1-1-1988

Computer Mathematics using Pascal, 2nd Edition

Stephen J. Sugden
Bond University, ssugden@bond.edu.au

John Simmond

1 COMPUTERS & INFORMATION REPRESENTATION

1.1 The Structure Of A Computer

What is a computer system? In time you will begin to appreciate that from the practical point of view of the user, it is just a collection of resources. Physically, a modern digital computer includes very rapid electronic data processing devices (integrated circuits or "chips"), stored programs of machine instructions, and sets of data (either numbers or words). Under the control of the stored program, data is automatically read, processed and ultimately the results written back out to some medium, often a humanly readable medium such as paper or a video (television) screen. Frequently the medium for information or data written out from a computer (called "output") is magnetic disk. Since magnetic media (floppy disks and tapes) are specifically designed to retain their contents indefinitely, the stored data on a computer disk or tape may be re-read back by the computer at some later time. This allows computers of course to store potentially very large volumes of information and retrieve it for later use.

We shall see later that computer resources are broadly classified into the categories **hardware** and **software.** The term **hardware** simply refers to any physical part of a computer system. For example, a video display screen is a computer hardware item. The term **software** refers to computer **programs** of any kind. These very general terms of classification will be used throughout this book.

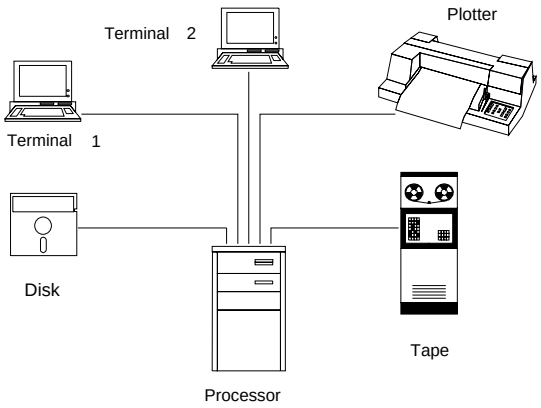


Figure 1.1 The Units of a Typical Medium Sized Computer.

Computers are also classified broadly according to their size, namely **mainframe computers, minicomputers,** and **microcomputers.** The distinction between each of these classes is somewhat arbitrary; a minicomputer to one person may be a "micro" to another. How do you define small, medium, and large cars, for example ? Traditionally, the term **mainframe** has been used to describe the very powerful computing machines used by government departments, banks, and other large organisations for the enormous volumes of data processing that they require. Payroll calculations, census records, savings bank transactions - including automatic teller machines (ATMs), are all very common applications for large mainframe computers. We present in Table 1.1 a comparison of typical characteristics for each of the three broad classes.

Let's now examine the physical components of a digital computer. To reiterate, any physical part of the computer system is referred to in general as **hardware.** If you can see it - it's hardware. By contrast, the term **software** is used to refer to **computer programs** of any type. Any form of information fed into a computer for processing is referred to as **input data,** or sometimes just **input**. This input data may represent the values to be used in a mathematical calculation to find the trajectory of a spacecraft, or perhaps the names and addresses of customers of a particular bank for storage and/or further processing. Computer input data enters the computer from the rest of the physical world in many and varied ways. The data processed by a computer will finally be communicated back to humans by the computer in any of a number of forms. Such data is referred to generally as **output**. Examples of computer output devices are printer, video screen, and magnetic disk drive. Operations which involve either input or output or both are usually labelled with the abbreviation **I/O**. Later on in this chapter, we consider the principal features of a number of **I/O** devices. Such devices are also termed **peripheral devices** since their connection is to the **central processing unit,** which is the prime component of a computer system.

Input and output may be in many forms and exist on many different media. Input may be derived from magnetic media such as **tape** or **disk,** or from punched cards, paper tape, keyboard, a bar-code reader in a supermarket, optical character recognition devices such as those found at many library loans desks, and from many other forms. Common output devices are the video screen, printing terminal, line-printer, graphical plotters and magnetic media (normally disk or tape). Figure 1.1 shows some of the functional units of a medium-sized computer.

	Mainframe Computer	**Mini-Computer**	**Micro-Computer**
Cost	$100,000 to ..	$10,000 to $500,000	Below $10,000
Manufacturing Technology	Large Scale Integration	Large Scale Integration	Large Scale Integration
Siting	Large Computer Room	Room or part of room	Office desk
Operating Staff	Yes. Maybe 24 hours.	Only if large	No
Specialist Programming Staff	EDP section	A few	Usually Not
Environmental Control	Yes (dust,heat)	Possibly not	No
Number of Users	100's	A few (maybe 10 or 20)	Single user

Table 1.1 A Comparison of Computer Types.

Before considering peripheral (I/O) devices, we first present a simple model of a computer, and then consider the processing unit and peripheral devices separately. A digital computer is an enormously complex device; made up of large numbers of electronic, electromechanical, and other components. It is useful therefore, to have a simplifying model so that we can gain some overall understanding of how this vastly complicated device functions. One way to view a computer is as a "black box" which accepts input data, processes it in predefined ways, and produces appropriate output. This simple model is illustrated in Figure 1.2.

Figure 1.2 A "Black Box" Model of a Computer

1.2 The Central Processing Unit & The Main Memory

First we take a look at the heart of the computer system - the Central Processing Unit (CPU). The CPU of a computer consists of an Arithmetic Logic Unit, a Control Unit and Main Memory. The vast bulk of all processing, i.e. calculations and data manipulations, is performed by this part of the computer. A simple diagram of the internal organization of a typical processor is given in Figure 1.3.

Nowadays the (central) processor in a computer is housed on one semiconductor **chip**. A more descriptive name for "chip" is **integrated circuit.** The processor chip is responsible for all arithmetic, comparison and logical operations required by the rest of the system. In earlier computers, the CPU occupied much more space; even the size of an entire room. It has been the development of the microprocessor chip during the 1970's that has given the great impetus to the

microcomputer revolution of this decade. Not only are the computers of today much smaller than ever before, but they are also much faster, much cheaper, consume far less power, and are capable of more complicated processing functions.

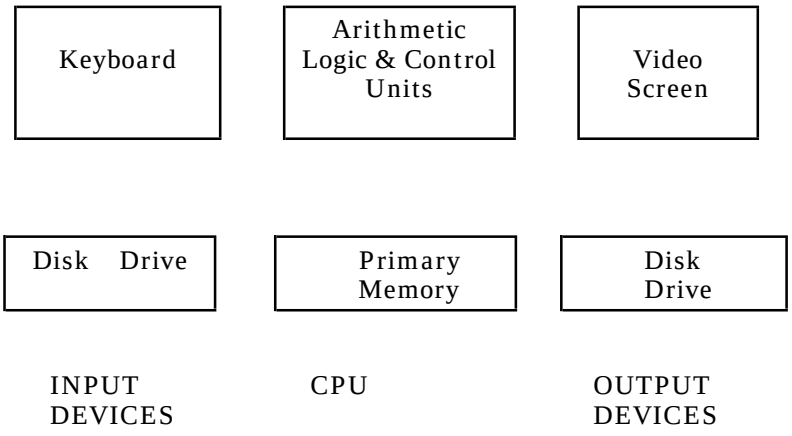


Figure 1.3 Inside The **Black Box.**

Closely coupled, in fact **directly connected,** to the CPU is the **main storage** or main memory; also called **primary memory** or RAM (Random-Access Memory). The main memory is used as an essential scratchpad or temporary storage area, both for data and for stored programs. It is very similar in function to the memory in your pocket calculator, although of much larger capacity. No calculations can proceed until the necessary data and instructions are available in main memory. The main memory may be viewed as consisting of a large array of **letterboxes** or pigeonholes, each having the capacity to store just one **character.** A character is just a single symbol on a computer keyboard, i.e. just a letter, digit, arithmetic symbol, punctuation symbol such as comma, or other, special symbol. The unit of storage which has the capacity to store just one character is called a **byte**.

Just as each house in a street has a letterbox with its address number on it, so each main memory location has a unique address associated with it. The CPU can directly access the contents of any location in essentially the same time as any other, just by specifying the address. This is what is meant by the term **random-access**. Figure 1.4 shows a simplified view of a part of the main memory of a computer.

P	**A**	**R**	**T**
0	1	2	3
	O	**F**	
4	5	6	7
M	**E**	**M**	**O**
8	9	10	11
R	**Y**		
12	13	14	15

Figure 1.4 A Portion of Main Memory

The sequential numbers at the bottom of each **cell** indicate the relevant **addresses** of the cells. As mentioned above, in most modern computers, each cell has a capacity of **one byte** (a unit of storage that can accommodate just 1 character). For example, location 8 above contains the character "M". However, if we wish to store numbers, then 4, 8 or even 16 cells may be required for each value. Such values normally occupy adjacent locations in the main memory. It is important to realise that the **interpretation** to be placed on the contents of a memory cell is entirely up to the program requiring the information.

Computer programs are obeyed as lists of instructions to the electronic circuitry (hardware). Temporary results of calculations are retained in the main memory for later use in the program currently running. For this reason, the primary memory is often said to be a **scratchpad** memory.

Most modern-day computer main memories (RAM's) are composed of a large number of semiconductor chips. These devices retain their contents only when power is applied. This type of memory is therefore termed **volatile** and any portion of its contents required for permanent storage must be saved onto some form of permanent memory (usually a magnetic medium such as magnetic disk) before the semiconductor RAM is reused by another program, or the power shut down.

Inside the Central Processing Unit (CPU)

The CPU contains a sub-unit known as the **control unit.** It is the responsibility of this device to direct the processing of the CPU and in particular to ensure that the correct sequencing of instructions is maintained.

The control unit contains small storage areas internally and these are of very high speed. They are known as **registers**. The two registers of greatest importance are the **instruction register** and the **program counter.** Since the program driving the computer is stored in main memory, instructions must be fetched from this storage area and loaded into a temporary store where they can be decoded and then carried out. Just as data items are encoded into **binary** for machine representation, so also are the program instructions.

Each instruction that the computer is capable of performing has its own **code**, and a computer program is just a list of these (coded) instructions, along with, of course, the data on which they are to operate. For example, we could design our own small computer with just four **operation codes,** or "opcodes" for short. For example, we may have code 1 for the operation of addition, 2 for subtraction, 3 for multiplication, and 4 for division. The values to be combined using the operator are called **operands.** Thus, if we write:

```
MOV   AX,4        ; store value 4 in register AX
MOV   BX,3        ; store value 3 in register BX
ADD   AX,BX       ; add contents of AX and BX and store result back in AX
```

then, in the last instruction, "ADD" is the operator and AX, BX are the operands. These three instructions form a small fragment of assembly-language code, which first places (**MOV**es) constants 4 and 3 into the registers **AX, BX** respectively, and then orders the computer to add the contents of the registers and finally place the result back into **AX** (the final destination of the result is implicit in the definition of the **ADD** instruction). A computer instruction will typically contain an opcode specifying **what** is to be done; one or more **operands** which either give the actual values to be processed or specify where in main memory to find them; and finally a specification as to where to place the result. As we have noted, in the example given above, it is implied that the result is stored in the AX register.

Different computers will have different **instruction formats;** typically consisting of several fields which represent the operation code, and addresses of operands and result. The temporary storage area which is used for holding a copy of the current instruction is the **instruction register.**

At any time, the control unit must know where the next instruction is coming from, i.e. which main memory location contains this instruction. It is the function of the **program counter register** (sometimes called the instruction pointer register) to hold the address in main memory of the next instruction. Once this location is found from the program counter, the relevant instruction may then be fetched from memory, loaded into the instruction register, decoded, and executed.

The machine instructions are generally stored in consecutive locations of main memory. Thus, in order to access the correct location for the next instruction, the program counter will be first incremented as part of the so-called **fetch-execute cycle.** This assumes of course that each instruction is the same length (in bytes). This is **not** the case for the **Intel** 8088/8086/80286 family of processors that are used in the **IBM PC** or compatibles.

1.3 Secondary Storage Devices

In order that we may save permanent, re-readable copies of our data and programs, some form of **mass-storage** is required. This normally takes the form of magnetic disk or tape. These magnetic media have the essential property that the information stored is retained even when the electrical power is removed.

Any form of computer memory which is specifically used for permanent or semi-permanent storage in addition to the main (primary) memory is termed **secondary storage** or **secondary memory.** Some books also refer to this kind of memory as **backing store** or **mass storage.** Secondary storage is in fact essential for any kind of practical computer system to operate. Separate **volumes** (units) of secondary storage may be kept either **online** or **offline**. (A secondary storage device, or any other peripheral device, is said to be **online** when electrically connected to the computer, and **offline** otherwise.)

As we have seen earlier, the vast majority of computer main memory does not retain its contents when the power is cut. Furthermore, main memory is relatively expensive, limited in capacity, and needs to be directly wired to the CPU. What is needed is a form of storage which is:

- Permanent
- Low-cost
- Potentially unlimited in capacity
- Relatively fast - but need not be as fast as main memory
- Reliable

Magnetic disks and tapes provide all of these requirements except that access to stored information on tapes is slow because of their sequential-access nature. Disks are superior to tapes in all respects, except that the cost of disks can be somewhat higher, although with the widespread use of floppy disks nowadays, prices have come down quite a lot (one dollar each for floppy disks in Australia; one third of this in the USA). These items are a very convenient form of secondary storage, and depending on physical size and density of data storage, are capable of storing in excess of one megabyte (1 million characters) of data. Also available are **hard disk** units for around $800 upwards for microcomputers. Storage space of 10 to 100 megabytes, or even more, is available on these devices. Hard disk data access times are much faster than for "floppies", since engineering tolerances can be made much finer (disk is rigid), and the entire unit is sealed against atmospheric and other contaminants.

The main advantage of disks over tapes is their ability to allow **random access** to the stored information. A simple illustration is provided by comparing an ordinary record player (random-access disk device) to a cassette player (sequential-access tape device). Any track on the disk can be located by the read device (pickup arm) in essentially the same time as any other track. This phenomenon is of course due to the physical shape of the disk. By contrast, any tape device is intrinsically **sequential access** since to find for example, any song on the cassette tape, we must sequentially, (i.e. in natural order) pass over all preceding songs. Access to stored information is therefore much slower for tapes than for disks. Once the desired access point is reached however, data transfer times for tapes and disks may be comparable.

A recent addition to the range of secondary storage devices is the **CD-ROM (Compact Disk Read-Only Memory)** optical storage device. It is essentially the same device as the audio compact disk (laser disk) player. With present technology, up to 1 gigabyte (1000 megabytes or 1,000,000,000 bytes) of information may be stored on a small disk comparable in size to an audio compact disk or 45 rpm record. As with the audio compact disk, to create the stored data, all information is digitally encoded to binary (0's and 1's) and a laser used to burn microscopic pits into the surface of the disk. To read the information back into a computer, a (non-destructive!) laser is used to sense the presence of the encoded binary (0's and 1's) by reflection of the laser beam. An entire encyclopaedia is presently available on this medium, and with appropriate software allow search times for any word or phrase in the multi-volume encyclopaedia of the order of only a few seconds. The only real disadvantage of this medium for general purpose use is that it is **read-only**, i.e. once recorded, the information cannot be altered. Of course, for some applications, this is an advantage: digital storage of recorded music, video programs, text of encyclopaedia, archive of legal documents, and so on. You may be able to imagine other applications in which large volumes of digital information need to be stored and which must not be alterable after being initially stored.

We conclude this section with Table 1.2 which gives a comparison of primary and secondary memory.

ATTRIBUTE	PRIMARY STORAGE	SECONDARY STORAGE
FUNCTION	Essential for storage of temporary or intermediate results and for use as a scratchpad by the CPU; program instructions also stored here.	Backing storage - used for permanent storage of programs and data. Usually magnetic media.
DATA RETENTION	Usually volatile, i.e. contents lost if the power is cut.	Permanent.
SPEED	Extremely fast.	Much slower than mainstore but still quite fast. Disks are faster than tapes.
CAPACITY	256K-640K bytes for a microcomputer; many megabytes for a mainframe.	On-line several gigabytes; off-line - unlimited.
COST/BYTE	Nowadays quite cheap.	Floppy disks very cheap - $1 for 360 kilobyte disk.
TYPES	Semiconductor chips; magnetic cores (older).	Magnetic disks, magnetic tapes; paper tape (older).

Table 1.2 A Comparison of Primary & Secondary Memory

1.4 Some Peripheral Devices

We now consider some peripheral devices in common use.

Terminals

A terminal is the usual and simplest way of accessing a multi-user computer system. It is connected via either dedicated cabling, or in the case of remote terminals, via telephone line, coaxial cable, or perhaps microwave radio link to the processing unit. Terminals are used for both input and output. Modern terminals include a keyboard and a visual display unit (VDU), which is similar to a television screen. Some, especially older terminals feature a printing device instead of the video screen. These devices are generally referred to as teleprinters or teletype machines.

Some of the more sophisticated terminals allow very high quality (termed "high-resolution") graphical displays to be constructed by the computer driving them. Most people have seen some of the very high quality computer graphics images generated or displayed by such devices. Quite a number of the modern movies, especially the science-fiction ones, contain some dazzling animated images and patterns generated by computer. Special display terminals and printers are required for such high-quality finished products.

Card Readers

Until the mid-1970's, the traditional input device for data processing systems was the card reader. The data are stored on punched cards, which are made of stiff paper with patterns of holes to encode the data. A set of cards for one program or a collection of data is called a **deck**. A card reader uses a set of photoelectric cells to determine the pattern of holes on the card. Fibre optics are sometimes used. Card readers can operate at approximately 300-1000 cards per minute.

Paper Tape

Except for older-style telex machines and similar equipment, paper tape is not often used nowadays. It has a number of tracks spanning the length of the tape. Holes are punched in these tracks in order to encode the data. Six, seven, or eight track tapes are commonly used.

Direct Source Readers

With punched cards and paper tape, data has to be copied from some other source document. To save time, and avoid errors, several methods of so-called **direct data input** have been devised. Methods for direct input include optical character recognition (OCR), magnetic ink character recognition, and use of bar-code reading devices.

Optical character recognition makes use of a data input device which can recognize printed or typed characters by an optical scanning process. Printed documents may therefore be input to a word processor without manual re-keying.

Magnetic ink character recognition is the process of reading characters which are printed in magnetic ink. It is almost exclusively used in the banking industry for the clearing of cheques.

Bar codes are a relatively new form of direct input strategy. A bar code consists of a number of vertical stripes in black ink on a white background. Characters are encoded by a combination of thick and thin stripes. A bar code reader interprets the pattern of stripes and produces the equivalent character code. A laser beam is used in the scanning process. At a supermarket checkout employing a bar-code reader, no keying of prices is required by the checkout operator. The customer receives an itemised list with a full description of his/her purchases since the computer can look up the prices and descriptive details once the bar code has been read. The store manager can know minute by minute the stock levels and sales item by item. Fast moving lines can be replenished and slow ones considered for deletion. Do you like waiting at a crowded supermarket checkout while a less-than-enthusiastic sales assistant searches the shelves for a price check ? Use of bar codes virtually eliminates this problem.

Voice Recognition Devices

Intensive research is taking place in this area. Currently, voice recognition systems are restricted to dealing with a relatively small number of words which have been previously "learned" by the system. These systems hold great potential for handicapped users of for example, word processing equipment.

Printers

These are the commonest form of **hardcopy** output devices. Many different types exist. Large **lineprinters** print an entire line at a time and are capable of speeds of 300-1200 lines per minute. Microcomputers often have small **character** printers, which according to their name, print one character at a time. Within the character printer group we have such varieties as **dot-matrix** printers, which form their characters from a number of small dots (much like a newspaper photograph or television screen image); **daisy-wheel** printers, which use a print-wheel shaped like a daisy with a character or two on each "petal"; and **ink-jet** printers, which form characters by spraying liquid, fast-drying ink onto the output medium (usually paper).

A very exciting, new kind of printer today is the **laser** printer. We can imagine these as operating like a modern, high-quality photocopier machine. Very-high quality images are possible and modern type-setting techniques will make use of these devices. An important advantage of the laser printer is that the print speed (typically 8 pages/minute) is independent of the complexity of the image. This means that high-quality graphics images can be printed in the same time as the equivalent pages of textual material. We must remember, however, that the time taken for the computer processor to **generate** a complex graphics image may be appreciable. A new use for computers, known as **Desktop Publishing** has sprung up, and some reasonably high quality documents (newsletters, bulletins, even entire books) can be produced with appropriate hardware and software. The final master copy for this textbook was produced using a word processing package called **Lotus Manuscript** and printed on an **Apple LaserWriter II** printer.

Microfilm

The main advantage of this technique is that it avoids the bother and expense of large quantities of printed pages. A page of output is photographed by a special camera. A page is represented by an area of approximately 0.5 cm x 0.5 cm. There are usually 100 pages per card. A microfilm reader is used to enlarge and project the image of a page onto a screen. You may have one in your school library. However with the advent of CD-ROM storage and retrieval devices this form of storage/archival has limited applicability in the future.

Digital Plotters

In engineering and architecture, computer output is commonly required in the form of drawings and maps. Digital pen-plotters have a pen whose motion across the surface of the output medium (usually paper or plastic) is controlled by a computer. Special software is required to control plotters and some are very slow. However, more modern types of plotters, such as ink-jet, electrostatic, and laser plotters can produce high-quality plots at respectable speeds.

Speech Synthesizers

Speech synthesis output is the counterpart of voice recognition input. Speech synthesizers work by storing a digitally-encoded form of a number of key sounds. Words are constructed by combining these **phonemes**, and directing the codes to a sound-generator circuit and then to a loudspeaker. Early versions of such devices had the typical machine-like "robot voice" sound about them, due primarily to lack of variation in intonation of phonemes (atomic components of speech) in spoken context. More modern systems have overcome this and other problems, and are rather impressive in some applications.

1.5 Information Representation

A digital computer represents all information ("data") in discrete units, i.e. numbers. An abacus is a good example of a digital computer. So are our own fingers and toes when used for counting. We may say that a digital computer counts. Remember that a computer can be regarded as a "black box" that:

ACCEPTS information, i.e. input data
PROCESSES it in a pre-defined way, and
REPORTS the results, i.e. produces output.

The Sequential Nature of Digital Computers

The modern digital computer is a **stored-program** device. The concept whereby the machine's memory not only holds the data for processing, but also the program of instructions, specifying the operations to be carried out on this data, was pioneered by the mathematicians John Von Neumann and Alan Turing. This allows for powerful, repetitive calculations to be performed automatically, (once the program has been stored), and accounts for much of the computer's versatility and power.

Computers execute the instructions given to them in a sequential manner, i.e. one after the other in discrete steps. Every translated, machine-language program consists of a finite number of well-defined, primitive instructions to the computer hardware. Their sequential execution is controlled internally by a special register called the program counter. Its function is to keep track of where the next instruction to be executed is to be found in main memory.

Sometimes the normal sequence of execution is interrupted and control is transferred back to a previous instruction, which may then be re-executed repeatedly, forming what is known as a **loop**. In this way, unlimited amounts of calculation may be expressed by a finite program. This is an essential feature of modern, digital computers.

The Deterministic Nature of Digital Computers

In broad terms, we say that digital computers process both textual and numeric information. These machines are very powerful and fast but are nevertheless completely mindless. They can only process their input data in preset, pre-defined, i.e. **programmed** ways.

The output of a computer program must be completely predictable, that is, the machine is **deterministic.** In other words, a given program run today must produce identical results if run tomorrow (assuming that the input data are the same). If you set your automatic dishwasher on a certain program cycle, you expect it to do the same thing every time. If it doesn't then it's time to call the serviceman.

Machine Representation of Information

All information stored and processed by a digital computer is represented by numeric **codes**. Information is **digitized**, in the sense that codewords consisting entirely of 0's and 1's (binary digits) are used to represent the original information.

At the character level, each symbol is represented by a sequence of 7 or 8 **binary digits**. (A binary digit is usually referred to by the contraction: **bit**.) In other words the upper and lower case alphabetical characters, the ten decimal digits, as well as a large number of special characters are all converted into numbers for convenience of internal machine representation.

Other modes of data representation exist, such as the coding of purely numeric quantities intended for arithmetic processing only, and we consider examples of this later in the chapter.

Number Representation Systems

Before we look at how numeric information is represented in a computer we need to consider some number representation systems. Firstly, let's clearly distinguish between the concepts of a **number** and a **numeral**. A number is a mathematical abstraction conveying some idea of **quantity.** It should be realized that **negative** or even **complex numbers** are no less "real" than the so-called natural

numbers, i.e. 1,2,3,4,5,.... and so on. These ideas are all abstractions of everyday concepts of quantity. All numbers other than natural numbers have been created by mathematicians in order to solve otherwise insoluble equations.

We distinguish here between the ideas of a number, which is a mathematical concept and the **representation** of a number, which is a concrete, usually visible, physical object usually known as a **numeral**.

The system of number representation with which we are most familiar is the **decimal system.** However, many other systems for representing numbers exist. Most people have heard of the **Roman system** of numerals. (Why aren't they called Roman numbers ?). For example, the Roman representation of the decimal 236 is CCXXXVI. In this system, the basic symbols (digits) are:

I , representing one
V , representing five
X , representing ten
L , representing fifty
C , representing one-hundred
D , representing five-hundred
M , representing one-thousand.

Of course, the decimal representations of these numbers are respectively 1,5,10,50,100,500,1000.

There are two very serious deficiencies with the Roman system of representation. The first relates to its very limited ability to compactly represent large numbers; the second to the inconsistent or context-dependent way in which use is made of the positions of the component digits of a numeral.

Positional Notation

The Roman system is a useful one to study since its deficiencies can be used to highlight the advantages of our decimal system, which uses a **consistent positional notation.** Such a system gives significance to the position of each digit as well as to its intrinsic "value". To be sure, in the Roman system, digits contribute to the total value according to their relative positions, but not, as we shall see, in a consistent manner. For example, MCM Roman = 1900 decimal, and the C is effectively contributing negative one hundred to the total value; whereas MMC Roman = 2100 decimal, and the C is worth positive one hundred.

The decimal system uses a **polynomial, positional** representation method. For example:

2659 decimal = 2 thousands + 6 hundreds + 5 tens + 9 units.

In fact, 2659 decimal = f(10), where

$$f(x) = 2x^3 + 6x^2 + 5x^1 + 9x^0, \quad \text{or}$$
$$f(x) = 2x^3 + 6x^2 + 5x + 9, \quad \text{or}$$
$$f(x) = ((2x + 6)x + 5)x + 9;$$

the last form being known as **Horner's** form, or the nested multiplication form. It leads to quite an efficient way to evaluate a polynomial for a given value of x, and, as we see in the next section, forms the basis for an important class of algorithms for the conversion of numeric representations from decimal to another base and vice-versa. We use it again when we look at numerical algorithms for polynomials.

The Bases 2,8,10,16

The adjective **decimal** is just a synonym for **base ten.** A base is to be regarded as the "x" used in the polynomial above. It is the number on which the numeric representations are based. These forms are built-up from linear combinations of powers of the base, i.e. polynomials with the base as the indeterminate (or variable). The coefficients are just the digits in the final representation, and range between 0 and b-1 (inclusive), where b is the base.

Example 1 Octal (base eight):

Legal Digits = {0,1,2,3,4,5,6,7}

For example, $7123_8 = f(8)$, where

$$f(x) = 7x^3 + 1x^2 + 2x^1 + 3$$
$$= ((7x + 1)x + 2)x + 3.$$

N.B. To avoid confusion, values for the argument of the function f are always expressed in decimal. Otherwise, they would always be "10". Why is this ?

Example 2 Binary (base two):

Legal Digits = {0,1}

For example, $101101_2 = f(2)$, *where*

$$f(x) = x^5 + x^3 + x^2 + x$$

$$= ((((1x + 0)x + 1)x + 1)x + 0)x + 1.$$

Example 3 Hexadecimal (base sixteen):

N.B. The symbols A,B,C,D,E,F respectively represent the numbers ten, eleven, twelve, thirteen, fourteen, and fifteen. These special symbols are necessary whenever the base exceeds ten.

For example, $F2B9_{16} = f(16)$, *where*

$$f(x) = ((15x + 2)x + 11)x + 9.$$

Conversion Algorithms

1. Base b to Decimal

e.g. $7123_b = ((7b + 1)b + 2)b + 3.$

The algorithm is therefore:

set DecimalValue to leading digit
repeat
  set DecimalValue to (DecimalValue*b) + NextDigit
until no digits left
output DecimalValue

Example 1

Let's consider the case where b=8.

7	1	2	3
7	56		
	57	456	
		458	3664
			3667

Therefore, 7123 octal = 3667 decimal.

In this table, we first write down the digits of the original representation in base 8 - one per column. We then write down the leading digit (7 in this case) underneath itself. This step corresponds to the statement "set DecimalValue to leading digit" in our algorithm pseudocode. To generate the remaining entries, and finally the decimal representation, we just keep multiplying by 8 (the base), putting the answer in the next column to the right, and then adding-in the digit already at the top of the column.

Example 2

We now consider the case where b=16.

F	2	B	9
15	240		
	242	3872	
		3883	62128
			62137

Therefore, F2B9 hexadecimal = 62137 decimal.

The steps here are identical to those above for b=8, except that now b=16, and of course, we must remember F=15 decimal, and B=11 decimal.

2. Decimal to Base b

The algorithm here is precisely the reverse all steps of the previous method (base b to decimal), i.e.

repeat
divide Value by b to get new Value + Remainder
until new Value = 0
output remainders in reverse order as the digits in base b representation

As an example let's convert 3667 decimal to octal.

```
8)3667
8) 458     +  remainder 3
8)  57     +  remainder 2
8)   7     +  remainder 1
8)   0     +  remainder 7
```

Therefore, 3667 decimal = 7123 octal.

3. Base b1 to Base b2 (neither is decimal)

Method 1
Convert the representation in base b1 to decimal using algorithm 1, then convert this to base b2 representation using algorithm 2.

Method 2 (only valid if one of b1, b2 is a power of the other)
This is best illustrated by an example. Since $8 = 2^3$, we may convert from binary to octal by grouping the binary digits ("bits") into threes and then translating each group into one octal digit, which incidentally will be the same as the corresponding decimal digit.

As an example let's convert 1101110111 binary to octal.

```
Group the bits as follows      :     001 101 110 111
Convert each group to octal    :      1   5   6   7
```

Hence, 1101110111 binary = 1567 octal.

Conversion from, for example, octal to binary is simply a reversal of this process.

Computer Representation of Numbers

Only two common methods of representing numbers inside a computer will be considered here. They are both systems for **integers**. Representation of rational numbers will not be considered.

1. Binary-Coded Decimal

Binary-coded decimal (BCD) is a simple and increasingly popular way of representing numbers within a computer. In this system, each decimal digit is coded separated in binary. For example:

379 (decimal) = 0011 0111 1001 (BCD).

2. Sign-and-Magnitude Code

This method also deals with negative numbers. The sign of number and its magnitude are represented separately. For example if one bit is used for the sign, the convention being 0 for positive and 1 for negative then:

+ 13 (decimal) is represented as 0 1101 (SM),
- 13 (decimal) is represented as 1 1101 (SM), where the sign bits are underlined.

