal with the following topics:

1. finding roots of equations

2. solving systems of linear algebraic equations
3. interpolation and regression analysis
4. numerical integration
5. numerical differentiation
6. solution of differential equations
7. boundary value problems
8. solution of matrix problems

In this book we will discuss some of the popular methods available in each of these areas.

1.2 NUMERIC DATA

Numerical computing may involve two types of data, namely, *discrete data* and *continuous data*. Data that are obtained by counting are called discrete data. Examples of discrete data are the total number of items in a box, or the total number of people participating in a race.

Data that are obtained through measurement are called continuous data. Examples of continuous data are the speed of a vehicle as given by a speedometer, or temperature of a patient as measured by a thermometer.

1.3 ANALOG COMPUTING

Analog refers to the principle of solving a problem by using a tool which operates in a way analogous to the problem. For example, the electronic circuits in an analog computer act analogously to the problem to be solved. Analog computing is based on inputs that vary continuously, such as current, voltage or temperature. The earliest computers were analog and functioned on the basis of electrical voltages. Calculations were performed by adding, subtracting, multiplying and dividing voltages. Analog computers are fast, but their accuracy is limited by the precision with which the physical quantities can be read.

Many real life measurable quantities are analog in nature: time, temperature, pressure, and speed, for instance. Analog methods are preferred when these quantities have to be represented in a calculation. An example of application of analog computers is a machine used in a postal department to convert the weight of a package into the cost of postage needed for mailing.

The basic requirement in the application of analog computers is the writing down of differential equations describing the physical system of interest. Given the differential equations, the analog result may be obtained either by direct method, in which equivalent electrical circuits are directly used to simulate the time variations of the dependent variables of the physical system, or the functional method, in which electronic circuits perform the mathematical operations indicated by the terms of the differential equation.

1.4 DIGITAL COMPUTING

A digital computer is a computing device that operates on inputs which are discrete in nature. The input data are numbers (or digits) that may represent numerals, letters, or other special symbols. Just as a digital clock directly counts the seconds and minutes in an hour, a digital computer counts discrete data values to compute the results.

Today's digital computers can cope with the analog information, but they have to convert it into digital form. They do this by measuring the value of analog quantity at regular intervals and converting that measurement into a number of electrical pulses corresponding to that measurement. In an analog watch, for example, time and hands on the watch face change continuously; a digital watch, however, converts the passage of time into tiny intervals, marked by the numbers changing on the dial.

Digital computers are more accurate than analog computers. Analog computers may be accurate to within 0.1 per cent of the correct value, whereas digital computers can obtain whatever degree of accuracy is required by choosing the correct number of decimal places. They are designed to read, store, manage, and output specific units like numbers, letters, or punctuation marks. Digital computers are widely used for many different applications and are often called *general purpose computers.*

1.5 PROCESS OF NUMERICAL COMPUTING

As stated earlier, numerical computing involves formulation of mathematical models of physical problems that can be solved using basic arithmetic operations. The process of numerical computing can be roughly divided into the following four phases which are illustrated in Fig. 1.1:

1. formulation of a mathematical model
2. construction of an appropriate numerical method
3. implementation of the method to obtain a solution
4. validation of the solution

The formulation of a suitable mathematical model is critical to the solution of the problem. A mathematical model can be broadly defined as a formulation of certain mathematical equation that expresses the essential features of a physical system or process. Models may range from a simple algebraic equation to a complex set of differential equations. Figure 1.2 shows various types of mathematical equations that might result while formulating mathematical models of physical processes.

The formulation of a mathematical model begins with a statement of the problem and the associated factors to be considered. The factors may concern the balance of forces and other laws of conservation in physics.

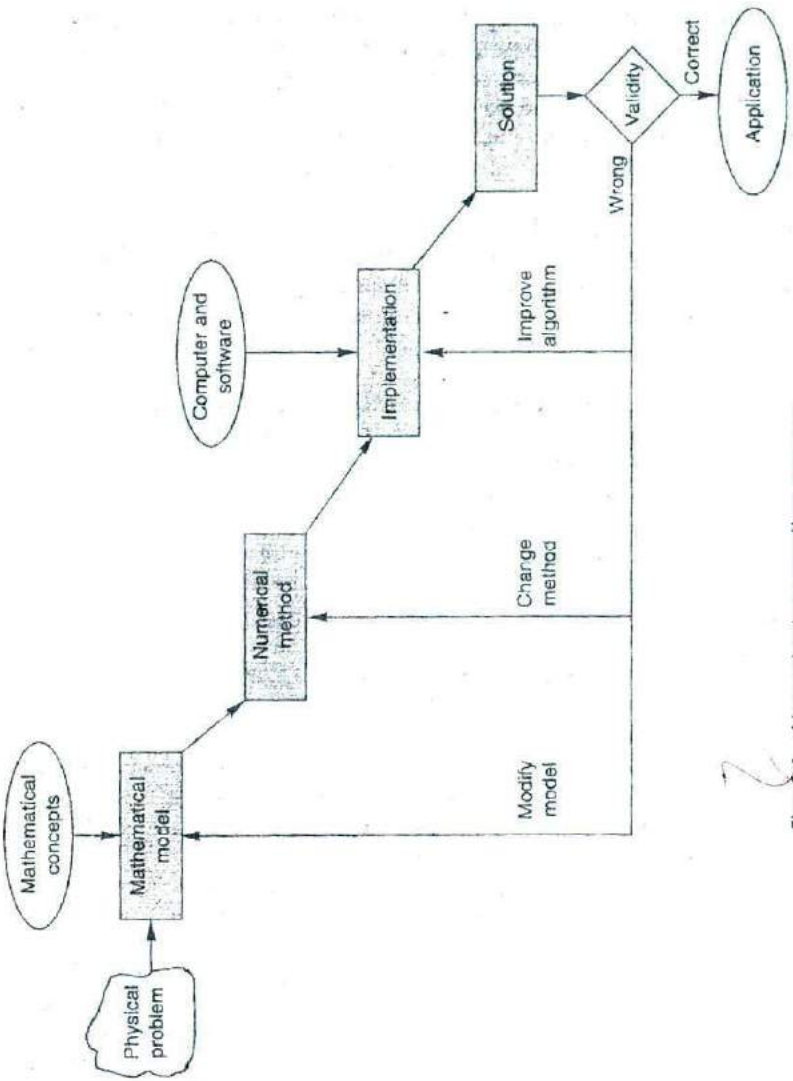


Fig. 1.1 Numerical computing process

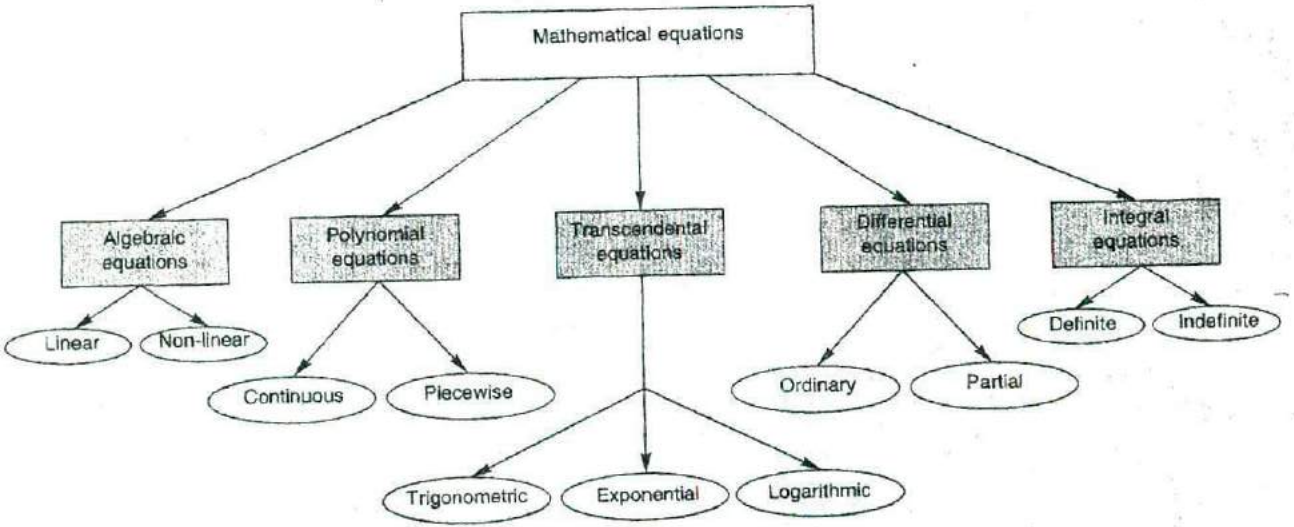


Fig. 1.2 Different forms of mathematical equations

Real life problems have many uncertainties and unknowns. It might, therefore, be necessary to make certain assumptions for approximating and to include only those features of the problem that are considered critical to the final solution. An over-simplified model may have only limited usefulness. The model may be enhanced later, if necessary. The model refinement may make the solution procedure more difficult. We must always maintain the balance of enhancement of the model and accuracy of the solution required.

Once a mathematical model is available, our first step would be to try to obtain an explicit analytical solution. In most cases, the mathematical models may not be amenable to analytical solutions or they may not be solved efficiently using analytical techniques. In such cases, we have to construct appropriate numerical methods to solve mathematical models. As mentioned earlier, a numerical method is a computational technique which involves only a finite number of basic arithmetic operations.

For a given problem, there might be several alternative numerical methods. We must consider different factors or trade-offs before selecting a particular method—such as type of equation, type of computer available, accuracy, speed of execution, and programming and maintenance efforts required.

Modelling is the process of translating a physical problem into a mathematical problem. The process involves

1. making a number of simplifying assumptions
2. identification of important variables
3. postulation of relationships between the variables

This book is mainly concerned with the solution of mathematical models using numerical techniques.

Example 1.1

Formulate a mathematical model for predicting the population growth of a city.

Assumptions:
Birth and death rates are proportional to population and time interval.
Parameters:

$P(t)$ —population at time t

ΔP —increase in population in time interval Δt

Then,

$$\begin{aligned}\Delta P &= \text{births in } \Delta t - \text{deaths in } \Delta t \\ &= C_1 P(t)\, \Delta t - C_2 P(t)\, \Delta t \\ &= (C_1 - C_2)\, P(t)\, \Delta t\end{aligned}$$

$$\text{Growth rate} = \frac{\Delta P}{\Delta t} = CP(t)$$

Taking the limits $\Delta t \to 0$, we get,

$$\frac{dp}{dt} = C\,P(t)$$

Solution of this differential equation is

$$P(t) = P_0\, e^{ct}$$

where P_0 is the population at time $t = 0$.

The population growth depends on the growth constant $C = C_1 - C_2$. The population will be stable if $C_1 = C_2$.

The third phase of the numerical computing process is the implementation of the method selected. This phase is concerned with the following three tasks

1. design of an algorithm
2. writing of a program
3. executing it on a computer to obtain the results

Once we are able to obtain the results, the next step is the validation of the process. Validation means the verification of the results to see that it is within the desired limits of accuracy. If it is not, then we must go back and check each of the following:

1. mathematical model itself
2. numerical method selected
3. computational algorithm used to implement the method

This may mean modification of the model, selection of an alternate numerical method or improving the algorithm (or a combination of them). Once a modification is introduced, the cycle begins again. Figure 1.3 illustrates how the numerical computing cycle moves from the real world to mathematical world and back.

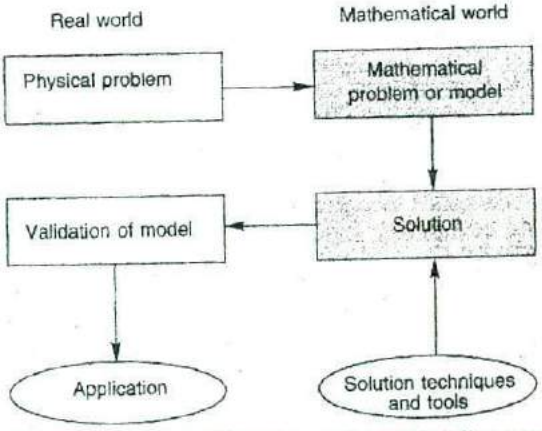


Fig. 1.3 Another way of looking at the computing process

1.6 CHARACTERISTICS OF NUMERICAL COMPUTING

Numerical methods exhibit certain computational characteristics during their implementation. It is important to consider these characteristics while choosing a particular method for implementation. The characteristics that are critical to the success of implementation are: accuracy, rate of convergence, numerical stability, and efficiency.

Accuracy

Every method of numerical computing introduces errors. They may be either due to using an approximation in place of an exact mathematical procedure (known as *truncation errors*) or due to inexact representation and manipulation of numbers in the computer (known as *roundoff errors*). These errors affect the accuracy of the results. The results we obtain must be sufficiently accurate to serve the purpose for which the mathematical model was built. Choice of a method is, therefore, very much dependent on the particular problem. The general nature of these errors will be discussed in detail in Chapter 4.

Rate of Convergence

Many numerical methods are based on the idea of an *iterative process*. This process involves generation of a sequence of approximations with the hope that the process will converge to the required solution. Certain methods converge faster than others. Some methods may not converge at all. It is, therefore, important to test for convergence before a method is used. Rapid convergence takes less execution time on the computer. There are several techniques for accelerating the rate of convergence of certain methods. The concepts of convergence and divergence are discussed in Chapter 4. They are also discussed in various places where specific methods are analysed for convergence.

Numerical Stability

Another problem introduced by some numerical computing methods is that of *numerical instability*. Errors introduced into a computation, from whatever source, propagate in different ways. In some cases, these errors tend to grow exponentially, with disastrous computational results. A computing process that exhibits such exponential error growth is said to be numerically unstable. We must choose methods that are not only fast but also stable.

Numerical instability may also arise due to ill-conditioned problems. There are many problems which are inherently sensitive to round off errors and other uncertainties. Thus, we must distinguish between sensitivity of methods and sensitivity inherent in problems.

When the problem is ill-conditioned, there is nothing we can do to make a method to become numerically stable.

Efficiency

One more consideration in choosing a numerical method for solution of a mathematical model is *efficiency*. It means the amount of effort required by both human and computer to implement the method. A method that requires less of computing time and less of programming effort and yet achieves the desired accuracy is always preferred.

1.7 COMPUTATIONAL ENVIRONMENT

The last phase of the numerical computing process, namely the implementation phase, requires resources such as computer hardware, operating system and other systems software, language compilers, actual application programs and other software tools to manipulate data and provide output in a desired form.

The computer hardware may range from a small personal computer to a large super computer depending on the nature and size of the problem. A program may not always produce the same results on two different types of computers due to difference in their round off errors.

Appropriate operating systems and compilers play an important role in developing portable programs. UNIX and MS-DOS have become popular operating systems for scientific computing. FORTRAN language has dominated the scientific computing field for the last four decades and it is expected to continue its predominant role for some more years. It has been continuously modified and extended to support the ever changing requirements of software engineering. The likely strong competitor for FORTRAN in the near future will be C and C++ languages which contain some unique features and powerful control structures. Portability is another strong point of these languages.

1.8 NEW TRENDS IN NUMERICAL COMPUTING

In recent years, the increasing power of computer hardware has affected the approach of numerical computing in several ways. It has forced scientists and engineers to search for algorithms that are computationally fast and efficient. An important new trend is the construction of algorithms to take advantage of specialised computer hardware such as *vector computers* and *parallel computers*. Another trend is the use of sophisticated interactive graphics, in which the user can view the results graphically and advise the computer, graphically, on how to proceed further.

One important development which is likely to have an increasing impact on scientific computing is *symbolic computation*. Symbolic computation systems would enable us to add, multiply and divide polynomials or rational expressions the same way we would do using pencil and paper. They can also solve certain mathematical problems

without rounding off errors. Symbolic computation is expected to play an increasing role in scientific computation.

Object-oriented numerical computing is gaining importance due to the popularity of languages like C++ and Java. They incorporate concepts such as encapsulation, inheritance, polymorphism and operator overloading. They support the idea that program units should interact with one another only through clearly defined interfaces. They also enable the extension (or reuse) of the existing code without modifying it.

1.9 MATHEMATICAL BACKGROUND

This book assumes that the readers have some mathematical background. They require basic knowledge of algebra, functions, matrices, and integral and differential calculus.

1.10 SUMMARY

In this chapter, we have introduced the concept of numerical computing and discussed the steps involved in solving a physical problem using numerical methods. We also discussed the characteristics of numerical computing and computing resources required for implementing a numerical method.

Key Terms

Accuracy
Algorithm
Analog computer
C
C++
Continuous data
Digital computer
Discrete data
Efficiency
FORTRAN
General purpose computers
Ill-conditioned problems
Iterative process
Mathematical model
Numerical computing
Numerical method
Numerical stability
Parallel computers
Rate of convergence
Round off error
Symbolic computation
Truncation error
Validation
Vector computer

REVIEW QUESTIONS

1. What is Numerical Computing?
2. Distinguish between analog computing and digital computing.
3. Describe, with the help of a block diagram, the process of numerical computing.

4. Newton's second law of motion states that the time rate of change of momentum of a body is equal to the resultant force acting on it. Using this law, formulate a mathematical model to determine the terminal velocity of a free falling body near the earth's surface.
5. The Newton's law of cooling states that the rate of heat from a liquid is proportional to the difference in temperatures between the liquid and the surroundings. Formulate a mathematical model to govern this law.
6. When a boat moves through water, the retarding force is proportional to the square of the velocity. Formulate a differential equation in terms of velocity given the mass m and the drag coefficient k.
7. State the four characteristics of numerical computing.
8. What is accuracy? How is it affected during the process of numerical computing?
9. What is convergence? How is it important in numerical computing?
10. What do you mean by numerical instability?
11. Distinguish between sensitivity of methods and sensitivity of problems.
12. Describe resources required for implementing a numerical computing process.

CHAPTER 2

Introduction to Computers and Computing Concepts

2.1 INTRODUCTION

In Chapter 1, we discussed that numerical computing requires two important tools, namely, mathematical methods and computers. Most numerical methods cannot be solved without the help of computers. Therefore, a background knowledge of computers and computing concepts will enhance the understanding of implementation of numerical computing solutions. This chapter provides some basic information on computing environment and problem solving approach using computers.

The spate of innovations and inventions in computer technology during the last two decades has led to the development of a variety of personal computers. They are so versatile that they have become indispensable to engineers, scientists, business executives, managers, administrators, accountants, teachers and students. They have strengthened humankind's powers in numerical computations and information processing.

Modern computers possess certain characteristics and abilities peculiar to them. They can

1. perform complex and repetitive calculations rapidly and accurately
2. store large amounts of data and information for subsequent manipulations
3. hold a program of a model which can be explored in many different ways
4. make decisions
5. provide information to the user

6. automatically correct or modify certain parameters of a system under control
7. draw and print graphs
8. converse with users interactively

Engineers and scientists make use of the high-speed computing capability of computers to solve complex mathematical models and design problems. Many calculations that were previously beyond contemplation have now become possible. But for computers, many of the technological achievements, such as landing on the moon, would not have been possible.

Computers have helped automation of many industrial and business systems. They are used extensively in manufacturing and processing industries, power distribution systems, airline reservation systems, transportation systems, banking systems, and so on. *Computer-aided design* (CAD) and *computer-aided manufacture* (CAM) are among the most popular industrial applications today.

Modelling and *simulation* is another area where computers are increasingly used. This has greatly accelerated research in such areas as physical and social sciences, medicine, astronomy and meteorology.

Business and commercial organisations need to store and maintain voluminous records and use them for various purposes such as inventory control, sales analysis, payroll accounting, resources scheduling and generation of management reports. Computers can store and maintain files and can sort, merge or update them as and when necessary.

The ability of computers to store large amounts of data has led to their application in libraries, documentation centres, employment exchanges, police departments, hospitals and other similar establishments. Computers are used in international games such as the Olympics to keep track of events and provide timely and reliable information and documentation to all concerned.

Since computers can bank a variety of information and converse with the users, they are being used as resources in teaching and learning at all levels of education and training. This process is known as *computer-assisted learning* (CAL). Here, learners can communicate directly with a computer in a conversational mode. Using this mode, a learner can learn a topic in his own time and pace.

Computers are also used to manage the learning processes. This is called *computer-managed learning* (CML). Computers can store students' responses, evaluate their performance and then direct them to the next appropriate learning unit.

The areas of computer applications are too numerous to mention. Computers have become an integral part of our everyday life. They continue to grow and open new horizons of discovery and application such as the electronic office, electronic commerce, and the home computer centre.

The microelectronics revolution has placed enormous computational power within the reach of every scientist and engineer. However, it

must be remembered that computers are machines created and managed by humans. A computer has no brain of its own. Anything it does is the result of human instructions. It is an obedient slave which carries out the master's orders as long as it can understand them, no matter whether they are right or wrong. In short, computers lack common sense. These instructions constitute the program or software.

2.2 EVOLUTION OF NUMERICAL COMPUTING AND COMPUTERS

The use of computing techniques is over 5000 years old. The Babylonians, Chinese, and Egyptians used numerical methods for the survey of lands and the collection of taxes as early as 3000 BC. Computing history starts with the development of a device called the *abacus* by the Chinese around this period. This was used for the systematic calculation of arithmetic operations. Since then the number system has undergone various changes and has been used in different forms in computing. The most significant development in computing was the formulation of the decimal number system in India around 800 AD. Another significant development was the invention of *logarithm* by John Napier in 1614, which made computing simple.

The modern age of mathematics emerged during the 17th century when Johannes Kepler and Galileo Galilee deduced the laws for planetary motion and Sir Isaac Newton formulated the law of gravity. The subsequent developments in mathematics and other sciences increased the need for new computing techniques and devices.

The principle of logarithm was later applied to a calculating device known as the *slide rule*, which was extensively used till recently. The first accounting machine was built in France by Blaise Pascal in 1642. Then came the Leibnitz calculator in 1671 designed by Gottfried Wilhelm von Leibnitz. These machines progressed in technology and variety and became the standard calculating machines of the business community. During the beginning of the 19th century, Joseph Marie Jacquard invented an automated loom operated by a mechanism controlled by punched cards.

The origin of the modern computer can be traced back to 1834 AD, when an English mathematician, Charles Babbage, designed an analytical engine. This is considered to be the first programmable digital mechanical computer. However, this kind of machine was not built until 1944, when Mark I, an electromechanical automatic computer, was developed by IBM. Subsequently, a series of technological improvements and innovations took place and the design of computers underwent continuous and dramatic changes. Some of the important developments since the slide rule are given in Table 2.1.

Table 2.1 Some developments in computing technology

Year	*Device*
1622	Slide rule
1642	Pascal calculator, an accounting machine by Blaise Pascal
1671	Leibnitz calculator
1801	Punched card loom by Jacquard
1822	Difference engine by Charles Babbage
1834	Analytical engine by Charles Babbage
1890	Punched card machine by Herman Hollerith
1930	Differential analyser by Vannevar Bush
1936	Paper on computational numbers by Alan Turing Link between symbolic logic and electric circuit by Claude Shanon
1937	Binary adder built by George Stibitz
1941	First general-purpose computer designed by Konrad Zuse
1943	Colossus machine built to crack German secret codes, by the British
1944	First automatic computer, MARK I, designed by Howard Aiken
1945	Critical elements of a computer system outlined by John Von Neumann
1946	First electronic digital computer, ENIAC, put to operation by Presper Eckert and John Mauchly
1947	Transistor invented by John Bardeen, William Shockley and Walter Brattain
1951	First business computer, UNIVAC, became operational
1956	Second generation computer (using transistors) introduced by Bell Laboratory
1959	Integrated circuits (ICs) demonstrated by Clair Kilby
1964	First third generation computer using ICs developed
1965	First commercial minicomputer, PDP-8, introduced by Digital Equipment Corporation
1971	Intel 4004 microprocessor designed by Ted Hoff
1974	First fourth generation computer (using microprocessors) built by Ed Roberts
1975	First personal computer software created by Bill Gates and Paul Allen
1977	Apple introduced its famous personal computer
1981	IBM PC introduced in the market
1982	Cray supercomputer marketed by Cray Research Company
1989	Optical computer demonstrated

Modern Computers

The era of modern computers began in 1951 when the UNIVAC (Universal Automatic Computer) became operational at the Bureau of Census in USA. Since then, computers started appearing in quick succession, each claiming an improvement over the other. They represented improvements in speed, memory (storage) systems, input and output devices and programming techniques. They also showed a continuous reduction in physical size and cost. The developments in computers are closely associated with the developments in material technology, particularly the semiconductor technology.

Computers developed after ENIAC have been classified into the following four generations:

First generation	1946 – 1955
Second generation	1956 – 1965
Third generation	1966 – 1975
Fourth generation	1976 – present

You may notice that from 1946, each decade has contributed one generation of computers.

In the *first generation* computers vacuum tubes were used. Magnetic tape drives and magnetic core memories were developed during this period. The first generation computers possessed the following drawbacks as compared to the later models:

1. large in size
2. slow operating speeds
3. restricted computing capacity
4. limited programming capabilities
5. short life span
6. complex maintenance schedules

The *second generation* computers were marked by the use of a solid-state device, called the transistor, in the place of vacuum tubes. These machines were much faster and more reliable than their earlier counterparts. Further, they occupied less space, required less power, and produced much less heat.

Research in the field of electronics led to the innovation of the integrated circuits, now popularly known as IC chips. The use of IC chips in the place of transistors gave birth to the *third generation* computers. They were still more compact, faster and less expensive than the previous generation.

Along with the third generation computers, newer and faster equipments were introduced for handling storage and input-output.

Continued efforts towards miniaturisation led to the development of large-scale integration (LSI) technology. Intel Corporation introduced LSI chips called microprocessors for building computers. The latest child of the computer family that uses VLSI chips has been named the *fourth generation* computer. The fourth generation computers are marked with an increased user-computer interaction and speed. Table 2.2 gives an idea of the main features of each generation.

Fifth Generation Computers

Japan and many other countries are working on systems that are known as *knowledge-based* or *expert systems* which will considerably improve the man-machine interaction. Such systems would integrate the advancements in both hardware and software technologies and would facilitate computer-aided problem-solving with the help of organised information in many specialised areas.

Table 2.2 Computer generations

Features	*Generation*			
	First	*Second*	*Third*	*Fourth*
Main component	Vacuum tube	Transistor	Integrated circuit (IC Chips).	LSI and VLSI circuit
Internal storage (Memory)	Electrostatic tubes	Magnetic core Magnetic drum	Magnetic core	Semiconductor memory
External storage (Auxiliary memory)	Paper tape Punched card Magnetic tape	Magnetic disk Magnetic drum Magnetic tape Paper tape Punched card	Magnetic disk Magnetic tape Magnetic drum Punched card Paper tape	Magnetic disk Magnetic tape Magnetic drum Floppy disk CD rom
Speed of operation (Additions/second)	40 to 300 thousands	3,000 to 30,000 thousands	30,000 to 3,00,000 thousands	3,00,000 to 30,00,000 thousands

This generation of computers is called the *fifth generation computers*. Although knowledge-based systems are expensive and time-consuming to build, they are likely to become more popular in the coming years.

2.3 TYPES OF COMPUTERS

Computers may be classified based on operating principles, size and capability, and applications.

Principles of Operation

Based on the operating principles, computers can be classified into any one of the following types: digital computers, analog computers, and hybrid computers.

Digital computers operate essentially by counting. All quantities are expressed as discrete digits or numbers. Digital computers are useful for evaluating arithmetic expressions and for manipulations of data (such as preparation of bills, ledgers, solution of simultaneous equations, etc.).

Analog computers operate by measuring rather than by counting. The name, which is derived from the Greek word *analog*, denotes that the computer functions by establishing similarities between two quantities that are usually expressed as voltages or currents. Analog computers are powerful tools to solve differential equations. Computers which combine features of both analog and digital types are called *hybrid computers*.

A majority of the computers used today are digital. As their name suggests, digital computers were originally designed to perform certain numerical calculations. They gradually replaced almost all mechanical calculating devices. Later, the concept of stored programs enabled them to store data and instructions and perform certain sequences and combinations of arithmetic operations automatically. This has led to the use of digital computers in a variety of applications.

Applications

Modern computers, depending upon their applications, are classified as special purpose computers or general purpose computers.

Special purpose computers are tailor-made to cater solely to the requirements of a particular task or application. They incorporate the instructions needed into the design of internal storage so that they can perform the given task on a simple command. They, therefore, do not possess unnecessary options and cost less.

On the other hand, *general purpose* computers are designed to meet the needs of many different applications. In a general purpose computer, the instructions needed to perform a particular task are not wired permanently into the internal memory. When one job is over, instructions for another job can be loaded into the internal memory for processing. Thus, a general-purpose machine can be used to prepare pay-bills, manage inventories, print sales reports, and so on.

Size and Capability

Computers are also available in different sizes and with different capabilities. Broadly, they may be categorised as microcomputers, minicomputers, mainframes and supercomputers. The selection of a particular system primarily depends on the volume of data to be handled and the speed of the processor.

Microcomputers A *microcomputer* is the smallest general-purpose processing system. Functionally, it is similar to any other large system. Microcomputers are self-contained units and are usually designed for use by one person at a time. Since microcomputers can be easily linked to large computers, they form a very important segment of the integrated information systems.

Minicomputers A *minicomputer* is a medium-sized computer that is more costly and powerful than a microcomputer. An important distinction between a microcomputer and a minicomputer is that the latter is usually designed to serve multiple users simultaneously. A system that supports multiple users is called multiterminal, time-sharing system. Minicomputers are the popular computing systems among research and business organisations today.

Mainframe computers Computers with large storage capacities and very high speed of processing (compared to micro or minicomputers) are known as *mainframe computers*. They support a large number of terminals for use by a variety of users simultaneously. They are also used as the central host computer in distributed data processing systems.

Supercomputers *Supercomputers* have extremely large storage capacities and computing speeds that are many times faster than other computers. While the speed of traditional computers is measured in terms of millions of instructions per second (mips), a supercomputer is rated in tens of millions of operations per second (mops) (an operation is made up of numerous instructions). Typically, the supercomputer is used for large-scale numerical problems in scientific and engineering disciplines. These include applications in electronics, petroleum engineering, weather forecasting, structural analysis, chemistry, medicine and physics.

Personal computers *Personal computers* are nothing but micro- computers that are specially designed for personal use of individuals. The name "personal computer" was coined by IBM when it marketed its first microcomputer in 1981. Since then, many companies have produced IBM compatible PCs. During the last fifteen years, the processor chips used in IBM compatible PCs have undergone dramatic improvements in their performance characteristics. Table 2.3 shows the characteristics of various PC processor chips. Note that today's PCs are far more powerful than the mainframes of just a few years ago.

Table 2.3 Characteristics of microprocessor chips

	8088 PC XT	*286 PC AT*	*386*	*486*	*Pentium*	*Pentium Pro*	*Pentium II*
Clock speed (megahertz)	4.77	6–12	16–33	16–50	66–200	120–200	200+
Data path (bits)	8	16	32 16 (SX)	32	64	64	64
Computation size (bits)	16	16	32	32	32	32	32
Memory-size (bytes)	640K	2 megs	4–16 megs	4–64 megs	4–64 megs	16–64 megs	16–64 megs
Floating point	Coprocessor	Coprocessor	Coprocessor	On chip	On chip	On chip	On chip
Speed (MIPS)	0.33	1.2	2.5–6	20–40	112	250	500
Number of transistors per chip	29,000	130,000	275,000	1.2 million	3.3 million	5.2 million	10+ million

Workstations There is a class of computers, known as *workstations*, which lie in between minicomputers and microcomputers in terms of

processing power. A workstation looks like a personal computer but is specially designed for engineering and graphics applications.

Parallel computers *Parallel computer* is a relatively new type of computer that uses a large number of processors. The processors perform different tasks independently and simultaneously, thus, improving the speed of execution of complex programs dramatically. Parallel computers match the speed of supercomputers at a fraction of the cost.

2.4 COMPUTING CONCEPTS

A computer, small or big, is basically a device used for processing of data (numbers) and text (words). It performs essentially the following three operations in a sequence:

1. receives data (and instructions)
2. processes data (as per the instructions)
3. outputs result (information)

This cycle of operation of a computer is known as the *input–process–output* cycle and is shown in Fig. 2.1.

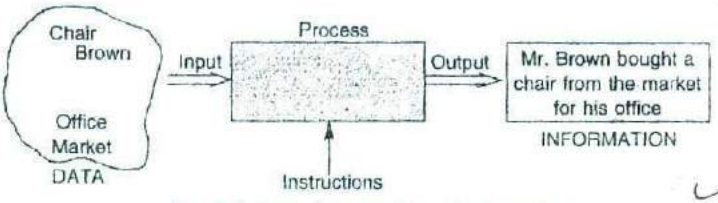


Fig. 2.1 Input-process-output cycle

Raw facts, known as *data,* are provided to the computer in bits and pieces. They are encoded in such a way that the computer can understand them. The computer then processes the data with the help of certain instructions provided to it, and produces a meaningful and desired output known as *information*. For example, if the data consists of two numbers, say, 10 and 15 and the instruction is to add them and print out the result, then the output information would be the sum of the two numbers, i.e. 25. A set of instructions designed to perform a particular sequence of functions is called a *computer program*.

Processing is nothing but manipulation of data in accordance with certain procedures to suit the need of the user (or application). The same basic data can provide several kinds of information depending upon the type of instructions.

Input is usually through a keyboard (like a typewriter) and output may be obtained either on a *display screen* or on a *printer*. While the printer produces typed copy on paper (usually known as *hard copy*), the screen display (*soft copy*) allows the user to verify the output before it is printed.

A computer often includes an external storage system to store (and retrieve) data and programme. The popular storage medium is a floppy disk. Other media, such as hard disks, magnetic tapes and CD ROMs are also used. All these physical components are known as *hardware*.

2.6 COMPUTER ORGANISATION

Although computers differ widely in their details, all of them follow a basic organisational structure as shown in Fig. 2.2. In order to carry out the three basic operations, namely, input, process and output, a computer includes the following hardware components: input devices, processing units, output devices and external storage devices.

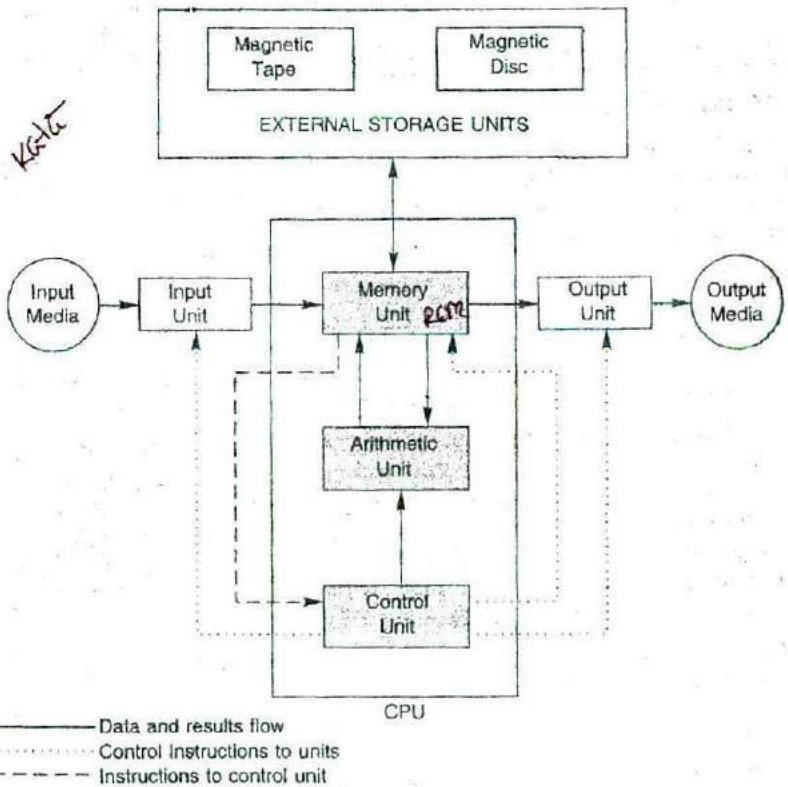


Fig. 2.2 Structure of a computer

Input Devices

An input device presents data to the processing unit in machine-readable form. Although the keyboard is a common input device for a small computer, a system may also support one or more of the input devices given in Table 2.4.

Table 2.4 Input devices

S.No.	Device	Medium of data storage	Remarks
1.	Optical character reader (OCR)	Special paper document	Only input
2.	Magnetic ink character recogniser (MICR)	Special paper document	Only input
3.	Mark sense reader	Special paper or card	Only input
4.	Graphics tablet	Document	Only input
5.	Mouse	Document	Only input
6.	Floppy drive	Floppy disk	Input, output, storage
7.	Hard disk (Winchester) drive	Magnetic disk	Input, output, storage
8.	Tape drive	Magnetic tape	Input, output, storage
9.	CD ROM drive	CD ROM	Input, storage

Processing Units

Processing units receive data and instructions, store them temporarily and then process the data as per the instructions. The processing units include: memory unit, arithmetic logic unit, and control unit. All three units together are known as the *central processing unit* (CPU).

Memory unit The memory unit holds (stores) all data, instructions and results temporarily. The memory consists of hundreds of thousands of cells called 'storage locations', each capable of storing one word of information. The memory unit is called by different names, such as storage, internal storage, primary storage, main memory or simply memory.

Arithmetic logic unit This unit is used to perform all the arithmetic and logic operations, such as addition, multiplication, comparison, etc. For example, consider the addition of two numbers A and B. The control unit will select the number A from its location in the memory and load it into the arithmetic logic unit. Then it will select the number B and add it to A in the arithmetic unit. The result will then be stored in the memory or retained in the arithmetic unit for further calculations.

Control unit This unit coordinates the activities of all the other units in the system. Its main functions are:

1. to control the transfer of data and information between various units
2. to initiate appropriate actions by the arithmetic unit

The program provides the basic control instructions. Conceptually, the control unit fetches instructions from the memory, decodes them, and directs various units to perform the specified tasks.

Output Devices

Output devices receive information from the CPU and present it to the user in the desired form. Although a printer is the most commonly used

output device, devices such as plotters are also becoming popular. Some common output devices are given in Table 2.5.

Table 2.5 Output devices

Device	*Medium of presentation*	*Remarks*
Printer	Paper	Only output
Plotter	Paper	Only output
Visual display unit (VDU)	Display screen	Only output
Floppy drive	Floppy disk	Input, output, storage
Disk drive	Magnetic disk	Input, output, storage
Tape drive	Magnetic disk	Input, output, storage

External Storage Devices

The purpose of external storage is to retain data and programs for future use. For example, a program may be required at regular intervals. If such information is stored in an external storage media, then one can retrieve it as and when necessary, thus avoiding the need to type it again. Any number of files containing information can be stored on external media. Since they are permanent (they are not erased when the equipment is turned off), one can store long files on external media, and later on work on them in sections, keeping all the sections in storage except the one currently in use.

The popular external storage media used with micro and mini computers are floppy disks, hard disks and CD ROMs.

Floppy disks The most common storage medium used on small computers today is a *floppy disk*. It is a flexible plastic disk coated with magnetic material and looks like a phonograph record. Information can be recorded or read by inserting it into a disk drive connected to the computer. The disks are permanently encased in stiff paper jackets for protection and easy handling. An opening is provided in the jacket to facilitate reading and writing of information (Fig. 2.3).

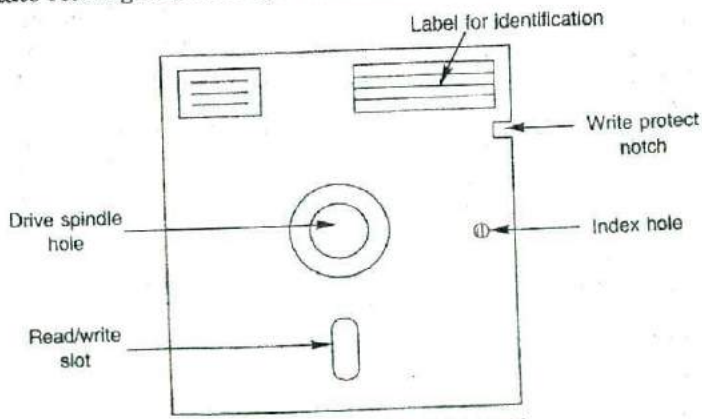


Fig. 2.3 Floppy disk (5.25 inch)

Floppy disks are available in two standard sizes—5.25 inch and 3.5 inch. The 3.5 inch floppy disk, which was introduced later, can store more information than the previous one.

Hard disks Another magnetic media suitable for storing large volumes of information is the *hard disk*, popularly known as the *Winchester disk*. A hard disk pack consists of two or more magnetic plates fixed to a spindle, one below the other, with a set of read/write heads. The disk pack is permanently sealed inside a casing to protect it from dust and other contaminations, thus increasing its operational reliability and data integrity.

Winchester disks possess a number of advantages compared to floppy disks:

1. They can hold much larger volumes of information than floppies.
2. They are very fast in reading and writing.
3. Hard disks are not susceptible to dust and static electricity.

Winchester disks are available in different sizes and capacities. Standard sizes are 5.25 inch, 8 inch, 10.5 inch and 14 inch. Storage capacities of 260, 540, 680, 1000, 1200, 2000 megabytes are typical on a personal computer.

CD ROMs Compact disk read-only memory (CD ROM) disks are used to distribute large volumes of data and text. Computer programs and user manuals are often distributed on CD ROMs.

2.6 DRIVING THE COMPUTER: THE SOFTWARE

Computers need clear-cut instructions to tell them *what to do*, *how to do*, and *when to do*. A set of instructions to carry out these functions is called a *program*. A group of such programs that are put into a computer to operate and control its activities is called the *software*. These programs must reside in the internal storage (memory) to execute their instructions. For example, if we want to delete some data stored in memory, the system uses one set of program instructions. Similarly, if we want to sort a list of names, it uses another set of instructions designed to perform this task.

Software is an essential requirement of computer systems. Just as a car cannot run without fuel, a computer cannot work without software. There are four major kinds of software that are implemented as shown in Fig. 2.4: operating system, utility programs, language processors and application programs.

Software is intangible but resides on or is stored in something tangible, such as floppy disks and magnetic tapes.

Operating System

The software that manages the resources of a computer system and schedules its operation is called the *operating system*. The operating

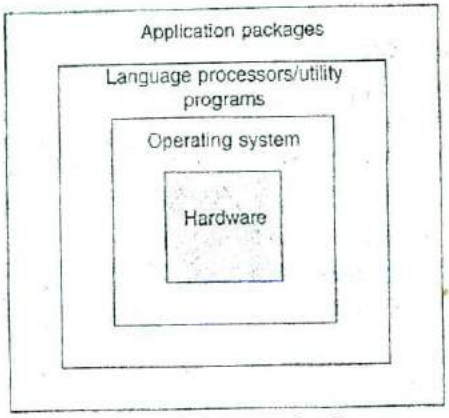


Fig. 2.4 Layers of software

system acts as an interface between the hardware and the user programs and facilitates the execution of the programs (Fig. 2.4). The principal functions of operating system include:

1. to control and coordinate peripheral devices such as printers, display screen and disk drives
2. to monitor the use of the machine's resources
3. to help the application programs execute its instructions
4. to help the user develop programs
5. to deal with any faults that may occur in the computer and inform the operator

The operating system is usually available with hardware manufacturers and is rarely developed in-house owing to its technical complexity. Small computers are built from a wide variety of microprocessor chips and use different operating systems. Hence, an operating system that runs on one computer may not run on the other. The popular operating systems include, among others, MS DOS and UNIX.

Utility Programs

There are many tasks common to a variety of applications. Examples of such tasks are:

1. sorting a list in a desired sequence
2. merging of two programs
3. copying a program from one place to another
4. report writing

One need not write programs for these tasks. They are standard, and normally handled by *utility programs*.

Like operating systems, utility programs are pre-written by the manufacturers and supplied with the hardware. They may also be obtained from standard software vendors. A good range of utility programs can make life much easier for the user.

Language Processors

Computers can understand instructions only when they are written in their own language called the *machine language*. Therefore, a program written in any other language should be translated into machine language. Special programs called *language processors* are available to do this job.

These special programs accept the user programs and check each statement and, if it is grammatically correct, produce a corresponding set of machine code instructions. Language processors are also known as *translators*.

There are two forms of translators: compilers and interpreters.

A *compiler* checks the entire user-written program (known as the *source program*) and, if error-free, produces a complete program in machine language (known as *object program*). The source program is retained for possible modifications and corrections and the object program is loaded into the computer for execution.

An *interpreter* does a similar job but in a different style. The interpreter (as the name implies) translates one statement at a time and, if error-free, executes the instruction. This continues till the last statement. Thus an interpreter translates and executes the first instruction before it goes to the second, while a compiler translates the entire program before execution.

The major differences between a compiler and an interpreter are:

1. Error correction (called *debugging*) is much simpler in the case of the interpreter because it is done in stages. The compiler produces an error list for the entire program at the end.
2. Interpreters take more time for the execution of a program compared to compilers because a statement has to be translated every time it is read.

Compilers and interpreters are usually written and supplied by the hardware vendors. Since a compiler (or an interpreter) can translate only a particular language for which it is designed, one will need to use a separate translator for each language.

Application Programs

While an operating system makes the hardware run properly, *application programs* make the hardware do useful work. Application programs are specially prepared to do certain specific tasks. They can be classified into two categories: standard applications, and unique applications.

Some applications are common for many organisations. Ready-to-use software packages for such applications are available from hardware and/or software vendors. Standard packages include, among others, Sales Ledger, Purchase Ledger, Statistical Analysis, Pay Roll, PERT/CPM, Production Planning and Control, Inventory Management, and Linear Programming.

In some situations one may have to develop one's own programs to suit one's unique requirements. Once developed, they come into the category of unique application packages.

2.7 PROGRAMMING LANGUAGES

The functioning of a computer is controlled by a set of instructions (called a *computer program*). These instructions are written to tell the computer:

1. what operation to perform
2. where to locate data
3. how to present results
4. when to make certain decisions

The communication between two parties, whether they are machines or human beings, always needs a common language or terminology. The language used in the communication of computer instructions is known as the programming language. The computer has its own language and any communication with the computer must be in its language or translated into this language.

Three levels of programming languages are available. They are:

1. machine languages (low level languages)
2. assembly (or symbolic) languages
3. procedure-oriented languages (high level languages)

Machine Language

Computers are made of two-state electronic components which can understand only pulse and no-pulse (or '1' and '0') conditions. Therefore, all instructions and data should be written using *binary codes* 1 and 0. The binary code is called the *machine code* or *machine language*.

Computers do not understand English, Hindi or Tamil. They respond only to machine language. Added to this, computers are not identical in design. Therefore, each computer has its own machine language. (However, the script 1 and 0, is the same for all computers). This poses two problems for the user.

First, it is difficult to understand and remember the various combinations of 1's and 0's representing numerous data and instructions. Also, writing error-free instructions is a slow process.

Secondly, since every machine has its own machine language, the user cannot communicate with other computers (if he does not know its language). Imagine a Tamilian making his first trip to Delhi. He would face enormous obstacles as the language barrier would prevent him from communicating.

Assembly Language

An *assembly language* uses mnemonic codes rather than numeric codes (as used in machine language). For example, ADD or A is used as a symbolic operation code to represent addition and SUB or S is used for

subtraction. Memory locations containing data are given names such as TOTAL, MARKS, TIME, MONTH, etc.

As the computer understands only machine code instructions, a program written in assembly language must be translated into machine language before the program is executed. This translation is done by a computer program referred to as an *assembler*.

The assembly language is again a machine-oriented language and hence, the program has to be different for different machines. The programmer should remember machine characteristics when he prepares a program. Writing a program in assembly language is still a slow and tedious task.

Procedure-Oriented Language (POL)

These languages consist of a set of words and symbols and one can write programs using these in conjunction with certain rules. These languages are oriented toward the problem to be solved or procedures for solution rather than mere computer instructions. These are more user-centered than the machine-centered languages. They are better known as *high-level languages*.

The most important characteristic of a high-level language is that it is machine-independent and a program written in a high-level language can be run on computers of different makes with little or no modification. The programmer need not know the characteristics of that machine. However, such programs need to be translated into equivalent machine-code instructions before actual implementation.

A program written in a high-level language is known as the *source program* and can be run on different machines using different translators. The translated program is called the *object program*. The major disadvantage of high-level languages is that they take extra time for conversion and thus, are less efficient compared to the machine-code languages. Figure 2.5 shows the system of implementing the three levels of languages.

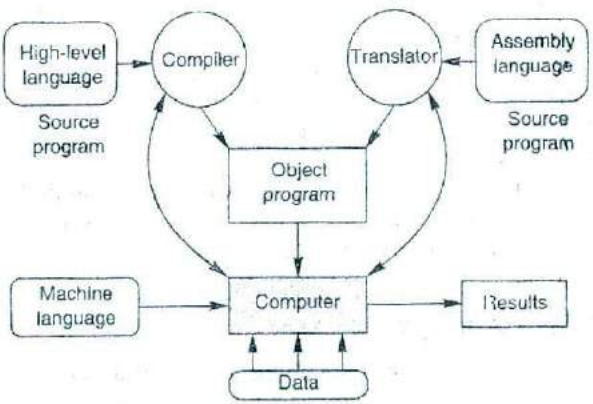


Fig. 2.5 Implementation of a program

Common High-level Languages

Many high-level languages have been developed during the last three decades. The most common high-level languages are FORTRAN, BASIC, COBOL, C, PL/1, C++ and Java. Although, they are less efficient than the machine or assembly languages, they relieve the programmers of the tedious task of remembering numeric codes for storage locations, operations, etc. In addition, these languages are easier to learn and use.

The choice of a language depends upon many factors such as the knowledge of the programmer, the computer, the problem to be solved, etc. The languages that are used more popularly are given in Table 2.6.

Table 2.6 Summary of common high-level languages

Year	*Language*	*Name derived from*	*Developed by*	*Application*
1957	FORTRAN	FORmula TRANslation	IBM	Science, engineering
1958	ALGOL	ALGOrithmic Language	International group	Science, engineering
1959	LISP	LISt Processing	MIT, USA	Artificial engineering
1960	APL	A Programming Language	IBM	Science, engineering
1961	COBOL	COmmon Business Oriented Language	Defence Dept., USA	Business
1964	BASIC	Beginner's All purpose Symbolic Instruction Code	Dartmouth College, USA	Engineering, science, business, education
1965	PL/1	Programming Language 1	IBM	General
1970	Pascal	Blaise Pascal	Federal Institute of Technology, Switzerland	General
1972	PROLOG	PROgramming in LOGic	University of Marseille	Artificial intelligence
1973	C	Earlier language called B	Bell Laboratory	General
1975	Ada	Augusta Ada Byron	U.S. Defence Dept.	General
1983	C++	Language C	Bell Laboratory	General, object-oriented programming
1991	Java	None	Sun Microsystems	General, internet, object-oriented programming

2.8 INTERACTIVE COMPUTING

A major breakthrough in programming took place in the early 1960s when interactive languages like BASIC were developed. With an interactive language, we can converse (interact) with a computer. Most of the modern languages including FORTRAN have incorporated interactive features. With the help of an interactive language, we may engage in a conversation with our computer like this:

```
I am computing sum of two values
Please input value of X
255.75
Please input value of Y
120.50
Sum of X any Y is 376.25
Do you want me to do one more sum?
No Thanks
Bye then, See you again!
```

The lower-case words are of the computer and the words underlined are ours. Such interactive computing would be useful in determining certain intermediate results and taking actions depending upon the values.