Representation of Characters

Input, output and secondary storage media and devices generally store and manipulate data as **characters.** Characters include letters, digits and characters such as punctuation marks. These are called alphabetic, numeric (together known as alphanumeric) and special characters respectively.

A **character code** is one in which each character is coded separately as a sequence of binary digits. Six, seven or eight bits per character are most commonly used. It is the set of characters which can be used by a given computer or programming language. Table 1.3 below shows a subset of a very common character code, the seven bit American Standard Code for Information Interchange (ASCII). This is the code that almost all microcomputers use.

The first 32 characters of the ASCII set, i.e. those with decimal ASCII codes less than 32, are known as **control characters**. They are non-printable characters and are used for various communications and device control purposes. For example, the BEL or alarm character is the one that the computer responds to when it 'beeps' at you following some blunder that you've made.

Character	Binary	Decimal
NUL	0000000	0
BEL	0000111	7
BACKSPACE	0001000	8
CARR RETURN	0001101	13
SPACE	0100000	32
!	0100001	33
"	0100010	34
#	0100011	35
$	0100100	36
0	0110000	48
1	0110001	49
2	0110010	50
3	0110011	51
4	0110100	52
5	0110101	53
6	0110110	54
7	0110111	55
8	0111000	56
9	0111001	57
A	1000001	65
B	1000010	66
C	1000011	67
D	1000100	68
E	1000101	69
a	1100001	97
b	1100010	98
c	1100011	99
d	1100100	100
e	1100101	101

Table 1.3 Part of the ASCII Character Set

Alphabetic data generally remains in character code form during processing by a computer, but numeric data is usually converted to one of the numeric codes described above. All conversion from one to another is carried out by hardware or software within the computer.

1.6 Software

Without software the hardware of a computer can do nothing. Software refers to any set of instructions (i.e. programs) for a computer to obey and can be divided into two very broad categories, these being:

(i) System Software, and
(ii) Application Software.

System Software

The term **system software** refers to any program, or collection of programs, that has been designed and written to coordinate the various processes running on a given computer system. Such programs typically perform the following functions:

- Management of the resources of a computer system.
- Provision of various essential utilities.
- The performance of the necessary "housekeeping" chores.

- Provision of language translation services (compilers and interpreters).

For example, users need to be able at any time to find their files, determine the amount of free disk space that they have, and so on. Such services are provided by a very complex program called an **operating system.**

What is an Operating System ?

An operating system is a very complex piece of software. It is a set of programs that allows the computer to manage its own hardware and software resources. It is a **control program** without which no useful work could be done. Its function is to control all activities of the computer and to provide essential utility services to the users. It must be able to provide facilities for listing or displaying files, displaying file directories and for compiling and executing programs written in various languages such as Pascal, FORTRAN, COBOL, and BASIC.

For micro-computers, the most common operating systems at present are CP/M, MS-DOS. PC-DOS, and UNIX (for larger machines).

Other examples of system software are **compilers**, **editors**, and **interpreters.** An editor is a program that allows the creation and editing (modification) of text files, such as computer programs, data files, or even text files intended for pure documentation (e.g. letters, reports, or even books such as this one). If an editor is used to create a computer program, then we need access to another item of system software - a language translator program, in order to convert the human-readable form of the program into binary machine instructions that the primitive hardware of the computer can execute. Such a translator is usually termed a **compiler**. It is important to realize that no computer can function without an explicit sequence of instructions (i.e. program) to direct its operation. The construction of such reliable, efficient, re-usable, and adaptable programs is the province of the discipline known as Software Engineering.

Interpreters and Compilers

All instructions given to a digital computer are expressed in some programming language such as BASIC, FORTRAN, COBOL, and Pascal. However the computer cannot directly execute sequences of instructions written in any of these languages. In fact, the only language that it can "understand" is its own, native machine language. Every computer has its own machine language to which all programs must ultimately be converted before they can run. Programs written in English-like high-level languages must therefore at some stage be converted to "low-level" machine-language codes (instructions).

Since computers can only "understand" programs written in the very low-level machine code mentioned above, there has to be a **translation process** by which a high-level language program must be converted to an equivalent **machine language** program before execution by the machine can begin.

Translation of a program in a high-level language into machine-code is an exceedingly complex process. This job is in fact so tedious, complex and error-prone, that it is done by the computer itself! There is a separate program called a **translator** within the computer system itself, whose sole purpose in life is to faithfully translate our Pascal or BASIC programs into machine instructions. It is most important for you to remember that your program must always be first processed (i.e. translated) by this system-resident translator program before it can actually be executed by the machine.

Two types of translator programs exist. Most computers using the BASIC language have an **interpreter** to do the translation. An interpreter is a translator program that initiates the execution of the **object-code** (translated instructions) resulting from each BASIC instruction as soon as it is generated. The process is then repeated with the next BASIC instruction and so on, i.e.

Translate Instruction 1, Execute resulting code,
Translate Instruction 2, Execute resulting code,
Translate Instruction 3, Execute resulting code, etc.

The other type of language translator is known as a **compiler**. Turbo Pascal is a compiler. A compiler differs fundamentally from an interpreter in that it translates the entire program before any attempt is made to execute the resulting object code. This object code may then be stored in a disk file so that it may be executed over and over again without recompilation (i.e. **retranslation)** each time. FORTRAN, Pascal, COBOL, and most other languages also use this approach. It should be remembered that a compiled program will generally run much faster than the same program run via an interpreter, since no time is wasted between execution of instructions for continual translation of source (e.g. BASIC) code. There is also a program called a **loader** which takes the translated form of your Pascal program from secondary storage (i.e. disk) and loads it into main memory, ready for execution. The above discussion is shown in diagrammatic form in Figure 1.5.

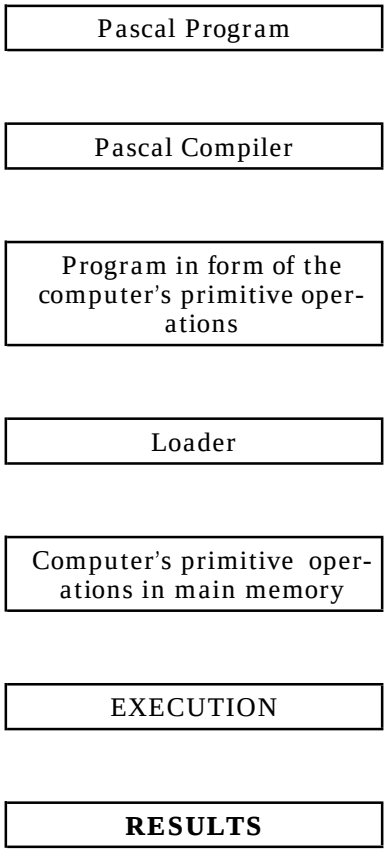


Figure 1.5 What Happens To A Typical Pascal Program

Programming Errors

The process of writing a computer program and then trying to get it doing what you want seldom goes smoothly. Many different kinds of errors can (and usually do!) occur. These may be broadly classified as:

- Compile-time - those which can be detected by the compiler.
- Run-time - those which cannot be detected until the processor executes the program.
- Algorithmic - you have solved the wrong problem.

Compile-Time Errors

The Pascal compiler that you use has another very important function apart from its basic task of translating your program. It must be able to detect and recover from any grammatical errors (**syntax errors**), or simple errors of meaning (**semantic errors**) that you may have made when you wrote your Pascal program. Every language has rules of grammar and Pascal is no exception. **Syntax errors** are those which occur when you have written a statement which breaks one of the rules of grammar (syntax) of the programming language being used. The syntax of a language is a set of rules which determines what constitutes a valid, well-formed statement in the language. These may be compared to the grammatical rules of English. For example, if you write the English sentence:

"HE READ THE NEW BOOK",

an English-language **parser** would determine that the sentence is **well-formed.** It has:

subject (pronoun)	HE
verb	READ
article	THE
adjective	NEW
object (noun)	BOOK

Some items are **optional,** that is, they could be left out and the sentence would still be well-formed (but almost certainly have a different **meaning**). For example, we could omit the word "NEW" from the sentence above and it would still make sense. None of the other words however could be omitted without creating a sentence which is grammatically incorrect. Poor English grammar and spelling is unfortunately tolerated by most human beings, but you will find that the computer will not tolerate the slightest deviation from its **rules of syntax.** Of course, the machine itself knows nothing about such rules; rather they are enforced by rules built into the translation program that we have called a **compiler.**

In Pascal, as in other computer languages, common examples of syntax errors occur when an essential language keyword is misspelt, or even omitted. For example, if we write the Pascal code fragment:

if b*b - 4*a*c > 0 write('This Quadratic Has Real Roots');

then our Pascal compiler should detect that an essential keyword has been omitted. The statement should read:

if b*b - 4*a*c > 0 **then** write('This Quadratic Has Real Roots');

Semantic errors are errors of **meaning.** Depending on the language, these can sometimes be detected at compilation time. Pascal and its derivatives have been specifically designed to detect as many of these errors as is reasonably possible at compilation time, rather than when the program is actually running. If you do not see the importance of this approach, then reflect for a minute on the consequences of such errors in the software which will be used to control the launch of anti-missiles in the United States' Strategic Defense Initiative Program (SDI or "Star Wars").

A semantic error would occur, for example, if we attempted to add a scalar to a vector. A good compiler would detect this and inform us of the error. In this manner, the erroneous program would not be translated and hence never executed.

A crucial property of programming languages, as opposed to natural languages like English, is that provided you abide by the syntax rules the **meaning** (semantics) of each statement is clear and unambiguous (i.e. the operation to be carried out by the processor is clear and unambiguous). Any deviation from the prescribed syntax of a programming language will result in a compile-time error.

Run-Time Errors

These errors arise when a computer is asked to do something which is beyond its capabilities, or is simply logically impossible. It is quite easy to write a program which is syntactically correct, but which describes a process the computer is incapable of carrying out, for example, dividing by zero.

Run-time errors are usually more difficult to correct than compile-time errors, because:

- They might occur only for particular inputs.
- Their causes may lie further back in the algorithm.

With practice you will learn to recognize the types of run-time errors which often crop up, and also to recognize their causes. Run-time errors usually occur because of a flaw in the algorithm. We study algorithms in considerable detail in the next chapter.

Summing Up on Errors

There are four things about errors you should remember:

- All programmers get them.
- When the computer says there is an error, there is. The computer is always right here.
- Do not run a failed program to see if it fails again - it will. A computer is a deterministic machine, i.e. given the same program and the same input, it will always produce the same results.
- Even if the program works, it is not necessarily correct. Would it work for other input? Does it produce the correct results?

1.7 Historical Note

Blaise Pascal (1623-1662)

Blaise Pascal invented the first practical calculating device at the age of nineteen. His father took responsibility for his education and Blaise did not spend a single day at school. However, at the age of eleven he started his first experiment which was concerned with the qualities of sound. He became interested in sound after striking a piece of china and noticing that the ringing hum stopped when the china was touched.

Pascal's father wanted him to study classical languages and would not allow him to have mathematics books. In 1656 after having discussions with a gambler who was losing badly at dice Pascal developed the fundamentals of our theories of probability and statistics. He also invented a type of wristwatch, and experimented to show properties of atmospheric pressure. His work in hydrostatics led to the hydraulic press. Pascal's Principle which describes the properties of a fluid at rest is named after him. Pascal's Triangle concerning the binomial theorem was also discovered by this gifted Frenchman.

In his early adult life Pascal turned more to the study of philosophy and theology and became a Jansenist (member of a Christian order). His writings in this area (most famous **Pensées**) are quite well-known. Pascal soon abandoned his scientific activities and devoted much of his life to meditation and theological writing.

Pascal grew very ill in his latter years. A few years before his death he had a serious attack of toothache and to take his mind off it he devoted 8 days to contemplating the properties of the cycloid (a cycloid is the path traced out by a point on the rim of a wheel). He discovered six interesting properties of the cycloid.

Although Pascal's writings were not extensive they were highly original. We can only speculate as to what further mathematical discoveries he may have made if he had put his mind to it.

1.8 Summary

- A computer is a set of resources, including digital electronic data processing devices, stored programs and sets of data, which, under the control of the stored programs, automatically inputs, processes and outputs data.
- The term **Hardware** refers to the physical components, solid-state and otherwise, which make up a computer.
- The term **Software** refers to the programs which direct the operation of a computer.
- Computers may be classified as mainframe, minicomputers, and microcomputers.
- Peripheral devices allow for input and output. Some examples are terminals, card readers, paper tape, direct source readers, voice recognition devices, printers, microfilm, digital plotters, and speech synthesizers.
- The CPU of a typical computer consists of an Arithmetic Logic Unit, a Control Unit, and Main Memory (RAM).
- A **bit** is a binary digit, i.e. a "0" or "1". A **byte** is the smallest unit of computer memory. It has the capacity to store one character and is made up of 8 bits.
- Secondary memory devices usually include disks and tapes.
- Familiar number representation systems are the Roman and decimal systems. Computers use the binary system because the devices used in constructing computers have only two states.
- Number representations can readily be converted between the decimal, octal, hexadecimal, and binary systems.
- Two common methods of representing numbers inside a computer are - Binary Coded Decimal and Sign-and-Magnitude.
- Characters are usually represented in ASCII code.
- A compiler translates statements in a programming language into primitive operations the CPU is capable of executing. A compiler is needed for each programming language to be used on a particular computer.
- The operating system supervises the running of all programming jobs.
- Compile-time errors are the result of not obeying the syntax rules (grammar) of the programming language being used.
- Run-time errors are the result of asking the computer to do something it cannot do.

1.9 Exercises

1. Define the following terms:

(i) hardware	(v) bit
(ii) software	(vi) byte
(iii) CPU	(vii) secondary storage
(iv) RAM	(viii) character set

2. Describe the structure of an electronic digital computer system, giving the function of each major component. Draw a block diagram to illustrate the interactions between the various components.

3. Identify the types of computer input devices which are also output devices.

4. Distinguish between the concepts of *number* and *numeral*, giving examples of each.

5. Why is the binary system used for arithmetic and data representation in digital computers ?

6. Express the numbers "forty-eight" and "one hundred and twenty-seven" in each of the following systems:

   (i) Roman
   (ii) Decimal
   (iii) Binary
   (iv) Octal
   (v) Hexadecimal

7. Rewrite the following polynomial equations in Horner's form:

   (*i*) $f(x) = 3x^3 - 6x^2 + 4x + 5$

   (*ii*) $f(x) = 4x^4 - 5x^3 - 5x^2 - 2x + 7$

   (*iii*) $f(x) = 5x^5 + 3x^4 + 5x^3 - 6x^2 + 8x - 9$

   (*iv*) $f(x) = 6x^5 + 5x^3 + 7x^2 - 8x + 9$

   (*v*) $f(x) = 8x^6 - 5x^4 - 7x^2 + 6x - 8$

8. A certain computer uses a 16-bit word to represent integers. The leading bit is reserved for the sign of the number - a zero to represent positive numbers and a one for negative numbers. The next bit is the most significant. Determine the bit pattern for the contents of a word holding the value -319.

9. Convert each of the following **decimal** representations into binary, octal, and hexadecimal:

   (i) 15
   (ii) 27
   (iii) 99
   (iv) 4095

   In each case convert the representations back to decimal. This of course, enables you to verify your answers.

10. Convert each of the following binary representations to both octal and hexadecimal:

    (i) 1101110101 (ii) 1111110001101

11. Distinguish between compile-time and run-time errors.

12. What are the functions of a compiler and a loader ?

2 PROBLEM SOLVING & ALGORITHMS

2.1 Elements of Problem-Solving

Only certain problems lend themselves to solution on a computer. Until you know more about computers you can't be expected to decide whether a problem is suitable or not. Fortunately all the problems selected for you in this book are suitable for solving using programming techniques. Solving problems using a computer is demanding! It is an intricate process requiring much thought, careful planning, logical precision, persistence, and attention to detail.

General Problem-Solving

Problem solving is a creative process for which there is no universal approach or method. Different strategies appear to work for different people. However, there are a number of steps that can be taken to improve one's ability. We now look at these briefly.

Problem Definition Phase

In this initial step, our main aim is to get a clear statement of the problem. Make sure you **understand** the problem and exactly what is required! You will have absolutely no chance of getting the computer to solve your mathematical problems if you don't understand the problem yourself. It is a common misconception among students learning to program a computer that the machine will save them from having to learn and understand some of their mathematical theory. The sooner that you realize the utter futility of this hope, the sooner you will make progress in successfully solving problems using a computer. In order to specify the precise steps of a mathematical calculation to a mindless automaton (i.e. a computer), we need to be doubly sure of our theory. In other words, in a very real sense, we must have an even better knowledge of our mathematical theory than would perhaps normally be required "just to pass exams", if we are to have any hope at all of programming solutions to mathematical problems on a computing device.

It really sounds like a trivial observation to say that we must **know what must be done before we can decide how to do it**, but you will find that many will forget this simple piece of advice and just jump straight into writing a computer program, getting half way before they even realize that they don't even understand what they are trying to solve, let alone how to solve it. It is not wise to jump, boots and all, until one knows what one is jumping into.

It is important to realize that we are not saying that we must know **how** to solve the problem at the problem definition phase. In fact, we are not concerned at all with **how** in this phase, but we are very concerned with **what**. We must pin down just **what** it is we are to do - the **how** comes later. We talk about the **how** in quite a bit of detail in the following sections.

In order to define the **what**, we make use of our training in mathematics so as to express the problem as carefully as possible. Ideally, one should try to specify what the computer is to do for every possible case that may arise. For example, suppose we are to write a program to solve a quadratic equation. We input the coefficients of the quadratic, since these completely define it, and expect the computer to come up with two solutions. If the equation is

$$Ax^2 + Bx + C = 0$$

then the solutions are given by the usual quadratic formula

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

If we wrote a program in Pascal to solve for x_1 and x_2 after being given A, B, C, and using the above quadratic formula, then we deserve to fall flat on our faces! What if A=1, B=0, C=1? Surely these are reasonable numbers to use - we could hardly have simpler numbers!

Can you solve the equation $x^2 + 1 = 0$?

Certainly not in terms of real numbers. (If you take the unit Complex Numbers then you will see that this equation becomes solvable once we extend our set of numbers). If no real solution exists to a mathematical equation, then it is simply nonsense to expect that a computer can find it. Can a computer do the impossible? Of course not.

Getting Started Phase

There are many ways to solve most problems and often many solutions to a given problem. How do we decide which lines of attack will be productive and which will be fruitless? There is no simple answer to this question. Often we simply don't have any idea as to where to start on the problem even after the problem definition phase. We remark in passing however, that those with training in problem-solving disciplines such as mathematics, tend to fare much better here than those who lack such a background.

The important piece of advice that can be given here is not to become too concerned with detail at this stage. The detail can come later - after the complexity of the problem as a whole has been brought under control.

Use of Specific Examples

An approach which sometimes allows us to get started on a problem is to pick a specific example of the general problem we are trying to solve and make some attempt to devise a method that will allow us to solve this particular case. For example, if we want to write a computer program to sort a sequence of integers into ascending order, we can make a start by choosing a dozen (or even less - don't make examples too complicated at this stage) and sort them by hand, carefully observing the processes that we use to get the job done. These processes, taken in the correct sequence, form a particular method for sorting integers. It also important for us to realize that we are not so much interested in the solution itself, that is the answer, but more in the **means** by which we managed to arrive at the answer. A particular method or means for solving a problem is referred to by mathematicians and computer programmers as an **algorithm**. In a sense, this entire unit of Computer Mathematics is all about algorithms, since we shall be studying algorithms (i.e. methods) for solving various mathematical problems such as solutions of equations, areas bounded by curves, and so on.

Although this approach can give us the foothold needed to start on a problem, care must be taken not to fall into the trap of thinking that the solution method adopted for specific examples or cases will necessarily work in the general case. Experience has shown however that it is a rare problem for which this technique provides no insight whatsoever. Examples usually give us some information, even if it is only negative, i.e. solutions may not exist at all for some values of the input data. We need only recall our quadratic equation above for a simple example of this.

Use of Similar Problems

After a little experience in mathematics and computing it becomes less likely that a new problem is completely different to others that have already been solved. A good habit is to always make an effort to be aware of the similarities among problems - in the hope of course that similarities among problems will lead to similarities among solutions. However, care must be taken not to overstudy the existing solution of a similar problem. There is a danger of proceeding down a path of reasoning which leads to a dead end. It is usually wise in the first instance to try to independently solve the problem.

As an example of this technique however, consider the problem of solving the cubic equation:

$$Ax^3 + Bx^2 + Cx + D = 0$$

Since we know that there is a formula to solve general cubics, we look it up in a reference book, and find that it involves solving some quadratics as sub-problems. We already have an algorithm for quadratics, so our solution to the cubic can be based on the quadratic to some extent. We should be able to proceed without much difficulty. The same approach may be used for the fourth degree equation:

$$Ax^4 + Bx^3 + Cx^2 + Dx + E = 0$$

with similar success. We find however that further progress (with higher degree equations) in this direction is impossible. Here we have an example of how this approach leads to a dead end. Any attempt to solve the general quintic equation:

$$Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F = 0$$

will be futile. Why is this? Is there some programming technique which cannot be applied for this case, but which works for the other cases? No. For the answer we turn once again to our mathematical resources. It is an established result of polynomial theory that no general formula exists for solving polynomials of degree higher than four. Any effort to develop an algorithm for this problem will therefore be wasted. Not only are algorithms tricky to find, but sometimes they do **not exist** at all, and this fact can on occasion be proved! Imagine wasting months or years of effort on an important

problem which is logically incapable of solution! Here is a good example where our knowledge of mathematics saves us from making enormous gaffes in design of algorithms, and from following fruitless paths of investigation.

Working Backwards From The Solution

If we still do not know where to start on a problem we can in some cases assume we already have the solution and try to work back to the starting conditions. Even a guess at the solution to the problem may just give us the start we need.

General Problem-Solving Strategies

The "divide-and-conquer" strategy is very commonly used in problem-solving. The basic idea is to divide the original problem into two or more sub-problems which hopefully are easier to solve, and whose solutions when combined in some way lead us to the solution of the original problem. If, after sub-dividing our problem in this way, at least one of the sub-problems is still too hard, we may again sub-divide. Thus we generate a tree-structure. Finally, we hope to reach a stage where each sub-problem can be solved without further splitting. We refer to this technique as stepwise-refinement or top-down design, and the development of the rest of the book is based on this approach. This general technique is of course used by professionals engaged in all forms of problem solving - it is by no means restricted to computer problem-solving, which we now consider in particular.

2.2 Computer Problem-Solving

How do we use a computer to solve a problem or carry out some task? The steps that generally need to be taken are as follows:

- Select an appropriate task (e.g. sort a list of names into alphabetical order, find the area under a curve, print examination reports). Only with experience will you be able to decide whether a specific task is suitable for a computing solution.
- Define the problem very clearly. If possible, try to allow for all cases that can occur - remember our quadratic example. (This is our problem definition phase mentioned earlier.)
- Design a way of carrying out the task. As we have seen, the technical term used for "a way of carrying out a task" is **algorithm**. (Remember that, in general, tasks need not be intended for execution on a computer. There are algorithms for making a cup of tea, for baking cakes, surf-board riding etc.).
- Translate the algorithm into a **language** intelligible to the computer. The finished product of such a translation is called a **program**, and the process of translation is called **coding**. A program is therefore a set of instructions which can be carried out by a computer. The language in which the algorithm is finally expressed by the coder (or programmer) is called a **computer programming language,** or more simply, a **programming language.** Some programming languages in common use are FORTRAN, Pascal, COBOL, and BASIC. Each of these languages was designed with a specific area of application in mind. In this course we use Pascal.
- Get the computer to actually **execute** or **run** the program. If the original algorithm was in fact correct, and there were no errors in the translation process, then we may reasonably expect that the computer will solve our problem for us. We shall find, however that this is very seldom the case. This leads us to our next point.
- Ensure that the task carried out by the computer is the one that was intended. This part is usually called **program testing,** and if there are errors detected, the process of removing them is referred to as **debugging**. Here we are attempting to verify that the program is a correct expression of the algorithm, and that the algorithm does actually solve the problem. This part of computing is usually very time-consuming (and hence often ignored), and also very difficult and taxing. It is a complete waste of time if you don't know the mathematical theory behind the problem beforehand. In fact the entire **programming process** (i.e. problem definition, algorithm development, coding of the algorithm, execution of program, debugging and testing) is a waste of time if you haven't the necessary knowledge (mathematical or otherwise) to understand the problem.

It is important to realize that a computer is a willing, obedient, and totally stupid slave, that works rapidly and precisely in doing exactly the task that you have **told** it to do - not necessarily what you **want** it to do! Computers **almost never** make mistakes but they do sometimes break down.

When you write your first program to perform some non-trivial task, it will almost certainly contain mistakes. Unfortunately, one of the major skills required in computing is that of debugging faulty programs. It would be nice if things worked first time, but we cannot really hope for this when it is realized that even **professional** computer programmers rarely have enough mathematical training to allow them to prove their algorithms are correct even before coding and execution on a computer is attempted. It seems that we shall have to continue to stumble along, forever writing incorrect programs and then "patching" them into shape so that they actually work! Computer science is a very young science, and one of the most important areas of modern research is the investigation of mathematical techniques which will make computer software (i.e. programs) more reliable. From a practical point of view though - ask your classmates and teacher to help you find some of the more difficult bugs, but always try to solve them first yourself. Otherwise the unit in Computer Mathematics will profit you little.

Programming Tip

> **Think first. Code later.**
>
> It is generally accepted that the sooner you start coding a program (i.e. translating an algorithm into a programming language the computer can understand) the longer it will take you to solve the problem successfully. Trying to write anything but the simplest of programs without a well organized solution outline is like a visitor traversing a large country without a road map or a builder building a house without plans. Chaos will soon follow.
>
> Think once, think twice, think some more, then write your algorithm, and finally write your program.

2.3 Algorithms

As we have already learned, an algorithm is a description of how to carry out a process. The agent which actually does the work is called a **processor**. A computer is therefore a particular kind of processor, which carries out processes, expressed as programs of instructions. Some examples are given in Table 2.1.

Process	**Algorithm**	**Processor**
Drive to Perth	Road map	Car Driver
Baking a fancy cake	Fancy recipe	Cook + Oven
Building a model plane	Assembly instructions	Boy + Dad

Table 2.1 Algorithms & Processes

In these examples the processors are either persons or a machine (oven), who with simple skills and tools obey instructions and if no mistakes are made, will produce the desired effect.

Many processes need to accept input and produce output. For example, in baking the cake the input is ingredients plus energy from the oven and the output is the baked cake (and an oven to clean!). It should be no surprise to us that computer algorithms often contain errors, since we are used to the fact that human beings make mistakes. What if the oven is too hot or the cake is left in it for too long? If you set your automatic washing machine on the wrong cycle and then go away and leave it, should you be surprised on returning to find that it hasn't done what you wanted it to do? It has of course done what you **told** it to do instead. Such frustratingly literal obedience is very hard for some people to come to terms with! They prefer to deal instead with ambiguous, imperfect people. We all do, really. It can indeed be very frustrating to have to deal with mindless machines - they have no imagination and cannot normally "read between the lines" of our instructions to them, as people usually do. They just follow them to the letter. Too bad if we have been imprecise in our specification of them. You will be spared much pain and frustration if you learn these points early.

The specification of input and output is a necessary part of the specification of any task that we want the computer to carry out. Many processes eventually terminate, for example, model plane completed, the cake baked, Perth reached. However, some processes never terminate, for example:

- Breathing.
- Keeping yourself fed.
- Updating library catalogue.
- Controlling traffic lights.
- Learning new ideas.

In this book, we consider only processes which terminate, i.e. the algorithm (and hence the program derived from it) must have a definite end point which must always be reached. We will have occasion to point out, however, that it is often rather tricky to ensure that the algorithms we create do in fact have this property for all possible input data. We also insist that the process arising from an algorithm must produce some output before termination. An algorithm which produces the right output every time it terminates is said to be **partially correct.** If it also always terminates, we call it **totally correct.**

An algorithm for carrying out a particular task is given to the processor as a sequence of unambiguous, primitive operations which the processor is capable of carrying out. An example is given in Table 2.2.

Algorithm	**Processor**	**Primitive Operation**
Road map	Car Driver	Proceed along route 1; turn right at Dogtown
Recipe	Cook	Add 2 eggs and heat for 20 minutes
Assembly instructions	Boy + dad	Glue part A to part B
Computer program	Computer	Subtract 2 from PAY

Table 2.2 Algorithmic Primitive Operations

For a computer the operations correspond to statements (instructions) in the programming language that is being used. For example, in Pascal "subtract 2 from PAY" would be written (coded) as:

```
PAY := PAY - 2 .
```

We remark in passing here that the := sign denotes assignment of the value of the expression on the right hand side to the variable PAY. It clearly cannot denote mathematical equality, as any student of algebra should quickly discern. The colon is included in order to emphasize this fact.

Algorithms which consist only of a simple sequence of primitive operations are normally not very useful. It may be that a sequence of operations needs to be performed only if a particular condition is true. For example, an automatic pilot for a new car uses the following algorithm for negotiating traffic lights:

```
set speed to 5 kmph
drive to lights
if light is red
  then brake
  else if light is orange
    then maintain speed
    else if light is green
      then increase speed
      else stop : lights not working.
```

Conditional execution as shown in this example gives much more flexibility and necessary control to our algorithms. In fact, it is essential to have this decision-making capability in our programs. To solve our quadratic equation properly we need to know that $B^2 - 4AC$ is greater than zero. Our algorithm would contain an instruction such as:

```
if B*B - 4*A*C >= 0  then .. (calculate solutions)
                     else .. (indicate no real solutions).
```

The ability to make decisions based on the current "state of things", i.e. while the program is actually executing, is absolutely essential for development of algorithms. In other words, when developing an algorithm, we must assume that the processor can make decisions "on the fly", and take action appropriate to the outcome of some **test.**

So far, we have really glossed over the question of "just what kinds of instructions are we allowed to use when writing algorithms?". This is a very important question. Before answering it though, we make the observation that the legal instructions for a **program** (not an algorithm) are simply those which the **processor** can "understand". When we write in Pascal, we use statements that are legal in that language. We assume that the computer can understand each of these Pascal statements (instructions). In actual fact, this is not the case, since as we have seen in Chapter 1, a **further** translation process is necessary before the computer can actually execute our Pascal program. For the present however, it will be convenient for us to pretend that the computer actually understands Pascal itself! This is a common and very convenient practice in computing, that is, to assume the existence of some mythical device which can understand and interpret instructions in a language convenient to us at the time. We refer to such an abstract device as a **virtual machine.** The language in which we can directly communicate to a **real machine** (i.e. computer) is called, naturally enough, **machine language.** We have mentioned something about machine language in the previous chapter. The main points to remember here are that:

- Machine language is far too tedious for people to write programs in.
- This being the case, some translation process must always precede the execution of any program written in other than machine language.

So the permissible instructions in a computer program are just those which are legal in the programming language that we have chosen to use. What about the permissible instructions in algorithms themselves ? Since programs are just one way of expressing algorithms, we would expect that an "algorithmic language" (one specifically designed for writing algorithms) may look rather similar to one or more **programming languages.** This is indeed the case. The language in which we have chosen to ultimately express our algorithms in this book is the programming language **Pascal**, which was designed in the early 1970's by Niklaus Wirth as a vehicle for teaching the principles of algorithm development and programming. It has proved an excellent language for this purpose. In some of the early chapters here, we do not use the precise syntax of Pascal itself, but a closely-related **pseudocode.** Some people call it **structured English**, but regardless of the name, the main aim is to be as clear and precise as possible while avoiding any possible ambiguities.

We shall find that, no matter what language we ultimately use, we still need three different ways of combining instructions (i.e. ways of specifying order of execution) to form a complete algorithm or program. Programming language elements which allow the specification of the order in which, and conditions under which statements or groups of statements are to be executed are called **control structures.** There is a fundamental theorem in computer science to the effect that any algorithm can be expressed using only the control structures **sequence**, **selection**, and **iteration**. We look at each of these in turn.

Sequence

This is simply the execution of instructions one after the other, in a strict, linear sequence. For example:

```
add eggs
add butter
add sugar
```

Selection

This refers to the ability to base the decision of the next instruction to be executed on the outcome of some test, i.e. a **decision** is made while the program is actually executing as to what to do next. For example:

```
if B*B - 4*A*C >= 0  then .. (calculate solutions)
                     else .. (indicate no real solutions).
```

This is just our earlier example repeated. Note however that since we don't know **in advance** the values of A, B, and C, we have no of way knowing which one of the three **branches** of this **if** statement will actually be executed.

Iteration

The word **iteration** simply means **repetition.** This refers to the ability to repeat an instruction or a sequence of instructions over and over again until we are satisfied that the job has been done. For example:

```
beat until thickened
cook until golden brown
keep going until you reach the turnoff.
```

Notice that the key word common to these examples of repetition is **until**. Note further that each process is repeated up to a point but eventually terminates. Most interesting algorithms involve some form of repetition, i.e. **iteration**. If you think carefully, you should realize that without the ability to write algorithms involving repetition, the size of a program would be directly proportional to the number of operations required to be done. In other words, to do 10,000 operations we may need 10,000 instructions. With repetition, we may need only 2 or 3. Consider the following two algorithms which both print all the even numbers less than 1000:

Algorithm 1

```
write 2
write 4
write 6
write 8
write 10
.....
.....
write 998
```

Algorithm 2

```
set x to 2
repeat
   write x
   add 2 to x
until x = 1000
```

Of course we are not allowed to use the "....." when actually writing the algorithm out. Every statement must be explicitly included. You may be able to get away with vagaries like "....." in other units of mathematics, but you certainly won't get away with it here! ***Algorithm 1 actually has 499 statements!***

It is instructive to look at the second algorithm in more detail. We shall rewrite it in Pascal. What! We haven't done Pascal yet! It doesn't matter at all. All we do here is just imagine a virtual machine with very simple properties. It has one memory cell in which a value called x is stored; it can do addition with its one register; we can store the result of addition in the memory location reserved for x; it can test two numbers and determine the larger; and it can write out the value of x at any time. It can execute a few statements of Pascal directly, and proceeds from instruction to instruction from top to bottom. Our algorithm 2 can now be expressed in Pascal as follows:

```
x:=2;
while x<1000 do begin
  write(x);
  x:=x+2
end.
```

The symbol **:=** is read as "becomes", or "gets the value of". For example, x:=2 means that x gets the value 2 (the value 2 is stored in the memory location reserved for the value of x). The statements bracketed by the **while .... end** pair form what is known as a **pre-tested loop.** The condition **x<1000** is initially tested, and if true, the body of the loop is executed once. After this, control returns automatically to the test of the **while** statement. The same test is again applied, though this time with a different value of x, since inside the loop it has been incremented by 2. If the test again succeeds, then the loop body is re-executed. This process continues until the **while** test fails, after which control passes to the next instruction after the loop. In this case there is none, so we stop. You should study this example very carefully and work through it on paper before any attempt is made to run it on a computer. This last activity is best left to the next chapter, although there will no doubt be some who are keen to try it out before that time is reached. Such students should not be discouraged from doing so, but are advised that maximum benefit will be gained if the program execution is first checked out by hand.

The Need for Iteration

As we have mentioned, the power of iteration is that we can write down a fixed length algorithm for a process of indeterminate length. With power comes responsibility however, and the price we pay for this extra processing power is that we must somehow be able to guarantee that any iteration that we build into an algorithm must eventually terminate. It seems fairly clear that algorithm 2 above has this property, but what of the algorithm below:

Algorithm 3

```
start with any given number (positive integer)
repeat
  add 1 to number
until number is prime
```

Does this algorithm terminate, no matter what the starting value is ? Once again, we need our knowledge of mathematics (number theory in this case) in order to prove termination. What is someone saying - the number of primes is infinite ? Yes, this is true. Will this result guarantee termination of our algorithm ? Once again, the answer is yes. Can you prove it ? (Be careful, because there is also a theorem in number theory which says that given a positive integer, no matter how large, one can always find two consecutive primes which differ by more than the given integer!)

While on the subject of termination of iterative algorithms, see if you can find a starting value of x for which the following algorithm fails to terminate:

Algorithm 4

```
start with any given positive integer for x
repeat
  if x is even then replace x by x/2 else replace x by 3x + 1
until x = 1
```

Try not to spend more than a few months on this one !

Let us consider another example of repetition.

Algorithm 5

This algorithm might be used to find an ideal partner:

```
repeat
  obtain $10 somehow
  send application to Ideal Playmate Company
  go out with the selected partner
until the partner is suitable
```

Can you see any problems with this algorithm ?

Final Remarks On Iteration

Repetition (iteration) necessarily uses memory i.e. it describes a process in which things are remembered from one step to the next (our toy virtual machine had only one memory cell). Further memory adds to the power of repetition and conditional execution by allowing information to be stored at one time and used at a later stage, possibly to determine what to do next. We will have much to say on iteration later in Chapter 3 and also as we study our mathematical applications.

Programming Tip

> Correct algorithm design is extremely important in computing. With simple problems students who are studying introductory programming courses often want to go directly to the coding stages, and eliminate almost entirely the separate and distinct algorithmic design phase. Hence they reinforce the notion that coding and language syntax are the most important parts of the **programming process.** The truly important part of programming is developing a correct, elegant and efficient algorithm in some language-independent representation such as pseudocode or structure chart.
>
> Computing can be considered as the study of algorithms and their implementation on digital computers.

2.4 The Development Of Algorithms

We have seen that the solution of a non-trivial problem in mathematics or computing is a surprisingly difficult task. When you start out trying to design algorithms it pays to follow a well thought-out general strategy that has worked for others. In the early pages of this chapter we mentioned the idea of stepwise refinement or top-down development as a general problem-solving strategy. This technique is developed here and will be constantly stressed in this book.

Top Down Development

The technique of top-down design of algorithms starts with only a broad statement of the solution to the problem and refines each step into its own component steps by branching out at each level. Sooner or later the branching process will stop since the component subtasks at that level are simple or primitive enough to be understood directly by us or the processor, meaning essentially that we already have a computer program to solve it or can write one quickly.

An Example Of Top Down Development

Let's look at a simple example to illustrate these ideas. Suppose that, to our great surprise, a rather strange-looking young Martian arrives on our doorstep one day looking for work. Unemployment is high on Mars and this enterprising young alien has decided to seek work outside of the usual avenues. Rather than send him away disappointed we decide to teach him how we live day-to-day on Earth and then engage his services as a houseboy. Unfortunately, this turns out to be a rather more difficult task than we had first thought. We find that this being is very fast and efficient at doing jobs for us, if he knows how to do them but that, on the negative side, he always obeys instructions to the very letter, never deviating at all from our explicit instructions. (Are these negative qualities ? - a philosophical point, since his claim is that we are not very good at describing our requirements, while we claim that he should use his imagination. Who do you think is right ? You may like to reserve your judgment until after we consider our top-down attack on making a cup of coffee). Since making a cup of coffee was not one of the Martian's primitive operations, we had to explain to him how to do it. Our first attempt is illustrated in Figure 2.1.

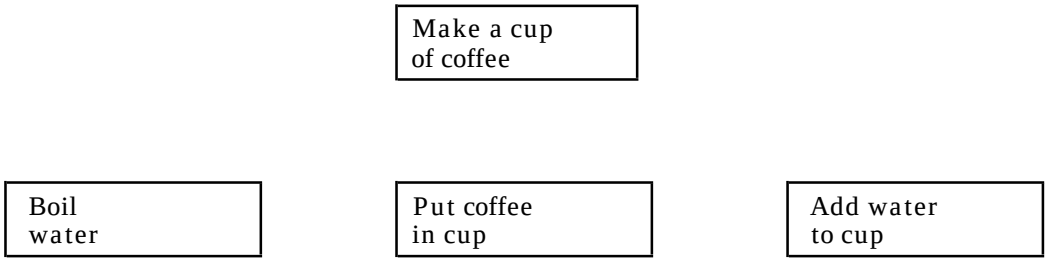


Figure 2.1 Making a Cup Of Coffee - First Attempt

The steps are executed in sequence from left to right, i.e. 200, 300, 400. We thought that anyone could understand this, so imagine our surprise when this little fellow tells us that he has not yet learned how to boil water ! Our only recourse is to do a similar breakdown for him on how to boil water. This is given in Figure 2.2.

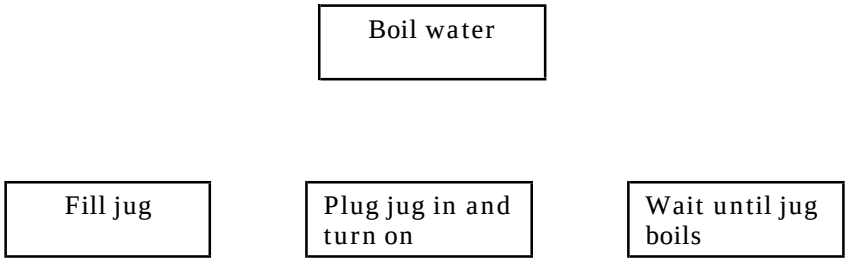


Figure 2.2 Making a Cup Of Coffee - Second Attempt

Surely each of these operations is simple enough! Perhaps not. After all, 230 involves repetition (i.e. keep checking jug until it boils), and 220 requires the little guy to know what to plug the jug into! Many more levels of decomposition may be required until the small steps are within his understanding and capabilities.

The entire algorithm (at the present level of decomposition) is given in Figure 2.3. Such a diagram is called a **structure chart.** Each of the blocks is often referred to as a **module**. There are some further "bugs" in the algorithm as expressed in Figure 2.3. What does he do if the tap is already on? Where is the jug to be found? What is he to do if there is no power? Perhaps a fuse is blown and he needs to replace it. Can you write a sub-algorithm for this contingency? Be assured that the design of a foolproof algorithm is an exceedingly difficult task.

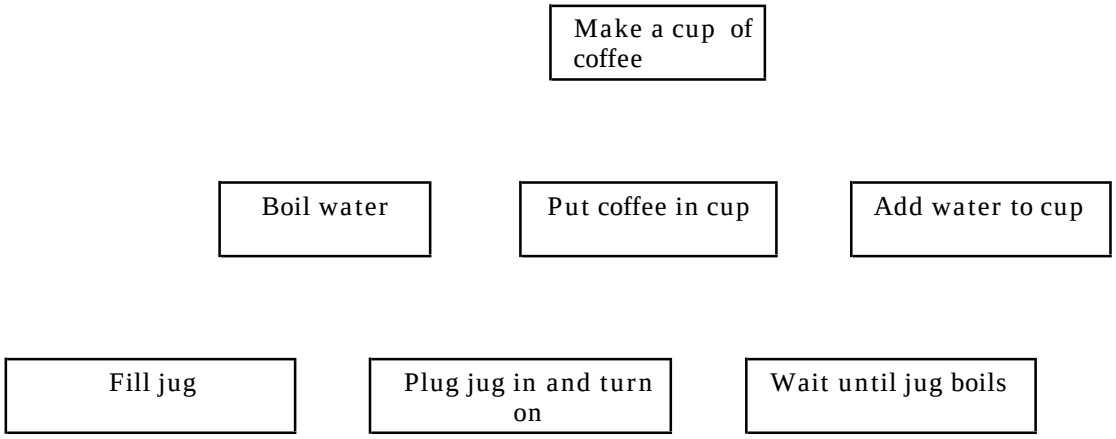


Figure 2.3 Making a Cup of Coffee - Third Attempt

2.5 Historical Note

Charles Babbage (1791-1871) & Ada Augusta Byron (1815-1852)

Born in Totnes, Devonshire, England, Charles Babbage was mathematics professor at Cambridge University from 1828. It was however while Babbage was still a student at the same institution that he began to work on his famous difference engine.

Babbage was a man of diverse interests, but his major contributions to science and mathematics were through his calculating machinery. These devices were entirely mechanical, with no idea that electricity would ever be used. His **analytical engine** was a digital, decimal computer. It used positions on wheels to store numbers, which were transferred by a system of racks to a central **mill** (processor), where all arithmetic was done. He used punched cards to direct the automatic operation of the machine. Since looms which used punched cards for a similar purpose already existed, he made use of these in his engine. It is unfortunate that he never fully implemented his analytical engine.

The Countess of Lovelace, Ada Augusta Byron (daughter of the poet Lord Byron) was an associate of Babbage in programming his analytical engine. She is considered by many to be the world's first computer programmer, and the modern high-level programming language **Ada** is named in her honour. **Ada** is the language adopted by the United States Department of Defence for use in embedded systems programming (e.g. flight control software on board a rocket).

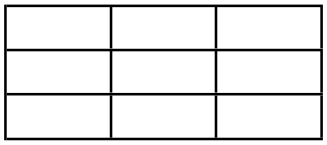
2.6 Summary

- Problem-solving ability can be improved by:
  - Taking care with the problem definition phase.
  - Not being too concerned about detail in the getting-started phase.
  - Using specific examples.
  - Using similar problems as a guide.
  - Working back from a known solution.
  - Using the divide and conquer strategy of top-down development
- The usual procedure in solving a computer problem is:
  - Select a suitable task.
  - Design an algorithm.
  - Express the algorithm as a program.
  - Have the computer execute the program.
  - Determine whether the program works correctly.
- An algorithm is a set of instructions as to how to carry out a process.
- A computer program is an expression of an algorithm as a sequence of primitive operations (statements in a programming language) which a computer can carry out.

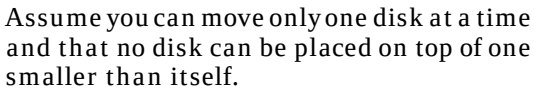
- All but the very simplest algorithms involve one or more of the following:
  - Conditional execution
  - Repetition
  - Memory.
- The technique of top-down development is often used in the design of algorithms.
- Structure charts are diagrams which aid in top-down development.

2.7 Exercises

1. Using the digits 1, 2, 3, 4, 5, 6, 7, 8, and 9, place exactly one of these in each position in the grid shown below so that the sum for each row, column, and diagonal is 15.

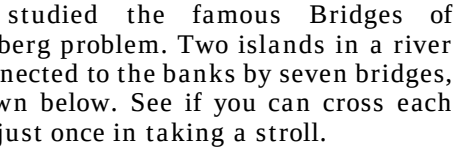


2. What is the minimum number of moves required to move the three disks in the diagram below from A to C?



Assume you can move only one disk at a time and that no disk can be placed on top of one smaller than itself.

3. Euler studied the famous Bridges of Königsberg problem. Two islands in a river are connected to the banks by seven bridges, as shown below. See if you can cross each bridge just once in taking a stroll.



4. The following directions are taken from a shampoo label: WASH HAIR, LATHER, RINSE, REPEAT. An algorithm can be defined as a sequence of operations that, when executed will always produce the correct results in a finite time. Implicit in this definition is that each step is unambiguously defined.

   (a) Which of the steps above are ambiguous? Explain why.
   (b) Rewrite the algorithm to remove these ambiguities.

5. Write an algorithm that specifies how to place a telephone call on STD. Be sure to include provision for the cases where the telephone number is initially unknown, lack of dial tone, getting a busy signal, and the case where no one answers. What happens if an answering-machine answers the call?

6. Each of the following represents an attempt to write an algorithm to compute the sum of the first 10 positive integers. State why each is either poor or invalid. For the purposes of this question, assume that a poor algorithm is one for which a small change in the specification of the problem requires a major change in the algorithm.

   (i) START
   Set sum to 1+2+3+4+5+6+7+8+9+10
   Write sum
   END OF ALGORITHM

   (ii) START
   Set n to 10 and sum to 0
   Add n to sum
   Decrement n by 1
   See if n>0
   Go back and repeat
   Write sum
   END OF ALGORITHM

(iii) START
Set i to 0 and sum to 0
Repeat the following 10 times
Add i to sum
End of loop
Write sum
END OF ALGORITHM

(iv) START
Set i to 0 and sum to 0
While i <= 10 do the following
Add i to sum
End of the loop
Write sum
END OF ALGORITHM

Write a valid algorithm to sum the first 10 integers.

7. Devise an algorithm which describes how to find a book in a library. Ensure that the algorithm describes some sensible process when (a) the book is not held by the library, and (b) the book is held by the library but is not currently on the shelves.

8. Write an algorithm to find the sum of the first k integers 1, 2, ..., k. The value of k will be external input to the algorithm. (Be sure to handle the illegal situation of k <=0).

9. Develop an algorithm to solve quadratic equations of the form $Ax^2 + Bx + C = 0$ using the quadratic formula:

$$\text{Roots} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Your algorithm should handle the usual case of two different, real roots as well as the special cases of:

(a) A double root (B*B - 4*A*C = 0)
(b) Complex roots (B*B - 4*A*C < 0)
(c) A non-quadratic equation (A = 0)
(d) A constant equation (A = 0, B = 0)

10. Write an algorithm to input an integer N and determine if N is prime. A prime number is an integer N > 1 that is not evenly divisible by an values other than 1 and N. The output of your algorithm should be either: "N" is prime or "N" is not prime, "M" is a factor . Make sure that your algorithm operates properly for all values of N.

11. **Challenge Problem.** Devise an algorithm to sort a list of surnames into alphabetical order.

12. Devise an algorithm for determining which letter occurs most often in a piece of text from a novel. Hint : use 26 separate counter variables to accumulate the frequency of each letter.

13. Explain how sequence, selection and iteration are expressed in Pascal.

3 INTRODUCTION TO PASCAL

3.1 Why Do We Use Pascal ?

A preliminary version of the programming language Pascal was drafted in 1968. Based on ALGOL (**ALGO**rithmic **L**anguage), this highly structured, yet simple and elegant language was created by Professor Niklaus Wirth of the Eidgenossische Technische Hochschule (ETH), Zurich, Switzerland. The language obtained its name from the famous French mathematician and philosopher, Blaise Pascal (1623-1662). The first compiler for Pascal became available in 1970. A revised version was published in 1973, and it has since become very popular at universities and colleges offering computing courses. It is also gaining large numbers of adherents amongst owners of personal computers. High-performance products such as **Turbo Pascal** on IBM and compatible personal computers have also given a renewed impetus to the use of the Pascal language. It was precisely the availability of such modern software tools at reasonable prices and the lack of a suitable numerical mathematics text based on Pascal that convinced the authors to write this book.

Professor Wirth gives his reasons for the new language as:

1. "To make available a language suitable for teaching programming as a systematic discipline based on fundamental concepts clearly and naturally reflected by the language", and
2. "To develop implementations of this language which are both reliable and efficient on presently available computers."

Pascal is one of the few programming languages specifically designed to help the programmer locate coding errors quickly. Extensive error checking is performed during compilation and (optionally) during execution. It is not unusual for Pascal programs, once free of syntax errors, to execute error-free almost immediately. Features of the language which promote this desirable situation are:

1. The requirement for declaring how each programmer-defined symbol is to be used.
2. The existence of a rich set of data types plus a variety of programming structures permitting the text of programs to be closely related to the intended meaning. Structured programming essentially involves the writing of programs in a way which is easy to read and understand. This in itself reduces the likelihood of errors.

The Pascal statements that make it an excellent host language for structured programming are:

```
repeat...until
case
if...then...else
while...do
for...to/downto...do.
```

3.2 A Simple Pascal Program

Let's get straight into it by writing a very simple program. The problem we solve is to take two integers which will be supplied as input and compute their product. The result will be printed along with a heading. We start by developing and refining an algorithm using the technique of top-down development described in the previous chapter.

The first step may be a structure chart such as that shown in Figure 3.1.

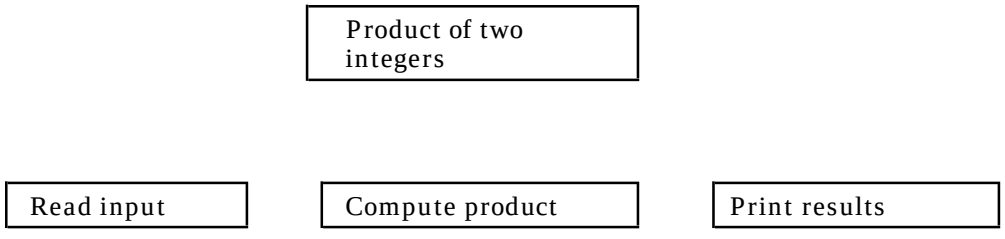


Figure 3.1 Structure Chart - First Attempt

Module 300 can be expressed in a single Pascal statement so it will not need to be refined further. However refinement of modules 200 and 400 is necessary, because they cannot be expressed as single statements in Pascal. As the two integers are read it would be useful to print them so that we could be sure that we had the intended integers for the multiplication. It is also very important to write headings for numbers which are to be printed on a piece of paper in order to make the output more readable. So clearly we need to refine module 200 further, somewhat along the lines given in Figure 3.2.

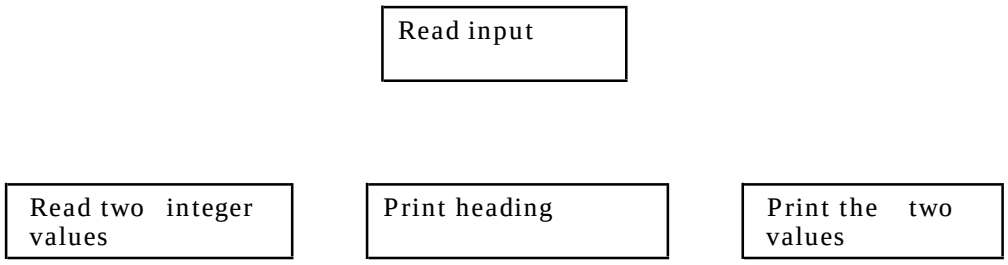


Figure 3.2 Structure Chart - Second Attempt

Each of the modules 210, 220, 230 can in fact be expressed as a single Pascal statement, so no further refinements are necessary. Module 300 also needs no further refinement. A refinement for module 400 is necessary and is shown in Figure 3.3.

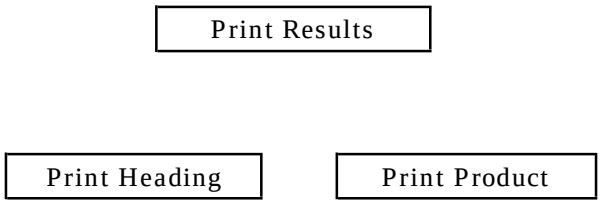


Figure 3.3 Structure Chart - Third Attempt

The complete structure chart appears in Figure 3.4

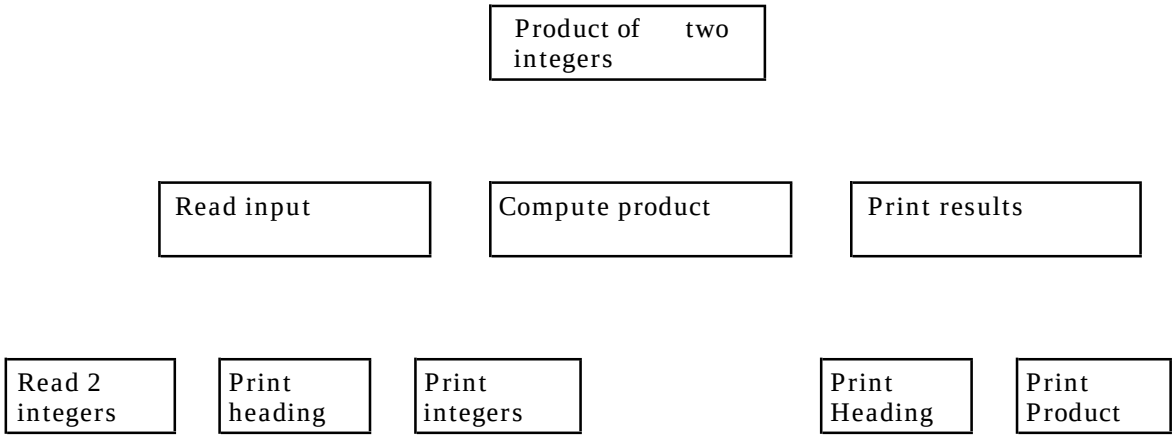


Figure 3.4 Structure Chart - Fourth Attempt

The program is written by coding into Pascal statements the "leaf" modules, i.e. 210, 220, 230, 300, 410, 420. The program with explanations is given below:

```
(0)    Program  SimpleExample;
(1)    { This is a program which takes two integers
(2)      supplied  as input and computes the product.
(3)      The two integer values along with the result
(4)      are printed under a suitable heading }
(5)
(6)    var
(7)       Number1, Number2, Product : Integer;
(8)
(9)    begin
(10)      readln(Number1, Number2);                  (210)
(11)      writeln('Integer values read in are : '); (220)
(12)      writeln(Number1,'  ',Number2);             (230)
(13)      Product := Number1 * Number2;              (300)
(14)      writeln('Product of numbers is :');        (410)
(15)      writeln(Product)                           (420)
(16)   end.
```

Let's now go through the program line by line.

The numbers down the right and left hand sides in ( ) parentheses are simply for reference purposes and are not part of a Pascal program. The numbers on the left are line numbers and those on the right refer to module numbers in the structure chart.

In line 0 we simply give the program a name.

Lines 1-4 contain a comment and are ignored by the compiler. They are put in to aid readability. Text in { } or (* *) brackets is regarded as comments, and ignored by the compiler.

Lines 5 & 8 are blank lines inserted to enhance readability.

Lines 6 & 7 constitute a declaration statement. Here the compiler is told that the program uses three **variables**, with the names given. Also the variables are to hold only integer values.

Lines 9 & 16 simply indicate the beginning and end of the program statements.

In line 10 two numbers are read from the input device (in this case the keyboard) and stored in the variables called Number1 and Number2.

In line 11 everything in apostrophes (also called single quotation marks, or even just single quotes ' ') is printed (on the screen in this case).

The values stored in Number1 and Number2 are printed in line 12.

In line 13 the numbers in Number1 and Number2 are multiplied together and the result stored in the variable Product.

Line 14 prints another message, while line 15 prints the value stored in Product.

Line 16 ends with a full-stop. All programs must end this way.

Points to note about the layout of the program:

- A mixture of upper and lower case letters has been used to improve readability. Blank lines and spaces are there for the same purpose.
- Reserved words like **var**, **begin**, and **end** are shown in bold print. They have a special meaning in Pascal. A semicolon (;) is used to separate a declaration, statement, or groups of statements in **begin** ..... **end**, from the succeeding declaration, statement, or group of statements. Reserved words are not considered statements and hence a ";" is not required after a **begin** or before an **end**.
- Going over a line is acceptable, that is, you can extend a single statement over more than one line. Also you can put more than one statement on a line. Always be careful not to crowd up your statements and thereby make them hard to read.

3.3 Variables, Identifiers, and Assignments

The concept of a **variable** is fundamental to all programming. It is essentially a memory cell or group of cells which hold information. The information held by the variable is called its **value**. Variables can hold integers, non-integer numbers, characters, strings of characters etc. It is important to understand that the value of a variable normally changes during the execution of a program.

In order that we may refer to a variable it must be given a name, or **identifier**. The term **identifier** in Pascal refers not only to a variable but also to the name of a constant, data type, procedure, function, or program. Certain practical rules apply to identifiers, so that the compiler has a fighting chance of recognising them. These are:

1. An identifier must begin with a letter.
2. After the first letter you can have any combination of letters and digits. Certain implementations of the Pascal language, for example, Turbo Pascal may allow additional characters to be legal in identifiers. The underline ("underscore" in American books) character "_" is one which is often permitted.
3. Standard Pascal limits the length of identifiers to at least one but no more than 8 characters, however most modern version of the language allow many more. For example, up to 127 are permitted in Turbo Pascal V3.0.
4. Some words are known as **reserved words** and have a special meaning to the compiler. These words therefore cannot be used by the programmer as identifiers. Some common reserved words are: **program**, **var**, **with**, **while**, **to**, **if**, **else**, **end**, **or**, **then.** The complete set of reserved words can be found in your Pascal manual.
5. In Pascal there are also standard identifiers which have pre-defined meanings, but they can be used as identifiers. However if in a program a standard identifier is used as an ordinary identifier, its pre-defined meaning is ignored.

Some examples of identifiers are:

x	FatCat
A1	THINdog
Radius	HoursWorked.
NUMBER	FAT_CAT

When you write programs you should choose the name of an identifier so that it will easily convey the type of information the variable stores, for example, PayRate, TotalTax are better than P, PR, or T.

The identifiers given in Table 3.1 would **not** be acceptable in Pascal.

Identifier	**Problem**
$money	Does not start with a letter
3Pay	Does not start with a letter
while	This is a reserved word
a20,00	Commas not allowed
NUMBER(1)	Parenthesis not allowed
0VERTIME	Does not start with a letter
Div	This is a reserved word

Table 3.1 Illegal Pascal Identifiers

Variables must be declared at the start of a program so that the compiler can allocate sufficient memory space to the variables, and to check that each is used to hold only one particular **type** of value (e.g. integer, character). An example of a sequence of **variable declarations** is:

```
var
   Radius, Circumference : Integer;
   Pi     : Real;
   Text   : Char;
   Answer : Boolean;
```

Integers

Data of type **Integer** can take the values (-1-maxint), ...0, 1, 2, 3,...., maxint, where maxint is an implementation-defined limit that is commonly 32,767 (2**15 -1) on small micro-computers. Therefore, on such a machine the range of integers would be -32768 to 32767. Turbo Pascal on the IBM PC and compatibles has this range. **Operators** are symbols that represent the common mathematical operations such as addition, multiplication, and so on. The **arithmetic** operators are given in Table 3.2.

Pascal Arithmetic Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
div	integral part of division (quotient)
mod	remainder after division

Table 3.2 Pascal Arithmetic Operators

Next, we have the **relational** operators; so named because they yield True/False values (called **Boolean** values in Pascal) depending on the relationship between two quantities. These operators should already be familiar to you from algebra, however since standard character sets on the computer do not have compound symbols such as ≥, we sometimes need to place two symbols side-by-side in Pascal. For instance, we represent the ≥ symbol in Pascal by >= , that is, the greater than symbol followed by the equals symbol. Full details are in Table 3.3.