2.9 PROBLEM SOLVING AND ALGORITHMS

Mathematical problems that can be solved through the computer may range in size and complexity. Since the computer does not possess any common sense and cannot make any unplanned decisions, the problem, whether it is simple or complex, has to be broken into a well-defined set of solution steps. It should be remembered that computers do not "solve" problems; rather, they are used to implement the solutions to problems.

In every instance of problem solving, the computer cannot be used to solve the problem until a method of solution has been evolved and a detailed procedure has been prepared by the user. It is assumed that the user has a certain amount of background knowledge, knows certain facts about the problem and possesses sufficient deductive and reasoning skills.

Problem solving involves the following steps:

1. studying the problem in detail
2. redefining or restating the problem
3. identifying output requirements, input data available and conditions and constraints to be used
4. comparing alternative methods of solution
5. selecting the method which is considered to be the best
6. preparing a logical and concise list of procedures or steps necessary for determining the solution
7. computing the results
8. examining the results for correctness

The computer's help may be necessary only in the seventh step. All the remaining steps are to be performed by the user. It is this fact that a beginner finds difficult to appreciate.

The logical and concise list of procedure for solving a problem is called an *algorithm*. It describes the steps that lead to unambiguous results in a finite number of operations. Figure 2.6 illustrates an algorithm for finding the square root of a set of N numbers.

Step 1: Find out the number of values for which square roots are to be evaluated.
Step 2: Take a value.
Step 3: See whether the value is positive or negative. If positive, go to *Step 4*, otherwise go to *Step 6*.
Step 4: Evaluate the square root.
Step 5: Record the value and its square root.
Step 6: Repeat Steps 2 to 5 until all the values are completed.

Fig. 2.6 Algorithm for finding the square root of a given set of values

An algorithm prepared for the first time might need review to:

1. determine the correctness of various steps
2. reduce the number of steps, if necessary
3. increase the speed of solving the problem

An algorithm should also include steps to identify any abnormal data or results and take corrective measures, if possible. In case of large problems, we can break them into parts representing small tasks, prepare several algorithms and later combine them into one large algorithm. This is known as the *modular approach*.

Developing computer programs using the modular approach is known as *modular programming*. A module is a program unit or entity that is responsible for a single task. Modules (known as subprograms) are arranged in a hierarchical structure (similar to an organisation chart) as shown in Fig. 2.7. This is essentially *top-down design* in which bigger modules are broken into smaller ones such that they are small enough to be understood and easily coded using simple logic.

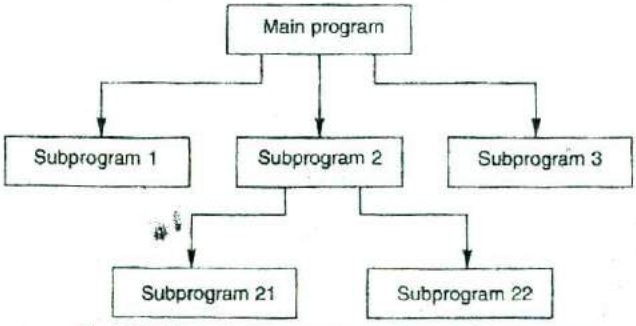


Fig. 2.7 Top-down modular design of a program

2.10 FLOW CHARTING

When organising a problem for computer solution, it is desirable to present the algorithm pictorially. A flow chart is a diagram that outlines the sequences of operations to be performed. The operating steps are placed in boxes that are connected by arrows to indicate the order of execution of steps. Figure 2.8 illustrates the flow chart for the algorithm shown in Fig. 2.6. It is perhaps the best available method for expressing what the computer must do. Some symbols commonly used in flow charts are shown in Fig. 2.9.

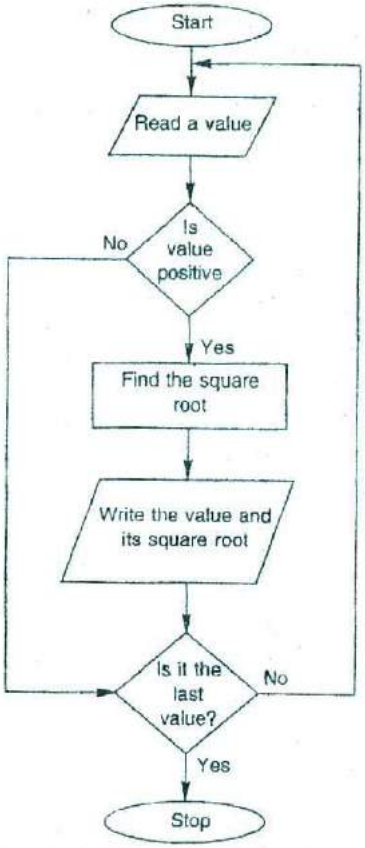


Fig. 2.8 Flow chart for finding the square root of a given set of numbers

The important functions of a flow chart are as follows:

1. It provides a graphic representation of the problem so that it is easier to understand the plan of solution.
2. It provides a convenient aid to writing computer instructions (program).

3. It assists in reviewing and correcting the program.
4. It helps in discussion of the solution logic with others.

While drawing a flowchart, one must remember the following:

1. First list the logical steps.
2. Complete the main path of the logic first and then complete all branches and loops.
3. Use descriptive terms or mathematical equations in the boxes.
4. Each box should represent a step that is meaningful.
5. Use unambiguous terms in the flow chart so that others can easily understand it.

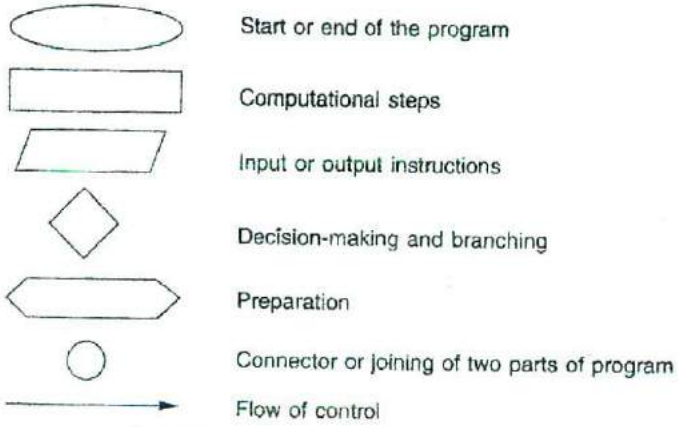


Fig. 2.9 Flow chart symbols

2.11 STRUCTURING THE LOGIC

Solution steps of all problems can be organised into one or combination of the following three control structures:

1. sequence structure
2. branching structure
3. looping structure

Sequence structure is used when the solution does not involve any repetitive operations or options. This is known as *straight-line logic* and is illustrated in Fig. 2.10.

Branching refers to the process of following one of two or more alternate paths of computations. This happens at a point where a test is performed to identify the conditions of certain variables in the process. The basis for selecting a particular path is stated within the decision box. The decision can be based on a comparison, on the value of a variable, on the sign of a variable, etc. The basic flow charts associated with branching are shown in Fig. 2.11.

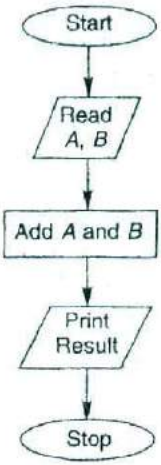


Fig. 2.10 Sequence structure

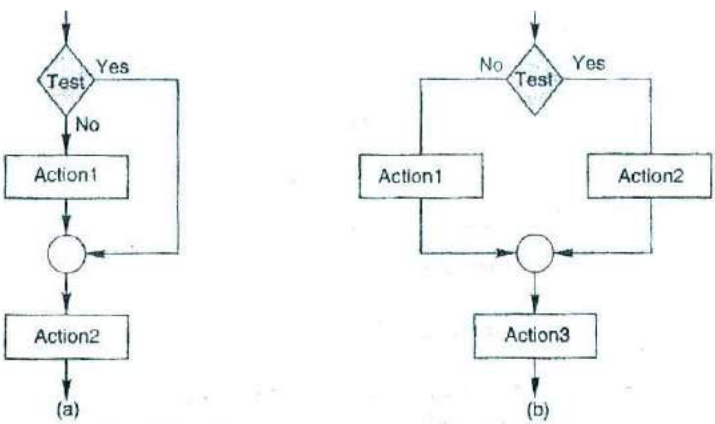
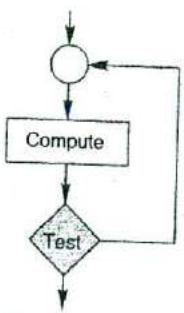


Fig. 2.11 Branching structure (IF THEN ELSE)

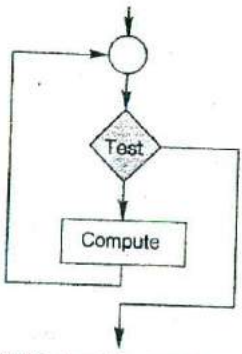
In Fig. 2.11(a), a few steps are bypassed and the program is rejoined at a later stage. This is known as *forward jump*. In Fig. 2.11(b), each branch contains one or more computational steps. The two branches may join up again in the main path or may contain completely different steps and only join up at the end.

Looping refers to the repeated use of one or more steps. There are two types of loops. One is the *fixed loop* where the operations are repeated a fixed number of times. In this case, the values of the variables inside the loop have no effect on the number of times that the looping operation is performed. The other is the *variable loop* where the operations are repeated until a specified condition is met. Here, the number of times that the loop is repeated may vary. Searching for a particular item in a list of items is an example of variable loop.

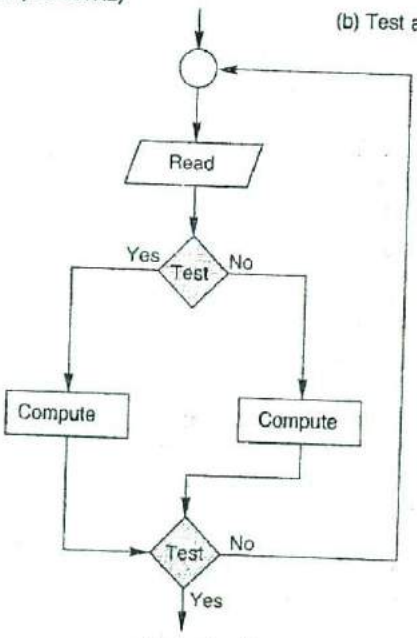
Loops are also referred to as *backward jumps*. These jumps may occur either after meeting a specified condition in the process or after doing a certain computation. These jumps (loops) are illustrated in Fig. 2.12.



(a) Do and Test (DO UNTIL)



(b) Test and Do (DO WHILE)



(c) A mixed loop

Fig. 2.12 Illustration of loops

2.12 USING THE COMPUTER

Computers can be used to solve specific problems that may be scientific or commercial in nature. In either case, there are some basic steps involved in using the computers. These are as follows:

1. *Problem analysis* Identify the known and unknown parameters and state the constraints under which the problem is to be solved. Select a method of solution.
2. *Collecting information* Collect data, information and the documents necessary for solving the problem and also plan the layout of output results.
3. *Preparing the computer logic* Identify the sequence of operations to be performed in the process of solving the problem and plan the program logic, preferably using a program flow chart.
4. *Writing the computer program* Write the program of instructions for the computer in a suitable language.
5. *Testing the program* There may be errors (bugs) in the program. Remove all these errors which may be either in using the language or in the logic.
6. *Preparing the data* Prepare input data in the required form.
7. *Running the program* This may be done either in batch mode or interactive mode. The computations are performed by the computer and the results are given out.

The selection of a particular input/output device depends upon the nature of the problem, type of input data and the form of output required.

2.13 SUMMARY

We have discussed in this chapter the following aspects of computers and computing technology:

- evolution of computing devices
- generations of modern computers
- different types of computers
- input-process-output cycle of computing
- organisation and structure of a computer
- functions of various input, output and storage devices
- need for various types of computer programs
- importance of programming languages and their applications
- steps involved in solving mathematical problems
- use of flow charts for representing problem-solving algorithms
- application of modular and structured programming techniques for implementing computer-based solutions

Key Terms

Abacus
Algorithm
Analog computer
Application programs
Assembler
Assembly language
Low level language
Machine code
Machine language
Mainframe computer
Mark I
Microcomputer

(Contd.)

(Contd.)

Binary code	*Microelectronics*
Branching structure	*Microprocessor*
Compiler	*Minicomputer*
Computer program	*Modelling*
Computer-aided design	*Modular programming*
Computer-aided learning	*Object program*
Computer-aided manufacture	*Operation system*
Computer-managed learning	*Parallel computer*
Data	*PCAT*
Debugging	*PCXT*
Digital computer	*Pentium*
ENIAC	*Personal computer*
Expert systems	*Second generation*
Fifth generation	*Sequence structure*
First generation	*Simulation*
Floppy disk	*Slide rule*
Flow chart	*Software*
Fourth generation	*Source program*
Hard disk	*Straight-line logic*
Hardware	*Structured programming*
High-level language	*Supercomputer*
Hybrid computer	*Third generation*
IC chips	*Top-down design*
Information	*Transistor*
Interactive computing	*Translators*
Interpreter	*UNIVAC*
Knowledge-based systems	*Utility programs*
Language processors	*Vacuum tube*
Large-scale integration (LSI)	*Winchester disk*
Logarithm	*Workstation*
Looping structure	

REVIEW QUESTIONS

1. Describe the abilities of modern computers that are directly relevant to numerical computing.
2. List at least two applications of computers in each of the following areas:
   (a) Industry
   (b) Business
   (c) Education
   (d) Engineering
3. Match the items in the following lists:

(a) China	(i) Punched Cards
(b) John Napier	(ii) Accounting Machine
(c) Blaise Pascal	(iii) Abacus

(d) IBM (iv) Logarithm
(e) Jacquard (v) Mark I

4. Describe briefly the developments in computing technology during the three decades from 1945 to 1975.
5. Describe the technology of fourth generation computers. How are they better than the earlier computer models?
6. What are fifth generation computers? How are they different from fourth generation systems?
7. Distinguish between analog and digital computers.
8. Distinguish between special purpose and general purpose computers.
9. A majority of computers used in the world today are digital. Why?
10. What are personal computers? How are they different from microcomputers?
11. Describe the relevance of supercomputers to engineers and scientists.
12. Define each of the following terms in one sentence:
    (a) Computer Program
    (b) Hardware
    (c) Information
    (d) Data
    (e) Software
13. Describe the functions of the following units in a computer:
    (a) Memory Unit
    (b) Arithmetic Logic Unit
    (c) Storage Unit
14. Describe how an application program is implemented in a computer.
15. Why do we need language processors? Describe the two forms of language processors available.
16. Compare the functions of application programs with that of operating systems.
17. What is machine language? What are its limitations?
18. How is assembly language better that machine language?
19. What are the features of high-level languages?
20. How is a program written in a high-level language implemented on a computer?
21. State the contributions of the following organisations to the development of high-level languages:
    (a) IBM
    (b) Bell Laboratory
    (c) US Defence Department
22. What are the advantages of interactive computing?
23. State the main steps involved in solving a mathematical problem.
24. What is an algorithm? How is it useful for a programmer?
25. What is modular programming? How does it help in solving a problem?
26. Why do we often use flow charts for developing computer programs?

27. Describe the three basic control structures used in executing the solution steps.
28. Critically compare the Do-and-Test and Test-and-Do looping structures.
29. Compare the following:
    (a) Forward jump versus backward jump
    (b) Fixed loop versus variable loop
30. Describe the basic tasks involved in solving a problem using a computer.

CHAPTER 3

Computer Codes and Arithmetic

3.1 INTRODUCTION

Computers store and process numbers, letters and words that are often referred to as *data*. How do we communicate these numbers and words to computers? How do computers store this data and process them? Since computers cannot understand the Arabic numeral or English alphabet, we should use some "codes" that can easily be understood by them.

In all modern computers, storage and processing units are made of a set of silicon chips, each containing a large number of transistors. A transistor is a two-state device that can be put "off" and "on" by passing an electric current through it. Since the transistors are sensitive to currents and act like switches, we can communicate with the computers using electric signals, which are represented as a series of "pulse" and "no-pulse" conditions. For the sake of convenience and ease of use, a pulse is represented by the code "1" and a no-pulse by the code "0". They are called bits, an abbreviation of "binary digits". A series of 1's and 0's are used to represent a number or a character and thus, they provide a way for humans and computers to communicate with one another. This idea was suggested by John Von Neumann in 1946. The numbers represented by binary digits are known as *binary numbers*. Computers not only store numbers but also perform operations on them in binary form. Although information is stored in the computer memory in combinations of 0's and 1's, binary numbers become cumbersome when expressing large numbers. For this reason, internal contents of a computer are not displayed in binary form. Instead, they are displayed as *hexadecimal* or *octal* systems. Number systems that are popularly used

in computing are the decimal system, binary system, hexadecimal system and octal system.

In this chapter, we will discuss the various number systems and their conversion from one system to another. We shall also discuss the internal representation of numbers and their arithmetic operations.

3.2 DECIMAL SYSTEM

The decimal number system, so familiar to us, is the oldest positional number system. In a positional system, a number is represented by a set of symbols. Each symbol represents a particular value depending on its position. The actual number of symbols used in a positional system depends on its *base*.

The decimal system uses a base of 10 and thus it uses 10 symbols, 0 to 9. Any number can be represented by arranging symbols in various positions. In the decimal system, each position represents a specific power of 10. Each successive position to the left of the decimal point represents a value ten times greater than the position to its immediate right as shown below :

Position ⟶	6	5	4	3	2	1	0
Place Value ⟶	10^6	10^5	10^4	10^3	10^2	10^1	10^0

For example, the decimal number 5704 represents:

3	2	1	0	⟵ Position
10^3	10^2	10^1	10^0	⟵ Value
5	7	0	4	⟵ Decimal point

$4 \times 10^0 = 4$

$0 \times 10^1 = 0$

$7 \times 10^2 = 700$

$5 \times 10^3 = 5000$

Sum = 5704

We can express this in general form as

$$d_m (10^m) + d_{m-1} (10^{m-1}) + \ldots + d_0 = \sum_{i=0}^{m} d_i \, 10^i$$

where d_i are the decimal symbols, 0 to 9 and $m - 1$ are the number of symbols. This is called the expanded notation for the integer.

Similarly, a fractional part of a decimal number can be represented as

$$\sum_{i=1}^{n} d_i \, 10^{-i}$$

where n is the number of symbols in the fractional part.

3.3 BINARY SYSTEM

The binary system is the positional number system to the base 2. It uses two symbols 0 and 1. Again, each position in a binary number represents a power of the base as shown below.

Position ⟶	6	5	4	3	2	1	0
Place Value ⟶	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	(64)	(32)	(16)	(8)	(4)	(2)	(1)

Note that each successive position in the integer part of the binary number has a value two times greater than the position to its right.

For example, the binary number 1101 represents the decimal values as shown below:

3	2	1	0	⟵ Position		
2^3	2^2	2^1	2^0	⟵ Value		
1	1	0	1	⟵ Binary point		
				1×1	=	1
				0×2	=	0
				1×4	=	4
				1×8	=	8
				Sum		13

That is, $1101_2 = 13_{10}$

The subscript 2 denotes a number in binary system and 10 denotes a number in the decimal system. In general form, it can be written as

$$d_m (2^m) + d_{m-1} (2^{m-1}) + \ldots + d_0 = \sum_{i=0}^{m} d_i \, 2^i$$

where d_i are the binary symbols, 0 or 1. We can further generalise the notation to any base b as $\sum_{i \le m} d_i \, b^i$

Note that the base b is usually an integer greater than one, and digits d_i are between 0 and $b - 1$. The base is sometimes called *radix* and the fractional point is called *radix point*.

3.4 HEXADECIMAL SYSTEM

The hexadecimal system is a number system that uses 16 as its base. This system requires 16 one digit symbols. The first ten symbols are represented by digits 0 through 9 and the remaining six by the letters A through F. The letter A denotes 10, B denotes 11 and so on. Table 3.1 shows equivalents of decimal, binary, and hexadecimal values.

Table 3.1 Equivalent values of different systems

Decimal System	*Binary System* Weight → 8	4	2	1	*Hexadecimal System*
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	1	0	1	0	A
11	1	0	1	1	B
12	1	1	0	0	C
13	1	1	0	1	D
14	1	1	1	0	E
15	1	1	1	1	F

In the hexadecimal system, each position represents a value 16 times greater than the position to its immediate right. The place values of hexadecimal system are shown below:

Position →	4	3	2	1	0
Place Value →	16^4	16^3	16^2	16^1	16^0

The following example illustrates the decimal value represented by a hexadecimal number.

3	2	1	0	← Position
16^3	16^2	16^1	16^0	← Value
1	2	A	F	← Hexadecimal point
				$15 \times 1 = 15$
				$10 \times 16 = 160$
				$2 \times 256 = 512$
				$1 \times 4096 = 4096$
				Sum 4783

Thus, $12AF_{16} = 4783_{10}$

To convert a binary number to hexadecimal, we need only to group the binary digits in sets of four and convert each group to its equivalent hexadecimal digit. Thus, the binary number 0111 1010 0001 0010 0001 becomes 7A121 in hexadecimal. This is illustrated below:

Binary quadruplets	0111	1010	0001	0010	0001
	↓	↓	↓	↓	↓
Hexadecimal point	7	A	1	2	1

This example clearly illustrates the advantage of hexadecimal system over binary system. For all large binary numbers, the hexadecimal representation is much more compact and, therefore, easier to write and manipulate than its binary equivalent.

3.5 OCTAL SYSTEM

The octal number system is a system having base b as 8. The eight octal symbols are 0 through 7. The place values in the octal system are powers of 8 as shown below:

Position ⟶	4	3	2	1	0
Place Value ⟶	8^4	8^3	8^2	8^1	8^0

The position values increase by a factor of 8 from right to left. The example below shows an octal number and its equivalent decimal value:

3	2	1	0	⟵ Position		
8^3	8^2	8^1	8^0	⟵ Value		
2	0	5	6	⟵ Octal point		
				6×1	=	6
				5×8	=	40
				0×64	=	0
				2×512	=	1024
				Sum		1070

Thus, $2056_8 = 1070_{10}$

Since $8 = 2^3$, each octal digit has a unique 3 bit binary representation. This is shown in Table 3.2.

Table 3.2 Binary representation of octal digits

Octal	*Binary Representation*
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Just as in the hexadecimal system, to convert a binary number to octal, it is only necessary to group the binary digits in sets of three and convert each set to its octal equivalent. For example, the binary number 1011010 can be represented in octal as follows:

Binary triplets	001	011	010
	↓	↓	↓
Octal equivalent	1	3	2

3.6 CONVERSION OF NUMBERS

We discuss here the following systems of conversion:

1. non-decimal to decimal system
2. decimal to non-decimal system
3. octal to hexadecimal system
4. hexadecimal to octal system

Non-decimal System to Decimal System

We can convert a number in base 2, base 8 or base 16 to a decimal number using the expanded notation discussed so far. This conversion can also be accomplished using the following algorithms.

Integral Part

1. Multiply the *leftmost* digit by the base b
2. Add the next digit to the right to the product
3. Multiply the sum by the base b and add the next digit
4. Continue the process until the last (rightmost) digit is *added*

The sum is the decimal equivalent of the given integer number

Fractional Part

1. Multiply the *rightmost* digit by $1/b$
2. Add the next digit to the left to the product
3. Multiply the sum by $1/b$ and add the next digit
4. Continue this process until the last (leftmost) digit in the fractional part is *added*
5. Multiply the last sum by $1/b$

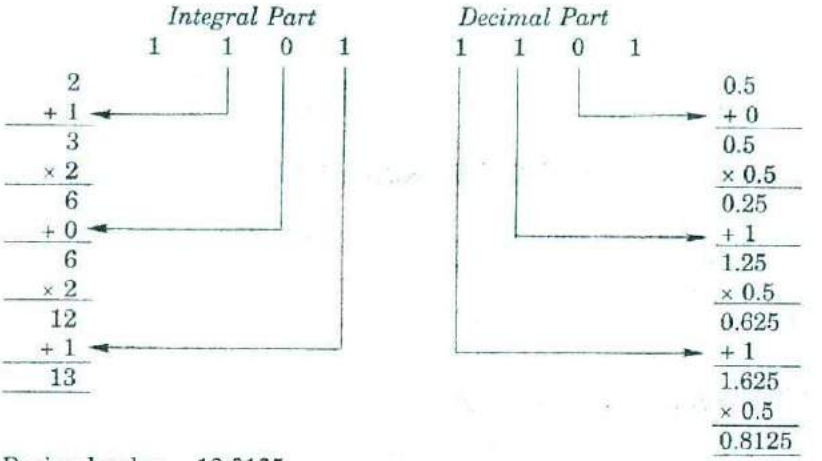
The product is the decimal equivalent of the given fractional number.

Note that, in the integral part algorithm, the process ends when the rightmost digit is added, but, in the case of fractional part algorithm, the process ends when the leftmost digit is added and the final sum is multiplied by $1/b$.

Example 3.1

Convert binary number 1101.1101 to its decimal equivalent.