Algebraic Relational Operator	Pascal Relational Operator	Meaning
=	=	Equals
<	<	Less Than
>	>	Greater Than
≤	<=	Less Than Or Equal To
≥	>=	Greater Than or Equal To
≠	<>	Not Equal To

Table 3.3 Pascal Relational Operators

Integers also can be arguments of the absolute value and squaring functions **abs()** and **sqr()** respectively.

Data of type **Real** consists of an implementation-defined subset of the rational numbers. A real number can also be written in Pascal scientific notation. In scientific notation, a real number begins with an integer or real value followed by the letter E and an integer exponent (possibly preceded by a sign). Examples of valid and invalid real numbers are given in Table 3.4.

Valid Real Numbers	Invalid Real Numbers	
3.143256	160.	(no digit after .)
-0.005	-.12345	(no digit before .)
+12345.0	.16	(no digit before .)
-15E-04 (= -0.0015)	-15.E-03	(-15. is an invalid real)
-2.345E2 (= -234.5)	12E.3	(.3 is an invalid integer)
1.2E+6 (= 1200000)	.123E	(.123 is an invalid real)
1.15E-3 (= .00115)	.0	(no digit before .)

Table 3.4 Valid & Invalid Real Numbers in Pascal

As shown by the last valid number above, 1.15E-3 means the same as $1.15x10^{-3}$. The exponent (-3) causes the decimal point to be moved left three digits. A positive exponent causes the decimal point to be moved right. The + sign may be omitted when the exponent is positive.

In a Pascal program, when we wish the computer to write real values, we may specify the format of the number. To do this, we must tell the computer the maximum number of characters in the number (the **field width)**, and how many decimal places we want. Such directions are known as **format specifications.** Table 3.5 shows some real values printed using different format specifications.

Value	Format	Printed Output
3.14159	:5:2	•3.14
3.14159	:5:1	••3.1
3.14159	:5:3	3.142
3.14159	:8:5	3.14159
3.15159	:9	••3.1E+00
-0.0006	:9	-6.0E-04
-0.0006	:8:5	-0.00060
-0.0006	:8:3	••-0.001

Table 3.5 Real Output Formatting In Pascal

Examples of output statements are:

```
writeln('Average speed in kph was ', Speed :5:1);
writeln('Distance travelled was ', Dist :5:1);
```

Operators which can be used on Reals are +,-,*, and /. Examples of functions available are: abs, sqr, sqrt, sin, cos, exp, ln (more about these later).

Boolean data can only take the values **True** and **False.** Operators used with Boolean data are: **and**, **or** and **not**.

A variable of type **Char** can store exactly one character. Possible values are generally characters taken from the set of ASCII characters, but this is implementation-dependent. On IBM and compatible micro-computers, the so-called **extended ASCII** character set is available. Character data can be compared with the same relational operators as for Integer and Real values. This means in particular that we can **sort** character data into order (a generalisation of the notion of **alphabetical order**). Intrinsic functions available for Char data are **Ord**, **Pred**, and **Succ**.

You should always realize that when variables are declared (i.e. memory cells are allocated) the contents of the chosen memory cells are likely to contain any old value left there from the previous program. The variable is said to be **uninitialized** - initialization takes place as soon as the first value is stored in the variable.

Values may be stored in a variable by means of an **assignment statement.** Examples are:

```
Pi := 3.142;
Count := 1;
Circumference := 2*Pi*Radius;
```

Remember also that a variable can only store the type of value you have declared it will hold. For example, Radius above can only store integer values. If you try to assign the wrong type of value you will get a compile-time error. A variable can only take one value at a time, but it can hold many different values during the course of the execution of the program. An assignment such as:

```
Count := Number;
```

can be used to copy the value of the variable Number into the memory location reserved for the variable Count.

An important restriction in writing assignment statements is that all items in an assignment statement - constants, variables, and functions must be of the appropriate type (integers, characters, reals, or booleans). In this way, the Pascal compiler is able to detect many errors and tell us before we try to run the program. For example, the Pascal compiler will prevent us from trying to add a character to an integer.

Another type of declaration statement in Pascal is the **constant declaration.** An example is given below:

```
const
  PI = 3.142;
```

This declaration assigns to PI the value 3.142 for the duration of the program. Once this is done, it is illegal to attempt to change the value of PI.

3.4 Arithmetic Expressions

Arithmetic expressions specify calculations which are to be carried out on the current values of certain variables in primary memory. We can place them on the right hand sides of assignment statements so that the value computed will be stored in the memory location given by the left hand side. In this way, arithmetic expressions allow us to carry out arithmetic. Examples are:

```
Circumference  := 2*Pi*Radius;
Discriminant   := B*B - 4*A*C;
NettValue      := Final - Initial;
TotalCost      := LabourCost + MaterialsCost.
```

These examples all illustrate arithmetic expressions as used in **assignment statements.** They are used to compute a value for storage in the memory location occupied by the variable given on the left hand side. We may also directly write the value of an arithmetic expression. Examples are:

```
writeln('Discriminant is ',B*B - 4*A*C);
writeln('Amount of Sales Tax is $ ',Price*0.2);
```

We must always ensure that the left hand side of an assignment statement is a variable identifier and not an expression, since the left hand side always specifies the **destination** for storing the result of the expression on the right hand side. We may write z:=x+y which means - compute the sum of x and y based on current stored values, and then store the result in the memory cell/s reserved for z. An arithmetic expression can also be just a single variable: z:=x; which means to copy the value of x into the z's memory cell. On the other hand, x:=z; means to copy the value of z into x's memory cell, which is quite different. We can even say x:=x+1; which means - replace the current value of x with one more.

It is illegal and nonsensical to write Pascal statements such as x+1:=y , since the left hand side does not identify a memory location. No Pascal program containing such a statement could ever be run on the computer, since this error would be detected and reported by the compiler. It is a **syntax** error.

Precedence Rules

Just as in ordinary algebra, where we have rules for the order in which operations are done, we also need to follow some **rules of precedence** when writing arithmetic expressions in Pascal. These rules are necessary so that there are no ambiguities:

1. The sub-expressions in the most deeply nested parentheses (brackets) are evaluated first.
2. The order of arithmetic operators is: * / div mod + - .
3. Subject to rules 1 and 2, operations are carried out from left to right.

Some further examples of mathematical expressions in Pascal are given in Table 3.6.

Mathematical Expression	Pascal Expression
$\frac{a+2}{b+3}$	**(a+2)/(b+3)**
$\frac{xy}{w+4}$	**x*y/(w+4)**
$\frac{-b+\sqrt{b^2-4ac}}{2a}$	**(-b+sqrt(b*b-4*a*c))/(2*a)**

Table 3.6 Algebraic Expressions In Pascal

3.5 Input and Output

Computers are very good at performing calculations and doing precisely what they are told. However clearly we must have a way of putting data into a computer and getting the results out of the computer's memory. For these purposes we have input and output statements in Pascal. The input

statements are **read** and **readln**, and the output statements are **write** and **writeln**. Every program should have at least one output statement so that the results can be seen. An input statement allows a program to obtain data values from an external device such as a keyboard or disk drive. The program then of course can assign the data values to certain variables.

Input

An example of a simple input statement is:

```
readln(Radius, Number)
```

The effect of this statement is that the values taken from the input device (keyboard in this case) are assigned to the variables Radius and Number. An input device can be compared to a conveyor belt from which items are taken on each read statement. Hence a particular value can only be read once.

Because input statements assign a value to a variable they can be thought of as an assignment statement, i.e. readln(variable) is essentially the same as - variable := the value of the next data item.

Any value assigned in an input statement must be of the **same type** as the variable into which it is read. If we assume that Radius and Number were declared to be integers and the data were presented as:

```
34
78
```

then Radius would have assigned to it a value of 34 and Number a value of 78.

The **read** statement is similar to **readln** except that after a value is read the next value is expected to be found on the same line. For example, if you used a **read** statement instead of **readln**, then to assign a value of 34 to Radius and a value of 78 to Number the data would have to be presented as:

```
34 78
```

Some rules concerning the physical layout of input data are:

- Values cannot extend over more than one line.
- Values are separated by a blank or they are found on the next line.
- Values to be read by a readln statement must be terminated by pressing the "enter" key, i.e. moving to the next line.

If you want to avoid a run-time error make sure that the number of executed input statements (for example, readln(Radius, Number) counts as two) does not exceed the number of data items supplied. However if the number of data items is more than the number of executed inputs then some data items will not be read.

Programming Tip

> On some occasions a program will produce wrong results even though there is nothing wrong with the program. What has happened is that the data that has gone into the program is not what was intended. It is imperative that input data be checked immediately for correctness and plausibility - otherwise the results will follow the well-known saying "garbage in garbage out" (GIGO).
>
> One simple way of validating input data is to print it back out immediately and check it visually. This is often referred to as "echo printing" and should be incorporated into every program, at least until you are confident that the input data are being read correctly.

Output

An example of a simple form of output statement in Pascal is

```
writeln(Radius,Circumference).
```

However, if just two numbers appeared on the screen then it might be difficult to know which was the radius and which was the circumference. We can improve the output by using headings or words to make the meaning more clear. For example the above writeln statement might be rewritten as:

```
writeln('Radius = ',Radius,' Circumference = ',Circumference)
```

and the output might appear as :

```
Radius = 5    Circumference = 31.416
```

A **literal** is any string of characters enclosed in single quotation marks (actually apostrophes). For example, 'This is a message for all you students to wake up'. This is the means by which you can write messages and headings. As an other example the statement:

```
writeln('Game won in ',N,'moves')
```

would produce the output:

Game won in 12 moves (if the current value of N were 12.)

Programming Tip

> The output of a program is there to be read by humans.
>
> Users of programs care little about the programmer's cleverness but are more concerned with correct answers presented in a clear, easy-to-read format.
>
> Always take care to ensure your program gives correct results but once it is working correctly, it is worthwhile to spend some time making the output reasonably attractive.

Let's take a look at a program to solve a quadratic equation, paying particular attention to readln and writeln statements.

```
Program QUADRATIC;
{This is a program which solves a quadratic equation
given the coefficients as input}
var
  a,b,c,root1,root2 : Real;
begin
  writeln('   ':25,'Roots of the Quadratic Equation');
  writeln;
  write(' ':14, 'a', ' ':14, 'b', ' ':14, 'c');
  writeln('root1', ' ':10, 'root2');
  readln(a, b, c);
  root1 := (-b+SQRT(b*b-4.0*a*c))/(2.0*a);
  root2 := (-b-SQRT(b*b-4.0*a*c))/(2.0*a);
  writeln(a:15:3, b:15:3, c:15:3, root1:15:4,root2:15:4);
end.
```

The output of the program is:

Roots of the Quadratic Equations

a	b	c	root1	root2
1.000	3.000	2.000	-1.0000	-2.0000

You should study the program along with its output. The use of the colons (:) for formatting should then become apparent.

Programming Tip

Comments should be used within a program to explain what the program is doing. Good comments can be of enormous value in trying to understand a program or part of a program. You should use comments not only for someone else but also for yourself. As you work on programs which are larger and larger and cannot be completed in a day you will start to forget what you have done. It will become important that you start to work in modules with suitable commenting.

Although you may adopt your own specific style for commenting here are some do's and don'ts:

- Don't undercomment
- Don't overcomment
- Do not rehash obvious program statements
- Include an extensive comment block at the beginning of a program
- Use comments to block or paragraph your program
- Keep the comments up-to-date

If you practice good commenting on the small programs you will be writing this year, it will be of great benefit to you if you need to write larger programs later on.

3.6 Conditional Execution

Let's take two numbers and suppose we want to know the absolute difference (i.e. the larger number minus the the smaller number). Clearly the way in which the difference is computed depends on which number is larger. Assuming that the numbers are held in variables FIRST and SECOND then a Pascal statement to compute the absolute difference is given given below:

```
if FIRST >= SECOND
then DIFFERENCE := FIRST - SECOND
else DIFFERENCE := SECOND - FIRST
```

A complete program to carry out the computation is shown below:

```
Program AbsoluteDifference;
{This is a program to compute and print the difference
 of two integers supplied as input }
var
  FIRST, SECOND, SUM, DIFFERENCE : Integer;
begin
  readln(FIRST,SECOND);
  writeln('Input values are : ', FIRST, SECOND);
  if FIRST >= SECOND
    then DIFFERENCE := FIRST - SECOND
    else DIFFERENCE := SECOND - FIRST;
  writeln('Absolute Difference is : ', DIFFERENCE)
end.
```

In Pascal the conditional statement with two branches has the general form:

```
if condition
then alternative 1
else alternative 2
```

Note that this is a single statement with no semi-colons. The indentation and the three lines are simply to make the condition and two branches obvious. The compiler does not care either way.

Alternatives 1 and/or 2 may be single statements or a sequence of statements. In the case of a sequence of statements, the sequence must begin with **begin** and finish with **end** (so that the compiler knows where the sequence starts and finishes). Such a sequence, enclosed in a **begin ... end** pair, is called a **compound statement**. An example to illustrate compound statements in conditional execution is given below:

```
if REQUEST <= BALANCE
  then begin
    BALANCE := BALANCE - REQUEST;
    writeln('Withdrawal OK')
  end
  else begin
    BOUNCED := BOUNCED + 1;
    writeln('This person is a fraud')
  end
```

We may have a conditional statement with only one branch. This is the same as having an *empty* **else** part. We do nothing if the condition turns out to be false. The conditional statement with only one branch has the general form:

```
if condition then statement
```

An example is:

```
if X < Y then begin
  DUMMY := X;
  X := Y;
  Y := DUMMY
end
```

What do these statements do?

The statements which occur after the **then** or **else** in a conditional statements may themselves be conditional. In this case a great deal of care needs to be taken so as not to have an extra **else** dangling around. Nested conditional statements can be a nightmare for the beginner. We illustrate (but will not consider further) nested conditional statements with the example below:

```
Program Quadratic;
{This is a program to find the roots of a quadratic equation
with given  coefficients}
var
  a,b,c,discriminant,re,im: Real;
begin
  readln(a,b,c);
  if (a=0) and (b=0)
    then writeln('The equation is degenerate')
    else if a=0
           then writeln('The only root is ', -c/b)
           else if c=0
                  then writeln('The roots are ',-b/a,' and ',0)
                  else begin
                         re := -b/(2*a);
                         discriminant := sqr(b) - 4*a*c;
                         im := sqrt(abs(discriminant))/(2*a);
                         if discriminant >= 0
                           then writeln('Roots: ',re+im,'and ',re-im)
                           else begin
                                  write('The roots are complex : ');
                                  writeln(re,'+I*',im,'and',re, '-I*',im)
                                end
                       end
end.
```

Pascal gives us a statement when we want do a conditional execution with several branches. For example, suppose we want to print the name and the corresponding day of the week. Suppose we are given an integer variable DayOfWeek with values in the range 1 to 7. We could then carry out our task with the (rather long) Pascal statement below:

```
if DayOfWeek=1
  then writeln('Sunday')
  else if DayOfWeek=2
         then writeln('Monday')
         else if DayOfWeek=3
                then writeln('Tuesday')
                else  ........
```

This is obviously very tedious and likely to disappear off the right-hand side of the page. Fortunately Pascal provides the **case statement** to help us out of the problem. The **case** statement which is the equivalent of the above **if** ... **then** ... **else** statement is given below:

```
case DayOfWeek of
  1: writeln('Sunday');
  2: writeln('Monday');
  3: writeln('Tuesday');
  4: writeln('Wednesday');
  5: writeln('Thursday');
  6: writeln('Friday');
  7: writeln('Saturday')
end
```

Depending on the value of DayOfWeek, one and only one of the seven possible branches is executed.

The general form of the case statement is:

```
case expression of
  CaseLabel1: Alternative1;
  CaseLabel2: Alternative2;
  .
  .
  .
  CaseLabelN: AlternativeN
end
```

To execute the case statement the expression is first evaluated and the result must equal one of the specified case labels. The alternative which corresponds to the case label is then executed. Each alternative can be any compound statement. If no case label matches the expression then the outcome is *undefined*, although many modern versions of Pascal such as Turbo allow an **else** or **others** clause at the end to cater for this possibility. If you wish to use this non-standard feature, consult your Pascal compiler manual. An example is given below:

```
var
  Month    : 1..12;
  LeapYear : Boolean;
  Year     : Integer;
begin
  LeapYear:= (Year mod 4 = 0);
  case Month of
    1,3,5,7,8,10,12 : writeln('31 days');
    4,6,9,11        : writeln('30 days');
    2               : if LeapYear then writeln('29 days')
                                  else writeln('28 days')
  else writeln('Illegal Month Number')
end
```

To write conditional statements which are a little more complex than we have been doing so far we will need to make use of **Boolean expressions** - expressions that have one of the two values **True** or **False.**

Boolean expressions can be formed by putting one of the relations:

<	less than
<=	less than or equal
=	equal
<>	not equal to
>=	greater than or equal to
>	greater than

between two arithmetic expressions. The resulting value is either True or False.

Some examples of simple and compound Boolean expressions are given below:

X+Y = 0
B*B > 4A*C
P+Q <> R-6
(X+Y = 0) or (P > Q)
(B*B > 4*A*C) and (A <> 8)

Using compound Boolean expressions we can write complex conditions in conditional statements, for example:

```
if (B*B > 4*A*C) and (A <> 0) then ..... else ........
```

The precedence rules which govern the order of carrying out logical operations are similar to the precedence rules of ordinary arithmetic. The result of a Boolean variable can of course only take the values **True** or **False**. Boolean variables are declared along with other variables at the start of a program. Some common operations on True/False values are **not**, **and**, and **or**. If we think of False as corresponding to 0, True to 1, then **not** corresponds to complement, **and** to multiplication, and **or** to addition (but 1+1=1 since true or true = true). The Boolean **or** operation is like set union with the universal set: if you keep taking set unions, it can still get no bigger. In fact, the entire algebra of Boolean expressions is like that of union and intersection of sets, as the following functions, called **truth tables**, in Tables 3.7 and 3.8 illustrate.

AND	0	1
0	0	0
1	0	1

Table 3.7 Boolean **AND** Operator

OR	0	1
0	0	1
1	1	1

Table 3.8 Boolean **OR** Operator

Example of Boolean variable declarations are:

```
var
  COUNT, MAX, MIN: Integer;
  FINISHED, FOUND: Boolean;
```

Boolean variables can be assigned one of the two Boolean constants, for example:

```
FINISHED:=True;

if NEXT <> WANTED then FOUND:=false;
```

Below is a program which uses Boolean variables. Study the program and try to predict the output if the input values are 6 and 9:

```
Program Compare;
{This is a program which takes two integers as input and compares
the first with the second. It indicates whether each of the tests
"less than", "equals", or "greater than" is true or false}
var
  a,b: Integer;
  less,equals,greater: Boolean;
begin  {Compare}
  {Input and echo the data}
  readln(a,b);
  writeln('a = ', a, 'and b = ', b);
  less:=(a<b);    {Perform the three relational comparisons}
  equals:=(a=b);
  greater:=(a>b);
  writeln('a<b is',less);   {Write the results of the comparisons}
  writeln('a=b is',equals);
  writeln('a>b is',greater)
end. {Compare}
```

3.7 Condition-Controlled Iteration

If we had a pair of integers for which we wanted the sum and product then you could easily write a program with the Pascal knowledge you have so far. However if you wanted to do the same calculations on several pairs of integers you would have to run the program several times with a different pair of integers as input each time. This of course would be very tedious. We would like to make the program keep processing pairs of integers until we told it to stop. In other words we want to introduce a means of carrying out **repetition** (or **iteration**) into the program.

As soon as we introduce repetition we must have a means of determining when the repetition is to stop. Suppose that we want to stop the processing when the difference between the integers is 100. The pseudocode might be written as:

```
repeat
  Input pair of integers
  Compute product & sum
  Print integers, product, and sum
until difference is 100
```

A program to carry out this processing is given below. It should be self explanatory.

```
Program ComputeSum&Product;
{This is a program which calculates the sum and product of two
integers. It terminates when the difference is 100}
var
  FIRST, SECOND, SUM, PRODUCT, DIFFERENCE: Integer;
begin
  repeat
    readln(FIRST,SECOND);
    SUM:=FIRST+SECOND;
    PRODUCT:=FIRST*SECOND;
    DIFFERENCE:=FIRST-SECOND;
    writeln('Input Values are : ', FIRST, SECOND);
    writeln('Sum & Product are : ', SUM,PRODUCT)
  until DIFFERENCE=100
end.
```

The repetition used above is an example of what is called a **post-tested** loop, i.e. carry on doing something **until** some condition is true. Some examples of this type of everyday repetition are:

```
Rinse until free of shampoo.
Stir until smooth.
Go out on dates until you find the perfect partner.
Fry until crisp.
Keep looking for the ball until you find it.
```

In the **repeat ... until** form of repetition we do something first and then we test to see if we have to do it again. There is another form of repetition called **pre-tested**, in which the test is performed first in order to see if anything has to be done at all (more about this type of repetition a little later in this section).

The Pascal expression of a **post-tested** repetition is:

```
repeat
  statement(s)
until condition
```

where "condition" may be any Boolean expression. The structure above is often referred to as a **loop**.

Clearly with a post-tested loop the body of the loop (the statement(s) between the **repeat** and **until**) must be carried out at least once.

We now look at the problem of finding the sum of a sequence of integers that is terminated with a zero. A program to carry out this task is given below:

```
Program Summation;
{This program adds up a sequence of integers.
The summation is terminated when a zero value is encountered.}
var
  SUM, NEXT: Integer;
begin
  SUM:=0;
  repeat
    readln(NEXT);
    SUM:=SUM+NEXT;
  until NEXT=0;
  writeln('Sum of integers is :', SUM)
end.
```

Notice that the variable SUM has been initialized to zero. If it hadn't then any "old rubbish" could have been held in the memory location reserved for SUM and would have given an incorrect answer for the summation, unless SUM happened to start off at 0.

Also you should notice that because the loop is post-tested the terminating value of zero is added to SUM the last time the loop is executed. Of course in this case it doesn't change our final answer, but could easily present difficulties for other types of problems.

Instead of wanting to find the sum of a sequence of integers terminated by a zero a student wanted to find the product. He wrote the program below:

```
Program Product;
{This program computes the product of a sequence of integers
terminated by a zero.}
var
  PRODUCT, NEXT: Integer;
begin
  PRODUCT:=1;
  repeat
    readln(NEXT);
    PRODUCT:=PRODUCT*NEXT;
  until NEXT=0;
  writeln('Product of integers is : ', PRODUCT)
end.
```

There is a major fault with the program. Can you spot it before reading on? The problem of course is that the answer is always zero. See if you can patch up the program before reading on. Some students may have changed the body of the loop in the following way:

```
PRODUCT:=1;
repeat
  PRODUCT:=PRODUCT*NEXT;
  readln(NEXT)
until NEXT=0;
```

This would certainly be a good try but there still is a problem. What is it? In this case when each integer is read it is tested before being used in the multiplication and of course a zero would not be used. However the first time around the program would not have a value for NEXT - in fact there would be but it would be the value left behind by the last program that used the memory cell allocated to NEXT. See if you can remove this shortcoming before reading on. A student's next version is shown below:

```
PRODUCT:=1;
readln(NEXT);
repeat
  PRODUCT:=PRODUCT*NEXT;
  readln(NEXT)
until NEXT=0;
```

This looks good - NEXT is properly initialized to the first integer and the terminating zero is not included in the product. But there is still a problem! Writing a program which gives correct results for all input values can be an exceedingly frustrating task. Have you spotted the problem? The program would work satisfactorily most of the time but trouble would start if the first integer was a zero. Since two readln statements are always executed the program will attempt to read beyond the terminating zero. Hence the result will either be wrong or if there are no more integers to read a fatal error will result. So despite all of our attempts at patching up the program the post-tested nature of the loop is still a problem for this particular example. What we really need is another type of loop - the pre-tested one.

Pre-tested loops are available in Pascal and have the form:

```
while condition do statement .
```

The important point to note is that the condition is tested before the body of the loop is executed. It is possible that sometimes the body might not be executed at all. Whereas the body of a **repeat** loop is executed as long as the condition is false, the body of a **while** loop is executed as long as the condition is true. It is possible to write **repeat** and **while** loops which are equivalent except that one is pre-tested and one post-tested, i.e.

```
repeat                    while not condition do begin
  statements                statements
until condition           end .
```

These loops are essentially equivalent except that, if the condition is initially TRUE, the repeat loop will execute at least once, whereas the **while** loop will never get off the ground. Strictly speaking, our equivalence should be written as follows:

```
repeat                    statements;
  statements              while not condition do begin
until condition             statements
                          end .
```

We go back to our previous example where we wanted to find the product of a sequence of integers. Using a pretested loop the program might look like:

```
     Program Product;
     {This is a program which computes the product of a sequence of
integers.
     It uses a pretested loop.}
     var
       PRODUCT, NEXT: Integer;
     begin
       readln(NEXT);
       PRODUCT:=NEXT;
       while NEXT <> 0 do begin
         PRODUCT:=PRODUCT*NEXT;
         readln(NEXT)
       end;
       writeln('Product of integers is : ', PRODUCT)
     end.
```

Notice that the main body of the loop is not executed in the case where NEXT=0.

Clearly the use of repetition requires that great care be taken - particularly with regard to **initialization** and **specification of the terminating condition.**

Remember, nearly all errors occur on either the first time through the loop or the last.

Here are some rules when using repetition:

- Use a **repeat** loop only when you know that the loop can **always** be executed safely at least once.
- If there are cases in which a loop should not be executed then you **must** use the **while** loop.
- If a **while** loop is used, the variables used in the terminating condition should be initialized before the loop is entered.
- If a **repeat** loop is used, all variables used in the terminating condition should be initialized either before the loop is entered or during the body of the loop.
- On the whole you will find **while** loops more useful than **repeat** loops. In fact use the **while** loop unless there is a specific reason not to.

Programming Tip

Always pay a lot of attention to the exact number of times a loop will be executed. Often although a loop is set up correctly it produces wrong results because the loop is executed once too often or one time too few - the so-called "off-by-one-error".

In the example below which attempts to sum the first n positive integers the result is incorrect because it is actually finding the sum of the first n-1 integers:

```
sum:=0;
number:=1;
while number < n do begin
 sum:=sum+number;
 number:=number+1
end
```

The correction is of course - **while** number <= n **do** .

The trouble with off-by-one errors is that they do not give error messages - just wrong answers. This is an example of the kind of error that in an earlier chapter we called an **algorithmic error.** We in fact solved the wrong problem. In some cases it may not be so obvious that the results are wrong. This type of error is often difficult to detect.

Make sure that you examine each loop very carefully as soon as you have written it.

3.8 Count-Controlled Iteration

Suppose we want to find the sum and difference of successive pairs of integers, and we know exactly the number of pairs (say 1000). So in this case we need to stop the repetition after 1000 cycles. This type of repetition is called **definite** iteration as opposed to the **indefinite** iteration considered in the previous section. Examples of definite and indefinite iteration in real life situations are given in Table 3.9.

Definite Iteration	Indefinite Iteration
Keep jogging for 7 kms	Wait till the train comes
Wait at the shop for 1 hour	Wash until clean
Jog around the block 4 times	Play it until you get it right

Table 3.9 Definite & Indefinite Iteration

To find out whether a given loop is an instance of **definite** iteration, we must ask ourselves the question: *"At run time, do we know **in advance**, how many times the loop will be traversed?"* If the answer to this question is "yes", then the loop in question is an instance of **definite iteration;** if "no" then it is **indefinite iteration.**

If we wanted to work out the sum and difference of 1000 pairs of integers we could use the **while** loop in which the terminating condition is based on a count of the number of repetitions. A program to do this is given below:

```
Program Computation;
{This program computes the sum and difference of 1000
pairs of integers}
var
  COUNT,FIRST,SECOND,SUM,DIFFERENCE: Integer;
begin
  COUNT:=1;
  while COUNT<=1000 do begin
    readln(FIRST,SECOND);
    SUM:=FIRST+SECOND;
    DIFFERENCE:=FIRST-SECOND;
    writeln('Sum and Difference are : ',SUM, DIFFERENCE);
    COUNT:=COUNT+1
  end
end.
```

Pascal however allows us to construct a loop without having to keep a running count. This can be done using the **for** loop. The above program is rewritten using this loop below:

```
Program Computation;
{This program uses the for loop to find the sum and
difference of 1000 pairs of integers}
var
  COUNT,FIRST,SECOND,SUM,DIFFERENCE: Integer;
begin
  for COUNT:=1 to 1000 do begin
    readln(FIRST,SECOND);
    SUM:=FIRST+SECOND;
    DIFFERENCE:=FIRST-SECOND;
    writeln('Sum and Difference are :', SUM,DIFFERENCE);
  end
end.
```

Notice that COUNT is simply a **control-variable**. It takes the values 1, 2, 3, ..., 1000 i.e. the body of the loop is repeated 1000 times. The general forms of the **for** loop are given below:

```
for ControlVariable := InitialValue to FinalValue do statement

for ControlVariable := InitialValue downto FinalValue do statement
```

where the control variable is any integer variable and the initial and final values can be any expressions which have an integer value. If the control value is descending in value then the **downto** keyword must be used.

Notice that the **for** loop is a pre-tested one and therefore if the final value is less than the initial value the loop is not executed at all, unless we have planned it to go backwards; in which case we must use **downto** rather than **to.** The increment to the control variable comes at the **end** of each iteration, i.e. the control variable maintains the value it had at the start of the loop. Also the initial and final values for the loop are evaluated before the first iteration.

3.9 Running, Debugging and Testing Programs

When you type in your program and try to run it you will find it rarely runs correctly the first time. "Life wasn't meant to be to easy", to quote an Australian prime minister. Murphy's Law, "If something can go wrong it will", certainly seems to apply in the area of computer programming. The process of removing the errors (commonly referred to as bugs) is called **debugging**.

When an error is detected an error message is given indicating what type of error it is. If you are using Turbo Pascal the position of the first compilation error in the program is given by a flashing bar (the cursor). Error messages can sometimes be difficult to interpret and sometimes misleading. With experience you will become more proficient at understanding them.

As mentioned in the previous chapter there are three main types of error - compile-time, run-time, and algorithmic. Syntax errors are detected by the compiler (at compilation time) as it tries to translate your program. Statements with syntax errors cannot be translated and hence your program cannot be executed. Run-time errors are detected during the execution of a program and occur because the computer is being asked to do something it is incapable of doing. When program execution is stopped because of a run-time error, a diagnostic message is usually given together with information indicating the location of the error.

As an example of syntax errors type in the following program and remove the syntax errors:

```
Program Payroll;
{This is a program which calculates gross and net pay
given the number of hours worked and the pay rate.
const
  Tax : 25;
var
  Hours, Gross, Rate : Real;
begim
  writeln('Enter hours worked');
  readln(Hours)
  writeln('Enter hourly rate);
  readln(Rate);
  Hours * Rate := Gross;
  Net := Gross - Tax
  writeln('Gross pay is $' Gross;
  writeln('Net pay is $', Net)
end
```

Remember the Pascal reserved words are in bold simply to aid readability and it will not be possible for you to use this feature in your programs.