We can convert the given binary number to the decimal equivalent using the above algorithm as follows:

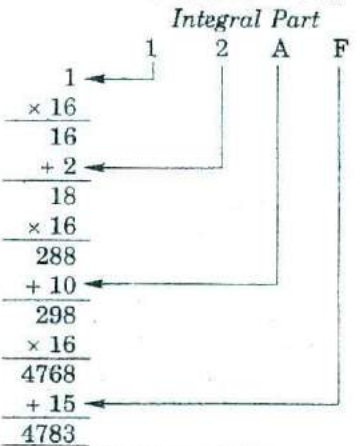


Decimal value = 13.8125

Example 3.2

Convert the hexadecimal number 12AF to a decimal number

Conversion is done as follows:



Thus, $12AF_{16} = 4783_{10}$

Decimal System to Non-decimal System

It is easy to convert a decimal number to a number of any other system. To do this, we must consider the integer and fractional parts separately as we did earlier. Algorithms to accomplish this are given below:

Integral Part

1. Divide the integer part of the decimal number by the base b of the new system. The *remainder* will constitute the rightmost digit of the integer part of the new number.

2. Divide the quotient again by the base b. The remainder is the second digit from right.
3. Continue this process until a *zero quotient* is obtained. The last remainder is the leftmost digit of the new number.

Fractional Part

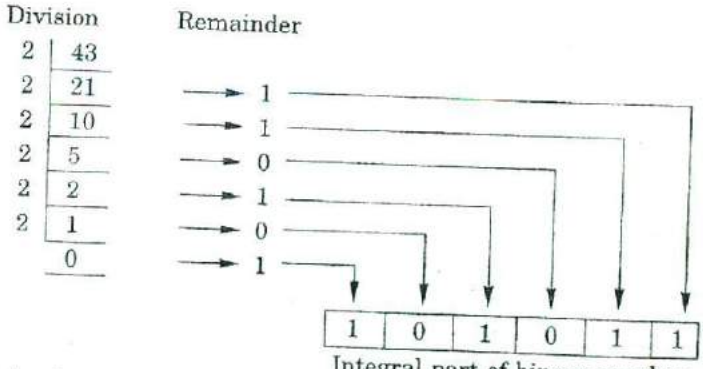
1. Multiply the fractional part of the decimal number by the base b of the new system. The integral part of the product constitutes the leftmost digit of the fractional part of the new number.
2. Multiply the fractional part of the product by the base b. The integral part of the resultant product is the second digit from left.
3. Continue the process until a *zero fractional* part or a *duplicate fractional part* occurs. The integer part of the last product will be the rightmost digit of the fractional part of the new number.

Note that a duplicate fractional part indicates that the sequence will be an infinite one. The particular block of digits will be repeated over and over again.

Example 3.3

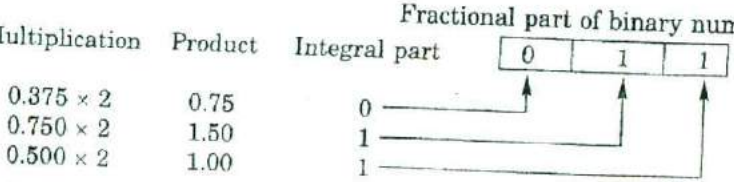
Convert the decimal 43.375 into its binary equivalent.

Integral Part



The digits in the remainder form the binary number when they are *dropped to the right.*

Fractional Part

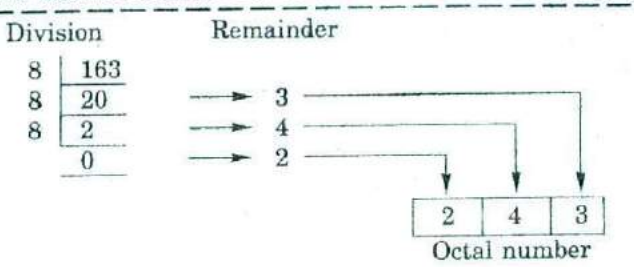


The digits in the integral part form the binary number when they are read from the top down (or *lifted up to the right* as shown).

Thus, $43.375_{10} = 101011.011_2$

Example 3.4

Convert the decimal number 163 to an octal number.



Thus, $163_{10} = 243_8$

Example 3.5

Convert the decimal number 0.65 to its binary equivalent.

Multiplication	Product	Integral part
0.65×2	1.3	1
0.3×2	0.6	0
0.6×2	1.2	1
0.2×2	0.4	0
0.4×2	0.8	0
0.8×2	1.6	1
0.6×2	1.2	1
0.2×2	0.4	0
0.4×2	0.8	0
0.8×2	1.6	1

Thus, $0.65_{10} = 1010011001 \ldots$

Note that a terminating decimal fraction need not have a terminating binary equivalent. This happens when a fractional part is repeated and therefore the process is terminated.

Octal and Hexadecimal Conversion

Using the binary system as an intermediate stage, we can easily convert octal numbers to hexadecimal numbers and vice-versa. The steps are as follows:

Octal to Hexadecimal

1. Write the octal number.
2. Place the binary equivalent of each digit below the number.
3. Regroup them as binary quadruplets from the binary point, with zeros added, if necessary.
4. Convert each group into its equivalent hexadecimal digit.

Hexadecimal to Octal

1. Write the hexadecimal number.
2. Place the binary equivalent of each digit below the number.

3. Regroup them as binary triplets from the binary point, with zeros added if necessary.
4. Convert each group into its equivalent octal digit.

Example 3.6

Convert the octal number 243 to a hexadecimal number.

Octal	2	4	3	← Octal point
	↓	↓	↓	
Binary equivalent	010	100	011	← Binary point
Regrouped as binary quadruplets	0000	1010	0011	
	↓	↓	↓	
Hexadecimal	0	A	3	

Thus, $243_8 = A3_{16}$

Example 3.7

Convert the hexadecimal number 39.B8 to an octal number.

Hexadecimal	3	9	.	B	8		
	↓	↓		↓	↓		
Binary equivalent	0011	1001	.	1011	1000		
Regrouped as binary triplets	000	111	001	.	101	110	000
	↓	↓	↓		↓	↓	↓
Hexadecimal	0	7	1	.	5	6	0

Thus, $39.B8_{16} = 71.56_8$

The grouping of binary digits into triplets or quadruplets plays an important role in the internal organisation of information in computers. They are often used to represent long binary strings with lesser number of symbols.

3.7 REPRESENTATION OF NUMBERS

As mentioned earlier, all modern computers are designed to use binary digits to represent numbers and other information. The memory is usually organised into strings of bits called *words*. Each such string has the same length in a particular computer, although different computers may use different word lengths. For example, IBM PC and AT systems use a word length of 16 bits, while VAX 11 systems use a word length of 32 bits.

The largest number a computer can store depends on its word length. For example, the largest binary number a 16 bit word can hold is 16 bits of 1. This binary number is equivalent to a decimal value of 65535. The

largest decimal number that can be stored in a computer is given by the following relation:

$$\text{Largest number} = 2^n - 1$$

where n is the word length in bits. Thus, we see that the greater the number of bits, the larger the number that may be stored.

Although the computer works well with the binary numbers, humans do not. Firstly, it takes too many bits to represent a number. Secondly, writing such long series of bits can be exhausting and may cause errors. This is why we have other systems such as octal, hexadecimal and decimal systems. Computers read decimal numbers supplied by humans but convert them automatically into binary numbers for internal use. These binary numbers may also be expressed in the octal or hexadecimal form for print-out or display. For output, the numbers are reconverted to decimal form for human use.

Integer Representation

Decimal numbers are first converted into the binary equivalent and then represented in either *integer* or *floating point* form. Let us first consider the integer representation.

For integers, the decimal or binary point is always fixed to the right of the least significant digit and therefore, fractions are not included. As mentioned earlier, the magnitude of the number is restricted to $2^n - 1$, where n is the word length in bits.

How do we represent negative numbers? Negative numbers are stored by using the 2's complement. This is achieved by taking the 1's complement of the binary representation of the positive number and then adding 1 to it.

Example 3.6

Represent –13 in binary form.

$$\begin{aligned} 13 &= 01101 \\ \text{1's complement} &= 10010 \\ &\quad +\ 00001 \\ \text{2's complement} &= 10011 \end{aligned}$$

Thus, –13 = 10011

Note that we have used an extra 0 to the left of the binary number representing 13. This is to indicate that the number is positive.

Then, if the leftmost bit is 1, the number is negative. The leftmost (or the most significant) bit of a binary number which is used to indicate the sign is called the *sign bit*.

Now we see that if we reserve one bit to represent the sign of the number, we have only $n - 1$ bits to represent the number. Thus, a 16 bit word can contain numbers -2^{15} to $2^{15} - 1$ (i.e. –32768 to 32767).

Example 3.9

Show that the number –32768 is represented in a 16 bit word as follows:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

–32768	=	(–32767) + (–1)	
32767	=	0111 1111 1111 1111	
1's complement	=	1000 0000 0000 0000	
	=	+ 0000 0000 0000 0001	
–32767	=	1000 0000 0000 0001	← (a)
1	=	0000 0000 0001 0001	
1's complement	=	1111 1111 1111 1110	
	=	+ 0000 0000 0000 0001	
–1	=	1111 1111 1111 1111	← (b)
–32768	=	1000 0000 0000 0000	← (a) + (b)

Floating Point Representation

We have just seen how integer numbers are represented. We have also seen that a 16 bit computer cannot store a positive number larger than 32767. What if we want to handle a fractional number like 35.7812 or a large number like 987654321? Such numbers are stored and processed in what is known as exponential form. These numbers have an embedded decimal point and are called *floating point numbers* or *real numbers*. For example, 35.7812 can be expressed 0.357812×10^2. Similarly, the number 987654321 can be expressed as 0.987654×10^9. By writing a large number in exponential form, we lose some digits. If x is a real number, its floating point form representation is

$$x = f \times 10^E$$

The number f is called *mantissa* and E is the *exponent*.

Floating point numbers are stored differently. The entire memory location is divided into three fields or parts as shown in Fig. 3.1. The first part (1 bit) is reserved for the sign, the second part (7 bits) for the exponent of the number, and the third (24 bits) for the mantissa of the number. Typically, floating numbers use a field width of 32 bits where 24 bits are used for the mantissa and 7 bits for the exponent.

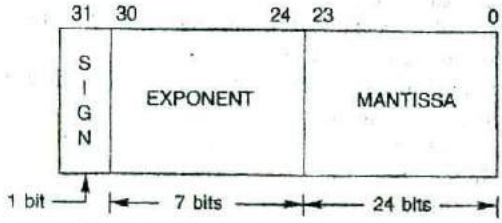


Fig. 3.1 Floating point representation

A-20696

Thus, we can represent very small fractions or very large numbers within the computer using the floating point representation.

Example 3.10

Convert the following numbers to floating point notation.

0.00596, 65.7452, −486.8

0.00596	is expressed as	0.596×10^{-2}
65.7452	is expressed as	0.657452×10^{2}
−486.8	is expressed as	-0.4868×10^{3}

The shifting of the decimal point to the left of the most significant digit is called *normalisation* and the numbers represented in normalised form are known as *normalised floating point numbers*. You may note that the mantissa should satisfy the following conditions.

For positive numbers: less than 1.0 but greater than or equal to 0.1
For negative numbers: greater than −1.0 but less than or equal to −0.1

That is, $0.1 \leq |f| < 1$

The normalised floating point numbers are written using the following notation:

0.596×10^{-2} written as .596 E − 2
-0.4868×10^{3} written as −.4868 E3

3.8 COMPUTER ARITHMETIC

Different systems of computer arithmetic are currently available. They include integer arithmetic, fixed point arithmetic, floating point arithmetic, interval arithmetic and karlsruhe accurate arithmetic.

They are either supported by hardware or software or some software/hardware combinations. Each system uses its own scheme for representing numbers in binary form within the machine. The most common and popular arithmetic systems are integer arithmetic and floating point arithmetic. These systems are discussed briefly in this section.

Integer Arithmetic

Virtually all computers offer integer arithmetic. The main property of integer arithmetic is that the result of any arithmetic operation with integers is an integer. The other property is that the result is always exact with the following two exceptions:

1. The range of integers that can be represented is not infinite but is bounded above and below.
2. The result of an integer division is usually given as a quotient and a remainder (since fractions cannot be represented in the integer scheme) which is truncated.

Example 3.11

Illustrate integer arithmetic

Addition:	25 + 12 = 37
Subtraction:	25 − 12 = 13
	12 − 25 = −13
Multiplication:	25 × 12 = 300
Division:	25 ÷ 12 = 2
	12 ÷ 25 = 0

Note that as only a finite range of integers can be represented, the product of two numbers may exceed the range.

Example 3.12

Show that the following rule does not generally hold good in integer arithmetic.

$$\frac{a+b}{c} = \frac{a}{c} + \frac{b}{c}$$

Let $a = 5$, $b = 7$ and $c = 3$

Then, $$\frac{a+b}{c} = \frac{5+7}{3} = \frac{12}{3} = 4$$

$$\frac{a}{c} + \frac{b}{c} = \frac{5}{3} + \frac{7}{3} = 1 + 2 = 3$$

The results are not identical. This is because the remainder of an integer division is always truncated.

Floating Point Arithmetic

Although integer arithmetic is adequate in many computing applications, it does not meet the requirements of many numerical computing methods as they often involve manipulation of fractional numbers. Hence, floating point arithmetic is the preferred choice for a majority of numerical computing applications.

In the floating point system, all the numbers are stored and processed in normalised exponential form. The most difficult operations in the floating point arithmetic are addition and subtraction.

Addition Let the two numbers to be added be x and y, and let z be the result. Let the fractional parts and exponents be f_x, f_y and f_z, and E_x, E_y and E_z, respectively. Then, the addition algorithm is as follows:

1. Set E_z = the larger of E_x and E_y. (Assume here $E_x >= E_y$. Then $E_z = E_x$).
2. *Shift Right.* Shift f_y to the right by $E_x - E_y$ places. (This makes the exponent of f_x and f_y the same).

3. *Add.* Set $f_z = f_x + f_y$
4. *Normalise.* If the absolute value of f_z is greater than one, shift the decimal point of f_z to the left of the most significant digit and then increase E_z by one.

Then $z = f_z \times 10^{E_z}$

In all the manipulations, the result of any operation is normalised and the mantissa is rounded or truncated to p digits, where p is the precision of the computer used. In the examples discussed here, we assume a precision of 6 and the mantissa is truncated to 6 digits.

Example 3.13

Add the numbers 0.964572 E2 and 0.586351 E5.

Let x = 0.586351 and y = 0.964572 E2

$E_z = 5$

$f_y = 0.000964$

$f_z = 0.000964 + 0.586351 = 0.587315$

Then, z = 0.587315 E5

Note that both the mantissa and exponent of the number with the smaller exponent are modified and the modified mantissa is truncated to six digits.

Example 3.14

Add the numbers 0.735816 E4 and 0.635742 E4.

$E_z = 4$

$f_z = 0.735816 + 0.635742 = 1.371558$

$z = 1.371558$ E4 = 0.137155 E5

Note that the mantissa of the result is truncated.

Subtraction Subtraction is nothing but addition of numbers with different signs. However, the subtraction of mantissas may result in a number less than 0.1. In such cases, the decimal point should be shifted to the left of the most significant digit and the exponent of the result should then be decreased accordingly.

Example 3.15

Subtract 0.994576 E–3 from 0.999658 E–3.

Let x = 0.999658 E–3 and y = 0.994576 E–3

$E_z = -3$

$f_z = 0.999658 - 0.994576$

$= 0.005082$

$z = x - y$

$= 0.005082$ E – 3

$= 0.508200$ E – 5 (normalised)

Multiplication Multiplication of two floating point numbers is relatively simple.

1. Multiply the fractional parts: $f_z = f_x \times f_y$
2. Add the exponents: $E_z = E_x + E_y$
3. Then, $z = f_z \times 10^{E_z}$
4. Normalise, if necessary.

Example 3.16

Multiply the numbers 0.200000 E4 and 0.400000 E – 2

$$f_z = 0.200000 \times 0.400000$$
$$= 0.080000$$
$$E_z = 4 - 2 = 2$$
$$z = 0.080000 \text{ E2} = 0.800000 \text{ E1 (normalised)} = 0.800000$$

Division Division is done as follows:

1. Divide the fractional parts: $f_z = f_x / f_y$
2. Subtract the exponents: $E_z = E_x - E_y$
3. $z = f_z \times 10^{E_z}$
4. Normalise, if necessary.

Example 3.17

Divide the number 0.876543 E – 5 by 0.200000 E – 3

$$f_z = 0.876543 \div 0.200000$$
$$= 4.382715$$
$$E_z = -5 - (-3) = -2$$
$$z = 4.382715 \text{ E} - 2$$
$$= 0.438271 \text{ E} - 1 \text{ (normalised)}$$

Note that the mantissa of the result is truncated.

3.9 ERRORS IN ARITHMETIC

In integer arithmetic, while all arithmetic operations are exact, we might come across the following two situations:

1. An operation may result in a large number that is beyond the range of the numbers that the computer can handle.
2. An integer division may result in truncation of the remainder.

When the result is larger than the maximum limit, it is referred to as an *overflow* and when it is less than the lower limit, it is referred to as *underflow*. Unfortunately, most computers do not issue any warnings or messages on integer overflow or underflow. Therefore, we should use integer arithmetic with utmost care.

The floating point arithmetic system is prone to the following errors:

1. Error due to inexact representation of a decimal number in binary form. For example, consider the decimal number 0.1. The binary

equivalent of this number is 0.0001100110011.... The binary equivalent has a repeating fraction and therefore must be terminated at some point.

2. Error due to rounding method used by the computer, in order to limit the number of significant digits. This was illustrated in the examples discussed in Section 3.8. In fact, if the numbers added are too different in magnitude, the smaller may be treated as if it were zero (see Example 3.18).
3. Floating point subtraction may induce a special phenomenon. It is possible that some mantissa positions in the result are unspecified. This happens when two nearly equal numbers are subtracted. This is known as *subtractive cancellation*. If the operands themselves represent approximate values, the loss of significance is serious since it greatly reduces the number of significant digits. The error can be arbitrarily large (see Example 3.21).
4. Overflow or underflow can occur in floating point operations when the result is outside the limits of floating point number system of the computer.

The following examples illustrate these errors

Example 3.18

Add the numbers 0.500000 E1 and 0.100000 E – 7.

Let

$$x = 0.500000 \text{ E1 and } y = 0.100000 \text{ E} - 7$$
$$E_z = 1$$
$$f_y = 0.000000001$$
$$f_z = 0.500000001 = 0.500000$$
$$z = 0.500000 \text{ E1}$$

Note that the value of z is the same as that of x.

Example 3.19

Multiply the number 0.350000 E40 by 0.500000 E70

$$E_z = 110$$
$$f_z = 0.175000$$
$$z = 0.175000 \text{ E110}$$

If we assume that the exponent can have a maximum value of 99, then the result *overflows*.

Example 3.20

Divide the number 0.875000 E – 18 by 0.200000 E95.

$$E_z = -18 - 95 = -113$$

$$f_z = 0.875000 + 0.200000 = 4.375000$$
$$z = 4.375000\ \text{E} - 113$$
$$= 0.437500\ \text{E} - 114$$

If we assume that the exponent can have a minimum value of – 99, then the result *underflows*.

Example 3.21

Subtract 0.499998 from 0.500000.

$$f_x = 0.500000$$
$$f_y = \underline{0.499998}$$
$$f_x - f_y = \underline{0.000002}$$
$$E_z = 0$$

Thus, $z = 0.000002 \times 10^0 = 0.200000 \times 10^{-5}$

The result contains only one significant digit. If the values of x and y are not exact, then the result may not reflect the true difference between them. In many systems, these unspecified digits are filled by arbitrary digits thus causing a further increase in the error.

3.10 LAWS OF ARITHMETIC

Due to errors introduced in floating point arithmetic, the associative and distributive laws of arithmetic are not always satisfied. That is,

$$x + (y + z) \neq (x + y) + z$$
$$x \times (y \times z) \neq (x \times y) \times z$$
$$x \times (y + z) \neq (x \times y) + (x \times z)$$

Although failure of these laws to be satisfied affects relatively few computations, it can be very critical on some occasions. The examples that follow illustrate the discrepancies.

Example 3.22

Associative law for addition

Let $x = 0.456732 \times 10^{-2}$, $y = 0.243451$, $z = -0.248000$

$$(x + y) = 0.004567 + 0.243451 = 0.248018$$
$$(x + y) + z = 0.248018 - 0.248000 = 0.000018$$
$$= 0.180000 \times 10^{-4}$$
$$(y + z) = 0.243451 - 0.248000 = -0.004549 = -0.4549 \times 10^{-2}$$
$$x + (y + z) = (0.456732 - 0.454900)\ 10^{-2}$$
$$= 0.183200 \times 10^{-2}$$

Thus, $(x + y) + z \neq x + (y + z)$

Example 3.23

Associative law for multiplication

Let $x = 0.400000 \times 10^{+40}$, $y = 0.500000 \times 10^{+70}$, $z = 0.300000 \times 10^{-30}$

$$(x \times y) \times z = (0.200000 \times 10^{+110})\ (0.300000 \times 10^{-30})$$

Note that $(x \times y)$ causes overflow and so the result will be erroneous.

$$x \times (y \times z) = (0.400000 \times 10^{40}) \times (0.1500000 \times 10^{40})$$
$$= 0.060000 \times 10^{80} = 0.600000 \times 10^{79}$$

This gives the correct result assuming that the exponent can take a value up to + 99.

Example 3.24

Distributive law

Let $x = 0.400000 \times 10^{1}$, $y = 0.200001 \times 10^{0}$, $z = 0.200000 \times 10^{0}$

$$x \times (y - z) = (0.400000 \times 10^{1}) \times (0.100000 \times 10^{-5})$$
$$= 0.400000 \times 10^{-5}$$
$$(x \times y) - (x \times z) = 0.800000 \times 10^{0} - 0.800000 \times 10^{0} = 0$$

3.11 SUMMARY

In this chapter, we have discussed a very important aspect of numerical computing, namely, the internal representation of numbers in a computer. We considered the following in detail:

- number systems that are popularly used in computing
- conversion of numbers from one system to another
- storage of numbers in the memory of a computer
- different systems of arithmetic operations that are commonly used in numerical computing
- errors introduced by arithmetic operations
- associative and distributive laws of arithmetic

Key Terms

Associative law
Base
Binary digits
Binary numbers
Bits
Computer memory
Data
Decimal numbers
Distributive law
Interval arithmetic
Mantissa
Normalisation
Normalised numbers
Octal numbers
Overflow
Processing
Quadruplets
Radix point

(Contd.)

(Contd.)

Exponent	*Real numbers*
Fixed point arithmetic	*Sign bit*
Floating point form	*Storage*
Floating point arithmetic	*Subtractive cancellation*
Hexadecimal numbers	*Transistor*
Integer arithmetic	*Triplets*
Integer form	*Underflow*

REVIEW QUESTIONS

1. How many binary digits are there? Which symbols are used for them? What are they usually called?
2. Binary digits are used to store and manipulate data in computers. Why? Why do then we use other number systems?
3. What is the complement of a number? Obtain the complement of the decimal number 5749?
4. How do we obtain one's complement and two's complement of a binary number?
5. What are the uses of complements of binary numbers?
6. What is sign bit? How does the computer store a negative number?
7. An 8-bit register stores numbers in two's complement form.
   (a) What is the largest positive decimal number that can be stored?
   (b) What is the smallest negative decimal number that can be stored?
8. Why do we need to represent numbers in exponential form? Explain how a decimal number is represented inside the computer using the exponential form?
9. Explain the following:
   (a) Overflow
   (b) Underflow
10. Discuss the errors that may occur during the floating point arithmetic operations.
11. The hexadecimal equivalent of the binary number 10011101 is
    (a) 5A (b) FF (c) 9D (d) 9E
12. The decimal equivalent of the binary number 10011101 is
    (a) 27 (b) 157 (c) 13 (d) 144
13. The binary equivalent of the octal number 42 is
    (a) 101110 (b) 111010 (c) 100010 (d) 101011
14. The octal equivalent of the hexadecimal number CD5 is
    (a) 3625 (b) 6325 (c) 3652 (d) 6352
15. The hexadecimal equivalent of the decimal number 163 is
    (a) B3 (b) A2 (c) A3 (d) 93

REVIEW EXERCISES

1. Convert the decimal numbers (i) 29, (ii) 123, and (iii) 432 to
   (a) binary system
   (b) octal system
   (c) hexadecimal system
2. Convert the octal numbers (i) 25, (ii) 52, and (iii) 563 to
   (a) decimal system
   (b) binary system
   (c) hexadecimal system
3. Convert the hexadecimal numbers (i) 8F, (ii) BC4, and (iii) AF3D to
   (a) decimal system
   (b) octal system
   (c) binary system
4. Convert the binary numbers (i) 0101, (ii) 0111.0111, and (iii) 1011.11 to
   (a) decimal system
   (b) octal system
   (c) hexadecimal system
5. Assuming that the computer stores each number in a 16-bit memory location, find the internal representations of the following numbers:
   (a) 498
   (b) –498
6. Write the following numbers in normalised exponential form and E-form.
   (a) 12.34
   (b) –654.321
   (c) 0.001234
   (d) –0.009876
   (e) 0.0
   (f) 12345
7. Assuming that the mantissas are truncated to 4 decimal digits, show how the computer performs the following floating point operations:
   (a) $0.5678 \times 10^4 + 0.6666 \times 10^4$
   (b) $0.1234 \times 10^4 + 0.4455 \times 10^{-2}$
   (c) $0.3366 \times 10^{-2} - 0.2244 \times 10^{-1}$
   (d) $0.6789 \times 10^2 \times 0.2233 \times 10^{-1}$
   (e) $0.6789 \times 10^2 + 0.2233 \times 10^{-1}$
8. Assuming that the mantissas are truncated to 4 decimal digits, compute the error in the following computations:
   (a) 5.6789 – 1.2345
   (b) 5.6789 + 9.2345
9. Illustrate with examples the concept of overflow and underflow.
10. Discuss an example to show that the distributive law of arithmetic is not always satisfied in numerical computing.

CHAPTER 4

Approximations and Errors in Computing

4.1 INTRODUCTION

Approximations and errors are an integral part of human life. They are everywhere and unavoidable. This is more so in the life of a computational scientist.

We cannot use numerical methods and ignore the existence of errors. Errors come in a variety of forms and sizes; some are avoidable, some are not. For example, data conversion and roundoff errors cannot be avoided, but a human error can be eliminated. Although certain errors cannot be eliminated completely, we must at least know the bounds of these errors to make use of our final solution. It is therefore essential to know how errors arise, how they grow during the numerical process, and how they affect the accuracy of a solution.

By careful analysis and proper design and implementation of algorithms, we can restrict their effect quite significantly.

As mentioned earlier, a number of different types of errors arise during the process of numerical computing. All these errors contribute to the total error in the final result. A taxonomy of errors encountered in a numerical process is given in Fig. 4.1 which shows that every stage of the numerical computing cycle contributes to the total error.

Although perfection is what we strive for, it is rarely achieved in practice due to a variety of factors. But that must not deter our attempts to achieve near perfection. Again the question is: How much near?

In this chapter we discuss the various forms of approximations and errors, their sources, how they propagate during the numerical process, and how they affect the result as well as the solution process.

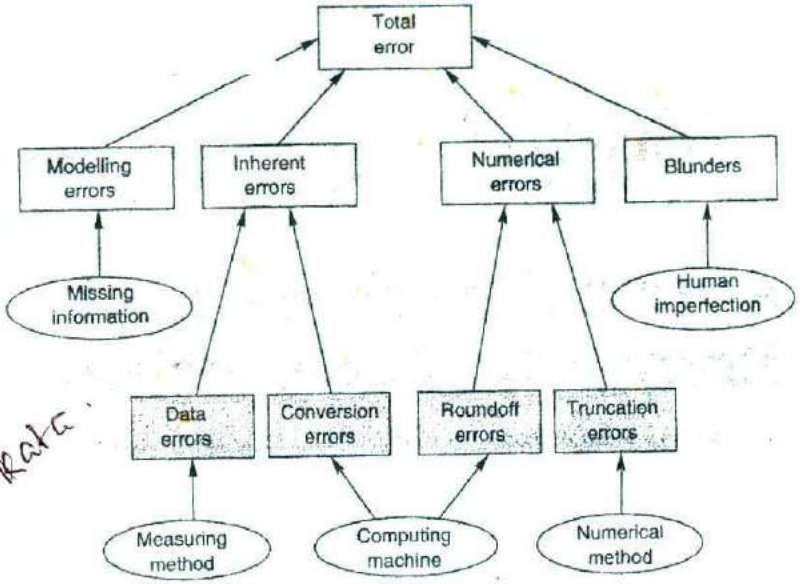


Fig. 4.1 Taxonomy of errors

4.2 SIGNIFICANT DIGITS

We know that all computers operate with a fixed length of numbers. In particular, we have seen that the floating point representation requires the mantissa to be of a specified number of digits. Some numbers cannot be represented exactly in a given number of decimal digits. For example, the quantity π is equal to

$$3.14159265358979323846...$$

Such numbers can never be represented accurately. We may write it as 3.14, 3.14159, or 3.141592653. In all cases we have omitted some digits.

Note that transcendental and irrational numbers do not have a terminating representation. Some rational numbers also have a repeating decimal pattern. For instance, the rational number $2/7 = 0.285714285714....$ Suppose we write 2/7 as 0.285714 and π as 3.14159. Then we say the numbers contain six *significant digits*.

The concept of significant digits has been introduced primarily to indicate the accuracy of a numerical value. For example, if, in the number $y = 23.40657$, only the digits 23406 are correct, then we may say that y has five significant digits and is correct to only three decimal places.

In general, when a number is said to be "good to four digits", it means that the number has four significant digits. The omission of certain digits from a number results in what is called *roundoff error*. The following statements describe the notion of significant digits.

1. All non-zero digits are significant.
2. All zeros occurring between non-zero digits are significant digits.
3. Trailing zeros following a decimal point are significant. For example, 3.50, 65.0 and 0.230 have three significant digits each.
4. Zeros between the decimal point and preceding a non-zero digit are not significant. For example, the following numbers have four significant digits.

$$0.0001234 \quad (1234 \times 10^{-7})$$
$$0.001234 \quad (1234 \times 10^{-6})$$
$$0.01234 \quad (1234 \times 10^{-5})$$

5. When the decimal point is not written, trailing zeros are not considered to be significant. For example, 4500 may be written as 45×10^2 and contains only two significant digits. However, 4500.0 contains four significant digits. Further examples are:

$$7.56 \times 10^4 \text{ has three significant digits.}$$
$$7.560 \times 10^4 \text{ has four significant digits.}$$
$$7.5600 \times 10^4 \text{ has five significant digits.}$$

Integer numbers with trailing zeros may be written in scientific notation to specify the significant digits.

The concept of *accuracy* and *precision* are closely related to significant digits. They are related as follows:

1. Accuracy refers to the number of significant digits in a value. For example, the number 57.396 is accurate to five significant digits (sd).
2. Precision refers to the number of decimal positions, i.e. the order of magnitude of the last digit in a value. The number 57.396 has a precision of 0.001 or 10^{-3}.

Example 4.1

Which of the following numbers has the greatest precision
(a) 4.3201 (b) 4.32 (c) 4.320106

(a) 4.3201 has a precision of 10^{-4}
(b) 4.32 has a precision of 10^{-2}
(c) 4.320106 has a precision of 10^{-6}

The last number has the greatest precision

Example 4.2

What is the accuracy of the following numbers?
(a) 95.763 (b) 0.008472 (c) 0.0456000 (d) 36 (e) 3600 (f) 3600.00

(a) This has five sd.
(b) This has four sd. The leading or higher order zeros are only place holders.
(c) This has six sd.

(d) This has two sd.
(e) Accuracy is not specified.
(f) This has six sd. Note that the zeros were made significant by writing .00 after 3600.

4.3 INHERENT ERRORS

Inherent errors are those that are present in the data supplied to the model. Inherent errors (also known as *input errors*) contain two components, namely, *data errors* and *conversion errors*.

Data Errors

Data error (also known as *empirical error*) arises when data for a problem are obtained by some experimental means and are, therefore, of limited accuracy and precision. This may be due to some limitations in instrumentation and reading, and therefore may be unavoidable. A physical measurement, such as a distance, a voltage, or a time period, cannot be exact. It is, therefore, important to remember that there is no use in performing arithmetic operations to, say, four decimal places when the original data themselves are only correct to two decimal places. For instance, the scale reading in a weighing machine may be accurate to only one decimal place.

Conversion Errors

Conversion errors (also known as *representation errors*) arise due to the limitations of the computer to store the data exactly. We know that the floating point representation retains only a specified number of digits. The digits that are not retained constitute the roundoff error.

As we have already seen, many numbers cannot be represented exactly in a given number of decimal digits. In some cases a decimal number cannot be represented exactly in binary form. For example, the decimal number 0.1 has a non-terminating binary form like 0.00011001100110011.... but the computer retains only a specified number of bits. Thus, if we add 10 such numbers in a computer, the result will not be exactly 1.0 because of roundoff error during the conversion of 0.1 to binary form.

Example 4.3

Represent the decimal numbers 0.1 and 0.4 in binary form with an accuracy of 8 binary digits. Add them and then convert the result back to the decimal form

$0.1_{10} = 0.0001\ 1001$
$0.4_{10} = 0.0110\ 0110$
Sum = 0.0111 1111

$$= 0.25 + 0.125 + 0.0625 + 0.03125 + 0.015625 + 0.0078125 + 0.00390625$$
$$= 0.49609375$$

Note that the answer should be 0.5, but it is not. This is due to the error in conversion from decimal to binary form. Remember, both the numbers have non-terminating binary representation.

Error is equal to $2^{-8} = 0.00390625$. It is clear that the error can be reduced by increasing the binary digits that represent the number. For example, if we use 16 bits, then the error will be equal to $2^{-16} = 0.15258789 \times 10^{-4}$.

4.4 NUMERICAL ERRORS

Numerical errors (also known as *procedural errors*) are introduced during the process of implementation of a numerical method. They come in two forms, *roundoff errors* and *truncation errors*. The total numerical error is the summation of these two errors. The total error can be reduced by devising suitable techniques for implementing the solution. We shall see in this section the magnitude of these errors.

Roundoff Errors

Roundoff errors occur when a fixed number of digits are used to represent exact numbers. Since the numbers are stored at every stage of computation, roundoff error is introduced at the end of every arithmetic operation. Consequently, eventhough an individual roundoff error could be very small, the cumulative effect of a series of computations can be very significant.

Rounding a number can be done in two ways. One is known as *chopping* and the other is known as *symmetric rounding*. Some systems use the chopping method while others use symmetric rounding.

Chopping

In chopping, the extra digits are dropped. This is called *truncating* the number. Suppose we are using a computer with a fixed word length of four digits. Then a number like 42.7893 will be stored as 42.78, and the digits 93 will be dropped. We can express the number 42.7893 in floating point form as

$$x = 0.427893 \times 10^2$$
$$= (0.4278 + 0.000093) \times 10^2$$
$$= [0.4278 + (0.93 \times 10^{-4})] \times 10^2$$

This can be expressed in general form as

$$\text{True } x = (f_x + g_x \times 10^{-d})10^E$$
$$= f_x \times 10^E + g_x \times 10^{E-d}$$
$$= \text{approximate } x + \text{error}.$$

where f_x is the mantissa, d is the length of the mantissa permitted and E is the exponent. In chopping, g_x is ignored entirely and therefore,

$$\text{Error} = g_x \times 10^{E-d},\ 0 \le g_x < 1$$

The absolute error introduced depends on the following:

1. the size of the digits dropped
2. number of digits in mantissa
3. the size of the number

Since the maximum value of g_x is less than 1.0,

$$\boxed{\text{Absolute error} \le 10^{E-d}}$$

Symmetric Roundoff

In the symmetric roundoff method, the last retained significant digit is "rounded up" by 1 if the first discarded digit is larger or equal to 5; otherwise, the last retained digit is unchanged. For example, the number 42.7893 would become 42.79 and the number 76.5432 would become 76.54.

As before, the value of unrounded number can be expressed as

$$\text{True } x = f_x \times 10^E + g_x \times 10^{E-d}$$

When $g_x < 0.5$, entire g_x is truncated and therefore,

$$\text{Approximate } x = f_x \times 10^E$$

and

$$\text{Error} = g_x \times 10^{E-d}, \qquad g_x < 0.5$$

When $g_x \ge 0.5$, the last digit in the mantissa is increased by 1 and therefore

$$\text{Approximate } x = (f_x + 10^{-d}) \times 10^E = f_x \times 10^E + 10^{E-d}$$

$$\text{Error} = [f_x \times 10^E + g_x \times 10^{E-d}] - [f_x \times 10^E + 10^{E-d}]$$

$$= (g_x - 1) \times 10^{E-d} \qquad g_x \ge 0.5$$

In either case, 10^{E-d} is multiplied by factor whose absolute value is no greater than 0.5. Therefore, the value of the absolute error is

$$\boxed{\text{Absolute error} \le 0.5 \times 10^{E-d}}$$

Note that the symmetric rounding error is, at worst, one-half the chopping error.

Sometimes a slightly more refined rule is used when the g_x is exactly equal to 0.5. Here f_x is unchanged if its last digit is even and is increased by 1 if its last digit is odd.

Example 4.4

Find the roundoff error in storing the number 752.6835 using a four digit mantissa.

$$\text{True } x = 0.7526 \times 10^3 + 0.835 \times 10^{-1}$$

Chopping method

Approximate $x = 0.7526 \times 10^3$

Error $= 0.0835$

Symmetric rounding

Error $= (g_x - 1) \times 10^{-1}$

$= -0.165 \times 10^{-1} = -0.0165$

Approximate $x = 0.7527 \times 10^3$

Truncation Errors

Truncation errors arise from using an approximation in place of an exact mathematical procedure. Typically, it is the error resulting from the truncation of the numerical process. We often use some finite number of terms to estimate the sum of an infinite series. For example,

$$S = \sum_{i=0}^{\infty} a_i x^i \text{ is replaced by the finite sum } \sum_{i=0}^{n} a_i x^i$$

The series has been truncated.

Another example is the use of a number of discrete steps in the solution of a differential equation. The error introduced by such discrete approximations is also called *discretisation error*. Consider the following infinite series:

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

When we calculate the sine of an angle using this series, we cannot use all the terms in the series for computation. We usually terminate the process after a certain term is calculated. The terms "truncated" introduce an error which is called *truncation error*.

Many of the iterative procedures used in numerical computing are infinite and, therefore, a knowledge of this error is important. Truncation error can be reduced by using a better numerical model which usually increases the number of arithmetic operations. For example, in numerical integration, the truncation error can be reduced by increasing the number of points at which the function is integrated. But care should be exercised to see that the roundoff error which is bound to increase due to increase in arithmetic operations does not off-set the reduction in truncation error.

We often use library functions to compute logarithms, exponentials, trigonometric functions, hyperbolic functions, and so on. In all these cases, a series is used to evaluate these functions. It is important to know the truncation errors introduced by these library functions. Truncation errors are discussed in detail in many places in this book.

Example 4.5

Find the truncation error in the result of the following function for $x = 1/5$ when we use (a) first three terms, (b) first four terms, and (c) first five terms.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!}$$

(a) Truncation error when first three terms are added

$$\text{Truncation error} = + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!}$$

$$= + \frac{0.2^3}{6} + \frac{0.2^4}{24} + \frac{0.2^5}{120} + \frac{0.2^6}{720}$$

$$= 0.1402755 \times 10^{-2}$$

(b) Truncation error when first four terms are added

$$\text{Truncation error} = 0.694222 \times 10^{-4}$$

(c) Truncation error when first five terms are added

$$\text{Truncation error} = 0.275555 \times 10^{-5}$$

Example 4.6

Repeat the above example for $x = -1/5$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!}$$

$$e^{-0.2} = 1 - 0.2 + \frac{0.2^2}{2} - \frac{0.2^3}{6} + \frac{0.2^4}{24} - \frac{0.2^5}{120} + \frac{0.2^6}{720}$$

(a) Truncation error (three terms) $= -0.1279255 \times 10^{-2}$

(b) Truncation error (four terms) $= +0.6665556 \times 10^{-4}$

(c) Truncation error (five terms) $= -0.257777 \times 10^{-5}$

Note that

$$|\text{T.E.}_3| < \frac{x^3}{3!}$$

$$|\text{T.E.}_4| < \frac{x^4}{4!}$$

$$|\text{T.E.}_5| < \frac{x^5}{5!}$$

4.5 MODELLING ERRORS

Mathematical models are the basis for numerical solutions. They are formulated to represent physical processes using certain parameters involved in the situations. In many situations, it is impractical or impossible to include all of the real problem and, therefore, certain simplifying assumptions are made. For example, while developing a model for calculating the force acting on a falling body, we may not be able to estimate the air resistance coefficient (*drag coefficient*) properly or determine the direction and magnitude of wind force acting on the body, and so on. To simplify the model, we may assume that the force due to air resistance is linearly proportional to the velocity of the falling body or we may assume that there is no wind force acting on the body. All such simplifications certainly result in errors in the output from such models.

Since a model is a basic input to the numerical process, no numerical method will provide adequate results if the model is erroneously conceived and formulated. It is obvious that we can reduce these type of errors by refining or enlarging the models by incorporating more features. But the enhancement may make the model more difficult to solve or may take more time to implement the solution process. It is also not always true that an enhanced model will provide better results. We must note that modelling, data quality and computation go hand in hand. An overly refined model with inaccurate data or an inadequate computer may not be meaningful. On the other hand, an oversimplified model may produce a result that is unacceptable. It is, therefore, necessary to strike a balance between the level of accuracy and the complexity of the model. A model must incorporate only those features that are essential to reduce the error to an acceptable level.

4.6 BLUNDERS

Blunders are errors that are caused due to human imperfection. As the name indicates, such errors may cause a very serious disaster in the result. Since these errors are due to human mistakes, it should be possible to avoid them to a large extent by acquiring a sound knowledge of all aspects of the problem as well as the numerical process.

Human errors can occur at any stage of the numerical processing cycle. Some common types of errors are:

1. lack of understanding of the problem
2. wrong assumptions
3. overlooking of some basic assumptions required for formulating the model
4. errors in deriving the mathematical equation or using a model that does not describe adequately the physical system under study

5. selecting a wrong numerical method for solving the mathematical model
6. selecting a wrong algorithm for implementing the numerical method
7. making mistakes in the computer program, such as testing a real number for zero and using < symbol in place of > symbol
8. mistakes in data input, such as misprints, giving values column-wise instead of row-wise to a matrix, forgetting a negative sign, etc.
9. wrong guessing of initial values

As mentioned earlier, all these mistakes can be avoided through a reasonable understanding of the problem and the numerical solution methods, and use of good programming techniques and tools.

4.7 ABSOLUTE AND RELATIVE ERRORS

Let us now consider some fundamental definitions of error analysis. Regardless of its source, an error is usually quantified in two different but related ways. One is known as *absolute error* and the other is called *relative error*.

Let us suppose that the *true value* of a data item is denoted by x_t and its *approximate value* is denoted by x_a. Then, they are related as follows:

$$\text{True value } x_t = \text{Approximate value } x_a + \text{Error}.$$

The error is then given by

$$\text{Error} = x_t - x_a$$

The error may be negative or positive depending on the values of x_t and x_a. In error analysis, what is important is the magnitude of the error and not the sign and, therefore, we normally consider what is known as *absolute error* which is denoted by

$$e_a = |x_t - x_a|$$

In many cases, absolute error may not reflect its influence correctly as it does not take into account the order of magnitude of the value under study. For example, an error of 1 gram is much more significant in the weight of a 10 gram gold chain than in the weight of a bag of rice. In view of this, we introduce the concept of *relative error* which is nothing but the "normalised" absolute error. The relative error is defined as follows:

$$e_r = \frac{\text{absolute error}}{|\text{true value}|}$$

$$= \frac{|x_t - x_a|}{|x_t|} = \left|1 - \frac{x_a}{x_t}\right|$$

More often, the quantity that is known to us is x_a and, therefore, we can modify the above relation as follows:

$$e_r = \frac{x_t - x_a}{x_a} = \left|1 - \frac{x_t}{x_a}\right|$$

The fractional form of e_r can also be expressed as the *per cent relative error* as

$$\text{Per cent } e_r = e_r \times 100$$

Example 4.7

A civil engineer has measured the height of a 10 floor building as 2950 cm and the working height of each beam as 35 cm while the true values are 2945 cm and 30 cm, respectively. Compare their absolute and relative errors.

Absolute error in measuring the height of the building is

$$e_1 = 2950 - 2945 = 5 \text{ cm}$$

The relative error is

$$e_{r,1} = 5/2945 = 0.0017 = 0.17\%$$

Absolute error in measuring the height of the beam is

$$e_2 = 35 - 30 = 5 \text{ cm}$$

The relative error is

$$e_{r,2} = 5/30 = 0.17 = 17\%$$

Although the absolute errors are the same, the relative errors differ by 100 times. It shows that there is something wrong in the measurement of the height of the beam. It should be done more accurately.

4.8 MACHINE EPSILON

Recall that the round off error introduced in a number when it is represented in floating point form is given by

$$\text{Chopping error} = g \times 10^{E-d}, \qquad 0 \le g < 1$$

where g represents the truncated part of the number in normalised form, d is the number of digits permitted in the mantissa, and E is the exponent. The absolute relative error due to chopping is then given by

$$e_r = \left|\frac{g \times 10^{E-d}}{f \times 10^{E}}\right|$$

The relative error is maximum when g is maximum and f is minimum. We know that the maximum possible value of g is less than 1.0 and minimum possible value of f is 0.1. The absolute value of the relative error therefore satisfies.

$$e_r \le \left| \frac{1.0 \times 10^{E-d}}{0.1 \times 10^{E}} \right| = 10^{-d+1}$$

The maximum relative error given above is known as *machine epsilon*. The name "machine" indicates that this value is machine dependent. This is true because the length of mantissa d is machine dependent. For a decimal machine that uses chopping,

$$\text{Machine epsilon } \varepsilon = 10^{-d+1}$$

Similarly, for a machine which uses symmetric roundoff,

$$e_r \le \left| \frac{0.5 \times 10^{E-d}}{0.1 \times 10^{E}} \right| = \frac{1}{2} \times 10^{-d+1}$$

and therefore

$$\boxed{\text{Machine epsilon } \varepsilon = \frac{1}{2} \times 10^{-d+1}}$$

It is important to note that the machine epsilon represents upper bound for the roundoff error due to floating point representation. It also suggests that data can be represented in the machine with d significant decimal digits and the relative error does not depend in any way on the size of the number.

More generally, for a number x represented in a computer,

$$\boxed{\text{Absolute error bound} = |x| \times \varepsilon}$$

For a computer system with binary representation, the machine epsilon is given by

$$\boxed{\begin{array}{l} \textit{Chopping} \\ \quad \text{Machine epsilon } \varepsilon = 2^{-d+1} \\ \textit{Symmetric rounding} \\ \quad \text{Machine epsilon } \varepsilon = 2^{-d} \end{array}}$$

Note that we have simply replaced the base 10 by base 2. Here d indicates the length of binary mantissa in bits.

We may generalise the expression for machine epsilon for a machine which uses base b with d-digit mantissa as follows:

$\varepsilon = b \times b^{-d}$ for chopping
$\varepsilon = b/2 \times b^{-d}$ for symmetric rounding

Example 4.8

When a computer uses a number base 2, how many significant decimal digits are contained in the mantissa of floating numbers?

Assume that the binary computer has p-bit mantissa. Then the error bound is 2^{-p}. This computer will have q significant digits with symmetric rounding, if,

$$2^{-p} = 1/2 \times 10^{-q+1}$$

Taking logarithms to the base 10, we get

$$q = 1 + (p - 1) \log_{10} 2$$

If we assume $p = 24$, then

$$q = 1 + 23 \log_{10} 2 \approx 7.9$$

We may say that the computer can store numbers with seven significant decimal digits.

4.9 ERROR PROPAGATION

Numerical computing involves a series of computations consisting of basic arithmetic operations. Therefore, it is not the individual roundoff errors that are important but the final error on the result. Our major concern is how an error at one point in the process *propagates* and how it effects the final total error. In this section, we will discuss the arithmetic of error propagation and its effects.

Addition and Subtraction

Consider addition of two numbers, say, x and y.

$$\begin{aligned} x_t + y_t &= x_a + e_x + y_a + e_y \\ &= (x_a + y_a) + (e_x + e_y) \end{aligned}$$

Therefore,

$$\boxed{\text{Total error} = e_{x+y} = e_x + e_y}$$

Similarly, for subtraction

$$\boxed{\text{Total error} = e_{x-y} = e_x - e_y}$$

Note that the addition $e_x + e_y$ does not mean that error will increase in all cases. It depends on the sign of individual errors. Similar is the case with subtractions.

Since we do not normally know the sign of errors, we can only estimate error bounds. That is, we can say that

$$|e_{x \pm y}| \leq |e_x| + |e_y|$$

Therefore, the rule for addition and subtraction is: *the magnitude of the absolute error of a sum (or difference) is equal to or less than the sum of the magnitudes of the absolute errors of the operands.*

This inequality is called the *triangle inequality*. The equality applies when the operands have the same signs, and the inequality applies if the signs are different.

Multiplication

Here, we have

$$x_t \times y_t = (x_a + e_x) \times (y_a + e_y) = x_a y_a + y_a e_x + x_a e_y + e_x e_y$$

Errors are normally small and their products will be much smaller. Therefore, if we neglect the product of the errors, we get

$$x_t \times y_t = x_a y_a + x_a e_y + y_a e_x$$
$$= x_a y_a + x_a y_a (e_x/x_a + e_y/y_a)$$

Then,

$$\boxed{\text{Total error} = e_{xy} = x_a y_a (e_x/x_a + e_y/y_a)}$$

Division

We have

$$\frac{x_t}{y_t} = \frac{x_t + e_x}{y_a + e_y}$$

Multiplying both numerator and denominator by $y_a - e_y$ and rearranging the terms, we get

$$\frac{x_t}{y_t} = \frac{x_a y_a + y_a e_x - x_a e_y - e_x e_y}{y_a^2 - e_y^2}$$

Dropping all terms that involve only product of errors, we have

$$\frac{x_t}{y_t} = \frac{x_a y_a + y_a e_x - x_a e_y}{y_a^2}$$

$$= \frac{x_a}{y_a} + \frac{x_a}{y_a}\left(\frac{e_x}{x_a} - \frac{e_y}{y_a}\right)$$

Thus,

$$\boxed{\text{Total error} = e_{x/y} = \frac{x_a}{y_a}\left(\frac{e_x}{x_a} - \frac{e_y}{y_a}\right)}$$

Again applying the triangle inequality theorem, we have

$$e_{x/y} \le \left|\frac{x_a}{y_a}\right|\left(\left|\frac{e_x}{x_a}\right| + \left|\frac{e_y}{y_a}\right|\right)$$

$$e_{xy} \le |x_a y_a|\left(\left|\frac{e_x}{x_a}\right| + \left|\frac{e_y}{y_a}\right|\right)$$

Note 1

The initial errors e_x and e_y may be of any type. They may be

1. empirical errors introduced in the measuring process
2. roundoff errors introduced in conversion
3. roundoff errors introduced due to arithmetic operations in the previous step, if x_t and y_t represent some intermediate results

4. truncation errors, if x_t and y_t represent the result of evaluation of infinite series
5. any combination of the above

Note 2

The final errors (after arithmetic operations) e_{x+y}, e_{x-y}, e_{xy} and $e_{x/y}$ are expressed in terms of only e_x and e_y and do not contain the roundoff errors introduced by the operations themselves. This results from the need to store the result in floating point representation. Therefore, we must add the roundoff error introduced in doing the operation in each case. For example,

$$e_{x+y} = e_x + e_y + e_o$$

Now, we can have relative errors for all the four operations as follows:

Addition and Subtraction

$$e_{r,x\pm y} \leq \frac{|e_x| + |e_y|}{|x_a \pm y_a|}$$

$$= \left|\frac{x_a}{x_a \pm y_a}\right| \cdot |e_{r,x}| + \left|\frac{y_a}{x_a \pm y_a}\right| \cdot |e_{r,y}|$$

Multiplication and Division

$$e_{r,xy} = |e_{r,x}| + |e_{r,y}|$$

$$e_{r,x/y} = |e_{r,x}| + |e_{r,y}|$$

Example 4.9

Estimate the relative error in $z = x - y$ when $x = 0.1234 \times 10^4$ and $y = 0.1232 \times 10^4$ as stored in a system with four-digit mantissa.