As an example of run-time errors type in the following example and attempt to correct the errors (at least two major ones are present):

```
Program Errors;
{This program is written to illustrate run-time errors}
const
  x=0.0;
var
  y,z: Real;
begin
  y:=5.0;
  z:=y/x;
  y:=y+sqrt(y-6.0);
  writeln(x,y,z)
end.
```

Debugging a program can be a very time-consuming task. Always make sure you design your algorithms carefully. In practice one of the first steps in finding a hidden error is to try to determine what part of the program is not working correctly. To aid in this you could insert extra writeln statements to provide a trace of your program. For example, if the summation loop in the section of code below is not computing the correct sum, you might insert an extra diagnostic writeln as shown by the last line in the loop:

```
for count:= 1 to NumItems do begin
  writeln('Next item to be summed? ');
  readln(Item);
  sum:=sum+Item;
  writeln('SUM = ',sum, 'COUNT = ',count)
end;
```

The extra writeln statement will display each partial sum and the current value of count.

Although it is not possible to treat program debugging in depth in this book, some common programming errors are considered below.

One common error that beginning programmers often make is getting themselves into an infinite loop. Consider the section of code:

```
writeln('Enter a number: ');
readln(NEXT);
while NEXT <> SENTINEL do
  SUM:=SUM+ITEM;
  writeln('Enter a number: ');
  readln(NEXT)
```

Programming Tip

> There are many reasonable approaches to debugging programs. Here are some suggestions for what to do and what not to do.
>
> One thing you shouldn't do is run the program again in the hope that it's the computer's fault and not yours. Hardware malfunctions are extremely rare and almost certainly you will get the same error message(s) or results.
>
> Another thing you shouldn't do is blindly try anything because you don't know what else to do. Also don't listen to advice like "I tried that once and it worked". If you do these things you will waste a lot of time and will probably get absolutely nowhere.
>
> One method of tracing through a program is to create on paper a small "memory box" for each variable. Step through the code and pretend you are the computer by doing the calculations and updating the contents of the relevant variables by writing values into the boxes.
>
> Take your program listing, get out your textbook and notes, and start studying the program. If you still can't find the problem then if you have written your program in modules you can start to test each module separately until you find the error(s). To test each module you will need to place into the program a number of write statements in strategic positions in the program and observe the output. These test statements can of course be removed later.
>
> Remember run-time errors can be due to serious flaws in your algorithm and may be difficult to remove. Resist the urge for a quick correction.

The body of the loop in the above section of code consists simply of one assignment. So provided the value in NEXT was not initially set to the value in SENTINEL they could never hold the same values since in the body of the loop neither the value of NEXT nor SENTINEL is changed - hence an infinite loop. What was meant of course was:

```
writeln('Enter a number: ');
readln(NEXT);
while NEXT <> SENTINEL do begin
  SUM:=SUM+ITEM;
  writeln('Enter a number: ');
  readln(NEXT)
end
```

Another common error with loops is concerned with the initialization of variables. Sometimes an initialization is placed inside a loop when it should be outside. Spot the error in the code below:

```
readln(count);
if count>0 then begin
  for i:=1 to count do begin
    sum:=0;
    readln(number);
    sum:=sum+number
  end;
  writeln('Sum of',count,'values is',sum)
end;
```

After all errors have been corrected and the program appears to be executing correctly this is still not the end of your task. The program should be tested thoroughly to make sure it works properly for input values. Enough sets of data should be used to give a reasonable level of confidence that all possible combinations would work properly. Notice that it is usually impossible to test computer programs will **all** possible combinations of input data - there are just too many possibilities.

3.10 Historical Note

Alan Turing (1912-1954)

Alan Mathison Turing was born in London in 1912. From a very early age, it was clear that he had an unusual aptitude for mathematics and science.

During World War II he worked on military research projects, and was awarded the OBE for his efforts. Turing was largely responsible for the British forces' cracking of the German cryptographic codes during World War II. Drawing on the knowledge of pulse signal processing he gained during the war, he later worked on the design of an electronic computer ACE (Analytical Computing Engine). After this he worked in numerical analysis (solution of mathematical problems, usually in terms of real numbers, on digital computers).

His fundamental mathematical model of any digital computing device (the **Turing Machine**), is of great theoretical importance in computer science, being used today in information theory, language theory, logic, and other fundamental areas of mathematics and computer science.

A recent biography of Turing refers to his breaking both the military codes (cryptographic codes) and also the codes of society; referring evidently to Turing's homosexuality, society's disapproval of which no doubt contributed to his tragic suicide in 1954. The Association for Computing Machinery (ACM) is the premier society for computing professionals in the USA. Its highest award for original contributions to computer science, the **Turing Award,** is named in honour of Alan Turing.

3.11 Summary

- A major reason for the popularity of Pascal is the fact that it is relatively easy to manipulate many different kinds of information.
- Pascal statements which make it an excellent language for structured programming are:

```
repeat ..... until
case
if .... then .... else
while ..... do
for .....to/downto. ....do
```

- While a program is running, information is stored in the computer's main memory in variables. A variable is referenced by its **name** or **identifier**. All variables must be declared at the beginning of the program.
- An assignment statement has the general form - VariableName := expression. These are used to assign values to variables.
- Arithmetic expressions are used to perform arithmetic. Precedence rules need to be obeyed.
- The simplest form of input statement is of the form - readln(list of variable names).
- The simplest forms of output statements are of the form - writeln(list of variable names) and writeln(literal).
- Comments are enclosed in { .... } or (* .... *), and are ignored by the compiler. Comments may **not** be nested.
- In Pascal conditional execution with one, two or several branches can be used. The forms of the statements are:

```
if condition then statement

if condition
  then statement1
  else statement2

case expression of
  CaseLabel1: Alternative1;
  CaseLabel2: Alternative2;
          .
          .
  CaseLabelN: AlternativeN;
end
```

- Boolean variables can only hold the values **True** or **False**. With these variables the logical operators are **and**, **or**, and **not**. Boolean expressions can be formed in similar ways to arithmetic expressions.
- Indefinite repetition can be expressed using either the **repeat** or **while** loops. The forms of these loops are:

```
repeat                              while ContinuationCondition do begin
  statements                          statements
until TerminationCondition          end
```

- The **repeat** loop is post-tested and the **while** loop pre-tested.
- When using loops particular attention should be paid to initialization, the termination condition, and to what happens on the first and last iterations.
- When the number of repetitions is known at the time of the start the iteration is referred to as **definite iteration.** In Pascal the **for** loop may be used for definite iteration. It has the forms:

  for ControlVariable := InitialValue **to** FinalValue **do** statement

  for ControlVariable := InitialValue **downto** FinalValue **do** statement

- The **for** loop is pre-tested and the value of the control variable is undefined when the loop is completed.
- Debugging refers to the process of removing errors from a program. Errors are either compilation errors, run-time errors, or algorithm design errors.
- After a program is apparently running correctly it should be thoroughly tested with a range of input data.

3.12 Exercises

1. Which of the following are valid identifiers in Pascal?

(i) HCl	(vi) begin
(ii) Na2CrO4	(vii) ALPHA
(iii) 2NaCl	(viii) COUNT(1)
(iv) X-Ray	(ix) Line_Number
(v) x222	(x) $100

2. For the assignment statements given below see if you can find any errors. Assume the declarations given apply to the examples:

```
const  Pi = 3.142;
var
  Grade1, Grade2, Grade3, a, b, c, d, x, AvGrade: Integer;
  Pay, Gross, FFFF, TotalPay, Average, Area, Sum,
  Hours, CommonwealthTax, StateTax, Rate, Deduct,
  Radius, Final: Real;
```

(a) `Pay := Gross - CommonwealthTax - StateTax - Deduct`

(b) `FFFF := Rate*(29000-TotalPay)`

(c) `Gross := $45.34*Hours-Tax`

(d) `Average := Sum/a`

(e) `AvGrade := (Grade1 + Grade2 + Grade3)/3`

(f) `Total := Total - FFFF - CommonwealthTax`

```
(g)  x := b - (2.0*a-b)/a
(h)  Area := 2.o*Pi*Radius*Radius
(i)  c := 2*a+b-d
(j)  Final := Begin - 34,786.89
(k)  x := FFFF + CommonwealthTax + StateTax
```

3. It is required to calculate the surface area and volume of a sphere. The formulae are given below:

   Surface Area = $4\pi r^2$

   Volume = $\frac{4}{3}\pi r^3$

   Write a program that computes and prints the surface area and volume of the sphere.

4. What is the purpose of each of the following in a Pascal program?

   (a) program heading
   (b) a comment
   (c) a declaration
   (d) the words **begin** and **end**
   (e) semi-colons
   (f) a full stop
   (g) blanks

5. Explain in your own words the meaning of the following terms:

   (a) a variable
   (b) an integer
   (c) a real
   (d) a Boolean variable
   (e) an assignment statement
   (f) a literal ('.....')
   (g) precedence rules
   (h) Boolean operators
   (i) relational operators

6. Are the following legal integer values?

   (a) 45
   (b) -78
   (c) 40987
   (d) 600,999
   (e) 45.98

7. Are the following legal real values?

   (a) 1234
   (b) 23.98
   (c) .8765
   (d) 678,876.76
   (e) 7.88E6
   (f) 0.86E-56
   (g) E12
   (h) 5.6E1.2

8. When would it be legal to to write - Number:=TwentyFive?

9. When is ((I div 2) * 2) = I true?

10. Write a Boolean expression that will be true if RATING is between 1 and 7 (inclusive) and SCORE is positive.

11. Which of the following are valid Pascal **var** declarations? Identify the error(s) in each of the invalid declarations.

```
(a) var 1a,2a,3a: Char;
(b) var a,b,c   : Integer;
          abc   : Real;
(c) var Letter  : Character;
(d) var a1,a2,a3 : Reals;
(e) var a1,a2,a3 : Char;
          b1 : Real;
          a1 : Integer;
```

12. Choose reasonable names and write a single **var** declaration for the following:

   (a) The three real coefficients of a quadratic equation.

   (b) The roots of a quadratic equation.

   (c) A Boolean variable indicating whether or not there was a solution.

   (d) A six digit student identification number.

   (e) A student's average mark obtained in six subjects.

13. Assume the following declarations:

```
a,b,c: Integer;
x,y,z: Real;
```

   Also assume that the following values have been assigned to the preceding variables:

a:=10; x:=1.52E1;
b:=-15; y:=0.4;
c:=7; z:=-5.1E3;

What is the value of the following Pascal expressions?

(a) a mod (c-1)
(b) a + 103 div (a-c)
(c) x * y + 1.0 - a
(d) x / y * 3.0 + z
(e) 2 + a * b mod c +1
(f) (2 + a * b mod c + 1) < 2

14. Write a program fragment which will evaluate the function defined below, for any Real **x** :

$$f(x) = \begin{cases} 5x + 3x + 6 & ;0<=x<=1 \\ 8.5x - 3x + .76x - 8 & ;1<=x<4 \\ 2x + 4 & ;4<=x<5 \\ 5 & ;5<=x<10 \\ 0 & ;\text{other real } x \end{cases}$$

15. The following program computes the roots of a quadratic equation. Identify possible sources of run-time errors and rewrite the program:

```
Program Quadratic;
{A program to compute the roots of the equation
 ax*x + bx + c = 0}
var
  a,b,c,PART,ROOT1,ROOT2: Real;
begin
  readln(a,b,c);
  Part:=sqrt(b*b-4*a*c);
  ROOT1:=(-b+PART)/(2*a);
  ROOT2:=(-b-PART)/(2*a);
  writeln('a=',a:10:5,'b=',b:10:5,'c=',c:10:5;
  writeln('First Root = ',ROOT1:15);
  writeln('Second root = ',ROOT2:15)
end.
```

16. Correct the syntax errors in the following program:

```
Progran Exercise
const
  a=10
  b=6
  c=9
var
  d;e;f: Integer
begin
  read(d,ef)
  if (d>f) then d=a+d;
    else d=a
    e:=d+f
    Write('This program does not make any sense,e,fd)
end;
```

17. Rewrite the following program correcting all syntax errors:

```
Program Test
  const
    w:=4;
  var
    FOUR : ONE : ZERO :Integers;
  begun
    FOUR:=FOUR+w+ONE
    if FOUR<=ONE
      FOUR:=ONE
    else ONE:=ZERO
    writeln ('This is the end' I Think !)
  the_end.
```

18. Determine the output of the following program:

```
Program Trace;
  const
    x=5;
  var
    a,b,c: Integer;
  begin
    b:=1;
    c:=x+b;
    a:=x+4
    a:=c;
    b:=c;
    a:=a+b+b;
    c:=c mod x;
    c:=c*a;
    a:=a mod b;
    writeln(a,b,c)
  end.
```

19. What is the output after the following section of code is executed (assume I,J,K are integers)?

```
K:=0;
I:=3;
while I<>7 do begin
  for J:=I downto 3 do
    K:=K+1;
    I:=I+2;
  end;
  writeln(K,I,J);
end
```

20. Write a section of code (using either a **while** or **repeat** statement) for each of the following:

(a) Read in integers, counting the number of positive and negative values. Stop when you encounter a value of zero.

(b) Find the largest integer whose square is less than 142619.

(c) Sum the even integers between 1 and 99.

(d) Output the squares of the first 30 odd numbers.

(e) Parts (c) and (d) could be done using a **for** statement. Write the corresponding statements.

4 ARRAYS & SUBPROGRAMS IN PASCAL

4.1 Arrays

Suppose that a modern technologically sophisticated rock-star promoter wanted to computerize her booking system so that she could easily obtain information about the status of seats. For a particular performance the promoter's computer programmer decides to use the value of a variable to indicate whether a particular seat is unbooked, reserved or paid for i.e. 1 - unbooked; 2 - reserved; 3 - paid for. If the promoter wanted to know the number of seats reserved then a section of code similar to that shown below could be written:

```
TotalReserved:=0;
if SEAT1=2 then TotalReserved:=TotalReserved+1;
if SEAT2=2 then TotalReserved:=TotalReserved+1;
if SEAT3=2 then TotalReserved:=TotalReserved+1;
                  .
                  .
```

Clearly for a large number of seats the program which would have to be written would be ridiculously long. Situations similar to this arise in programming frequently and we must have a facility for handling them. The concept of an array allows us to handle this situation efficiently. Just as an **iteration control structure** allows us to express a potentially infinite number of instructions with a finite program, the **array data structure** gives us the capability of addressing large ranges of memory with a single reference.

Singly Subscripted Arrays

We first consider singly-subscripted arrays (often referred to as vectors). A singly-subscripted array is a sequence of data items, all of which must be of the same type. A list of phone numbers, a shopping list and a string of characters are all examples of arrays. An individual item of an array is called an **element** and it can be accessed by means of an **index** or **subscript**. You may have already met the idea of subscript in connection with the study of **sequences** and **vectors.**

An array is declared by specifying its name, the range of subscripts which can be used in referring to its elements, and the type of elements. For example consider the declaration below:

```
var
  Grades:array[1..7] of Integer;
```

A singly-subscripted array called **Grades** is declared with 7 variable elements - Grades[1], Grades[2], ... , Grades[7]. The structure of the array Grades is shown diagrammatically in Figure 4.1.

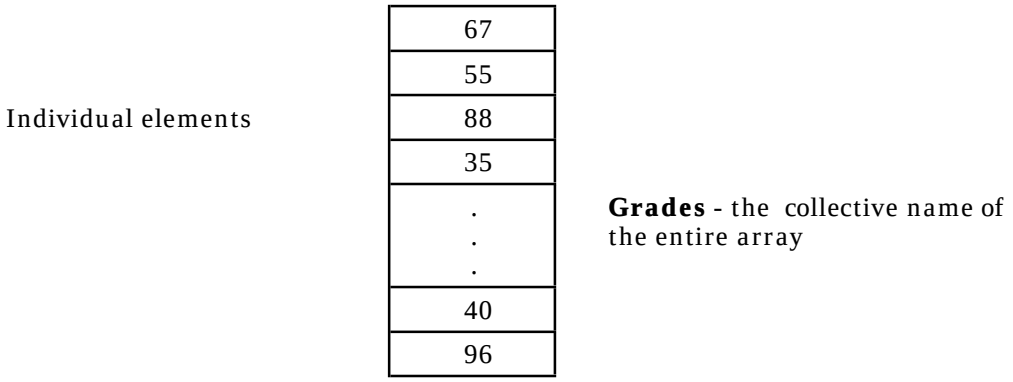


Figure 4.1 The Organization Of The Array **Grades**.

Example

For Figure 4.1, we have the following:

Grades[1] = 67
Grades[2] = 55
Grades[3] = 88
Grades[4] = 35, etc.

A singly-subscripted array declaration in Pascal has the following general form:

```
var ArrayName:array[LowerBound .. UpperBound] of type
```

where the array name is any identifier, the lower and upper bounds are integers which denote the highest and lowest values which can be used as subscripts, and type is the type of elements (Integer, Boolean etc.). For example, if we had 30 students in a class, we could use the declaration:

```
var Grades : array[1..30] of Integer;
```

The advantage of array declarations is that a large number of similar but different variables can be replaced by a single declaration. The real power of arrays is seen in repetition where the control variable of a loop can be used as a subscript of the array. Arrays can be used in the previous example where the number of reserved seats was found. A section of code to find the number of reserved seats is given below:

```
var
  SEAT:array[1..500] of Integer;

TotalReserved:=0;
for I:=1 to 500 do
  if SEAT[I]=2 then TotalReserved:=TotalReserved+1;
```

The trick in reducing the amount of code so dramatically involves the use of the control variable I as the subscript of SEAT. In fact one of the most common uses of **for** loops is for subscripting (or indexing) arrays. If our technologically advanced promoter wanted a list of the seats together with their status then the following loop would give it to her:

```
for I:=1 to 500 do
case SEAT[I] of
  1: writeln(I,'Unbooked');
  2: writeln(I,'Reserved');
  3: writeln(I,'Paid for');
end;
```

What if our rock star promoter had a number of concert halls and the number of seats in each was different or she wanted to change the number of seats? It would certainly be a nuisance to have to change all the occurrences of the integer 500 in the program. It would be both a tedious and error-prone task. We can get out of this dilemma by defining a **named constant** early in the program. This means that when the number of seats is changed only the named constant needs to be re-defined as shown below:

```
const
  NumberOfSeats=450;
var
  SEAT[I]:array[1..NumberOfSeats] of Integer;
                .
                .
  for I:=1 to NumberOfSeats do
                .
                .
```

Remember that the value of a named constant is fixed from the time of the definition and cannot change during the course of the execution of a program. It is good programming practice to use named constants where possible.

Examples of Array Usage

Now let's look at examples of some loops which you will find useful in many of your programming tasks. The declarations for the loops are given below:

```
const N =....;
var
  Vector:array[1..N] of Integer;
```

For input and output the following loops could be used:

```
for I:=1 to N do readln(Vector[I])
for I:=1 to N do writeln(Vector[I])
```

The loop below initializes the all the elements of the array Vector to 0:

```
for I:=1 to N do Vector[I]:=0
```

The following loop sums all the values of the elements of a vector:

```
Sum:=0;
for I:=1 to N do Sum:=Sum+Vector[I]
```

Here's one for you. What do you think is the intention of the following loop?

```
I:=0;
repeat
  I:=I+1
until Vector[I]=0;
```

The intention is probably to search through the array for the first element whose value is 0. Could there be a problem with this loop? Try to write a loop which overcomes the possible difficulty.

The problem is that there might be no value of 0 in the vector and hence the loop will never terminate. The loop is corrected below:

```
I:=0;
SEAT[1]:=0;
repeat
  I:=I+1
until (Vector[I]=0) or (I=N).
```

Going back to our rock star promoter example, a program fragment to find the **first** reserved seat is given below:

```
I:=0;
repeat
  I:=I+1
until (SEAT[I]=2) or (I=NumberOfSeats);
if SEAT[I]=2
  then writeln('Seat',I,'has been reserved')
  else writeln('No seats have been reserved')
```

Doubly-Subscripted Arrays

Sometimes we need to consider doubly-subscripted arrays. A doubly-subscripted array is one in which two subscripts are needed to identify an element. The elements of such arrays are like a grid. If **A** is a doubly-subscripted array then individual elements would be identified as:

```
A[1,1]  A[1,2]  A[1,3] .....
A[2,1]  A[2,2]  A[2,3] .....
A[3,1]  A[3,2]  A[3,3] .....
 .       .       .
 .       .       .
```

The form of the general declaration must of course have two subscripts. It is given below:

```
var A:array[LowerBound1..UpperBound1, LowerBound2..UpperBound2] of type.
```

To see how a doubly-subscripted array might be used we consider the addition of two 2 x 2 matrices **A** and **B** where

$$A=\begin{pmatrix}5 & 8\\ 4 & 5\end{pmatrix} \quad \text{and} \quad B=\begin{pmatrix}3 & 4\\ 8 & 9\end{pmatrix}.$$

Let the resultant matrix be **C**. Clearly,

$$C=\begin{pmatrix}8 & 12\\ 12 & 14\end{pmatrix}$$

The main steps in our program will be the input of the matrices **B** and **A**, computation of their sum, and finally printing the resulting matrix **C**. A program to compute the sum of two 2 x 2 matrices is given below. You will notice that the **for** loops are nested. Since this is our first example of nested loops you should trace through the program carefully.

```
program MatrixAddition;
{This is program reads two 2 x 2 matrices and computes their sum}
var
  A,B,C:array[1..2, 1..2] of Integer;
begin
  {Input matrices A and B}
  for I:=1 to 2 do begin
    for J:=1 to 2 do readln A[I,J]
  end;
  for I:=1 to 2 do begin
    for J:=1 to 2 do readln B[I,J]
  end;
  {Add matrices A and B}
  for I:=1 to 2 do begin
```

```
      for J:=1 to 2 C[I,J] := A[I,J] + B[I,J]
    end;
    {Print the resultant matrix C}
    for I:=1 to 2 do begin
      for J:=1 to 2 do begin
      write(C[I,J]);
      write('  ');
      end;
      writeln
    end
  end.
```

In the loops concerned with the input and addition of matrices **A** and **B**, the **begin** and **end** are not required. They are there to show the structure of the nested loop. Similarly, the outer **begin** ... **end** pair is not required in the nested loop for printing the matrix **C.**

Have a go at changing the addition part of the program so the program becomes one that will multiply the two matrices **A** and **B**. If you have not been taught matrix multiplication you may leave the exercise till later. The solution is given below but don't look at it until you've had a go yourself:

```
for I:=1 to 2 do begin
  for J:=1 to 2 do begin
    Sum:=0;
    for K:=1 to 2 do Sum:=Sum + A[I,K] * B[K,J];
    C[I,J]:=Sum
  end
end
```

Although we've only considered 2 x 2 matrices it is very easy to extend to n x n matrices where n is any positive integer.

Programming Tip

> The most common error made when using arrays is a subscript range error. This occurs when the subscript value is outside the range specified. These types of errors are not syntax errors, i.e. they will not be detected until program execution begins. They are often caused by a loop not terminating. You should pay particular attention to subscripts, and ensure that the legal range is not violated.
>
> Take care to ensure that all your **begins** have a matching **end** since this is often the reason why a loop control variable is not updated as expected. If the error can't be determined by inspection of the program, diagnostic output statements may be inserted to print subscript values which may be out-of-range.
>
> Always double check the subscript values at the loop boundaries. If these values are in-range, it is likely that all other subscript references in the loop will also be in-range.

4.2 Procedures

Suppose you are asked to write a program which will read in three integers and then print them out in descending order. One way to sort is to keep on swapping adjacent pairs until the values are sorted. Let's assume the integers are contained in variables A, B and C. Before looking at a program to carry out our task we consider some code which will swap the contents of two variables. If X,Y and Dummy are integer variables, then the following statements will swap the contents of X and Y:

```
Dummy:=X;
X:Y;
Y:=Dummy
```

Why can't we just write:

```
X:=Y;
Y:=X;
```

to do the exchange? If you now had a go at writing the program you would probably come up with something like the one below:

```
program Descending;
{This program reads three integers and prints them in
descending order.  Written by :  A GOODSTUDENT, DATE : 27/3/87 }
var
  A,B,C,Dummy: Integer;
begin
  readln(A,B,C);
  if A<B then begin
    Dummy:=A;
    A:=B;
    B:=Dummy
  end;
  if B<C then begin
    Dummy:=B;
    B:=C;
    C:=Dummy
  end;
  if A<B then begin
    Dummy:=A;
    A:=B;
    B:=Dummy
  end;
  writeln(A,B,C)
end.
```

It is obvious that the program is a little tedious even with only three integers. We would like to write the main part of the program somehow like:

```
if A<B then SWAP(A,B);
if B<C then SWAP(B,C);
if A<B then SWAP(A,B);
```

where SWAP is a procedure which carries out the swapping on the variables identified in the brackets.

In Pascal, as in most other languages we can define an independent program module, called a **procedure** (or subprogram) which can be called into action on a number of occasions during the execution of a program. The definition of the procedure in terms of operations already known is made at the beginning of the program by using a **procedure declaration**. We can rewrite our Descending program using procedures:

```
program  Descending;
{ Read three integers and prints them in descending order}
var
  A,B,C: Integer;

procedure SWAP(var X,Y:Integer);
var
  Dummy: Integer;
begin
  Dummy:=X;
  X:=Y;
  Y:=Dummy
end;

begin {main program}
  readln(A,B,C);
  if A<B then SWAP(A,B);
  if B<C then SWAP(B,C);
  if A<B then SWAP(A,B);
  writeln(A,B,C)
end   {main program}.
```

There are many new ideas here so let's consider them now. The procedure declaration or definition consists of three parts, namely:

1. The procedure heading which gives the name of the procedure and the variables X and Y the procedure will operate on. The variables X and Y are called **formal parameters.**
2. The declaration of the integer variable Dummy within the procedure.
3. The main body of the procedure, i.e. the operations to be carried out.

To call the procedure SWAP from within the main program the name of the procedure is written followed by a list of the variables it is to operate on. In our example we write SWAP(A,B). The variables A and B are called **actual parameters.** When a procedure call is executed the body of the procedure is executed with the formal parameters replaced by the actual parameters. So in our case SWAP(A,B) becomes equivalent to:

```
begin
  Dummy:=A;
  A:=B;
  B:=Dummy
end;
```

and similarly for SWAP(B,C). After the execution of a procedure a return is made back to the original point of the call. Some of the reasons why procedures are very useful in programming are:

- They lead to much less code, i.e. more compact programs.
- They make programs easier to understand, i.e. to understand the program we only need to known **what** its procedures do, not **how** they do it.
- They can be used as modules to build programs. Remember the importance of our structure charts in program design and development. A procedure generally corresponds to any leaf (box) in a structure chart.
- They are helpful when trying to debug or test a program, i.e. procedures could be incorporated into programs only when they have been debugged and tested.

We've only just touched on procedures in Pascal. Let's keep going. Repeating, all procedures used in a program must be defined or declared at the head of the main program. Procedure declarations come after any variable declarations of the main program but before the **begin** of the main program. The general form of a procedure declaration is:

```
procedure name(list of formal parameters);
  {declarations of variables or constants}
begin
  procedure body
end;
```

So far we've only considered **var** type formal parameters - there is another type. Notice that apart from the heading the syntax of a procedure is the same as that of programs without procedures.

The variables or constants declared in a procedure can only be used inside that procedure, and nowhere else in the program. The variables are said to be **local** to the procedure and exist only while the procedure is being executed. Variable names cannot be the same as the parameter names. Although it doesn't matter whether variable names inside a procedure are the same as those outside the program it is probably at this stage of your Pascal development best to keep them different so that you don't have to worry about learning the so-called "scope" rules.

Procedure calls are a single statement of the form:

```
procedure name(list of actual parameters separated by commas).
```

When a procedure call is made the formal parameters of the procedure are replaced by the actual parameters and the body of the loop then executed. The formal and actual parameters must be of the same type and number with substitution made one for one in the order in which the parameters occur in the procedure heading and procedure call respectively. The names of the actual and formal parameters can be quite different.

An example of a procedure to read the next non-zero integer from a set of integers is given below:

```
procedure ReadNonZero(var Number:Integer);
begin
  Number:=0;
  while Number=0 do readln(Number)
end;
```

Notice that there are no local variables in this procedure. An example of a procedure call might be ReadNonZero(X) where X is an integer variable. After execution, the value of X will be the first non-zero value in the data.

An example of a procedure which determines the greatest value of a set of integers is shown below. You should study this procedure carefully.

```
procedure FindMaxInt(var Max:Integer);
var
  Next: Integer;
begin
  readln(Max);
  while not Eof do begin
    readln(Next);
    if Next>Max then Max:=Next
  end
end;
```

Notice the Eof (end of file) which has been used for the first time. Eof is a Boolean-valued function that is intrinsic (built-in) to Pascal. It becomes true if and only if an end-of-file marker has just be reached. In our case, use of this function means that the body of the **while** loop is executed just as long as there are integers available. A possible call might be:

```
if not Eof then FindMaxInt(X);
```

The other kind of parameter is the **value parameter**. It is distinguished by not having the word reserved word **var** in the procedure heading. The corresponding actual parameter may be a variable, an expression or a constant.

In procedures using value parameters when a call is made the actual parameter is evaluated, and the resulting value stored stored in an anonymous memory cell somewhere accessible to the body of the procedure. Each reference to the formal parameter in the procedure body is treated as a reference to this anonymous memory cell. The cell is discarded as soon as the procedure body finishes execution.

As an example of the usage of value parameters we rewrite our SWAP procedure:

```
procedure SWAP(X,Y:Integer);
var
  Dummy: Integer;
begin
  Dummy:=X;
  X:=Y;
  Y:=Dummy
end;
```

The thinking student might be starting to get confused (while the non-thinking one was confused long ago) and might ask the question - if a value parameter can be used to accept variables, constants and expressions as corresponding actual parameters, while a variable parameter can only accept variables, why bother with variable parameters at all? **Variable parameters are used in cases where information is to be passed from a procedure back to its calling program while value parameters are used when information is to passed only from the calling program to the procedure.**

The use of value parameters helps protect the calling program from errors in the procedure, since the values of the actual parameters cannot be changed. So make all your parameters value parameters unless you really want to change the corresponding actual parameters. This rule should normally be relaxed for arrays.

We've left the simplest type of procedure i.e. the one with **no** formal parameters, until last. Sometimes you might have a complicated report heading and it would be sensible to confine it to a procedure. An example is given below:

```
procedure PrintHeading;
begin
  writeln; writeln('COMPUTER  MATHEMATICS');
  writeln; writeln('by  S.J. Sugden & J. Simmond')
end;
```

In this example the heading is fairly simple and might not justify a separate procedure. Procedures can also have arrays as parameters but they will not be considered in this course.

4.3 Functions

A function can be thought of as a procedure which produces a result and returns it via its name. In a procedure the body of a loop out carries a set of operations. Sometimes these produce a result. Instead of assigning the result to a variable in order to pass the result back to the main program we can assign it to the function name. Here's a simple example which finds the value of:

$$f(x) = x^3 + x^2 + x + 5, \quad when \quad x = 3.$$

```
function POLYNOMIAL(n : Integer) : Integer;
  begin
  POLYNOMIAL:=n*n*n + n*n + n + 5
end;
```

Clearly there is an assignment to the function name in the body (not much of a body in this case) of the function. Also the result is declared to be of type integer. To call the function, POLYNOMIAL(3) would appear somewhere in the body of the main program.