We know

$$e_{r,z} \leq \frac{|e_x| + |e_y|}{|x - y|}$$

Since the numbers x and y are stored in a four-digit mantissa system, they are properly rounded off and therefore,

$$|e_{r,x}| \leq \frac{1}{2} \times 10^{-3} = 0.05\%$$

$$|e_{r,y}| \leq \frac{1}{2} \times 10^{-3} = 0.05\%$$

Then

$$e_x = 0.1234 \times 10^4 \times 0.5 \times 10^{-3} = 0.617$$

$$e_y = 0.1232 \times 10^4 \times 0.5 \times 10^{-3} = 0.616$$

Therefore

$$|e_z| \leq |e_x| + |e_y| = 1.233$$

$$|e_{r,z}| \leq \frac{1.233 \times 10^{-4}}{|0.1234 - 0.1232|} = 0.6165 = 61.65\%$$

Although the relative errors in x and y are very small, the relative error in z is very large. If we use this result as an input to further calculations, the final result will be disastrous. The error due to subtraction of two nearly equal numbers is known as *subtractive cancellation*.

Rules for error propagation discussed above can also be derived using the concepts of differential calculus. We will find this approach more convenient when we deal with complex functions. For example, consider a power function

$$w = x^n$$

$$\text{Error } \Delta w = nx^{n-1}\,\Delta x$$

Relative error,

$$e_{r,w} = n \times \Delta x/x = n \times e_{r,x}$$

The relative error in w is n times the relative error in x.

Sequence of Computations

We have seen how errors in the operands propagate to the result of an operation. As we know, the computer can do only one operation at a time. It performs a sequence of operations in order to evaluate even a simple expression; such as

$$w = x^2 + y/z$$

In such cases, the result of one operation is stored in the machine in the floating point form before it is used as an input for the next operation. At each stage of computation, a roundoff error is therefore introduced in the result before it is used again. Thus, each stage becomes a source of new errors. This is illustrated in Fig. 4.2, for evaluating the above expression. The intermediate value u contains the propagated error due to error in x and its own roundoff error r_1. Similarly, v contains the propagated error due to errors in y and z and also the roundoff error r_2. Finally, w contains the propagated error due to errors in u and v and the roundoff error r_3.

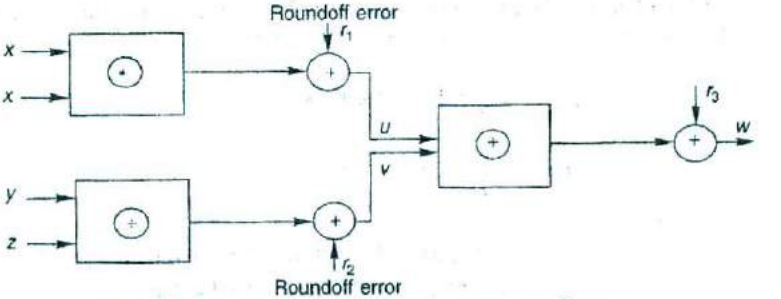


Fig. 4.2 Block diagram for evaluation of $x^2 + y/z$

Example 4.10

Find the absolute error in $w = xy + z$ if $x = 2.35$, $y = 6.74$ and $z = 3.45$

$$x_a = 2.35,\ e_x = 0.005|2.35| = 0.01175$$

$$y_a = 6.74,\ e_y = 0.005|6.74| = 0.03370$$

$$z_a = 3.45,\ e_z = 0.005|3.45| = 0.01725$$

$$e_{xy} = |x_a|e_y + |y_a|e_x$$

$$= 2.35 \times 0.03370 + 6.74 \times 0.01175 = 0.15839$$

$$e_w = |e_{xy}| + |e_z| = 0.15839 + 0.01725 = 0.17564$$

Addition of a Chain of Numbers

As we pointed out earlier, many standard mathematical ideas do not hold good in computer arithmetic. One such case is the floating point addition. In computer arithmetic, the floating point addition is *not always associative*. That is,

$$x + y + z \neq z + y + x$$

The examples 4.11 and 4.12 illustrate this rule.

Example 4.11

Evaluate $w = x + y + z$, where $x = 9678$, $y = 678$ and $z = 78$. Remember, the computer performs arithmetic operations one at a time and from left to right. We assume that there is no inherent error (for the sake of simplicity) in x, y and z and the length of mantissa is four.

Let $u = x + y$

Then,

$$u = 0.9678 \times 10^4 + 0.0678 \times 10^4 = 1.0356 \times 10^4$$

$$w = u + z = 0.1035 \times 10^5 + 78$$

$$= 0.1035 \times 10^5 + 0.00078 \times 10^5$$

$$= 0.1042 \times 10^5 = 10420$$

True $w = 10444$

$$e_w = 24$$

$$e_{r,w} = 2.3 \times 10^{-3}$$

Example 4.12

Evaluate $w = z + y + x$ using the data in the above example.

Here, $u = 78 + 678 = (0.078 + 0.678)10^3 = 0.756 \times 10^3$

$w = u + x$

$$= 0.0756 \times 10^4 + 0.9678 \times 10^4 = 1.0434 \times 10^4$$

$$= 0.1043 \times 10^5 = 10430$$

True $w = 10444$

$$e_w = 14$$

$$e_{r,w} = 1.3 \times 10^{-3}$$

Examples 4.11 and 4.12 show that the errors are not the same in both the cases. It also shows that the error is less when the numbers are arranged in the increasing order of their magnitude. See Example 4.13 for a more general proof.

Example 4.13

Prove that the procedure $w_1 = (y + z) + x$ is better than the procedure $w_2 = (x + y) + z$ when $|x| > |y| > |z|$.

(a) Procedure $w_2 = (x + y) + z$

Let $u = x + y$

Then, $$e_{r,u} = \frac{x}{x+y} e_{r,x} + \frac{y}{x+y} e_{r,y} + r_1$$

where r_1 is the relative roundoff error introduced at this stage.

$$e_{r,w_2} = \frac{u}{u+z} e_{r,u} + \frac{z}{u+z} e_{r,z} + r_2$$

$$= \frac{x+y}{S}\left[\frac{x}{x+y} e_{r,x} + \frac{y}{x+y} e_{r,y} + r_1\right] + \frac{z}{S} e_{r,z} + r_2$$

$$= \frac{1}{S}\left[x \cdot e_{r,x} + y \cdot e_{r,y} + z \cdot e_{r,z} + (x+y) r_1 + (x+y+z) r_2\right]$$

where $S = x + y + z$

Now, let us use

$$R_1 = \max\left(|e_{r,x}|, |e_{r,y}|, |e_{r,z}|\right) \qquad R_2 = \max\left(|r_1|, |r_2|\right)$$

Then, we get

$$e_{r,w_2} = \frac{1}{S}\left[(x+y+z) R_1 + (2x+2y+z) R_2\right]$$

$$e_{w_2} = (x+y+z)R_1 + (2x+2y+z)R_2$$

If we further assume that R_1 and R_2 are only due to conversion, then

$$R = R_1 = R_2 \; e_{w_2} = (3x + 3y + z)R$$

(b) Procedure $w_1 = (y + z) + x$

Similarly we can show that $e_{w_1} = (3z + 3y + x)R$

Comparison

$$e_{w_2} = (3x + 3y + z)R = (3x + 3y + 3z - 2z)R = (3S - 2z)R$$

$$e_{w_1} = (3x + 3y + 3z - 2x)R = (3S - 2x)R$$

Since $x > z$, $e_{w_1} < e_{w_2}$

Therefore, the procedure $w_1 = (y + z) + x$ gives better results than the procedure $w_2 = (x + y) + z$.

Polynomial Functions

Suppose we wish to evaluate a function $f(x)$ where f is differentiable and the approximate value x_a of x is given. In such cases, we can estimate the error bound in $f(x)$ using the *mean-value theorem* of calculus.

According to this theorem,

$$f(x) - f(x_a) = (x - x_a)f'(\theta)$$

where θ is some value between x and x_a and f' is the first derivative of the function f. Then the error in $f(x)$ is

$$e_f = |f(x) - f(x_a)| = |e_x f'(\theta)|$$

Since the value of θ is unknown, we take the maximum of $f'(\theta)$ in the interval for estimating the bound for e_f. Then,

$$e_f \leq e_x \max |f'(\theta)|$$

This means that we have to evaluate the function $f'(\theta)$ at various values of θ and find the upper bound. This is sometimes a difficult task.

Normally, the error e_x is small and, therefore, we can make a reasonable approximation as follows

$$\boxed{e_f \approx e_x f'(x_a)}$$

Note that this e_f does not include the errors that occur during the evaluation of the function itself due to conversion at various stages.

Example 4.14

Estimate the absolute and relative errors for the function

$$f(x) = \sqrt{x} + x \text{ for } x_a = 4.000$$

We assume that x is correct to four significant digits. Then

$$e_x = 0.0005 = 5 \times 10^{-4}$$

$$f'(x_a) = \frac{1}{2} x^{-1/2} + 1 = \frac{1}{2}\sqrt{x} + 1$$

$$f'(x_a) = \frac{1}{4} + 1 = 1.25$$

Then

$$e_f = 5 \times 10^{-4} \times 1.25 = 6.25 \times 10^{-4}$$

$$e_{rf} = \frac{e_r}{f(x_a)} = \frac{6.25 \times 10^{-4}}{6} = 0.104 \times 10^{-3}$$

The mean-value theorem approach can be readily extended to functions with more than one variable, using partial derivatives. For functions with two variables, x and y, we have

$$e_f = \left| e_x f_x'(x_a, y_a) \right| + \left| e_y f_y'(x_a, y_a) \right|$$

where f_x' and f_y' denote partial derivatives with respect to x and y.

Example 4.15

Estimate the error in evaluating $f(x, y) = x^2 + y^2$ for $x = 3.00$ and $y = 4.00$

We assume that

$$e_x = e_y = 0.005$$

$$f_x'(x, y) = 2x \text{ and } f_y'(x, y) = 2y$$

Therefore,

$$e_f = 2x\, e_x + 2y\, e_y$$
$$= (2 \times 3.00 + 2 \times 4.00) \times 5 \times 10^{-3} = 0.07$$

4.10 CONDITIONING AND STABILITY

We know that uncertainties exist in all stages of numerical processing. We have discussed in detail how these uncertainties, particularly roundoff errors, are introduced at various stages and how they are propagated during the evaluation of an expression or implementation of a numerical method. Induced errors such as roundoff errors accumulate with the increasing number of computations in a process. There are situations where even a single operation may magnify the roundoff errors to a level that completely ruins the result. A computation process in which the cumulative effect of all input errors is grossly magnified is said to be *numerically unstable*. It is, therefore, important to understand the conditions under which the process is likely to be "sensitive" to input errors and becomes unstable. Investigations to see how small changes (or *perturbations*) in input parameters influence the output are termed as *sensitivity analysis*.

Numerical instability may arise due to sensitivity inherent in the problem or sensitivity of the numerical method (or algorithm). This is

illustrated in Fig. 4.3. As we know, a mathematical model can be solved either by analytical methods or by numerical methods. In either case, when a small disturbance in an input parameter (known as *inherent error*) causes unacceptable amount of error in the output, we say that the problem is *inherently unstable*. Such problems are said to be *ill-conditioned*. When a problem itself is sensitive to small changes in its parameters, it is almost impossible to make a numerically stable method for its solution.

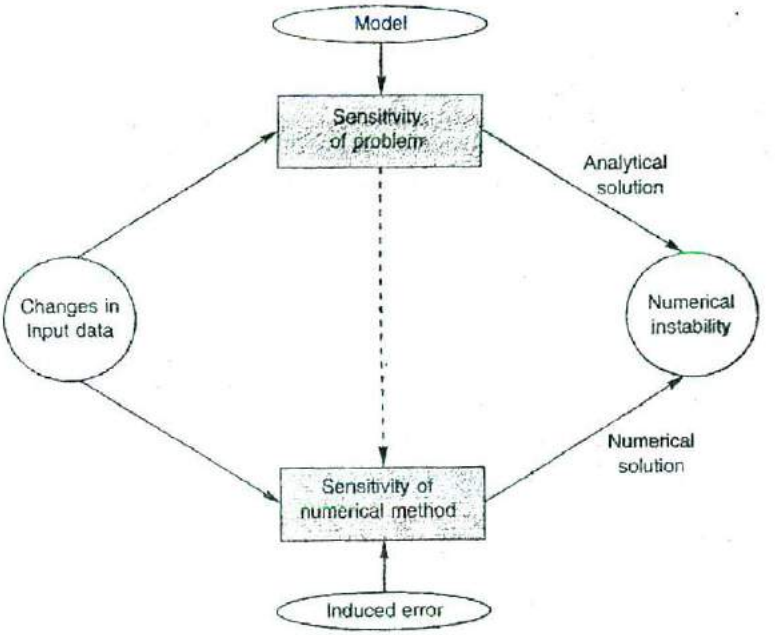


Fig. 4.3 Instability of numerical process

The term "condition" is used to describe the sensitivity of problems or methods to uncertainty. Let us suppose we are evaluating a function $f(x)$ and a small change in x produces a change in $f(x)$. We can quantify the condition of this function by a number called *condition number* which is defined as follows:

$$\text{Condition number} = \frac{\text{relative error in } f(x)}{\text{relative error in } x}$$

The relative error in $f(x)$ is

$$e_{r,f} = \frac{\Delta f}{f(x)} = \frac{f'(x)\Delta x}{f(x)}$$

The relative error in x is

$$e_{r,x} = \frac{\Delta x}{x}$$

Then

$$\text{Condition number} = \frac{xf'(x)}{f(x)}$$

The condition number provides a measure of extent to which an error in x is magnified in $f(x)$. If the condition number is large, then the function $f(x)$ is said to be ill-conditioned and its computation will be numerically unstable. There are different situations when a problem can have a large condition number.

1. small $f(x)$ compared to x and $f'(x)$
2. large $f'(x)$ compared to x and $f(x)$
3. large x compared to $f(x)$ and $f'(x)$.

When several parameters are involved, we may have instability with respect to some parameters and stability with respect to others. In such cases, we should use the partial derivatives to estimate the total change. That is,

$$\Delta f = \left|\frac{\partial f}{\partial x}\Delta x\right| + \left|\frac{\partial f}{\partial y}\Delta y\right| + \left|\frac{\partial f}{\partial z}\Delta z\right| + \ldots$$

Example 4.16

Show that the following system of equations is ill-conditioned for computing the point of intersection when m_1 and m_2 are nearly equal.

$$y = m_1 x + C_1$$
$$y = m_2 x + C_2$$

Solving the equations for x and y we get

$$x = \frac{C_1 - C_2}{m_2 - m_1}$$

$$y = m_1 \times \left[\frac{C_1 - C_2}{m_2 - m_1}\right] + C_1$$

Let us assume that $C_1 = 7.00$, $C_2 = 3.00$, $m_1 = 2.00$ and $m_2 = 2.01$. Then

$$x = \frac{7-3}{2.01-2.00} = 400$$

$$y = 2.00 \times 400 + 7 = 807$$

Now, let us change the value of m_2 from 2.01 to 2.005. Then

$$x = \frac{7-3}{2.005-2.00} = 800$$

$$y = 2.00 \times 800 + 7 = 1607$$

It shows that a small change (0.25 per cent) in the parameter m_2 results in almost 100 per cent change in the values of x and y. Therefore, the problem is absolutely ill-conditioned.

Example 4.17

Compute and interpret the condition number for

$$f(x) = \sqrt{(x-1)}$$

$$f'(x) = \frac{1}{2} \times (x-1)^{-1/2}$$

$$\text{Condition number} = \frac{xf'(x)}{f(x)} = \frac{x}{2} \frac{(x-1)^{-1/2}}{(x-1)^{1/2}}$$

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

The function is numerically unstable for the values of x close to 1.

Note that the term "ill-conditioned" is ill-defined. If we are to take floating point seriously then we should say "relatively small changes" and "relatively large changes".

If the ill-conditioned effect is present in the original physical system itself, then there is nothing that we can do to achieve numerical stability. In many instances, the ill-conditioning arises from mathematical formulation of the problem. In such cases, the instability may be removed by reformulating the mathematical models. For example, consider the quadratic equation

$$ax^2 + bx + c = 0.$$

We know that the two roots are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

When $b^2 >> 4ac, \sqrt{b^2 - 4ac}$ will be very close to b and therefore, when b is positive, the expression for x_1 may have the effect of *subtractive cancellation*. Here, we can reformulate the formula for x_1 as follows:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \times \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}}$$

$$= \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$

If b is negative, we must perform the same operation for x_2.

Another approach to the same problem is to change the algorithm of calculating x_1 and x_2. First find the larger root from the formula

$$x_1 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

and then find the smaller root from the relation

$$x_1 x_2 = c/a$$

Example 4.18

Compute the difference of square roots of two numbers $x = 497.0$ and $y = 496.0$.

Assume x and y are exact. Assuming a mantissa length of 4,

$$\sqrt{x} = \sqrt{497.0} = 0.2229 \times 10^2$$

$$\sqrt{y} = \sqrt{496.0} = 0.2227 \times 10^2$$

$$z = \sqrt{x} - \sqrt{y} = 0.0002 \times 10^2 = 0.02$$

Let us try another approach by rearranging the terms as follows:

$$z = \sqrt{x} - \sqrt{y} = \frac{x - y}{\sqrt{x} + \sqrt{y}}$$

$$= \frac{1}{0.4456 \times 10^2} = 0.2244 \times 10^1 = 0.02244$$

The correct answer is 0.02244. This shows that by rearranging the terms we improve the result.

Example 4.19

Suggest an algorithm to compute the binomial co-efficient.

$$B = \frac{n!}{(n-r)!\,r!}$$

A simple algorithm to calculate B is to find the factorials $n!$, $(n-r)!$ and $r!$ and combine them to get B. That is,

$$B = \frac{F_1}{F_2 \times F_3}$$

where $F_1 = n!$, $F_2 = (n-r)!$ and $F_3 = r!$

The problem with this algorithm is that when n is large, the factorial $n!$ may be too large for the computer to store and thus, may result in overflow error. This problem can be overcome by modifying the algorithm as follows:

$$B = \binom{n}{r} = \left(\frac{n}{r-1}\right)\frac{n+1-r}{r}$$

$$B = n \quad \text{for} \quad r = 1$$

This can be expressed recursively as

$$B = n \prod_{i=2}^{r} \frac{n+1-i}{i}$$

This algorithm will compute B without causing an overflow error unless the final answer itself is too large.

Example 4.20

Reformulate the following expressions to avoid loss of accuracy due to subtractive cancellation.

(a) $x - \sqrt{x^2 - 1}$ for large x

(b) $\dfrac{1 - \cos x}{\sin x}$ for small x

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

(b) $f(x) = \dfrac{1 - \cos x}{\sin x} = \dfrac{(1 - \cos x)(1 + \cos x)}{\sin x (1 + \cos x)}$

$= \dfrac{\sin^2 x}{\sin x (1 + \cos x)} = \dfrac{\sin x}{1 + \cos x}$

Even when the problem is formulated in a reasonable way and the input data is accurate, the method of solution may make the process unstable. For example, in a step-by-step algorithm where we use an interval h to increment a variable, the error may increase if h is decreased (or increased beyond some limit). If such *induced errors* are large, then our method of solution may exhibit what is known as *induced instability*. Another example is the "pivoting" technique used in solving simultaneous linear equations (see Chapter 7). Here, pivoting can make a well-conditioned system into an ill-conditioned one, if proper care is not taken in the design of algorithm.

Example 4.21

Show that the series $e^x = 1 + x + \dfrac{x^2}{2!} + \dfrac{x^3}{3!} + \ldots +$

becomes unstable when $x = -10$.

The series can be represented as

$$S(x) = \sum_{i=0}^{n} T_i + T_E$$

where

$$T_i = \frac{x_i}{i!}$$

T_E is the truncation error.

For $-1 < x < 1$, T_i decreases as i increases, but for large values of $|x|$, T_i will grow in magnitude until the factorial in the denominator dominates, when once again T_i will decrease in size. When $x = -10$, we have:

T_0	1
T_1	−10
T_2	50
T_3	−166.66767
T_4	416.66767
T_5	−833.33333
T_6	1388.8888...

Assuming a six-digit mantissa machine, the roundoff errors in representing the large values of T_i will be of greater magnitude than the final value $e^{-10} = 0.45 \times 10^{-4}$ itself. Therefore, the roundoff error totally dominates the computed solution. The method becomes unstable.

This difficulty may be overcome by using a simple technique known as the range reduction scheme for x. We know that

$$e^x = (e^{x/2})^2$$

Thus,

$$e^{-10} = (e^{-1})^{10} = ((e^{-0.5})^2)^{10}$$

4.11 CONVERGENCE OF ITERATIVE PROCESSES

As pointed out earlier, most of the numerical computing processes are iterative in nature. We start with an approximate value of the solution and compute iteratively the next approximate value till the difference between two consecutive values is negligible or within a specified limit. The number of iterations required to reach the given limit depends on the rate at which the iterates converge to the result.

Suppose that x_i, $i = 0, 1, 2,...$ is a sequence of iterates and x is the expected value of x. Let e_i be the error in the iterate x_i. Then

$$e_i = x_i - x \qquad \text{for each } i$$

We would like the iterates to converge to x and this would happen if the numerical process is stable. The process is said to converge if there exists positive constants p and c such that

$$\lim_{n \to \infty} \frac{|e_{i+1}|}{(|e_i|)^p} = c$$

The constant p is known as the *order of convergence* and c is known as *asymptotic convergence factor*. This shows that if the error in x_{i+1} is proportional to the pth power of the error in x_i (i.e. the previous iterate), then the iterative method is said to be of order of p. It is clear that the higher the order of iteration, more rapid is the *rate of convergence*.

The rate of convergence is a measure of how fast the truncation error goes to zero. This measure is used for comparing various iterative

methods. The rate of convergence is expressed in different ways. For example, if the method converges like h^2, the order of convergence is N^2, and so on. It means the value of p is 2. We shall consider the rate of convergence in detail when we discuss the iterative methods later.

4.12 ERROR ESTIMATION

It is now clear that it is almost impossible to know the exact error in a computed result. Nevertheless, it is possible at least to have some estimate of the error in the final result. There are three approaches that are popularly used in error estimation:

1. forward error analysis
2. backward error analysis
3. experimental error analysis

In *forward error analysis*, we try to estimate error bounds in the computed result using information such as uncertainties in the input data and the nature and number of arithmetic operations involved in the computing process. We can estimate the contribution due to

1. errors in the input data
2. roundoff errors in arithmetic operations
3. truncation of the iterative process
4. errors in formulation of the model

For example, we have seen in section 4.9 that the total error of a sum of three values is given by

$$e_s \le (x + y + z)R_1 + (2x + 2y + z)R_2$$

where

$$R_1 = \max(|e_{rx}|, |e_{ry}|, |e_{rz}|)$$
$$R_2 = \max(|r_1|, |r_2|)$$

This can be easily generalised for addition of n values:

$$e_s \le [(x_1 + x_2 + \ldots + x_n)]R_1 + [(n-1)x_1 + (n-1)x_2 + (n-2)x_3 + \ldots + 2x_{n-1} + x_n]R_2$$

$$= R_1 \sum_{i=1}^{n} x_i + R_2 \left[(n-1)x_1 + \sum_{i=1}^{n-1} i \cdot x_{n-i+1} \right]$$

Similarly, we can estimate bounds for product of n numbers.

Error estimated through forward analysis is always pessimistic and is often much higher than the actual error.

In *backward error analysis*, we try to show that the computed results satisfy the problem within the given bounds. For example, we can put back the roots computed in the equation and see to what extent they satisfy the original equation. By comparison, we can then decide on how much confidence we can place in the computed results. Backward analysis is usually easier to perform than forward error analysis.

Experimental error analysis involves a series of experiments by using different methods and step sizes and then comparing the results. We may also perform *sensitivity analysis* to see how any change in parameters affects the result.

When the application is very critical in nature (such as space and defence applications) the problem may be solved by more than two independent specialists groups and the results can be compared.

4.13 MINIMISING THE TOTAL ERROR

Assuming that the mathematical model has been properly formulated and the input data are accurate, the total numerical error primarily consists of two components, namely, truncation and roundoff errors. Any effort to minimise the total error should, therefore, be concentrated on the ways to reduce these two types of errors. The steps may include:

1. increasing the significant figures of the computer
2. minimising the number of arithmetic operations
3. avoiding subtractive cancellations
4. choosing proper initial parameters

In many iterative processes such as numerical integration, it is possible to minimise the truncation error by decreasing the step size. But this would necessarily increase the number of iterations and thereby, arithmetic operations. This would certainly increase the roundoff error. This phenomenon is illustrated in Fig. 4.4. We must, therefore, judiciously choose a step size that would minimise the sum of these errors.

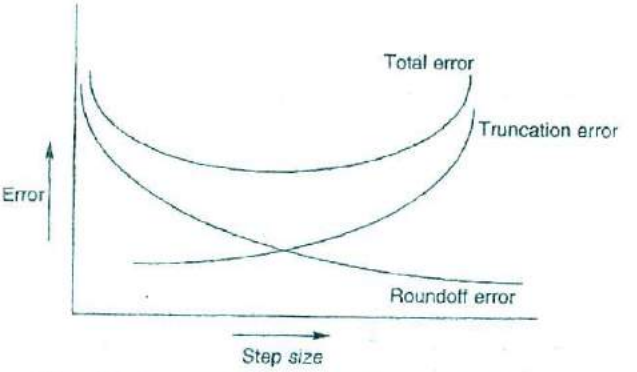


Fig. 4.4 Dependence of error on step size

4.14 PITFALLS AND PRECAUTIONS

We have seen that the floating point arithmetic system is full of pitfalls such as conversion, roundoff, overflow and underflow errors. In many cases, we may have to consider some precaution techniques to get the

most accurate results. The type of precaution techniques that might be used depends both on the computer hardware and the nature of the mathematical models. Here are some hints that might help improve the accuracy of the results.

1. Rearrange the formula so that you can avoid subtraction of two nearly equal numbers. For example,

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

can be replaced by

$$x + y$$

when x and y are nearly equal.

2. If necessary, use double precision for floating point calculations. This would improve the accuracy considerably but would take more execution time and computer memory space.
3. Rearrange your formula to reduce the number of arithmetic operations. An example is evaluation of a polynomial. The polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$$

may be rearranged as

$$(\ldots((a_n x + a_{n-1})x + a_{n-1})x \ldots + a_0)$$

This requires much less arithmetic operations.

4. When finding the sum of set of numbers, arrange the set so that they are in the ascending order of absolute value. That is, when $|a| > |b| > |c|$, then $(c - b) + a$ is better than $(a - b) + c$.
5. Wherever possible, rearrange your formula so that you use the original data rather than derived data.
6. Do not test a floating point number for zero in your algorithm.
7. Wherever possible, use integer arithmetic to avoid conversion and roundoff errors.
8. Avoid multiplication of large numbers that may lead to overflow.
9. Use alternative arithmetic such as interval arithmetic, if necessary.

4.15 SUMMARY

In this chapter, we studied various types of errors and how they can affect numerical calculations. We considered, in particular, the following:

- concept of significant digits and its relation to accuracy and precision of numbers
- inherent errors that are present in input data
- procedural errors introduced during the process of computing
- modelling errors that arise due to certain simplifying assumptions in the formulation of mathematical models
- importance of absolute and relative errors and their relation to the machine epsilon

- propagation of errors during computing and how it affects the result
- causes of numerical instability and how to overcome instability problems
- convergence of iterative processes
- estimation of errors and some steps that might help to reduce the final error

Key Terms

Absolute error
Accuracy
Algorithm
Asymptotic convergence factor
Backward error analysis
Blunder
Chopping
Condition number
Conditioning
Convergence
Conversion error
Data error
Discretisation error
Drag coefficient
Empirical error
Error propagation
Experimental error analysis
Forward error analysis
Human error
Ill-conditioned problem
Induced errors
Induced instability
Inherent error
Inherently unstable
Input error
Machine epsilon
Mean-value theorem
Modelling error
Numerical error
Numerical instability
Numerically unstable
Order of convergence
Perturbations
Precision
Procedural error
Rate of convergence
Relative error
Representation error
Rounding
Roundoff error
Sensitivity analysis
Significant digits
Stability
Subtractive cancellation
Symmetric rounding
Triangle inequality
Truncating
Truncation error

REVIEW QUESTIONS

1. Why is the study of errors important to a computational scientist?
2. Explain the concept of significant digits.
3. Describe the relationship between significant digits and the following:
   (a) round-off errors
   (b) accuracy
   (c) precision
4. What are inherent errors? How do they arise?
5. Distinguish between roundoff errors and truncation errors.
6. What is chopping? When does it occur?

7. What is symmetric round-off? Show that the symmetric error is, at worst, one-half the chopping error.
8. How does a truncation error occur? Give two examples.
9. How do mathematical models contribute to errors in numerical computing?
10. What are blunders? How can we minimize them?
11. What do you mean by relative error? How is it important in error analysis?
12. What is machine epsilon? How is it related to significant digits?
13. State and explain triangular inequality as applied to error propagation.
14. What is subtractive cancellation? How does its presence affect the result of a computation?
15. Define condition number. What is its significance to numerical computing?
16. What is range reduction technique? Give an example of its application.
17. How will you decide the convergence of an iterative process?
18. Explain briefly the three approaches used in error analysis.
19. In an iterative process, how does step size affect the total error?
20. Enumerate a few precautionary steps that might help improve the accuracy of numerical computing.

REVIEW EXERCISES

1. Find the accuracy and precision of the following numbers:
   (a) 12.345
   (b) 0.0002932
   (c) 0.0029320
   (d) 750
   (e) 750.5
   (f) −68.3705
2. Add the decimal numbers 0.4 and 0.65 in binary form using 6 binary digits and then estimate the error in the sum. Show that the error can be reduced by using more binary digits to represent the numbers.
3. Find the round-off error in the results of the following arithmetic operations, using four digit mantissa.
   (a) 27.65 + 22.20
   (b) 87.26 + 31.42
   (c) 1250.0 × 40.0
   (d) 3543.0 × 16.78
   (e) 25.68 ÷ 6.567
   (f) 456.7 − 1.531
   (g) 456.7 − 4.566
4. Calculate absolute and relative errors in the arithmetic operations in Exercise 3.

5. Estimate the relative error of the final result in evaluation of
   (a) $w_1 = (x + y)z$ and
   (b) $w_2 = x^2 + y/z$
   Given that $x = 1.2$, $y = 25.6$ and $z = 4.5$.
6. Find the absolute and relative errors in evaluating the following expressions:
   (a) $\sqrt{x^2 + y^2}$
   (b) $x\, e^y$
   Assume $x = 1.25$ and $y = 2.16$.
7. Find out which procedure (p_1 or p_2) produces better results:
   (a) $p_1 = x(x + 2)$, $p_2 = x^2 + 2x$
   (b) $p_1 = (x + 1)(x + 2)$, $p_2 = x(x + 3) + 2$
8. Determine the condition of the following functions:
   (a) $f(x) = \sin(x)$
   (b) $f(x) = 1/(1 - x)$
   (c) $f(x) = x^5$
   (d) $f(x) = x^{1/3}$
9. Rearrange the following expression to avoid loss of accuracy due to subtractive cancellation:
   (a) $\cos x - \sin x$ for x close to 45°
   (b) $\sqrt{1+x} - \sqrt{1-x}$ for small x
   (c) $1 - \cos x$ for small x
   (d) $\sqrt{x^2 + 1} - x$ for large x.
   (e) $\ln(x + 1) - \ln(x)$ for large x
10. Estimate the maximum error in evaluating the expression
    $x^3 - 2.5x^2 + 3.1x - 1.5$ at $x = 1.25$

CHAPTER 5

FORTRAN 77 Overview

5.1 NEED AND SCOPE

After a sound algorithm and a detailed flow chart comes the development of computer program, known as *coding*. Codes are written in a high-level computer language. Hundreds of high-level languages have been developed during the last four decades. Among these, a few have direct relevance to numerical computing. They include, among others, BASIC, FORTRAN, C, and C++.

FORTRAN, which stands for FORmula TRANslation, was the earliest scientific language developed in the 1950s. Since it was specially designed for mathematical computations, it has been the most widely used language for scientific and engineering applications. It is well suited for implementing the numerical methods discussed in this book. In spite of development of numerous other languages, FORTRAN continues to play a dominant role in engineering applications. Consequently, we are going to use FORTRAN for developing programs for implementing our algorithms. Our programs and algorithms are concise and general enough to be used as the basics for developing programs in other languages, if necessary.

A complete description of FORTRAN 77 is beyond the scope of this book. We only give here an overview of the language. However, enough material has been included so that the reader can easily understand the programs given in the book and also modify and implement them effectively. Wherever necessary, FORTRAN 90 features are also included.

5.2 A SAMPLE PROGRAM

For solving any problem in FORTRAN, we have to write a sequence of instructions using certain statements known as FORTRAN statements.

These instructions are required by the computer to perform the following tasks:

1. get data into the computer memory
2. perform arithmetic and logical operations on data
3. provide results on an output media

Program 5.1

```
*-------------------------------------------------------*
      PROGRAM SAMPLE
*-------------------------------------------------------*
* Main program                                          *
*   A program to evaluate a function at different       *
    points                                              *
*-------------------------------------------------------*
* Functions invoked                                     *
*   NIL                                                 *
*-------------------------------------------------------*
* Subroutines used                                      *
*   NIL                                                 *
*-------------------------------------------------------*
* Variables used                                        *
*   X - Independent variable                            *
*   F - Function value                                  *
* COUNT - Counter to store number of evaluations        *
*-------------------------------------------------------*
* Constants used                                        *
*   N - Number of function values                       *
*-------------------------------------------------------*
      REAL X, F
      INTEGER COUNT, N
      PARAMETER( N = 5 )
      WRITE(*, *) 'Input value of X'
      READ(*, *) X
      WRITE(*, *) '  OUTPUT OF SAMPLE PROGRAM''
      WRITE(*, *) '        X          F        '
      COUNT = 0
100   F = X * X
      WRITE(*, *) X, F
      X = X + X
      COUNT = COUNT + 1
      IF( COUNT .LT. N ) GO TO 100
      STOP
      END
*-------------------------------------------------------*
```

A sample FORTRAN 77 program to evaluate the function $f(x) = x^2$ for n values of x is shown in Program 5.1. When we run this program, it displays first the following message:

```
Input Value of X
```

and then waits for the input from the keyboard. Let us enter a real value, say 1.0 and then press the RETURN key. Execution now continues and produces the following output on the screen:

```
Input value of X
1.0
    OUTPUT OF SAMPLE PROGRAM
        X              F
     1.0000000      1.0000000
     2.0000000      4.0000000
     4.0000000     16.0000000
     8.0000000     64.0000000
    16.0000000    256.0000000
Stop - Program terminated.
```

Program 5.1 illustrates some of the FORTRAN statements and the overall format of a FORTRAN 77 program. This program is intended to give only an overview of a FORTRAN program. The details of FORTRAN features will be discussed in the sections to follow.

The first line of Program 5.1 is a FORTRAN statement known as *program unit header* or *program statement*. This statement is not essential in all systems. You must consult the system manual before using it.

> This is a recommended style in FORTRAN 90.

The lines starting with * or C in the first column are known as *comment lines* (only C in the FORTRAN IV version). These lines are used to insert explanatory remarks to help readers to understand the program. They are not instructions to the computer and, therefore, they are ignored by the compiler. Comment lines should be used liberally to explain various aspects within the program.

> FORTRAN 90 permits the use of the character '!' in the first column to mark a comment line. This can also be used as an in-line comment.

The next two lines

```
REAL X, F
INTEGER COUNT, N
```

declare the types of storage associated with the variables. That is, the variables X and F are declared as type *real* and COUNT and N as type *integer*. These statements are called *type declaration* statements.

> In FORTRAN 90, they are written as `REAL:: X, F` and `INTEGER:: COUNT, N`.

The identifier N represents the number of points at which the function is evaluated. The value of N is going to be constant throughout the program execution. Such identifiers are known as *symbolic constants* or *named constants*. Symbolic constants may be given values using a PARAMETER statement. (Some version of FORTRAN 77 may not include the feature of PARAMETER statement). The value of a symbolic constant cannot be changed during execution.

Some variables need to be given initial values like

```
COUNT = 0
```

before they are used in any expression. The process of setting variables to initial values is known as *initialisation*.

The set of statements

```
PRINT *, 'Input Value of X'
...
...
...
IF (COUNT .LT. N) GOTO 100
```

is known as *processing block*. It includes all *executable* statements such as input/output statements (READ, WRITE), assignment statements and control statements (such as IF). Note that the value of x^2 is evaluated and assigned to the variable F in this block. Similarly, the variables COUNT and X are incremented in this block. The statement

```
IF (COUNT .LT. N) GOTO 100
```

is known as a *control* statement.

This statement is responsible for creating a loop of operations and thereby making the function evaluated exactly N times. This is done with the help of the variable COUNT, usually known as a *counter*, which keeps counting the number of times the function has been evaluated.

Note that the statement

```
GOTO 100
```

directs the control to the statement

```
100 F = X * X
```

The number 100 is known as *statement number* or *statement label*. We need to use labels only to those statements to which the control is transferred from another part of the same program.

The last statement in our sample program is the END statement. This statement (which is a must in every FORTRAN program) serves two purposes:

1. it marks the end of source code during compilation
2. it terminates the execution of the program

> In earlier versions of FORTRAN, we need to use two statements:
>
> STOP — to stop the execution of the program
> END — to mark the physical end of the program.

FORTRAN 90 implements the same as follows:

```
END PROGRAM SAMPLE
```

Note the structure of the sample program. A FORTRAN program generally consists of a series of blocks of code in the following order:

Program name
Program description
Variable declaration
Initialisation of symbolic constants
Initialisation of variables
Executable statements
The END statement

FORTRAN requires certain coding formats to be followed. Table 5.1 lists them.

Table 5.1 FORTRAN 77 line format

Columns	*Use*
1	For typing the comment character.
1 - 5	For typing statement number.
6	For typing a non-zero FORTRAN character to indicate that the previous statement is continued.
7 - 72	For typing FORTRAN statement. The statement can begin anywhere in the region.
73 - 80	Not used (or used for typing line number).

FORTRAN supports the following major programming elements that have direct relevance to numerical computing discussed in this book.

1. constants
2. variables
3. input-output instructions
4. computational instructions
5. control instructions
6. documentation remarks
7. subprograms

The sample program has illustrated the use of all the first six elements. We shall discuss further details about them as well as the last element in this chapter.

5.3 FORTRAN CONSTANTS

Constants are the means by which numbers and characters are represented in a program. They are quantities that do not change. FORTRAN supports the following five built-in data types:

1. Integer type
2. Real type
3. Complex type

4. Logical type
5. Character type

Integer constants are numbers that do not contain decimal points (i.e. whole numbers). They can be positive, negative or zero. For examples

```
25    -10    0    +123
```

Real constants are numbers containing decimal points. They may be expressed in *positional form* or *exponential form*. Examples:

```
12.5        -1.756    0.0    5.    (Positional form)
.23 E+09    15E3      -2.3E -5    (Exponential form)
```

The exponential form (also known as *scientific form* or *floating point form*) is used where very large or very small numbers are to be written but not all digits need to be represented. (Number of significant digits depends on the computer.)

Complex constants are ordered pairs of real numbers, separated by commas and enclosed in parentheses, like (a, b). Examples:

```
(3.0, 4.0)    (-1.0, 0.92E2)    (1.2 E-2, 4.1 E1)
```

The first number is called the *real part* and the second is called the *imaginary part* of the complex number. (Complex numbers are usually written in $a + jb$ notation in mathematics.)

Logical constants are data that are used to represent the two truth values "true" and "false". Therefore, there are only two logical constants which are written as

```
.TRUE.
.FALSE.
```

Character constants represent a string of characters enclosed in apostrophes (single-quotes). Examples:

```
'John'    'January 26'    'NEW DELHI 20'    '123'
```

In FORTRAN 90, we may also use double quotes.

4.4 FORTRAN VARIABLES

Variables represent quantities that can change in value. In FORTRAN, they indicate storage locations where the values are stored. These values can be changed whenever required.

A variable name may consist of one to six characters, chosen from the letters A through Z and 0 through 9, the first of which must be a letter. Examples:

```
ALPHA    X1    SUM    NAME
```

FORTRAN 90 permits names with a length of 31 characters and also allows the use of underscore character.

> We can have longer names in FORTRAN 90. Examples:
>
> DISTANCE_TRAVELLED AVERAGE_HEIGHT
>
> This facilitates to create more meaningful names.

All variables must be declared for their storage types corresponding to the five data types discussed in section 5.3, namely, integer, real, complex, logical and character. Examples:

```
REAL NUMBER, SUM, X1
INTEGER TOTAL, COUNT, Y
COMPLEX ROOT1, Z
LOGICAL PACKED, L
CHARACTER * 20 NAME, CITY
```

The variables NAME and CITY can hold up to 20 characters. We can also increase the number of significant digits held in a real type variable by declaring it a "double precision" variable as follows:

```
DOUBLE PRECISION SUM, X1
```

> *Declaring a variable creates a storage location of appropriate type but it does not store any initial value. It contains some unknown bit pattern stored previously, i.e. the variable contains garbage.*

Any variable that is not declared explicitly for its type assumes default (*implicit*) type as follows:

Names beginning with	*Type*
Any letter I through N	Integer
Any other letter	Real

5.5 SUBSCRIPTED VARIABLES

FORTRAN variables can have subscripts to store a set of related values in one-dimensional vector or multidimensional matrices. A subscripted variable is called an *array*.

An array can be used to represent a collection of data of the same type and the subscripts can be used to access the individual data items. For instance, the third element of a one-dimensional array X is given by X(3). Examples of array variables are:

```
NAME(2, 10)    CITY(5)    GRADE(I)
```

We can use integer variables to represent subscripts and by assigning a suitable value to the subscript variables, we can access the desired element of the array.

All array variables must be declared for their type and size. Example:

```
REAL X(10)
```

or

```
REAL X(1:10)
```

Both these statements declare X as a one-dimensional array with elements numbered 1 to 10, which are of type real. The second form specifies the lower and upper bound of the subscript. In this form, the value of either bound may be positive, negative, or zero. The value of the upper bound should be greater than the value of the lower bound. Examples:

```
INTEGER   X(0:5), M(-10:20), N(-5:0)
REAL      P(5,5), VALUE(-3:3,5)
```

The second line declares P and VALUE as real type, two-dimensional arrays.

We may also declare type and size in separate statements like

```
REAL X, M
DIMENSION X(10), M(0: 10,10, 0:20)
```

Character arrays are declared as follows:

```
CHARACTER * 30 NAME(40)
```

or

```
CHARACTER * 30 NAME(1:40)
```

where the number 30 specifies the maximum number of characters to be stored in an array element.

In FORTRAN 90 arrays may be declared as follows:

```
REAL, DIMENSION(1:9) :: X,Y
CHARACTER(LEN = 30), DIMENSION(1:40) :: NAME
LOGICAL (-5:5) :: FOUND
CHARACTER, DIMENSION(10) :: CITY * 20
```

5.6 INPUT/OUTPUT STATEMENTS

Input/output statements are data transfer statements that are required in every program. FORTRAN supports two kinds of I/O statements

1. list-directed I/O statements
2. format directed I/O statements

We have already seen (in sample program 5.1) the use of list directed I/O statements. They are

```
READ *, X
PRINT *, F
```

The general form of these statements are:

```
READ *, v1 , v2 , ... , vn
PRINT *, v1 , v2 , ... , vn
```

where v_1, v_2, ..., v_n are data items. READ * reads input data from the standard input device, usually the keyboard, and assigns them to the

variables in the list. PRINT * outputs the values of data items in the list on the standard output device, usually the screen. The data items v_1, v_2, ..., v_n should be valid variables in case of READ and may be variables or constants in case of PRINT. Examples:

```
READ *, A, B, COUNT
PRINT *, X, 'TOTAL', SUM, 40.75
```

READ * is usually used to provide input data interactively through the keyboard. The data should be entered with either a comma or one or more spaces between the items.

Format directed I/O statements are used when the data should be read or written using a specified format. The general form of format directed I/O statements are

```
READ(n1, n2) v1, v2, ..., vn
WRITE(n1, n2) v1, v2, ..., vn
```

where n_1 is the number assigned to the device giving input or receiving output and n_2 is the number of the FORMAT statement which specifies the format of input data or output values. The FORMAT statement (a non-executable statement) takes the following form:

```
n2 FORMAT (list of specifications separated by commas)
```

Examples:

```
READ(5, 100) X, Y              (Reads values from unit 5)
WRITE(6, 200) X, Y, SUM        (Writes values to unit 6)
```

The unit may refer to keyboard, screen, printer, disk drive, and so on. If we are using only the standard devices as specified by the computer system, then we can use the following forms:

```
READ(*, 100) X, Y
WRITE(*, 200) X, Y, SUM
```

The FORMAT statements may look like

```
100 FORMAT (I5, F5.2)
200 FORMAT (I10, F7.2, F10.2)
```

The letter I indicates that the number to be handled is integer and F indicates that the number is floating point type. For more details about format specifications, you must consult the manual.

We may also give initial values to variables using the DATA statement as follows:

```
DATA X, Y / 25, 7.25 /
```

This statement assigns 25 to X and 7.25 to Y.

3.7 COMPUTATIONS

FORTRAN was specially designed to evaluate complex mathematical

expressions. We can write a FORTRAN expression for a given mathematical expression and assign it to a variable using an assignment statement as follows:

```
v = expression
```

This statement directs the computer to replace the previous value of the variable on the left-hand side of the equals sign with the result of the expression on the right. The expressions are written using variables, constants and arithmetic operators (see examples shown in Table 5.2).

Table 5.2 FORTRAN expressions

Algebraic expression	*FORTRAN expression*
$a = x + \frac{y}{z} - r^2$	`A = X + Y/Z - R ** 2`
$b = \frac{x + y}{z} + rt$	`B = (X + Y)/Z + R * T`
$c = (xy)\,(z + 2)$	`C = (X * Y) * (Z + 2.0)`

The following are the accepted arithmetic operators in FORTRAN:

- `+` Addition
- `-` Subtraction or Unary minus
- `/` Division
- `*` Multiplication
- `**` Exponentation

According to the precedence rule

1. all exponents are performed first, all multiplications and divisions next, and all additions and subtractions last,
2. for the same precedence, the operations are performed from left to right, and
3. when parentheses are used, the expressions are evaluated from innermost to outermost parentheses (using the same precedence rule each time)

In numerical computations, we often come across an assignment statement of the type

```
SUM = SUM + N
```

This means, replace the "old value" of `SUM` by the "new value".

Mixed-Mode Expressions

It is possible to combine integer, real and double precision quantities using these arithmetic operations. Expressions involving different types of numeric operands are called *mixed-mode expressions* and are evaluated as shown in Table 5.3.

Table 5.3 Mixed-Mode Evaluation

Mixed-mode expression	*Evaluation*	*Result type*
integer *op* real	Convert the integer to the corresponding real value and evaluate the expression.	Real
integer *op* double precision	Convert the integer to the corresponding double precision value and evaluate the expression.	Double precision
real *op* double precision	Extend the real to a double precision value (by adding zeros) and evaluate the expression.	Double precision

5.8 CONTROL OF EXECUTION

Control of execution means the transfer of execution from one point to another in the same program, depending on the conditions of certain variables. This may involve a *forward jump* thus skipping a block of statements, or a *backward jump* thus repeating the execution of a block of statements. This is known as *conditional execution* of statements. Examples of such conditional execution are:

1. If the value is negative, skip the following four statements.
2. If the item is the last one, go to the end.
3. Execute the following ten lines 100 times.
4. Evaluate the following statement until a given condition is satisfied.

FORTRAN contains two central structures which could be used to implement such conditional execution of statements. They are

1. IF-ELSE structure
2. DO-WHILE structure

Block IF-ELSE Structure

The block IF-ELSE structure (also known as *selection* structure) consists of a logical expression that tests for a condition or a relation followed by two alternative paths for the execution to follow. Depending on the test results, one of the paths is executed and the other is skipped. This is

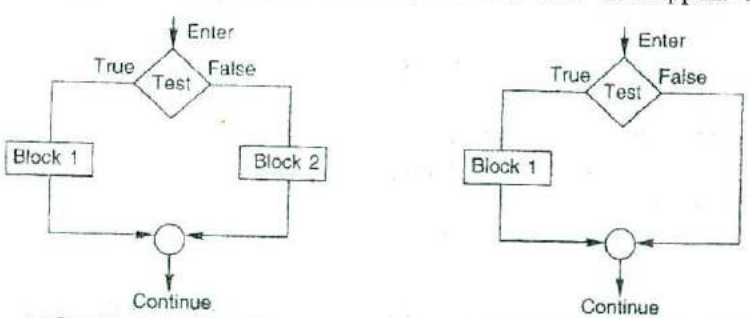
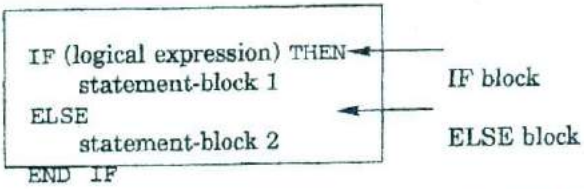


Fig. 5.1 Flow chart of IF-ELSE structure

illustrated in Fig. 5.1.
The FORTRAN statement to code a block IF-ELSE structure takes the form:



The statement blocks may contain zero or more statements. If the logical expression is true, the program executes statement-block 1 and then goes to the statement next to the `END IF` statement; if the logical expression is false, the program executes statement-block 2 (skipping statement-block 1) and then goes to the statement next to the `END IF`.