Now for a slightly more complex example. In this case the sum of the first n integers is found, for example, when n = 5, we have: 1 + 2 + 3 + 4 + 5 = 15.

```
function SumNum(n : Integer) : Integer;
  var
    I,NEXT,Sum: Integer;
  begin
    Sum:=0;
    for I:=1 to n do begin
      readln(NEXT);
      Sum:=Sum+NEXT
    end;
    SumNum:=Sum
end;
```

In this example the function is declared to be of type Integer. The result is assigned to the function name SumNum which acts as a variable of type Integer.

As you can see functions are declared in exactly the same way as procedures except that the reserved word **function** replaces **procedure**, the type of function is specified in the heading and there is an assignment of the result to the function name somewhere in the body.

Functions differ from procedures in the way they are called. A procedure call, which consists of a procedure name followed by a parameter list (possibly empty), is a complete program statement. A function call has the same format but it is not a complete statement. It may occur in an expression where a variable of the same type could occur. When a function call is carried out the function is evaluated (after appropriate substitution of parameters) and the result is then used at the point where the function call occurred. In our example above a call to the function might be:

```
if SumNum(5) < 100 then ....
```

The body of the function would take the value of n as 5 and hence find the sum of the first 5 counting numbers. Since this is less than 100, something would be done in the main program.

Assume that you're given a list of integers and asked to determine the first non-zero one. Write a function which will do this. There is one below for you to compare yours with:

```
function FirstNonZeroValue : Integer;
var
  Next:Integer;
begin
  Next:=0;
  while Next=0 do readln(Next);
  FirstNonZeroValue:=Next
end;
```

An example of a possible function call might be:

```
if FirstNonZeroValue > 700 then ....
```

As another example of a function consider the computation of factorial n (n!) where n is a positive integer. An example of a factorial is: 5! = 5x4x3x2x1 = 120. The function is given below:

```
function Factorial(n : Integer) : Integer;
var
  I,Product:Integer;
begin
  Product:=1;
    for I:=1 to n do Product:=Product*I;
  Factorial:=Product
end;
```

Pascal has some standard procedures and functions. In fact we've used some already. For example, **readln** and **writeln** are intrinsic procedures in Pascal, and Sqr, Abs are intrinsic functions. The functions we've just been considering are **user-defined** functions. Examples of some standard functions are: Abs, Cos, Exp, Int, Ln, Random, Round, Sqr, Sqrt, Trunc.

Programming Tip

> Procedures and functions can be used to write readable programs even when the programs are only a few hundred or more lines long. The names you choose should indicate clearly what the procedures or functions do. A reader should be able to obtain a good idea of what a program does by simply looking at the main program part of the program. He or she should only need to look inside a procedure or function to learn how it works.
>
> Although comments should be placed at the beginning of a procedure or function they should not simply rehash the procedure or function name. Examples of meaningful procedure or function names are - ReadData, SortIntegers and FindMax.

4.4 Historical Note

A Short History of the Computer (1600-1987)

Before we had advanced to our current sophistication in calculating devices, human beings used sticks, stones, shells, knots in a rope etc. for counting. Later on they progressed and used their fingers for simple computations. Then came beads on rods - the abacus. This calculating device has been used for the past 2,000 years and is still widely used in many Far-Eastern countries.

In the 1600's, John Napier came up with an ingenious device for multiplying and dividing. The device became known as Napier's Bones and was used for many years. It was not until 1642 that a 19 year-old Frenchman called Pascal built the first practical calculating device. Pascal's machine could however only add and subtract and it was not until some 30 years later that Leibniz developed a similar machine that could also multiply and divide.

Charles Babbage probably made the greatest contribution in the area of mechanical calculators. In 1833 this English mathematician designed but never built one of the modern digital computers. It was not until 1884 that the first commercially produced adding machine appeared. William Burroughs was the inventor.

Towards the end of the 1800's cards were used to control patterns in textile looms. This principle was used by Herman Hollerith to develop the punched card system to process census data. This system was the fore-runner of the electromechanical data processing system in use today.

The development of the computer, as with other calculating devices took many years. In 1938 George Stibitz at Bell Laboratories in the USA built several electromechanical computers. In the years that followed during the Second World War, workers at Harvard University designed a large machine that used a program to guide it through a long series of calculations. It performed an addition in 3/10 of a second. Compared with today's machines it was very slow and had very limited storage capacity - but still this was the first electromagnetic computer.

By the mid-1960's computers had already assumed an important role in Western society, particularly in of the USA. Applications were mostly in major business and government areas.

From the early 1970's to the current date there has been a veritable explosion of activity in the microcomputer areas.

4.5 Summary

- An array is a collection of data items of the the same type which can be referred to by a single identifier. The items can be distinguished from each other by attaching a unique subscript to each item.
- Singly-subscripted arrays use only one subscript to identify the individual elements, whereas doubly-subscripted arrays use two.
- **for** loops are most commonly used for indexing (or subscripting) arrays.
- A named constant takes a fixed value which may not be altered in the program. It is defined in the following way:

```
const identifier = value
```

- Procedures are independent program modules which can be called from anywhere in a main program. When a procedure has been executed a return is made back to the point of call in the main program.
- When a procedure is called, the formal parameters are replaced by the actual parameters before the body of the procedure is executed.
- There are two types of parameters - **variable** and **value**.
- Any reference to a **variable parameter** is treated as a reference to the corresponding actual parameter. The value of the actual parameter can be changed.
- Any reference to a **value parameter** is treated as a reference to a local copy of the value of the corresponding actual parameter (which is made when the procedure is called). The values of the actual parameters are not changed.
- Variables which are declared inside a procedure are said to be local and cannot be referred to outside the procedure.
- A function is a procedure which returns a scalar result. The result is assigned to the function name at some point in the body of the function. When a function call is executed the function is evaluated and the result is used in the place from where the function call was made.

4.6 Exercises

1. An important characteristic of arrays is:
   (a) The elements of an array can be of different types.
   (b) The value of an element is used to select the index.
   (c) The number of elements in an array determines the maximum value.
   (d) A single identifier refers simultaneously to more than one value.

2. An example of a valid element of a doubly-subscripted array is:

```
(a) Grades['A']      (d) Grades[15]
(b) Grades[2,3]      (e) Grades[8,5,'g']
(c) Grades[13,'C]
```

3. Which of the following declarations contain(s) a syntax error?

```
(a) var Grades:array['A'..'F'] of Integer
(b) var Grades:array[Limits] of Integer
(c) var Grades:array[1,50] of Integer
```

4. Which of the following is an invalid assignment statement?

```
(a) X[I] := X[I] + 1
(b) LINE[I+1] = LINE[I] +1
(c) LINE[I] := I*I*2 + 2*(NEXT[I]/GRADES[J])
```

5. Write a declaration for each of the following:

   (a) An integer array X of 100 elements.
   (b) An integer array X having a lower bound of -50 .
   (c) An integer array GRADES starting with subscript 0.
   (d) Real arrays TAXES and STATETAX of 50 elements each.

6. Why doesn't the following Pascal code swap the contents of the variables X and Y?

```
X:=Y;
Y:=X;
```

7. Given the following declarations and section of code:

```
var
  X:array[1..100] of Integer;
  I,NUMBER: Integer;
          .
          .
  NUMBER:=20;
  for I:=1 to NUMBER do X[I]:=2*I
          .
```

   (a) What is the value of X[1]?
   (b) What is the value of X[20]?
   (c) What is the value of X[21]?

8. Write a program that reads the elements of a 25-element integer array and then prints the elements in the following formats:
   (a) One element per row in the order read.
   (b) Five rows of five elements.
   (c) One element per row in reverse order read.

*9. Write a program to compute the mean and standard deviation of a set of class marks. The formula for standard deviation is given by:

$$\sigma = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n}}$$

*10. Write a program to read N data items into two arrays X and Y of size 20. Store the product of corresponding elements of X and Y in a third array Z, also of size 20. Print a three-column table displaying the arrays X,Y, and Z. Make up your own data with N less than 20.

*11. Let A be an array containing 20 integers. Write a program which reads up to 20 data items into A, and then finds and prints the subscript of the largest item in A and that item.

*12. Write a program to print out the values for the cubic function $y = 4x^3 + 5x^2 + 7x + 9$.

x	-4	-3	-2	-1	0	1	2	3	4
y									

13. Write Pascal code to:
   (a) read in the matrices:

$$A = \begin{pmatrix} 3 & 8 \\ 2 & 9 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 1 & 0 \\ 6 & 3 \end{pmatrix}.$$

   (b) Print these matrices.
   (c) Compute and print their sum, difference and product.

14. Write a procedure to print the headings for a report. Make up your own headings.

*15. Write a procedure which computes the slope of a line given two points on the line.

16. Write a procedure that calculates **C** = **A X B** where **A, B** and **C** are square matrices of dimension n.

*17. Write a program to print out the first 100 prime numbers. In your program include a procedure to determine whether a number is prime.

18. Write a program with three procedures to print the message "HI HO" in block letters, i.e.

```
*   *          *
*   *          *
*******        *        etc.
*   *          *
*   *          *
```

19. Write a function which computes the cube of any number.

20. Write a function that computes sin(x) using the approximation below:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots..$$

21. Write a procedure to find and print all the prime numbers less than 200 (to decide whether a number is prime it is only necessary to test for a zero remainder after division by every integer from 2 to $\sqrt{x}$ - if any is found it is not prime).

22. Write a procedure to find all the prime factors of a given number.

*23. Write a procedure to calculate and print the binomial coefficients ${}^{z^n}C_r$ over the range n=0, 1, 2, ...,10 and r=0, 1, 2,..., n.

*24. Write a program which will convert the decimal representation of a non-negative integer to one in any base from 2 to 9.

25. Evaluate $\sin 50°$ using 8 terms of the series:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots..$$

Compare your result with the value given by the SIN function. Remember that π radians = 180 degrees.

26. Show that the partial sums of the series below are always less than 2:

$$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots\ldots\ldots\ldots..$$

*27. Given any number write a program which tests whether the number is square, i.e. 4, 9, 16, 25, ... Do not use the Sqrt function in Pascal.

*28. Write a program to test for triangular numbers, i.e. 3, 6, 10, 15, ... (These are partial sums of the A.P. 1, 2, 3, 4, 5, 6, ...).

29. Write a program to evaluate Pascal's triangle (in Pascal's triangle , each number is the sum of the two above it, with each row being generated from the one above it).

*30. The square root of a number N can be approximated by repeated calculation using the formula:

$$NextGuess = \frac{1}{2}\left(\frac{LastGuess^2 + N}{LastGuess}\right)$$

Test this formula out by writing a program and trying a few numbers.

*31. **Challenge Problem.** Write a program to read examination marks ranging from 1 to 100 and print the number of students in each of the ranges below:

90 - 100	Excellent
80 - 89	Very Good
70 - 79	Good
50 - 69	Satisfactory
< 50	Unsatisfactory

Test your program on the following set of marks:

```
34  65  36  56  78  89  77  77  99  12
44  55  34  67  16  35  23  47  66  57
19  34  55  89  77  48  39  50 100   1
```

32. Write a program which reads in coordinates of two points $A(x_1, y_1)$ and $B(x_2, y_2)$ and determines the gradient and length of AB. Try not to repeat the same calculations in your program.

33. Write a program to convert 12 hour am/pm time to the 24 hr decimal system of hours:minutes.seconds. For example, 9.35 pm is 21:35.00.

34. Write a program to simulate a radar gun. The program should be able to read the speed and print the message "speeding" if the speed exceeds 60 kmph. Modify the program to give the fines and points as shown below:

speed	**fine**	**points**
60 - 70 kmph	$20	2
71 - 80 kmph	$40	3
81 - 100 kmph	$100	4
> 100 kmph	$1000	jail

5 DIFFERENTIATION

5.1 Limits

In our calculus studies, we first learn how to construct a **tangent** to a curve in the (x,y) plane. This process is referred to as **differentiation**. The aim of differentiation is to find the gradient (or slope) of the tangent at a given point or set of points on the curve. The curve is usually represented in the form $y = f(x)$. The functions we usually consider first are the **polynomials** - in fact initially we just look at **monomials**. These are simply powers of the independent variable, x.

For example $f(x) = x, g(x) = x^2, h(x) = x^3$. By combining these simple functions with various multiplying constants we build up the familiar **polynomials.** Some simple polynomials are:

$$g_2(x) = x^2 - 3x + 5 \qquad\qquad f_3(x) = 2x^3 + 0x^2 - x + 12$$

In this chapter, we concern ourselves with the examination of **limits**, approximate differentiation by computer, and some applications of the derivative. The methods we consider will be applied in general to "simple" functions, usually polynomials. The main reasons for this are that the polynomials are generally simplest to analyse, and that numerical results obtained are then easily verified by use of exact algebraic and analytical methods. However, we should remark that the methods, or **algorithms,** used in most cases apply equally to more general functions. These include the trigonometric, exponential and logarithmic functions, and some of the examples and exercises deal with such functions.

When we first study calculus, we encounter the idea of a **limit.** Limits can often be a tricky concept to swallow; even for students who are quite competent at other branches of mathematics such as algebra or trigonometry. This being the case, let's see if we can use the computer to illuminate some of the concepts of limits and differentiation. We define the limit of f(x) as x tends to **a** in Figure 5.1.

DEFINITION OF LIMIT

We say that $f(x) \to L$ as $x \to a$ if, given any positive number ε, there exists a corresponding positive number, δ such that $| f(x) - L | < \varepsilon$ whenever $0 < | x - a | < \delta$.

Figure 5.1 Definition Of Limit

What does this mean? First of all, we need to remember that $|p - q|$ means "the distance between p and q on the number line".

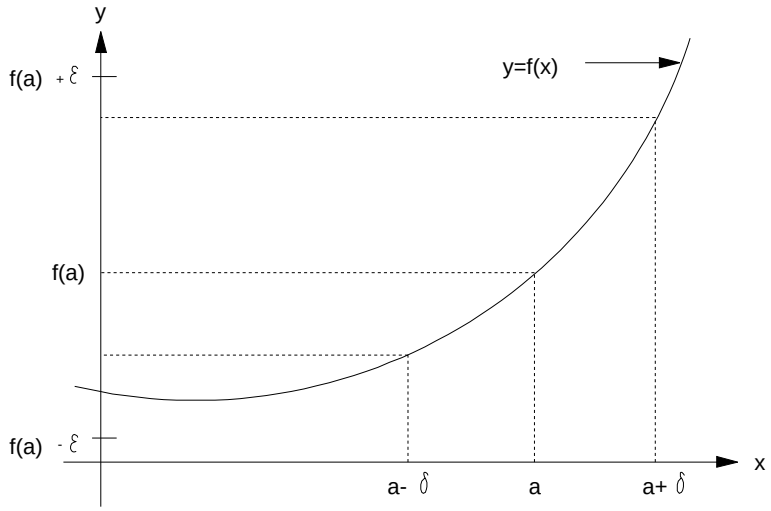


Figure 5.2 The Limit Concept

Our limit definition is then telling us that if we select any positive number ε (no matter how small it is), there will always be another number, δ, such that the value of the function will differ from its limit L by less than ε, just so long as the distance from x to a is less than δ. **In other words, there is an interval** $(a-\delta, a+\delta)$ **which has the property that as long as x is in the interval (but not at a), then f(x) will automatically lie in the interval** $(L-\varepsilon, L+\varepsilon)$. A diagram to help illustrate the limit concept is given in Figure 5.2.

Notice that the definition says nothing about the value of f(x) at x=a. We are not concerned with the value of f(a); in fact, f(x) may not even be defined at x=a.

Example Of A Limit

$$Let\ f(x) = \frac{\sin x}{x},\ \text{for } x \neq 0 \text{ and consider } \lim_{x \to 0} f(x) = \lim_{x \to 0} \frac{\sin x}{x}$$

The graph of y = (sin x)/x is given in Figure 5.3.

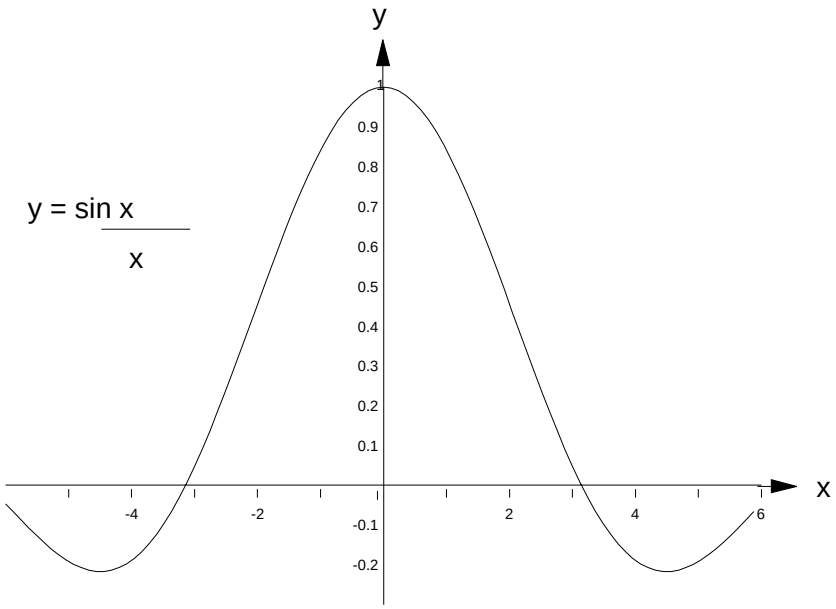


Figure 5.3 The Graph of sin x / x

Since division would be involved, it is implicit from the formula for f(x) that we have not defined the value of the function at x=0. Therefore, we cannot yet speak of f(0). We can of course define f(0) to be any value we please and then we can talk about its value! From the graph however, it seems clear that the limit of the function as x tends to 0 is 1. The diagram is not leading us astray here since it can be shown algebraically that the limit is in fact 1. We must always remember that neither the graph nor the output of the computer program that we will soon write **proves** that the limit equals 1; it simply helps our intuition. Some comments on the notion of **continuity** are in order.

A function f(x) is said to be continuous at x=a if and only if the limit as x tends to **a** is equal to the function value at x=a, i.e. f(a). This means in particular that the function has to be defined at x=a, that the limit of f(x) as x tends to a must exist, and finally, that these two values must be equal. If we define f(0)=1 (the value of the limit), then we have continuity of the function at x=0 (and everywhere else).

The very abstract-looking limit definition is really just saying that we can make f(x) get arbitrarily close (i.e. as close as we please) to the value L, just by making x sufficiently close to **a**, but not equal to **a**.

Pascal Program To Investigate A Limit

One of the first limits we look at when learning differentiation is one like this:

$$L = \lim_{h \to 0} \frac{(2+h)^2 - 2^2}{h}$$

Since h is approaching zero, we can use the computer to compute gradients of very short chords at x=2. We simply write a Pascal program to evaluate the quotient:

$$\frac{(2+h)^2-2^2}{h}$$

for various small values of h. Notice that the output of this program is a sequence of estimates of $f'(2)$, where $f(x)=x^2$.

```
program Limit;
{This program finds an approximation to the slope of f(x)=x*x at x=2}
var
  h, quotient : Real;
begin
  h := 1.0;
  writeln('     h          quotient'); writeln;
  while h > 0.000001 do begin
    quotient := (sqr(2.0 + h) - sqr(2.0))/h;
    writeln(h:12:8, quotient:12:8);
    h := h/10.0;
  end
end.
```

Run this program on your computer. Your output should be similar to the following:

```
Output

     h            quotient

 1.00000000     5.00000000
 0.10000000     4.10000000
 0.01000000     4.01000000
 0.00100000     4.00099998
 0.00010000     4.00009987
 0.00001000     4.00000936
```

Discussion of Output

Notice how the values get closer and closer to 4.0 as h gets smaller. Since we can evaluate this limit algebraically to get L=4, this confirms the output of our program. Notice that we are using the **algebraic** result to check our program, and not vice-versa. You should realize that it is impossible to **prove** convergence to a limit in this manner, i.e. by simply calculating a sequence of values. Our purpose here is to illustrate notion of a limit by calculating a sequence of **apparently converging values** in a computer program. The intention is to give us some insight into the mathematical concept of a limit, and there is no implication that the process of computing a sequence of numbers replaces the algebraic exercise of evaluating limits.

Using Limits To Find The Slope Of A Tangent

As we have said, the aim of differentiation is to find the slope of a tangent to a curve at a given point on the curve. The slope of the tangent to the curve y=f(x) at x=a is given by:

$$f'(a)=\lim_{h\to 0}\frac{f(a+h)-f(a)}{h}$$

We may use our techniques of the previous section to see how slopes of chords (secants) can be used to approximate the slope of the tangent. The quotient:

$$\frac{f(a+h)-f(a)}{h}$$

is simply the slope of the chord between the points P(a,f(a)) and Q(a+h,f(a+h)) on the curve. As $h\to 0$, the second point, Q, moves closer to the first point, P, which is fixed. The chord becomes shorter and shorter; its slope approaching that of the tangent at P. This slope is the derivative of f(x) at P (x=a), denoted f'(a). A diagrammatic representation of the definition is given in Figure 5.4.

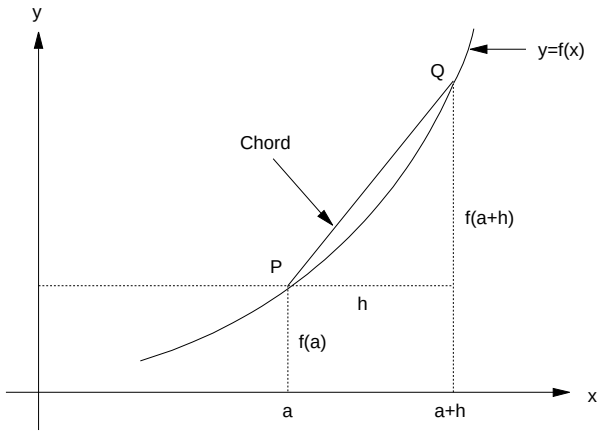


Figure 5.4 The Definition of a Derivative

5.2 Numerical Differentiation

In general, if we wish to determine an approximation to the derivative $f'(a)$ of the function f(x) at x=a, then the simplest procedure is to evaluate:

$$\frac{f(a+h)-f(a)}{h} \quad (1)$$

for suitably small h. It may be that the function f is very complicated and a human effort at algebraic manipulation using the usual rules of differentiation would be very tedious and error-prone. In cases such as this, it may be simpler just to use **numerical differentiation.**

Numerical differentiation in its simplest form is just what we have been doing, namely, the calculation of (1) for various small values of h. The result is usually a good approximation to f'(a) provided h is small enough. Numerical differentiation belongs to a whole class of numerical techniques for calculus applications. These methods deal with a calculus of **finite differences,** rather than precise mathematical limits. Included in the scope of finite difference techniques are numerical differentiation, numerical integration, and the numerical solution of differential equations. It is unlikely that you will meet differential equations at secondary school level, however those students who go on to tertiary education in mathematics or any of the physical sciences will study them in some detail.

Example

Suppose we have a function whose derivative (gradient) at a certain point is needed. The formula defining this function however is very complicated. We could, in principle, differentiate it using the product rule, chain rule etc., and then substitute for x, but this process would be very error-prone because of the large amount of algebra involved. We decide to use numerical differentiation instead. As an example of differentiation of a complicated function, let's write a Pascal program to estimate the derivative of:

$$y = \frac{x\sin(\exp(x\cos x))}{x^2-1} \quad at \quad x = 2.$$

The graph of this function is given Figure 5.5.

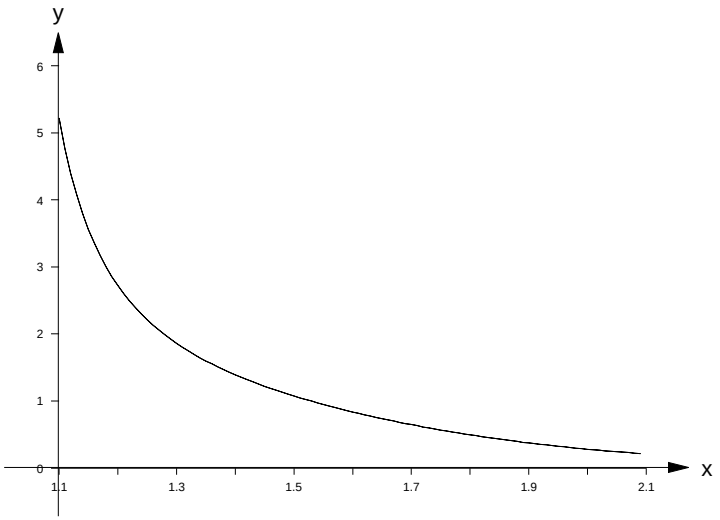


Figure 5.5 The Graph of A Complicated Function

Since the program simply computes f(2+h)-f(2) and divides by h, we need not worry about the algebraic details of simplifying the quotient, or using the rules of differentiation to find the derived function. A solution is given below.

```
program NumDiff;
{ This program finds an approximation to the gradient
  of the tangent to f(x)=x*sin(exp(x*cos(x)))/(x*x-1.0) at x=2 }
var
  h, k : Real;

function f(x : Real) : Real;
begin
  f:=x*sin(exp(x*cos(x)))/(x*x - 1.0)
end;

begin { main }
  h:=1.0;
  writeln('          h                Gradient');
  writeln;
  repeat
    k:=f(2.0 + h) - f(2.0);
    writeln(h:20:16,k/h:20:16);
    h:=h/10.0;
  until h < 0.00000000000001
end. { main }
```

Output

```
         h                Gradient

1.0000000000000000  -0.2617399287700000
0.1000000000000000  -0.7188904170800000
0.0100000000000000  -0.8109634454700000
0.0010000000000000  -0.8208144208800000
0.0001000000000000  -0.8218060247600000
0.0000100000000000  -0.8219049959700000
0.0000010000000000  -0.8219108167400000
0.0000001000000000  -0.8218830771500000
0.0000000100000000  -0.8217284630600000
0.0000000010000000  -0.8199094736500000
0.0000000001000000  -0.7958078640600000
0.0000000000100000  -0.5911715561600000
0.0000000000010000   0.0000000000000000
0.0000000000001000   0.0000000000000000
```

Discussion Of Output

We can see from the output of our program that the derivative (or gradient) seems to be around -0.8218 or -0.8219. Observe carefully that the accuracy begins to deteriorate after about h=0.0000001, and then the approximations quickly go completely haywire. Why is this? The floating-point arithmetic used by the Pascal compiler has reached its limit of accuracy at about six or seven decimal places, especially in the computation of **transcendental** functions such as sin, cos, exp. We have stated several times throughout this book that one must always be wary of results generated by floating-point (real) arithmetic. This point cannot be stressed too often, and here we have yet another example of the **finite-accuracy** of computer arithmetic.

Practical Use Of Numerical Differentiation

In scientific work, results obtained from experiments are usually sets of real numbers. These results from laboratory or other experiments are not usually functions; at least not in the sense of continuous algebraic expressions that we have been using. Nevertheless, empirical results must be analysed and it is common to fit ordered pairs of results to a theoretical curve, i.e. a function, which, it is assumed, is governing the underlying physical process being studied.

Analysis of such results often calls for some estimate of the derivative (or rate of change) of the function in question. The problem is that we don't have a formula to differentiate, but merely a set of points (ordered pairs). How can we possibly find derivatives? In cases where the data collected have closely-spaced abscissae (x-values), we may resort to **numerical differentiation.** A word of caution is appropriate here. Experimental data are of course subject to experimental error, and the process of numerical differentiation is particularly sensitive to such errors. This is because we are trying to find the quotient of very small quantities, and a small error can have an influence which is magnified many times.

Example

The data obtained from an ideal gas experiment at constant temperature are shown in Figures 5.6 and 6.7.

Pressure, P (kPa)	Volume, V (l)
1.00	2.96
1.02	2.90
1.04	2.85
1.06	2.81
1.08	2.73
1.10	2.69

Figure 5.6 Data For Ideal Gas Experiment

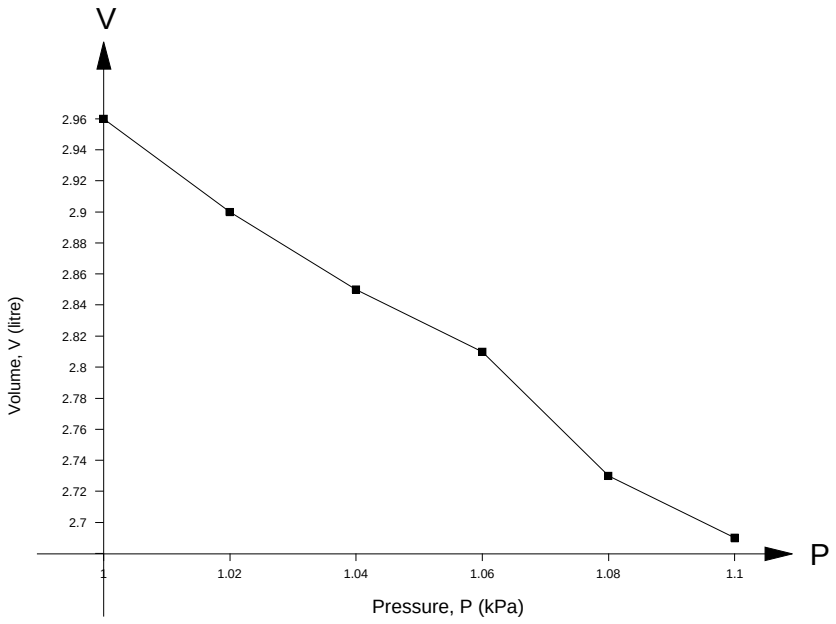


Figure 5.7 Graph For Ideal Gas Experiment

The theoretical model is PV=constant, i.e. P=k/V. We consider how to find the approximate rate of change of volume (V) when the pressure, P=1.04 kPa.

Solution

By differentiation, we have

$$\frac{dV}{dP} = -\frac{k}{P^2}.$$

Also, k = 2.96 kPa.l., approximately (why?), so that:

$$\frac{dV}{dP} = \frac{-2.96}{1.04^2} \approx -2.74 \quad \frac{litres}{kPa}$$

These values are of course approximate.

Checking this against an estimate obtained by numerical differentiation, we obtain:

$$\frac{\Delta V}{\Delta P} = \frac{2.81 - 2.90}{1.06 - 1.02} = \frac{-0.09}{0.04} = -2.25 \quad \frac{litres}{kPa}$$

Notice that we have chosen points which straddle the one of interest. The agreement here is not spectacular; nor should it be expected to be so. A number of factors contribute to this rather poor result. First, it should be remembered that the data do not exactly fit PV=k anyway. This being the case, it is just plain silly to expect any kind of accuracy in estimates of derivatives. Secondly, even if the values were exact, we are using the slope of a chord to approximate the slope of the tangent, and therefore cannot expect an exact result. Thirdly, and this point was mentioned earlier, we are taking the ratio of some fairly small quantities, and any error in the denominator will be magnified by division.

Numerical differentiation is a dubious process at best, however, when we study numerical integration, we shall see that the reverse is generally the case, i.e. relatively small errors in function values tend to have very little effect on numerical estimates of integrals. Integration is basically a "smoothing" process, and small "blips" tend to be smoothed out by what is basically a very stable process.

5.3 Maxima & Minima

One of the most important applications of differentiation is its use in finding **extreme** (i.e. maximum or minimum) values of functions. The usual approach to finding extreme points of a smooth function is to first differentiate it, and then solve the resulting expression equal to zero. (For our purposes, we define a smooth function to be one which possesses derivatives of all orders throughout its domain of definition.) Thus, for example, to find the maximum and minimum of:

$$f(x) = x \sin x + \cos x$$

we must first find the derivative, i.e.

$$f'(x) = x \cos x + \sin x - \sin x$$

$$= x \cos x.$$

We then solve f'(x) = x cos x = 0, in order to find the set, S, of **stationary** points of f(x). If we let E be the set of extreme points of f, then E is a subset of S. A stationary point is quite simply one where the derivative is zero. This condition is necessary but not sufficient for the point to be an extreme point. A point of horizontal inflection is a simple example of a stationary point which is neither a maximum nor a minimum. At such a point, the derivative (slope) is of course zero, however, there is no turning point.

Let's first consider an alternative method of finding approximate extreme values of f(x) on an interval which does not involve differentiation. Our strategy is to compute a large number of values of the function at closely-spaced values of x, all the while keeping track of which function values are the largest and smallest. We may refer to this technique as a **simple linear search,** and it is a rather primitive method for finding extrema. Indeed, it may fail spectacularly if our spacing of x-values is too coarse, since we may "step around" a peak in the graph of the function, in a region of high rate-of-change. Examples can always be constructed where this method will perform poorly, no matter how fine our spacing of points. As with all numerical methods, it pays us to be aware of the limitations of the algorithm being used. Blind application of any numerical method without regard for its range of usefulness or conditions under which it may be expected to work correctly is just plain foolishness and a waste of time.

Program For Linear Search For Maxima & Minima

In the next Pascal program, we consider a cubic polynomial with known zeroes:

f(x)=(x-4)(x+2)(x-6).

The zeroes are clearly x=-2,4,6. This may be seen directly from the factorised form of the cubic equation just presented, and also from the graph of the function, which is given in Figure 5.7.

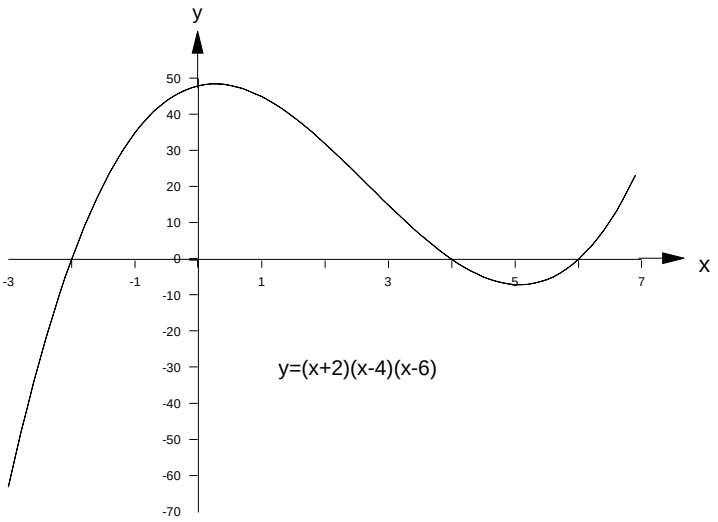


Figure 5.7 The Graph of f(x)=(x-4)(x+2)(x-6).

The program finds the overall maximum and minimum values of the function on the interval [0,10]. These may or may not correspond to turning points of the function.

```
program Extrema1;
{ This program finds maximum and minimum values of a given function }
var
  a,b,h,x,y            : Real;
  i,n                  : Integer;
  SmallestSoFar,xSmall : Real;
  BiggestSoFar, xBig   : Real;

function f(x : Real) : Real;
begin
  f:=((1.0*x - 8.0)*x + 4.0)*x + 48.0; { (x-4)(x+2)(x-6) }
end;

begin { main }
  a:=0.0; b:=10.0; n:=100;
  h:=(b - a)/n;
  SmallestSoFar:=1.0E25; BiggestSoFar:=-SmallestSoFar;
  xSmall:=0.0; xBig:=0.0;
  for i:=0 to n do begin
    x:=a + i*h;
    y:=f(x);
    if y<SmallestSoFar then begin
      SmallestSoFar:=y;
      xSmall:=x;
    end;
    if y>BiggestSoFar then begin
      BiggestSoFar:=y;
      xBig:=x;
    end;
  end;
  writeln('Min is ',SmallestSoFar:10:6,'; occurring at x =
',xSmall:5:2);
  writeln('Max is ',BiggestSoFar :10:6,'; occurring at x =
',xBig:5:2);
end. { main }
```

```
Output

Min is -7.029000; occurring at x =       5.10
Max is 288.000000; occurring at x =      10.00
```

Discussion Of Output

From the graph it would seem that the function we considered has a maximum of roughly 50 at a value of x which is not much greater than zero. We must be very careful to distinguish between a **local maximum** and a **global maximum.** A local maximum is simply a point at which the function has a value which is greater than or equal to its value at every other point in some neighbourhood; even a very small neighbourhood - it doesn't matter how small. For smooth functions, the derivative will be zero at such points, i.e. these points will be **stationary** points of the function. By contrast, if we seek the **overall** maximum of a function on a closed interval [a,b], that is, the largest value actually attained by the function on [a,b], then this value is referred to as the **global maximum of f on [a,b].** Even for a smooth function, this global maximum may be attained at a smooth turning-point, or **at an end point of the interval.** The rule for finding global maxima for smooth functions on [a,b] is therefore to first find the stationary points, check if they are smooth maxima, if so, calculate the function value at each such point. These are the **local maxima.** Then, calculate **f(a)** and **f(b).** Compare these two values with the set of local maxima and determine the largest. This value is the global maximum, and it is this value, viz. f(b)=f(10)=208, that has been found by our program above. It is the correct global maximum since it is certainly larger than f(a)=f(0)=48, and our only local maximum on [0,10] of value approximately 50 near x=0. Re-run the program with the interval [a,b]=[-1,7], and compare your results.

We should also realize that some very simple functions **do not have maxima**, even on small open intervals such as (0,1). For example, consider the function f(x) = 1/x on the open interval (0,1).

This function has no maximum on (0,1). For a proof, we suppose the contrary, that is, suppose this f(x) has as maximum M>1 which is attained on (0,1). M must be be greater than 1 since, e.g. f(1/2)=2. We then take $x = x_0 = \frac{1}{2M}$ which lies in (0,1). Since $f(x_0) = 2M > M$, we have a contradiction.

Therefore no M, no matter how large we choose it, will suffice for a maximum on (0,1). Of course, this means that f(x) becomes arbitrarily large on (0,1), but didn't we know this already? We should have, but it is still a good example to illustrate the simple fact that **some functions just do not have maxima.** The same comments apply equally of course to minima.

5.4 Historical Note

Sir Isaac Newton (1642-1727)

For more than twelve centuries after Christ, Europe was dominated by a corrupt church which stifled the development of science and mathematics. During the Renaissance period, we find the discoveries of men such as Galileo, Brahe, Copernicus, Newton transforming attitudes to science, nature and religion. It may seem paradoxical to young minds today that men of such enormous intellect such as Newton, Leibniz, Pascal - men of science, were also deeply religious. They considered the study of spiritual truths at least as important as the pursuit of scientific or "natural" truths. Here we have counterexamples to the modern popular misconception that religious belief and scientific rationality are entirely incompatible.

Let us consider for a moment the work of Sir Isaac Newton. It is rarely emphasized in a history course the considerable extent to which Newton made contributions to human knowledge in the areas of mathematics, physics, and theology. This man, an intellectual giant - perhaps the greatest mathematician the world has ever known - discovered the laws of gravitation, the laws of motion which bear his name, the refraction of light through a prism, invented the differential and integral calculus, discovered the binomial theorem, the iterative solution of f(x)=0 which bears his name, and many other scientific and mathematical achievements.

Prior to the time that he formulated his equations for the motion of astronomical bodies, no theory of gravitation existed, let alone equations of motion, and further still - the means by which to solve them! If only these equations could be solved, the paths of the planets and the moon could be predicted with accuracy. Here was the breakthrough that was needed for compilation of accurate tables for navigation at sea. Kepler, Brahe and Copernicus had spent painstaking years of observation and calculation related to heavenly bodies in order to solve problems of navigation.

Newton's work was of immense worth. It is little appreciated by people in our day. How was he to solve these equations of motion that he realised would lead to such tremendous advancement in the science of navigation so desperately needed by the British government? As Isaac Asimov says: "fortunately, he was Newton", meaning that he just invented the whole differential calculus in order to solve his equations of motion! Newton's researches into physics and mathematics are staggering. He also found time to run the Royal Mint for the British government and much time for theological writing, including his famous commentary on Daniel and The Revelation **("Observations On The Prophecies Of Daniel & The Apocalypse").**

During the latter part of the 17th century, and on into the 18th and 19th centuries, the work of Newton and others following him had a profound influence on the scientific and philosophical outlook of the time. The success of Newton's theories had the effect of permanently breaking the stifling stranglehold in which the religious authority of the time had held intellectual freedom itself. Newton had accurately predicted the motion of the planets; the **geocentric** view of the universe or solar system had given way to the **heliocentric** view (i.e. sun is the centre); the earth was round and not flat; and the motion of other bodies (not just the heavenly bodies) could be predicted according to Newton's laws as long as their initial position and velocity were known, along with the forces acting on them. These were enormous achievements, and totally unprecedented in the history of science.

A recent television advertising campaign in Queensland for The Capita Financial Group uses some well-known quotations to get its message across. Although the originators of these quotations are not acknowledged, one is clearly recognizable as Sir Isaac Newton's:

> *" If I have seen farther than others, it is because I have stood on the shoulders of giants. "*

5.5 Summary

- The whole of the calculus rests on the notion of a **limit.** To say that a limit of a sequence or a limit of a function f(x) exists is to assert that a fixed value (the limit) may be approached by values of the sequence or function to any pre-specified degree of closeness. This is done by taking sufficiently many terms of the sequence, or by taking values of x sufficiently close to a given fixed number (the limit point).
- Limits may exist at points where the function is not even **defined**. If a limit exists at a point, and the function value is defined at the point, and the function value is equal to the limit, then the function is said to be **continuous** at that point.
- Certain aspects of limits may be examined on a computer. In this chapter we looked at very simple programs to just compute function values and ratios and display them in a table. In this way, it was seen intuitively that a limiting value was being approached. It is important to remember that such a procedure in no way constitutes a mathematical proof either of the existence of a limit, nor of its value. The intention is merely to give us a better insight into just what is going on geometrically and help us to comprehend the algebraic processes of evaluating limits. It is easy to construct examples where a limit apparently exists, but a valid proof can be constructed to show that no limit is possible.
- Differentiation is the process where the limit concept is used to determine the instantaneous rate of change (gradient) of a function at a given point. Such values may be approximated on a computer by the ratios of **finite differences.** This process is called **numerical differentiation.**
- Extreme values of functions (maxima and minima) may be approximated on a computer by a number of methods. One of the simplest is to just **search** through a set of function values generated from regularly-spaced x-values, keeping track of the most extreme values so far. This technique is known as **simple linear search.**

5.6 Exercises

*1. (i) Write a Pascal program to obtain a sequence of approximations to the function:

$$f(x) = \frac{\sin x}{x} \text{ as } x \to 0.$$

Use h = 1.0, 0.1, 0.01 etc. as in the example in the text. Use the Pascal intrinsic sine function, and make an "educated guess" as to the value of:

$$\lim_{x \to 0} \frac{\sin x}{x}$$

(ii) As above but use h = -1.0, -0.1, -0.01 etc.

(iii) As in (i) but consider:

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

for h = 1.0, 0.1, 0.01 etc. and x=0.

*2. Calculate the relative errors of the data of the ideal gas example in the text, assuming the exact answer to be PV=k=2.95 (even this is not exact physically, but it will serve our purpose here). Different approaches are possible. One of the simplest is to assume the P values are without error, and then deduce the absolute and relative errors of corresponding values of V.

*3. Use the numerical differentiation program to estimate the derivative of the following functions. Differentiate these functions algebraically if possible/feasible and compare your program's output with the exact value.

(i) $y=\frac{x\sin(\exp(x\cos x))}{x^2-1}$ at $x=2.$

(ii) $y=x^3+3x^2-x+2$ at $x=-3.$

(iii) $y=\frac{\tan x}{x}$ at $x=0,\frac{\pi}{2},\pi,\frac{3\pi}{2},2\pi.$

(iv) $y=\sin x$ at $x=0,\frac{\pi}{2},\pi,\frac{3\pi}{2},2\pi.$

(v) $y=x^2+3x-4$ at $x=-3.$

4. Use differentiation to determine the exact extreme points on [0,10] of the function in the EXTREMA1.PAS program example. Compare with those obtained by the program. Why does the program come up with a maximum which is not at a turning point of the polynomial function?

*5. Use the numerical differentiation program to estimate derivatives of the following functions. Compare them with the exact value obtained by ordinary differentiation in each case.

(i) $f(x)=x^2+5x+2$ at $x=1.$

(ii) $f(x)=\ln x$ at $x=2.$

(iii) $f(x)=\sin x$ at $x=\pi/2$

(iv) $f(x)=\ln(\sec x)$ at $x=0.$

(v) $f(x)=arctan x$ at $x=0.$

*6. Use the Simple Linear Search method to locate approximate local maxima and minima for the following functions on the intervals given:

(i) $f(x)=(x-2)(x+1)(x-7)$ on [-1,9]

(ii) $f(x)=(x+1)(x-5)(x+3)$ on [-3,3]

(iii) $f(x)=\sin x$ on $[0,\pi]$

Verify your answers in each case by the usual method, namely, differentiation and solving $f'(x)=0.$

*7. Find the global maximum and minimum values of the following functions on the interval [-10,10].

(i) $f(x)=2x-3$

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

(iii) $f(x)=\cos x$

(iv) $f(x)=x^3-4x$

(v) $f(x)=-x^3+5$

(vi) $f(x)=\frac{1}{x}$

(vii) $f(x)=(x-2)(x+4)(2x-1)$

(viii) $f(x)=\cos\frac{x}{2}$

*8. Modify the simple linear search program (EXTREMA1) to print a complete table of (x,y) values. One pair will be printed for each point examined by the program.

9. **Challenge Problem**: Extend the simple linear search program (EXTREMA1) further to print an asterisk beside the maximum and minimum values.

6 SOLVING NON-LINEAR EQUATIONS

6.1 Indirect Methods For Solving f(x)=0

A very common requirement in science, engineering and quantitative business analysis is the solution of mathematical equations arising from physical or business/economic problems. Some simple examples of equations are:

$ax + b = 0$ (Linear), and $ax^2 + bx + c = 0$ (Quadratic).

In these cases we seek the solution (or solutions) for x with a, b, c being supplied as constants for any given problem. For equations of such simple form, we may express the solution, x, in terms of the **coefficients** a, b, c. In an earlier chapter we have already studied an algorithm to solve quadratics for real solutions using the quadratic formula for the general case. We have:

$$x = \frac{-b}{a} \quad \textit{for the linear case}, \text{ and}$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \textit{for the quadratic case},$$

provided, of course that in either case, a is different from zero, and in the quadratic case, the discriminant $b^2 - 4ac$ isn't negative.

The linear and quadratic equations have the **unusual property** that their solutions may be expressed in terms of their coefficients. Any algorithm which directly calculates the solution(s) of an equation from a formula is said to be a **direct** algorithm. Therefore, an algorithm employing our quadratic formula given above to solve a quadratic equation would be a direct algorithm. By contrast, an **indirect** algorithm is one which uses a series of **iterations** to gradually "home-in" on the solution. The equation to be solved is usually expressed in the form **f(x)=0.** Before considering iterative indirect algorithms for solving f(x)=0, we re-emphasize that we already have a direct algorithm to solve a quadratic equation. It is just the quadratic formula given above, which we used as the basis for a Pascal program in an earlier chapter. A comparison between the direct and indirect methods is made in Table 6.1.

	Direct Method	**Indirect Method**
Definition	Compute solution of f(x)=0 directly from a formula in terms of the coefficients or parameters of f(x).	Generate a sequence of approximations to solution of f(x)=0 and stop when latest approximation is close enough.
Example 1	Linear Formula.	Bisection Method.
Example 2	Quadratic Formula.	Newton's Method.

Table 6.1 Direct & Indirect Methods For Solving f(x)=0.

We should also realise that the indirect methods we study in this chapter are equally applicable to non-polynomial functions. Even though polynomials are very common, many non-polynomial equations occur naturally in physical problems, and most equations requiring solution are not polynomial equations. Such equations are often referred to as **transcendental** equations. For example, solutions to equations such as:

$$e^x = 2 - x \tag{1}$$

are commonly required in science and engineering problems. One way to find a crude solution to equations such as these is to accurately graph both sides as separate functions of x. The point(s) of intersection will then be the solutions. As an example, for the equation given just above, we would graph $y = e^x$ and $y = 2 - x$ on the same piece of graph paper. This is illustrated in Figure 6.1, in which the point, s, is the solution.

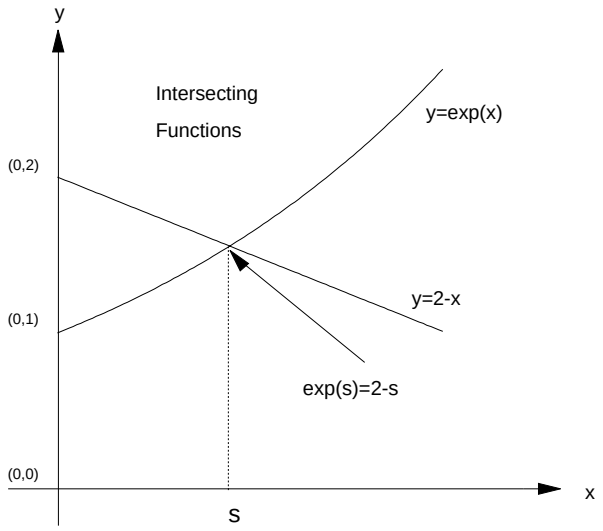


Figure 6.1 Graphical Solution of exp(x)=2-x

To solve the same equation using one of the indirect methods that we soon study, we would first express the equation in the standard form **f(x)=0.** This can be done in many ways. Perhaps the simplest would be:

$$f_1(x) = e^x + x - 2 = 0.$$

Alternatively, we could rearrange it to read:

$$f_2(x) = \frac{e^x + x}{2} - 1 = 0,$$

or, by taking logarithms of both sides of (1):

$$f_3(x) = x - \ln(2 - x) = 0.$$

Any given algorithm for solution may behave quite differently when presented with each of these functions, even though the solution is the same in each case.

Basic Graphical Ideas

Figure 6.2 shows us that the function f(x)=(x-2)(x-5) crosses the axis at x=2 and x=5. Of course, to students who are familiar with polynomial theory and factorization, this will be obvious. Suppose though, that we have to solve the equation f(x)=0, for some given function f(x) which may not be a polynomial.

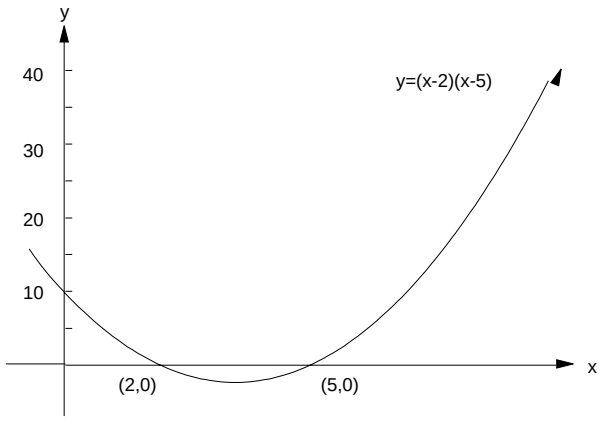


Figure 6.2 The Graph of y = (x - 2)(x - 5)

If we draw the graph of y=f(x), then from a geometrical point of view, we are looking for **points where the function intersects the x-axis.** This is the point that Figure 6.2 is meant to convey. When we consider indirect methods later in this chapter, geometrical concepts will be used to help us understand how the methods work. For example, Newton's method uses the **tangent** to the graph of the function to direct us to the next approximation. As a general rule, no matter what method we use, it is always a good idea to have some reasonably-accurate plot of the function before we start. This gives us some idea of the general shape of the curve y=f(x), and may also point out some regions where problems may arise for our particular method of solution. An example of this is stationary points for Newton's method. We consider these and other pitfalls later in the chapter. In the chapter titled **Computer Graphics** you will find a program to plot functions.

General Iterative Approach To Solving f(x)=0.

The general **indirect** approach to solving f(x)=0 is by iterative generation of a sequence of approximations and is summarised in Figure 6.3.

ITERATIVE APPROACH TO SOLVING f(x) = 0.

Express any equation to be solved in the form f(x) = 0. To solve this equation on a computing device, the basic approach is to generate a sequence of approximations $\{x_n\}$ to the true solution, s.

Figure 6.3 Iterative Approach to Solving f(x)=0.

The various indirect algorithms differ as to how this sequence of approximations to the solution is generated. Since we cannot always use a direct method, we must resort in the general case to **indirect** methods. The methods in effect, conduct a **smart search** of the set of real numbers (actually, rational numbers, since the computer cannot represent irrationals), in order to locate an acceptably close approximation to the solution we seek. Although the methods described here are generically the same in that they search for solutions rather than directly calculating them, they differ markedly in many ways. Characteristics such as rate of convergence (some don't always converge anyway!), behaviour in the presence of multiple roots or stationary points of the function, and other aspects must be considered.

We now consider a general formulation of the indirect approach to searching for solutions to f(x)=0. As stated earlier, we use the computer to somehow generate a sequence of values, which we hope will lead to an acceptable approximation to a solution of f(x)=0 in a reasonable number of iterations (cycles).

A Generic Indirect Algorithm For Solving f(x)=0

The following general approach is used to solve equations in an **iterative** or **indirect** manner. The algorithm is expressed in a Pascal-like pseudocode. Figure 6.4 has the details.

GENERIC INDIRECT ALGORITHM FOR SOLVING f(x)=0.

```
set x to x0   {initial approximation to s}
set ε to small value  {convergence tolerance}
repeat
  generate next element of sequence and call it x
  evaluate f(x)
until | f(x) |< ε
```

Figure 6.4 Generic Indirect Algorithm for Solving f(x)=0.

We can see from Figure 6.4 that to start the whole process off, we need to have some reasonable estimate for the first approximation. Finding this initial estimate is in fact a problem in itself. For the bisection method, we have to find an interval [a,b] on which the function differs in sign at the endpoints, i.e. f(a)f(b)<0 - convergence is then absolutely guaranteed. For Newton's method, there

is no foolproof way of finding the initial estimate, although a slower, always-convergent method such as bisection is frequently used to find an x_0 which is close to the solution. Newton may then be used to rapidly converge to the required degree of accuracy.

It can be seen that the search for a suitable x continues as long as $|f(x)| \geq \varepsilon$, i.e. as long as the absolute value of the function is not close to zero. Graphically, this means that the value of f at the current value of x lies outside a small horizontal band of width 2ε and centred on the x-axis. Iterations will continue while the function value at the current value of x is still too large in modulus to satisfy the convergence criterion. In typical applications, we might take $\varepsilon = 0.0000005$. Figure 6.5 shows $y = e^x \pm \varepsilon$.

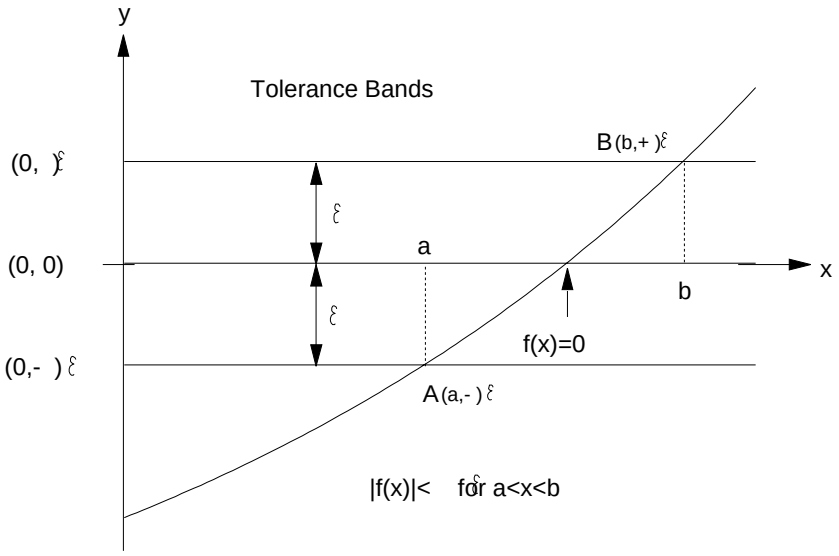


Figure 6.5 Tolerance Bands

Uniqueness Of Solution

As a general point, it is important to be aware that there may exist more than one solution to f(x)=0, for a given function f(x). In other words, it can easily happen that there are at least two distinct values, say s and t, such that f(s)=f(t)=0. The methods that we use here (bisection and Newton), are two of the commonest ones, and, in common with most other methods, give no **a priori** (that is, **in advance)** guarantees as to which member of a set of multiple solutions convergence will occur. In other words, even if the algorithm is known to converge, we normally have no way of knowing which solution will be found first.

We now consider these two specific sequence-generating algorithms (bisection and Newton). In summary, once given a starting value, each of these methods will produce a **sequence of values**, which we expect to ultimately converge to a solution of f(x)=0, but we normally have no way of knowing in advance which of the set of possible alternative solutions may be found.

6.2 The Bisection Method

For this method, we start by choosing an interval [a,b] for which f(a) and f(b) differ in sign. If f is **continuous,** then there must exist a point, s, between a and b such that f(s)=0. This is a consequence of the **Weierstrass Intermediate Value Theorem** which states that a continuous function on a closed interval [a,b] assumes every real value between its values at the endpoints, a and b. Since one of f(a) and f(b) is positive, and the other is negative, there must exist a point x on [a,b] such that f(x)=0. The bisection method conducts a **smart search** of the interval [a,b] for this value of x by continually halving the sub-interval in which the root is known to lie. The search terminates when this sub-interval is sufficiently narrow. The bisection method is illustrated in Figure 6.6.

The basis of the bisection method is simply to take as our first approximation, x, the mid-point of the interval [a,b], that is, x=(a+b)/2. We then examine the sign of the function at this value of x. There are three possibilities, of course: f(x) will be either zero, negative, or positive. In the (rather unlikely) case that f(x) is zero, then the algorithm will terminate since this is exactly the point we are looking for. Otherwise, f(x) will differ in sign from one of the original f(a) and f(b). We then

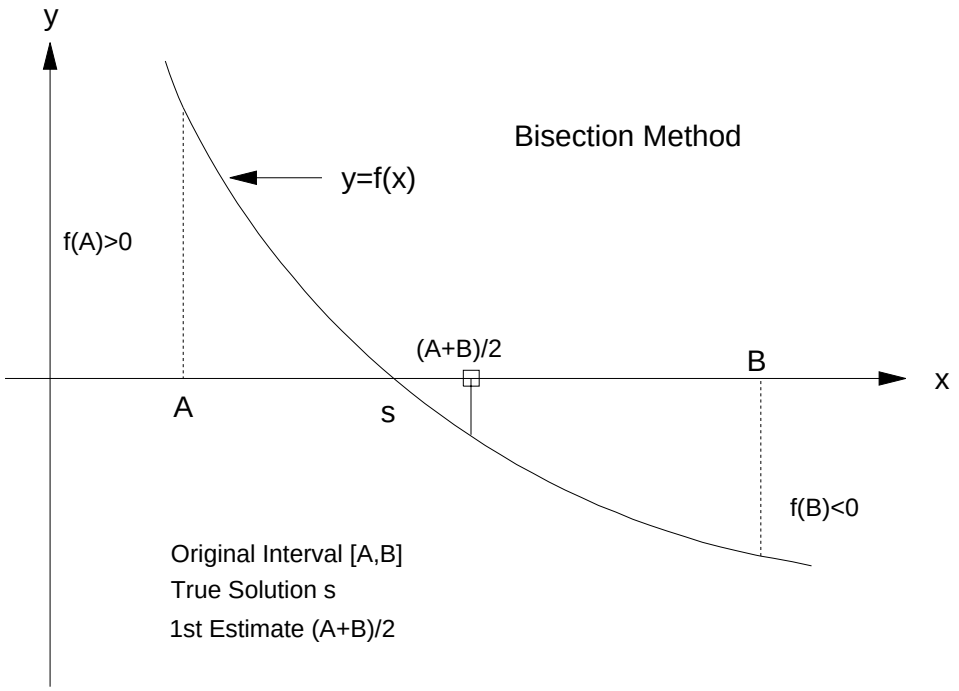


Figure 6.6 The Bisection Method

simply discard the other half of the interval, apply the Weierstrass Theorem and bisect the halved interval once again. This bisection process can be continued as long as desired; it is usually terminated when the required accuracy is reached or when an iteration limit is reached.

We now consider a Pascal program to implement the bisection algorithm to solve $f(x) = x^2 - 2 = 0$ on the interval [1,2]. This interval is acceptable since f(1)f(2)=(-1)(2)=-2<0. The answer is, of course, the square root of 2, or 1.41421, approximately.

```
program Bisection;
{ This program based on the bisection Method locates a root of a
 function given an interval within which the function changes sign }
var
  Epsilon, a, b, x : Real;
  NumIterations    : Integer;

function f(x : Real) : Real;
begin
    f:=x*x - 2.0   { the particular function being considered }
end;

begin {main}
  epsilon:= 0.0000005;
  a:=1.0; b:=2.0;  { a chosen interval within which the root exists }
  x:=(a + b)/2.0;
  NumIterations:=0;
  writeln('ItNo      a           b           x          f(x)');
  writeln;
  while abs(f(x)) >= epsilon do begin
    NumIterations:=NumIterations + 1;
    writeln(NumIterations:4,a:12:8,b:12:8,x:12:8,f(x):12:8);
    if f(a)*f(x) < 0 then b:=x else a:=x;
    x:=(a+b)/2
  end;
end. {main)
```

```
Output

ItNo      a           b           x           f(x)

  1  1.00000000  2.00000000  1.50000000  0.25000000
  2  1.00000000  1.50000000  1.25000000 -0.43750000
  3  1.25000000  1.50000000  1.37500000 -0.10937500
  4  1.37500000  1.50000000  1.43750000  0.06640625
  5  1.37500000  1.43750000  1.40625000 -0.02246094
  6  1.40625000  1.43750000  1.42187500  0.02172852
  7  1.40625000  1.42187500  1.41406250 -0.00042725
  8  1.41406250  1.42187500  1.41796875  0.01063538
  9  1.41406250  1.41796875  1.41601563  0.00510025
 10  1.41406250  1.41601563  1.41503906  0.00233555
 11  1.41406250  1.41503906  1.41455078  0.00095391
 12  1.41406250  1.41455078  1.41430664  0.00026327
 13  1.41406250  1.41430664  1.41418457 -0.00008200
 14  1.41418457  1.41430664  1.41424561  0.00009063
 15  1.41418457  1.41424561  1.41421509  0.00000431
 16  1.41418457  1.41421509  1.41419983 -0.00003884
 17  1.41419983  1.41421509  1.41420746 -0.00001726
 18  1.41420746  1.41421509  1.41421127 -0.00000647
 19  1.41421127  1.41421509  1.41421318 -0.00000108
 20  1.41421318  1.41421509  1.41421413  0.00000162
```

Properties Of The Bisection Algorithm

Notice that we have output the current sub-interval [a,b], as well as its mid-point x, and the function value, f(x), at each step (iteration) of the method. If we examine each row carefully, we can see that at each stage, the length of [a,b] (which is of course just b-a), is **halved.**

Observe also that at iteration 7, the estimate x=1.41406250 is actually correct to 3 decimal places, and subsequent estimates are actually **worse** until we get to iteration 15! This is a characteristic of the bisection algorithm, which is really quite blind, since, as we amplify below, it does not take into account the value of the function f(x) **at any time**, but only works on the **sign** of f. Convergence is however **guaranteed**, and, as we shall see now, an upper bound can be placed on the number of iterations required to achieve a predetermined accuracy.