Relational Expressions

Relational expressions are meant for comparing the values of two arithmetic expressions and have logical values `.TRUE.` or `.FALSE.` as results. Arithmetic expressions may contain single variable, simple constant, intrinsic function, or a complex expression. In numerical computing, we often want our programs to test for certain relationships and make decisions based on the outcomes. We may use the *relational operators* given in Table 5.4 for comparing the expressions.

Table 5.4 Relational operators

Operator	*Meaning*
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Examples of rational operators are

```
1. IF(X .LT. Y) THEN
       PRINT * 'Small is', X
   ELSE
       PRINT * 'Small is', Y
   END IF
2. IF(TOTAL .GT. 1000) THEN
       TAX = 0.15 * TOTAL
   ELSE
       TAX = 0.10 * TOTAL
   END IF
       PRINT * 'GRAND_TOTAL = ', TOTAL + TAX
```

```
3. IF(C - D .GE. A - B) THEN
       X = C - D
   ELSE
       X = A - B
   END IF
```

> *When arithmetic expressions are used along with the relational operators, arithmetic expressions are evaluated first and then the results are compared.*

Logical Expressions

In some cases, we may need to make more than one comparison. It is possible to combine two relational expressions using the following logical operators:

.AND.	Both relations are true
.OR.	One or both of the relations are true
.NOT.	Opposite is true

Such expressions are known as *logical expressions.*
Examples of logical expressions are:

```
1. IF(SUM .GT. 100 .OR. N .GT. 20) THEN
       ...
       ...
   ELSE
       ...
       ...
   END IF
2. IF(AGE .LT. 30 .AND. DEGREE .EQ. 'ME') THEN
       ...
       ...
   ELSE
       ...
       ...
   END IF
```

FORTRAN permits nesting of IF-ELSE blocks. That is, we can place an IF-THEN-ELSE code within an IF block or ELSE block.

> *Warning!*
> Be careful when comparing real values. They are never exact!

> We may also use the following relational operators in FORTRAN 90.
> < Less than
> <= Less than or equal to
> = = Equal to

```
/= Not equal to
>= Greater than or equal to
 > Greater than
```

DO-WHILE Structure

The DO-WHILE structure (also known as *looping* structure) performs a set of operations repeatedly while a certain condition is true. When the condition is not true, the repetition ceases. This kind of structure is implemented in FORTRAN by the DO statement. The general format of DO statement is:

```
DO n i = e1, e2, e3
   ...
   ...   <-- Body of the loop
   ...
n CONTINUE
```

where

n	number of the last statement in the loop
i	loop control variable
e_1	initial value of the control variable
e_2	final value of the control variable
e_3	increment value.

The control variable i may be a real or integer variable. The parameters e_1, e_2, and e_3 may be real or integer variables (or expressions or constants).

The default value of e_3 is 1. The logic of DO loop is as follows:

1. initialise the loop control variable to the initial value e_1
2. test to see if the value of loop control variable is less than or equal to the final value e_2. If it is true, continue the loop; otherwise exit the loop
3. execute the body of the loop
4. increment the loop control variable by e_3
5. go back to step 2 (beginning of the loop)

This can be written in pseudocode form as follows:

```
i = e1
DO WHILE i <= e2
     execute statements
     i = i + e3
END DO
```

Figure 5.2 shows a flow chart showing the execution of the DO structure. The number of times the loop is executed (unless terminated by an EXIT statement) is given by the formula

$$m = \left[\frac{e_2 - e_1 + e_3}{e_3} \right]$$

$[x]$ denotes the greatest integer less than or equal to x.

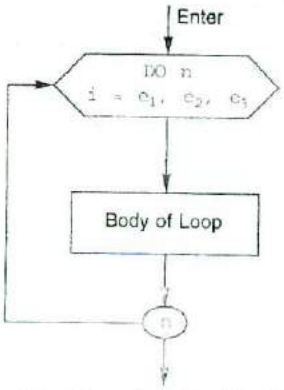


Fig. 6.2 Flow chart for the DO loop

Examples of DO loop are

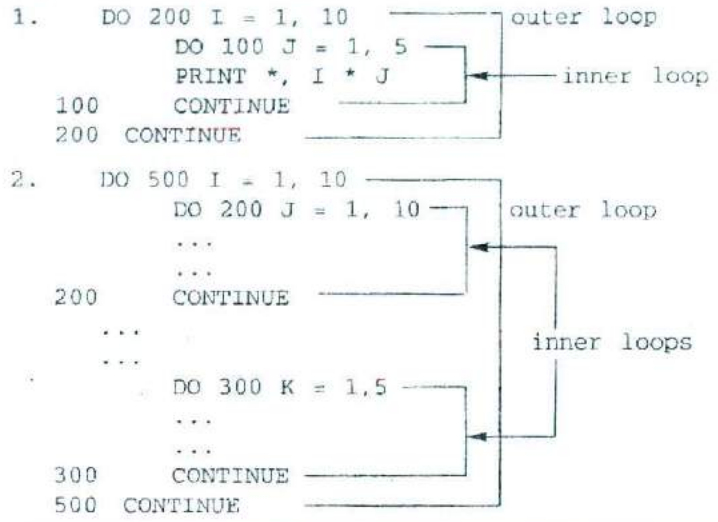
```
1.      DO 10 P = X/Y, Y5, Z/10.0
        ...
        ...
   10 CONTINUE
2.      DO 20 I = -4, 10, 0.25
        ...
        ...
   20 CONTINUE
3.      DO 30 N = 2, 20
        ...
        ...
   30 CONTINUE
4.      DO 40 J = 1, 100
        ...
        ...
        IF (...) GOTO 50   (Exit from the loop)
   40 CONTINUE
   50 ...
```

Warning !

Avoid the use of real variables for DO loop parameters. They cause roundoff errors and, therefore, cannot always guarantee the correct number of loop executions.

A DO loop can contain DO loops within its range. This is known as *nesting*. When nesting DO loops, the inner loop must be entirely contained within the range of the outer loop.

Examples of nesting DO loops are



The general form of DO structure in FORTRAN 90 is:

```
DO    loop control
      block of statements
END DO
```

This is implemented in two forms:

Form 1

```
DO i = e1, e2, e3
   ...
   ...
END DO
```

Form 2

```
1.  DO
       ...
       ...
       IF (...) EXIT
       ...
    END DO
    ...
2.  DO
       ...
       ...
       IF (...) CYCLE
       ...
       ...
    END DO
```

Leave the loop

Go to the beginning

5.9 SUBPROGRAMS

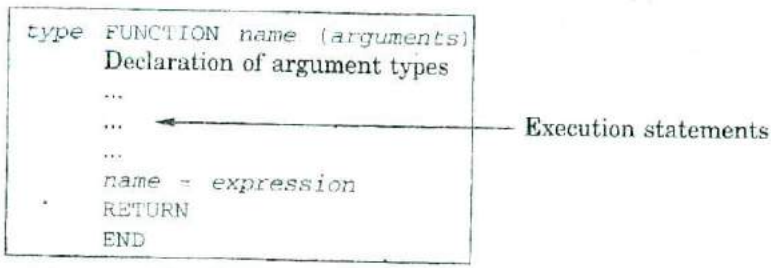
One of the features of any modern programming language is the provision for *subprograms*. A subprogram is a separate program unit that can be called into operation by other programs. Subprograms are heavily used in numerical computing for tasks such as evaluation of a function, matrix multiplication, sorting, reading a table of values, printing a report, etc.

The concept of subprograms allows us to break a complex problem into subtasks so that we may develop subprograms and later integrate them into a single program known as *driver* or *main* program. These subprograms can be independently designed, coded, and tested. Subprograms are usually called *modules* and the programming approach using modules is called *modular programming*.

FORTRAN supports two kinds of subprograms, namely, *functions* and *subroutines*. A function subprogram returns a single value to the calling program while a subroutine subprogram can compute and return several values.

Function Subprograms

A function subprogram (or simply a function) is an independent program unit written to compute and return a single value. It takes the following form:



where *type* specifies the type of the function value that is being returned and *arguments* are *dummy* variables that must be declared for their type inside the function. They may vary in number from zero to many. There should be at least one statement of the form

```
name = expression
```

which assigns a value of appropriate type to the function name, which is in turn returned to the calling program.

A function can be called as follows:

```
variable = name (arguments)
```

When the function is called, the values of the arguments in the calling statement are assigned to the corresponding arguments in the function header. The arguments, therefore, must agree in order, number and type. An argument may be a variable name, an array name, or a subprogram name. Example:

```
PROGRAM MAIN
REAL A, B, R, MUL  <-- MUL declared in main
READ * A, B
R = MUL(A, B)      <-- Calling MUL
PRINT * , R
END
-------------------------------
REAL FUNCTION MUL(X, Y)  <-- MUL defined
REAL X, Y
MUL = X * Y
RETURN
END
```

When an array is passed as an argument, then its corresponding dummy argument should be an array variable and its size must be declared properly. Note that a function may be called and used in an expression, like any other variable. Example:

```
R = A * MUL (A, B)
```

Subroutine Subprogram

A subroutine, unlike a function which always returns only one value, can return many values (or no values). Therefore, we use a subroutine when either several values are to be computed and returned or no values are to be returned (such as printing the values of some variables). The general structure of a subroutine is:

```
SUBROUTINE name (arguments)
Declaration of arguments
...
...                <-- Execution statements
...
RETURN
END
```

where *name* is the subroutine name and *arguments* are *dummy* variables that must be declared for their type. When subroutine has no arguments, the parentheses are omitted (note that in case of function, parentheses are necessary even if there are no arguments). The outputs of subroutine are returned to the calling program by means of the arguments.

A subroutine can be invoked using the CALL statement as follows:

```
CALL name (arguments)
     or
CALL name
```

The actual arguments in the calling statement must agree in a one-to-one manner with the order and type of the arguments in the subroutine. Example:

```
PROGRAM MAIN
REAL A, B, R
READ * , A , B
CALL MUL (A, B, R)
PRINT *, R
END

SUBROUTINE MUL(X, Y, XY)
REAL X, Y, XY
XY = X * Y
RETURN
END
```

The calling program assigns the values of A and B to the variables X and Y in the subroutine which in turn assigns the value of XY (computed in the subroutine) to the variable R. Compare this with the function subprogram.

Note that the variables that are not passed as arguments may be passed to the subroutine using a COMMON statement.

FORTRAN 90 greatly extends the power of function subprograms by allowing the result to be an array or structure. Function subprograms are designed as follows:

```
FUNCTION name(arguments) RESULT (result -
                                 variable)
Declaration of arguments and result-variable
...
...
result-variable = expression
END FUNCTION name
```

Instead of function name, the result-variable is assigned the value that is to be returned to the calling function. The result-variable is a variable name that has been placed like a function argument with the RESULT keyword, immediately after the function name. Both the arguments and the result-variable are declared for their types.

Note that, in FORTRAN 90, all programs and subprograms use the name of the program or subprogram in the END statement as follows:

```
END FUNCTION F
END SUBROUTINE SWAP
END PROGRAM SORT
```

FORTRAN 90 also includes features such as optional arguments, keyword-identified arguments and array sizes which are very powerful compared to FORTRAN 77. These features must be used wherever possible.

FORTRAN allows us to write out a formula for a function and define it using the assignment statement inside the program itself (instead of using an "external" function subprogram). Since such functions are "one-line" functions, they are called *statement functions*. A statement function is defined as follows:

Function-name (*arguments*) = expression

where *expression* is the FORTRAN expression of the formula (or function) to be evaluated and *arguments* is a list of variables used in the expression. The arguments are simple integer or real variables. Examples:

```
AREA  (R)          = 3.1416 * R * R
VALUE (P, R)       = P * (1.0 + R) ** N
POLY  (X, Y, M, N) = X ** M + Y ** N
```

A variable which appears in the expression but is not defined as an argument is called the *parameter* of the function. Values of such variables should be defined before using the function.

The function can be used in any subsequent lines of the program by writing the name of the function with actual arguments, like

```
CIRCLE  = AREA(X)
FVALUE  = VALUE(AMOUNT, INTEREST)
POLY1   = POLY (A, 2, B, 2)
RING    = AREA(X1) - AREA(X2)
```

Note that the functions can be used on the right side like any other variables. The actual arguments may be variables or constants (or even expressions). However, they must agree in number order and type with the dummy arguments in the function definition statement.

A statement function may use other statement functions if they are defined before it. Like function subprograms, the statement functions must be declared for their type in the program and defined after all declarations, but before the first executable statement.

5.10 INTRINSIC FUNCTIONS

In numerical computing, we use mathematical functions like logarithm, square root, absolute value, sine, etc., very frequently. FORTRAN supports a library of such functions which can be invoked in our programs. Since these functions are part of FORTRAN, they are also called *intrinsic* or *built-in functions*. An intrinsic function can be invoked by simply typing the name of the function followed by the arguments enclosed in parentheses. Example:

```
ABS(X)    COS(THETA)    SQRT(X * X + Y * Y)
```

The most commonly used intrinsic mathematical functions are summarised in Table 5.5. When using any of these functions, it is a good practice to declare them using the INTRINSIC statement in the declaration section.

Table 5.5 Commonly used mathematical functions

Function	*Description*
ABS(x)	Absolute value
ACOS(x)	Arccosine (result in radians)
ASIN(x)	Arcsine (result in radians)
ATAN(x)	Arctangent (result in radians)
ATAN2(x_1, x_2)	Arctangent of x_1/x_2 (result in radians)
COS(x)	Cosine (x in radians)
COSH(x)	Hyperbolic cosine
DBLE(x)	Conversion to double precision real
EXP(x)	Power of e
INT(x)	Truncation to integer
LOG(x)	Natural logarithm (base e)
LOG10(x)	Common logarithm (base 10)
MAX(x_1, x_2, ...)	Maximum value
MIN(x_1, x_2, ...)	Minimum value
MOD(x_1, x_2)	Remainder of division x_1, x_2 (e.g. MOD(5,3) is 2)
NINT(x)	Conversion to nearest integer
REAL(x)	Conversion to single-precision real
SIN(x)	Sine (x in radians)
SINH(x)	Hyperbolic sine
SQRT(x)	Square root $x \geq 0.0$
TAN(x)	Tangent (x in radians)
TANH(x)	Hyperbolic tangent

5.11 DEBUGGING, TESTING AND DOCUMENTATION

Errors in programming are common. Therefore, a program must be tested for any errors before it is used. Errors in computer code are called *bugs* and the process of correcting them is called *debugging*.

The program may be grammatically error free but may produce wrong results. Sometimes, a program may produce correct results for one set of data and wrong results for another set. Such errors are due to improper

program logic and are therefore known as *logic errors* or *run-time errors*. It is a good practice to test the program for all possible range and combination of data.

Documentation is a most important but often neglected activity by the programmers. Documentation provides all details about the program intent, its variables and other requirements that allow the users to immediately understand and implement the program more easily.

Documentation includes two parts – internal documentation and external documentation. Internal documentation means the use of explanatory remarks throughout the program, which describe how various parts of the program work. This is very important from the maintenance point of view.

External documentation includes instructions to the users on how to implement the program and what actions should be taken in certain special circumstances. Such a document is called *user manual*.

5.12 SUMMARY

We presented an overview of FORTRAN 77 in this Chapter. We discussed briefly the following features that are frequently used in developing numerical computing software:

- various categories of data types used to represent numbers and characters
- rules of defining variable names and creating storage space for them
- creation and use of subscripted variables to represent tables of data
- input/output statements required to read data values and print results
- operators used for evaluating mathematical and logical expressions
- control structures supported by FORTRAN 77
- design and use of subprograms in building a large application program

We have also highlighted the FORTRAN 90 features wherever applicable.

Key Terms

Algebraic expression
Arguments
Arithmetic operators
Array
Assignment statement
Backward jump
Bugs
Built-in functions
Calling program
Calling statement
Logic errors
Logical constants
Logical expression
Logical operators
Logical type
Looping structure
Main program
Mixed-mode expression
Modular programming
Modules

(Contd.)

(Contd.)

Character constants	*Multidimensional array*
Character type	*Named constant*
Comment line	*Nesting*
Complex constants	*Non-executable statement*
Complex type	*One-dimensional array*
Conditional execution	*Operators*
Constants	*Outer loop*
Control statement	*Output statement*
Control variable	*Parameter statement*
Counter	*Positional form*
Debugging	*Precedence rule*
Dimension	*Processing block*
DO loop	*Program statement*
DO..WHILE structure	*Program unit header*
Documentation	*Real constants*
Double precision	*Real type*
Driver program	*Relational operators*
Dummy variables	*Run-time errors*
Executable statement	*Size*
Exponential form	*Statement functions*
Expressions	*Statement label*
Flow chart	*Statement number*
Format-directed I/O statement	*Subprogram*
FORTRAN expression	*Subroutine*
Forward jump	*Subscripted variable*
Functions	*Subscripts*
IF..ELSE structure	*Symbolic constant*
Initialisation	*Testing*
Inner loop	*Two-dimensional array*
Input statement	*Type declaration*
Integer constants	*User manual*
Integer type	*Variable declaration*
Intrinsic functions	*Variables*
List-directed I/O statement	

REVIEW QUESTIONS

1. What are FORTRAN constants?
2. What are logical constants? Where are they used?
3. When do we use the exponential form to represent real numbers?
4. What are variables? State the rules of naming variables?
5. What is an array? When do we use arrays in computing?
6. What is meant by declaration of variables? How are array variables declared in FORTRAN?
7. Describe the actions of the following statements:
   (a) READ *
   (b) PRINT *

(c) READ (*, 100)
(d) WRITE (*, 200)
(e) WRITE (6, 200)
(f) WRITE (*,*)

8. What is the function of a FORMAT statement? Give an example.
9. How are DATA statements used to provide values to variables?
10. State the hierarchy of operations followed by FORTRAN in evaluating expressions.
11. List FORTRAN statements that are used to implement conditional execution statements. How are they different in terms of implementation?
12. Give two examples of each of the following expressions:
    (a) Relational expression
    (b) Logical expression
    (c) Mixed-mode expression
    Is there any special caution to be exercised in writing these expressions? Explain.
13. Why should we avoid the use of real variables as DO loop parameters?
14. What is nesting? When do we need to use nesting in numerical computing?
15. What are subprograms? How are they used in program development?
16. Distinguish between the function subprogram and subroutine subprogram.
17. What is a statement function? How is it different from function subprogram?
18. Give at least two examples of using statement functions in numerical computing?
19. What is testing? How is it different from debugging?
20. Describe the importance of documentation for programmers and program users.

REVIEW EXERCISES

1. Which of the following are illegal FORTRAN names? Why?
    (a) TOTAL (b) PART - 1
    (c) % MARK (d) REAL
    (e) A+ (f) 3M
    (g) X23 (h) X AXIS
2. Classify each of the following constants as an integer constant or a real constant. If they are neither, state the reasons.
    (a) 123 (b) 123 (c) '123'
    (d) 12 + 3 (e) −12.3 (f) +12
    (g) 12/3 (h) 1,234 (i) FOUR
    (j) $6.5 (k) 0.12E3 (l) −1.5E02
    (m) E5 (n) 5E.5 (o) 25.3−

3. Which of the following are legal character constants?
   (a) 'A'
   (b) TOTAL'
   (c) '1.23'
   (d) 'TOTA MARKS'
   (e) 'NEWTON'S LAW'
   (f) 'A.B.RAM'
4. Write the FORTRAN expressions for the following:

   (a) $\frac{a}{c+d} \times \frac{e}{f}$

   (b) $ax^2y + bxy^2 + c$

   (c) $\frac{a+b}{c} - b(d-e)(a \times d)^2$

   (d) $\left(c + \frac{x}{y}\right) n - 1$

5. Find the values of M and A when each of the following arithmetic statements is executed.
   (a) A = 2.5 + 3.0 ** 2/3.0
   (b) A = 2.0 ** 2 + 3.0 ** 2 – 4.0 * 3.0/2.0
   (c) A = 16/2 ** 3 + 5/2 * (2 * (6 – 4))
   (d) M = (9/4)/(3/2)
   (e) M = 4 ** 2 * (2/3)
6. Identify errors in the following assignment statements:
   (a) X = SUM/N
   (b) A = COS(X) + FLOAT(N)
   (c) N = (X/Y) * (X/Z)
   (d) M–1 = (A + B + C)/D
   (e) W = X ** –2 + SQRT(N)
   (f) D = P * ALOG (–3.5)
7. Following statements contain mixed mode expressions. Correct them using
   (a) the type declaration statements
   (b) the type conversion functions
   (c) none of the above
   (1) AREA = LENGTH * WIDTH
   (2) FORCE = MASS * ACCEL
   (3) DIST = SQRT(N ** 2)
8. Using library functions construct FORTRAN statements for the following:

   (a) $A = \sqrt{s(s-a)(s-b)(s-c)}$

   (b) $c = \sqrt{a^2 + b^2 - 2ab\cos(x)}$

   (c) $z = \sin\left|\frac{x-y}{x+y}\right|$

   (d) $f = \frac{1}{2\pi} \times e - \frac{(x+y)^2}{2}$

9. Given below are three sets of expressions. The two expressions in each set, though appear to be identical, do not produce the same results for certain values of integer variables *I*, *J* and *K* and the

real variable X. Identify those values for which the two expressions are not equal.

(a) $(I + J) / K$ and $I/K + J/K$
(b) $I * (J/K)$ and $I * J/K$
(c) $X * I/J$ and $X * (I/J)$

10. Write a program to read two values from the keyboard and to display their sum along with their values on the screen as follows:
    (a) All the three values in one line
    (b) Values one below the other in separate lines
11. Given the lengths of the two sides of a right triangle, write a program to do the following:
    (a) To read the values of two sides from the keyboard
    (b) To calculate the area of the triangle (one-half the product of two sides)
    (c) To calculate the length of hypotenuse (square root of the sums of squares of the sides) and
    (d) To print the results with labels like
    Area =
    Hypotenuse =
12. Write a program to evaluate the expression

$$w = \frac{(x + y)^2 - 2xy - y^2}{x^2}$$

and print the results for the following values of x and y:

(a) $x = 0.05$ and $y = 900$
(b) $x = 0.005$ and $y = 900$
(c) $x = 0.002$ and $y = 900$

Are the results different?
Note that the above expression, on simplification, reduces to $w = 1$. If the results are different, why?

13. Declare the variables to be double precision in the above program and see the results. Is there a difference? If so, why?
14. Write a program which requests the user to type in a number. If the number is four digit long or longer, then the computer should provide a message that the number is too large. If it is two digit long or shorter, then the message should be that the number is too small. Otherwise, print a message

WELL DONE, WE THINK ALIKE

15. Write a program to solve the quadratic equation

$$ax^2 + bx + c = 0$$

by using the formula

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The program should display the real roots or message indicating that there are no real roots (if $b^2 - 4ac$ is negative).

16. Write a program to read the marks obtained by 25 students in a class and count the number of students with marks in the following range:
    (a) 0 to 39 (Fail)
    (b) 40 to 59 (Pass)
    (c) 60 and above (Pass with I class)
17. Write programs to evaluate the following functions to 0.0001 per cent accuracy.

    (a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$

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

18. Write a program to read a set of numbers, count them, and find and print the largest and smallest numbers in the list and their positions in the list.
19. Write a program to calculate and print the mean, variance, and standard deviation of a set of N numbers.

    $$\text{Mean} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

    $$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \frac{1}{N^2}\left(\sum_{i=1}^{N} x_i\right)^2$$

    $$\text{Standard Deviation} = \sqrt{\text{Variance}}$$

20. Write a program to find the largest element of a given matrix and print out the value with location details.
21. Write a subprogram to evaluate the factorial of a number which is given by

    $$n\,! = n(n-1)(n-2) \ldots 1$$

    Using this subprogram write a main program to calculate the binomial coefficient

    $$b = \frac{n!}{(n-r)!\,r!}$$

    This gives the number of combinations of n objects take r at a time.
22. Write a subroutine subprogram that will interchange the values of two variables when called.
23. Write a menu-driven program that allows the user to use one of the following options:
    (a) To convert miles to kilometres
    (b) To convert feet to metres
    (c) To convert degrees Fahrenheit to degree Celsius

    Note: 1 mile = 1.60935 kilometres
    1 foot = 0.3048 metres
    C = 5/9 (F – 32)

24. Rewrite the following program so that it will take minimum time for execution.

```
      READ (5, 111) W, C1, C2, G, R, V
      CRT1 = W * C1 * (1.0 + G * R) * V
      CRT2 = W * (C2 + C1 * (1.0 + G * R)) * V
      CAP = C2 + C1 * (1.0 + G * R)
      WRITE (6, 222) CRT1, CRT2, CAP
  111 FORMAT (6F10.2)
  222 FORMAT (1Hb, 2F10.2, 5X, F10.5)
      STOP
      END
```

25. Improve the following program segments:

1.
```
      READ (5, 11) X, Y
      DO 50 I = 1, 100
      AI = (X * X + Y * Y)/AI
   50 WRITE (6, 22) A
   11 FORMAT (2F5.2)
   22 FORMAT (1Hb, F10.5)
      STOP
      END
```

2.
```
      . . .
      . . .
      DO 50 I = 1, 50
      . . .
      . . .
      X1 = X + Y/B * C
      X2 = Z + Y/B * C
      . . .
      . . .
      CONTINUE
      . . .
      . . .
```