Accuracy Of The Bisection Method

In any numerical method, it is useful to have some knowledge of the degree of accuracy of the solution we have obtained on the computer. For the bisection method, we can always place an upper bound on the error in our estimate of s. We reason as follows - at any stage of the process, x is the mid-point of the current sub-interval [a,b]. Since we know that the solution is somewhere in between the current a and b, then the mid-point estimate cannot be in error by more than half the length of [a,b]. Denoting the **original interval** by [A,B], the true solution by s, and the current (n th) estimate by x, we can write:

$$|x_1 - s| = \text{absolute error of 1st approximation} < \frac{(B-A)}{2^1}$$

$$|x_2 - s| = \text{absolute error of 2nd approximation} < \frac{(B-A)}{2^2}$$

$$|x_3 - s| = \text{absolute error of 3rd approximation} < \frac{(B-A)}{2^3}$$

In general, therefore, we have:

$$|x_n - s| = \text{absolute error of nth approximation} < \frac{(B-A)}{2^n}$$

This pattern is fairly plain to see. Because the interval containing the solution is halved at each step, and both the solution and current estimate are confined to the interval, we can write these explicit upper bounds. Since the right-hand side of the general bound is a **geometric progression** with common ratio 1/2, we know that it can be made as small as we please - just by taking N large enough. In other words, given any $\varepsilon > 0$, there exists an integer N such that the absolute error is less than ε. The error of our estimate can be made as small as we please, i.e.

$$\frac{(B-A)}{2^N} < \varepsilon$$

Solving for epsilon, we get:

$$N > \frac{\log_2(B-A)}{\varepsilon}$$

This means that convergence to $s \pm \varepsilon$ will be guaranteed if more than N iterations are done.

Example

Find the square root of 2 to 6 decimal places; starting with [1,2] and using the bisection method. Use $f(x) = x^2 - 2$. To solve this problem we let x_n be our final approximation to root 2. If it is correct to 6 decimal places then we must have $|x_n - \sqrt{2}| < 0.0000005$

Substituting $A = 1, \quad B = 2, \quad \varepsilon = 0.5 \times 10^{-6}$ we arrive at:

$$\begin{aligned} N &> \frac{\log_2(B-A)}{\varepsilon} \\ &= \log_2(2000000) \\ &= 1 + \log_2(1000000) \\ &= 1 + \frac{\log_{10}(1000000)}{\log_{10} 2} \\ &\approx 1 + \frac{6}{0.3010} \\ &= 20.93 \\ &\approx 21. \end{aligned}$$

We can therefore guarantee that convergence to required accuracy will occur after 21 iterations at most. In actual fact, we sometimes get there with less (20 in this case with the Pascal program). The estimate of maximum iterations we have derived is the most pessimistic, but it does give us a useful indicator of the speed of the method. The bisection method is rarely used in practice, because of its slow convergence; however it does have one thing going for it - it always works! In the next part, we study Newton's method, which, along with its relatives, is probably the most widely used. Its main attributes are conceptual simplicity and fast convergence; however, it is not guaranteed to converge.

6.3 Newton's Method

We have seen that the bisection method is very slow to converge. The reason is that it takes no account whatsoever of the magnitude of the function f at any time, but simply works on the sign of the function. The method which we now discuss, known as Newton's algorithm (or sometimes the Newton-Raphson algorithm), is generally the best since it is very fast. It not only takes the value of y at the latest estimate (x,y) into account, but also the direction of the curve at that point. That is, the slope of the tangent is found and used to direct us to the next estimate for x. Geometrically, we just "slide" down the tangent line at (x,y) until we hit the x-axis. This new value of x becomes the next approximation or estimate of the solution to f(x)=0. This idea is illustrated in Figure 6.7. So far, we have been discussing mainly polynomials. Actually, this method works for most functions of interest - not just polynomials. The main reason for our use of polynomials in the examples is that the are the "simplest" kinds of functions, and therefore it is usually easy to verify algebraically that our numerical methods are giving us the correct answers. This verification process is very important for algorithms and will be stressed throughout this book.

To derive the algorithm, we let x_0 be our initial estimate of a solution to f(x)=0. As explained at the start of this section, we wish to generate a sequence of approximations $\{x_n\}$ to the true solution, which we call **s**, that is f(s)=0 exactly. The first step is to construct the tangent at $(x_0, f(x_0))$ on the curve. The equation of this tangent is:

$$y - f(x_0) = f'(x_0).(x - x_0) \quad (1)$$

This is just the point-slope formula for a straight line with slope $f'(x_0)$. Remember that $f'(x_0)$ is just a number - it is a constant. How do we find the point where this line intersects the x-axis? Simply put y=0 in equation 1, with $x = x_1$ (the new approximation for x). This gives:

$$0 - f(x_0) = f'(x_0).(x_1 - x_0) \quad (2)$$

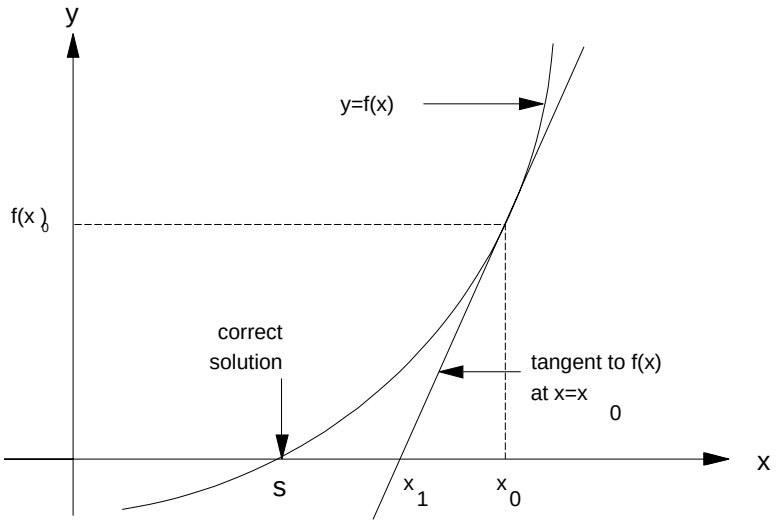


Figure 6.7 Newton's Method

Solving for x_1, we have:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \quad (3)$$

Equation 3 is Newton's iteration formula. It tells us how to get x_1 (the next approximation) if we know x_0 (the current approximation), and of course can compute the function and its derivative at x_0. Since we wish to continue this process of generating better estimates of the solution, s, we use the general formula given in Figure 6.8.

NEWTON'S ITERATION FORMULA

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

x_{n+1} is where the tangent to f(x) at $(x_n, f(x_n))$ cuts the x - axis. It is the next approximation after x_n in the sequence.

Figure 6.8 Newton's Iteration Formula

Notice that Newton's method will fail if $f'(x_n) = 0$, since the denominator of equation 3 will be zero. This corresponds to x_n being a stationary point of f, and of course, any tangent to f at such a point will be horizontal, and never intersect the x-axis.

A Pascal Program For Newton's Method

We consider solving the same function as for the bisection method. In this way comparisons may be made between the two algorithms. Notice that we now need to write another function subprogram for the derivative of f.

```
program Newton;
{ This program uses Newton's Method to locate a root
of a function given an approximate root }
var
  Epsilon, x    : Real;
  NumIterations : Integer;

function f(x : Real) : Real;
begin
  f:=x*x - 2.0  { the particular function being considered }
end;

function df(x : Real) : Real;
begin
  df:=2.0*x  { the derivative of the function being considered }
end;

begin { main }
  epsilon:= 0.0000005;
  x:=1.0;  { an approximate root - the starting point }
  NumIterations:=0;
  writeln('ItNo      x          f(x)        f''(x)');
  writeln;
  while abs(f(x)) >= epsilon do begin
    NumIterations:=NumIterations + 1;
    writeln(NumIterations:4,x:12:8,f(x):12:8,df(x):12:8);
    x:=x - f(x)/df(x)
  end;
end. { main }
```

```
ItNo    x          f(x)        f'(x)

 1  1.00000000 -1.00000000  2.00000000
 2  1.50000000  0.25000000  3.00000000
 3  1.41666667  0.00694444  2.83333333
 4  1.41421569  0.00000601  2.82843137
```

Properties Of The Newton Iteration

1. The method fails if it encounters a stationary point, unlike the bisection method, which is guaranteed to converge, once we locate an interval enclosing the root.
2. When convergence does occur, it is usually very rapid, except in the neighbourhood of multiple roots.

Another Example Of Newton's Method

Let us now consider an application of the Newton algorithm to solve a non-polynomial function. As we have already stated, such equations arise very often in science and engineering problems. The application that we consider is projectile motion. The mathematical study of projectile motion received a great deal of funding during the second world war (1939-1945), as did mathematical cryptography and cryptanalysis (the sciences of devising and breaking codes respectively). Advances in the Allied Forces' research in these areas were major factors, along with the development of the atomic bomb in the USA, of course, in their final victory in World War II. It is a great pity indeed that it often requires a cataclysm of the magnitude of a global war for politicians to realise the value of mathematical research. We now consider an application of Newton's method to the solving of the equations of projectiles.

Application Of Newton's Method To Projectiles

The simplest mathematical model of projectile motion assumes a two-dimensional path (trajectory) with constant gravity, flat earth, constant mass of projectile, and no air resistance. None of these assumptions is correct, however this simple model and the equations resulting from it are surprisingly accurate for short-range calculations at moderate speeds.

Equations Of Motion

The standard equations of motion for a body in free-fall from Newton's Second Law of Motion (gets around, doesn't he!) are:

$$\frac{d^2y}{dt^2} = -g \tag{4}$$

$$\frac{d^2x}{dt^2} = 0 \tag{5}$$

with the initial conditions:

$$\dot{y}(0) = u\sin\alpha \tag{6}$$

$$\dot{x}(0) = u\cos\alpha \tag{7}$$

$$x(0) = 0 \tag{8}$$

$$y(0) = 0 \tag{9}$$

In these equations,

u = launch speed,
g = acceleration due to gravity at the earth's surface, and
α = launch angle with respect to the + x axis.

Integration of these equations is a pretty easy matter since, by our assumptions, u, g, and alpha are all constants. The first integration yields:

$$\dot{y}(t) = u\sin\alpha - gt \tag{10}$$

$$\dot{x}(t) = u\cos\alpha \tag{11}$$

A second integration then gives:

$$y(t) = ut\sin\alpha - \frac{gt^2}{2} \tag{12}$$

$$x(t) = ut\cos\alpha \tag{13}$$

Equations 12 and 13 are the standard parametric equations of motion for a simple projectile. (Students requiring further detail should consult their calculus teacher or textbook for the derivation of these equations). The variable t indicates time elapsed from the time of launch, which is, of course, t=0. We can obtain a cartesian equation for the path, i.e. y and x in one equation with no other variables present by eliminating t from equations 12 and 13. When this is done, a quadratic equation for y in terms of x with the constants forming expressions for the coefficients results:

$$y = x\tan\alpha - \frac{gx^2}{2u^2\cos^2\alpha} \tag{14}$$

We see from equation 14, since alpha, g, and u are all constants, that the path, or trajectory, is the familiar parabola; with axis vertical (x=0) and inverted so that it reaches a maximum (the maximum height attained by the projectile) for some x=xm>0. We know from theory of parabola symmetry that xm=R/2, where R is termed the horizontal range of the projectile. It is easy to find R (in terms of u, g, and alpha) since y=0 when x=R. The trajectory is shown on Figure 6.9.

Problem

Suppose that a target projectile has just been launched vertically, i.e. alpha=90 degrees for it; launch position is [a,0]. Our task is to launch our air-to-air projectile from (0,0) to intercept and destroy it. Now there is a ten-second time delay from when the target launch was made to when we can get our projectile mobile. With a given launch speed of twice that of the target, our problem is then to determine the correct launch angle such that our projectile hits the target in mid-flight. This is obviously an idealised problem, but it does give us some insight into practical applications of calculus and numerical approximation techniques. We should remark that the equations obtained for solving by Newton's technique can also be solved by trigonometric manipulation. This however does not prevent us from illustrating the use of the method for a non-trivial problem. In any case, it is a useful exercise for the student to algebraically solve the equation to be derived below, and then compare and reconcile the results with the numerical solution. A program to solve the problem is given below.

The equation to be solved for alpha is $\sin\alpha + p\cos\alpha - q = 0$

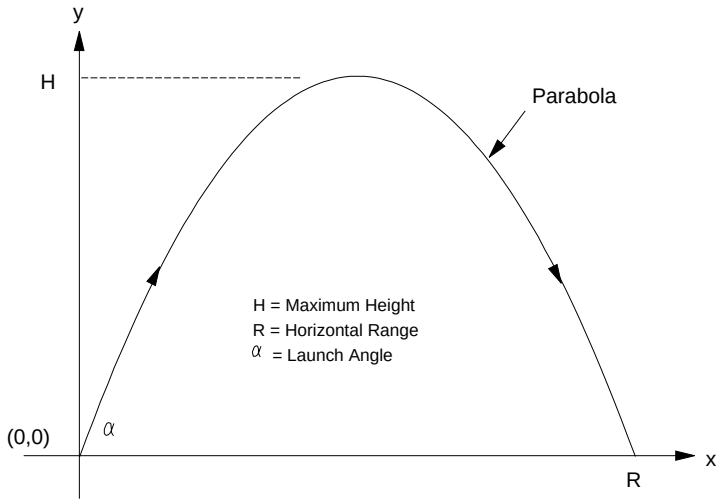


Figure 6.9 Trajectory of a Projectile

```
program Projectile;
{ This program solves the projectile collision problem above }
var
  xdot,ydot,t0,p,q,factor,x,y,t,Epsilon,u,v,a,g,angle : Real;
  NumIterations : Integer;

function f(alpha : Real) : Real;
begin
  f:=sin(alpha) + p*cos(alpha) - q;
end;

function df(alpha : Real) : Real;
begin
  df:=cos(alpha) - p*sin(alpha)
end;

begin { main }
  u:=1400.0; v:=1000.0;
  a:=100000.0; g:=9.8; factor:=3.14159/180.0;
  t0:=30.0;
  p:=(g*t0 - 2.0*v)*t0/2.0/a;
  q:=(v - g*t0)/u;
  epsilon:= 0.0000005;
  angle:=45.0*factor;
  NumIterations:=0;
  writeln('ItNo       angle           f(angle)        f''(angle)');
  writeln;
  while abs(f(angle)) >= epsilon do begin
    NumIterations:=NumIterations + 1;
    angle:=angle - f(angle)/df(angle);
    writeln(NumIterations:4,angle:16:8,f(angle):16:8,df(angle):16:8);
  end;
  t:=t0 + a/u/cos(angle);
  y:=u*(t - t0) - 0.5*g*(t - t0)*(t - t0);
  x:=u*(t - t0)*cos(angle);
  xdot:=u*cos(angle);
  ydot:=u*sin(angle) - g*(t - t0);
  writeln;
  writeln('Time Of Flight Of Target    = ',2.0*v/g:9:4);
  writeln('Angle              = ',angle/factor:9:4,' degrees');
  writeln('Time Of Collision = ',t:9:4,' sec after launch of target');
  writeln('Coordinates Of Collision Pt = (',x:9:4,',',y:9:4,')');
  writeln('Velocity Vector At Collision= (',xdot:9:4,',',y-
dot:9:4,')');
end. { main }
```

```
 ItNo       angle          f(angle)        f'(angle)

   1      0.76076858     -0.00015736      0.90074381
   2      0.76094328     -0.00000001      0.90065573

 Time Of Flight Of Target    =  204.0816
 Angle                       =   43.5989 degrees
 Time Of Collision           =  128.6329 sec after launch of target
object
 Coordinates Of Collision Pt = (100000.0000,90416.6731)
 Velocity Vector At Collision= (1013.8602,  -1.1559)
```

6.4 Historical Note

Niels Henrik Abel (1802-1829)

The son of a Norwegian pastor, Niels Abel, as was the case with most outstanding mathematicians, established himself early as a man of considerable mathematical talent. Unfortunately, his life, like that of Galois, was brilliant but very short. His father died when the young Abel was only 18, and so he was required to support the family. Despite this burden, at the age of 19 he proved that the general quintic (5th degree) equation cannot be solved by rational operations and extraction of roots. This achievement is remarkable when we realize that the greatest mathematicians of the previous 300 years had failed in this endeavour.

Abel did much valuable original work on the theory of elliptic functions, and was lucky enough to find a new journal that was prepared to publish his work. The first volume of the **Journal fur die reine und angewandte Mathematik,** contained five of his papers. Apart from this piece of fortune however, Abel's life seemed to be dogged by bad luck. He sent some work to Cauchy in Paris, but the great Cauchy misplaced it! It was later found and published in 1841, 12 years after Abel's death. Abel searched in vain for an academic position, and died of tuberculosis in 1829. Two days later he was appointed professor at Berlin University, where they had not yet heard of his death.

Groups in which the operation is commutative are called **Abelian**, in honour of Abel's work in this branch of mathematics. This man is honoured by a statue by Vigeland in Oslo Royal Park, and we can only speculate on the further discoveries that he may have made, had his life been more than a mere 26 years.

6.5 Summary

- The solution of equations is a fundamental requirement of modern science, engineering, and business modelling. We usually express the equation to be solved in the standard form f(x)=0.
- Methods for solving equations can be divided into two very broad categories. These are **direct** and **indirect** methods. Direct methods use a **formula** to compute the solution in terms of the problem parameters or coefficients. An example of a direct method is the solution of a quadratic equation by the quadratic formula. Indirect methods use a **sequence of approximations** to eventually arrive at an acceptable approximation to the solution of an equation. Almost all problems require **indirect** methods to be solved.
- We studied two indirect methods in common use - **Newton's method,** and the **bisection method.**
- The bisection method relies on the property of a continuous function which has opposite signs at the ends of a closed interval - it must cross the x-axis at least once in the interior of the interval. In this method, the interval is repeatedly halved until the crossover point lies in a very narrow range.
- Newton's method is used for smooth functions and uses the tangent to the graph of the function at the current estimate to direct us to the next estimate.
- The bisection method **always** converges to a solution, provided we start with an interval on which the function changes sign. The Newton method may fail to converge, and this is likely if the function has areas of very small, or even zero, slope in the neighbourhood of interest. In particular, turning points, and points of horizontal inflections pose problems for this method.
- The bisection method is not used as commonly as Newton and Newton-related methods because of its poor rate of convergence. The convergence of the Newton algorithm is very rapid indeed, except in regions of small derivative.

6.6 Exercises

1. If there is more than one root for the bisection method, which one will the algorithm converge to?

2. Solve the examples of this chapter algebraically and compare the solutions to those obtained on the computer.

*3. Using the bisection method find the root(s) of the function:

$f(x) = x^3 - \frac{1}{2}$, *where* $a = 1.0, \quad b = 2.0$.

Estimate the number of bisections needed to obtain a root correct to 4 decimal places and compare with your result.

*4. Use the bisection method to find (to 3 decimal places) a solution of the function sin x - 0.750 = 0, where a = 0 and b = 1.

*5. Derive an iterative method based on Newton's method for finding the cube root of N. Hint: consider the function $f(x) = x^3 - N$. Apply the method to find $\sqrt[3]{2}$ correct to three decimal places.

6. Use both the Newton and bisection methods to solve f(x)=0 for each of the following. Use the mid-point of the interval given in each case for the starting value. Tabulate the number of iterations taken for each method to solve to 5 decimal places accuracy.

(a) $f(x) = x^3 - 8$ *on* $(0,4)$

(b) $f(x) = (x-1)^2$ *on* $(0,2)$

(c) $f(x) = (x-1)(x-2)(x-3)$ *on* $(0,4)$

(d) $f(x) = e^x - 2x$ *on* $(0,8)$

*7. Find π correct to four decimal places, by solving f(x)=sin(x)=0 on the interval [3,4]. Use both the bisection method and Newton's method.

*8. Use Newton's method to find a real root of the polynomial $P(x) = x^3 - x - 1$Start at x = 1.5.

*9. Improve the Newton algorithm and program so that if convergence fails after MaxIt iterations then the loop will terminated with an appropriate error message. MaxIt may either be a Pascal **const** or input at run-time.

*10. Further improve the Newton program so that if $|f'(x)| < 0.000001$ then no division will be attempted but the loop will terminate with the message "function too flat". This precaution will avoid any potential overflow or division by zero problems. Any decent implementation of the Newton algorithm should contain provision for this contingency.

11. Show that the equation $e^x = x - 2$ given earlier in this chapter may be written in each of the three forms shown.

12. Solve the three equations of the previous question using both bisection and Newton and discuss the results.

*13. Apply Newton's method to solve the following equations using the starting point x_0 given:

(a) $f(x) = (x-1)(x+1)^5 = 0, \quad x_0 = 0.$

(b) $\sin x = x^3 + 1, \quad x_0 = -1.$

(c) $x = \log(x+3), \quad x_0 = 2.$

(d) $x^3 - x - 1 = 0, \quad x_0 = 1.5$

(e) $\sin x = x - 2, \quad x_0 = 2.5.$

7 STATISTICS & PROBABILITY

7.1 Arithmetic, Geometric & Harmonic Means

This chapter is devoted to a few short programs which illustrate simple algorithms to compute commonly used statistics: arithmetic mean, standard deviation, geometric mean, harmonic mean, range, and mode. Monte-Carlo methods for estimating areas are also examined. Finally, a program for the compilation of frequency distributions, and a random number generator algorithm are also presented. The topics of graphing data points on a scatter diagram, and linear regression (or "least squares fit") are considered in later chapters.

First, we need a few definitions:

1. Arithmetic Mean: $\mu = \dfrac{x_1 + x_2 + x_3 + \ldots + x_n}{n} = \dfrac{\sum_{i=1}^{n} x_i}{n}$

2. Geometric Mean: $g = \sqrt[n]{x_1 x_2 x_3 \ldots x_n} = \sqrt[n]{\prod x_i}$

3. Harmonic Mean: $h = \dfrac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \frac{1}{x_3} + \ldots + \frac{1}{x_n}} = \dfrac{n}{\sum \frac{1}{x_i}}$

Although, by far the most commonly-used mean is the **arithmetic mean** (or "average"), each of these means is useful in different circumstances.

Example Of Use Of Arithmetic Mean

A class of 30 students received the following % scores in a French test:

```
82, 45, 91, 55, 60, 63, 63, 72, 29, 33, 66, 51, 20, 96, 46,
82, 70, 39, 50, 40, 42, 86, 12, 88, 60, 34, 45, 95, 32, 75.
```

We may compute the arithmetic mean of these values. The result is usually termed **the class average,** and it is considered desirable for a student's score not to be below this somewhat arbitrary figure. However, this mean is generally a useful indicator of general class performance. For the arithmetic mean to be more useful, it should normally be accompanied by the **standard deviation,** and some other indicator of data spread ("dispersion"), such as the **range.**

Example Of Use Of Geometric Mean

The geometric mean is found by multiplying the n(say) items of the distribution together, and then finding the n th root of the product.

The advantage this has over the arithmetic mean is that it reduces the effect of a single large item on the average.

The arithmetic mean of 1, 2, 3, 4, 4, 4, 5, 100 is 15.5 - and this does not seem to be representative of the group in any reasonable way.

The geometric mean is $\sqrt[8]{1 \times 2 \times 3 \times 4 \times 4 \times 4 \times 5 \times 100} = 4.6$

This seems to be representative of the group, if we regard 100 as a freak.

Example Of Use Of Harmonic Mean

Suppose we travel from A to B at an average speed of 40 km/h, and then return from B to A at an average speed of 60 km/h. What is our average speed for the round trip? Before starting on this problem, we need to know what we mean by the term **average speed.** Our definition is:

$$\text{Average Speed} = \frac{\text{Total Distance Travelled}}{\text{Total Time Taken}}$$

The point to note about this example is that the answer is **not** the arithmetic mean of the two average speeds, i.e. the answer is **not** 50 km/h. To find the average speed according to our definition above, it would seem that we need to know the distance between A and B. Once we know this, we can calculate the time taken , since we also know the average speeds. It turns out that the answer is independent of the distance between A and B. Let the distance between A and B be x kilometres.

Therefore, according to the definition of average speed, the time taken to travel from A to B is given by:

$$t_{AB} = \frac{x}{40} \text{ hours.}$$

Similarly, the time taken to travel from B to A is given by:

$$t_{BA} = \frac{x}{60} \text{ hours.}$$

Total time for the round trip is therefore:

$$t_{AB} + t_{BA} = \left(\frac{x}{40} + \frac{x}{60} \right) \text{ hours.}$$

The average speed for the round trip, according to the definition is thus:

$$\frac{2x}{t_{AB} + t_{BA}} = \frac{2x}{\frac{x}{40} + \frac{x}{60}} = \frac{2}{\frac{1}{40} + \frac{1}{60}} = 48 \text{ km/h.}$$

This is precisely the **harmonic mean** of these two average speeds. You should verify these steps.

Program To Calculate Means

It is a fairly simple matter to write a Pascal program to compute all three of the means in one loop. Students should note carefully that each variable used for accumulating these quantities must be appropriately initialized before the loop is commenced.

```
program Means;
{ This program calculates the arithmetic, geometric and harmonic
means of a set of numbers. }
var
  i, n           : Integer;
  x              : Real;
  Sumx           : Real;
  Product        : Real;
  ReciprocalSum  : Real;
  ArithmeticMean : Real;
  GeometricMean  : Real;
  HarmonicMean   : Real;
begin { main }
  Sumx:=0.0; Product:=1.0; ReciprocalSum:=0.0;
  write('How many values to be used ? '); readln(n);
  for i:=1 to n do begin
    write('x = '); readln(x);
    Sumx         :=Sumx + x;
    Product      :=Product*x;
    ReciprocalSum:=ReciprocalSum + 1/x;
  end;
  ArithmeticMean:=Sumx/n;
  GeometricMean :=exp(ln(Product)/n);
  HarmonicMean  :=n/ReciprocalSum;
  writeln('Arithmetic Mean = ', ArithmeticMean:12:8);
  writeln('Geometric  Mean = ', GeometricMean :12:8);
  writeln('Harmonic   Mean = ', HarmonicMean  :12:8);
end. { main }
```

Output

```
How many values to be used ? 5
x = 1
x = 2
x = 3
x = 4
x = 5

Arithmetic Mean =   3.00000000
Geometric  Mean =   2.60517108
Harmonic   Mean =   2.18978102
```

7.2 Calculation of Maximum, Minimum, and Range

Let's now consider how we might extend our program to determine the largest and smallest of the n input values of x. Our strategy is to maintain two variables, which we designate LargestSoFar and SmallestSoFar respectively. Whenever a value is entered which is larger than LargestSoFar, we update the value of this variable; similar considerations apply to SmallestSoFar. At the finish, the range is simply the difference of these two variables.

```
program Range;
{ This program determines the range and maximum and minimum
  values as well as the various means of a set of numbers }
var
  i, n           : Integer;
  x              : Real;
  Sumx           : Real;
  Product        : Real;
  ReciprocalSum  : Real;
  ArithmeticMean : Real;
  GeometricMean  : Real;
  HarmonicMean   : Real;
  LargestSoFar   : Real;
  SmallestSoFar  : Real;
begin
  LargestSoFar :=-1.0E-20;
  SmallestSoFar:= 1.0E20;
  Sumx:=0.0; Product:=1.0; ReciprocalSum:=0.0;
  write('How many values to be used ? '); readln(n);
  for i:=1 to n do begin
    write('x = '); readln(x);
    if x > LargestSoFar  then LargestSoFar :=x;
    if x < SmallestSoFar then SmallestSoFar:=x;
    Sumx         :=Sumx + x;
    Product      :=Product*x;
    ReciprocalSum:=ReciprocalSum + 1/x;
  end;
  ArithmeticMean:=Sumx/n;
  GeometricMean :=exp(ln(Product)/n);
  HarmonicMean  :=n/ReciprocalSum;
  writeln('Arithmetic Mean = ', ArithmeticMean:12:8);
  writeln('Geometric  Mean = ', GeometricMean :12:8);
  writeln('Harmonic   Mean = ', HarmonicMean  :12:8);
  writeln('Minimum Value   = ', SmallestSoFar :12:8);
  writeln('Maximum Value   = ', LargestSoFar  :12:8);
  writeln('Range           = ', LargestSoFar-SmallestSoFar:12:8);
end.
```

Output

```
How many values to be used ? 5
x = 1
x = 2
x = 3
x = 4
x = 5
Arithmetic Mean =   3.00000000
Geometric  Mean =   2.60517108
Harmonic   Mean =   2.18978102
Minimum Value   =   1.00000000
Maximum Value   =   5.00000000
Range           =   4.00000000
```

Comments On The Range Program

1. We must be careful to initialise the values of SmallestSoFar and LargestSoFar before entering the forloop, otherwise, the first comparisons will be meaningless, and completely unpredictable.
2. Pascal has no power operator, and we use the Exp and Ln functions to achieve the desired result for the geometric mean. Students should understand this point very thoroughly.

7.3 Standard Deviation

To extend our program further and calculate the standard deviation is also a straightforward matter. Firstly, we define the variance, v, as "the mean of the squares minus the square of the mean" (both these means are in the arithmetic sense). Thus we have:

$$V = E(x^2) - (E(x))^2$$

This is the same as:

$$V = \frac{\sum x^2}{n} - \left(\frac{\sum x}{n}\right)^2$$

where the function E(t) is just the arithmetic mean of t. The standard deviation of x is simply the square root of the variance, V. An equivalent alternative formula for standard deviation is:

$$\sigma = \sqrt{\frac{\sum (x_i - \overline{x})^2}{n}}$$

but this is not used in the program for means and standard deviation given below.

```
program StdDev;
{ This program calculates the various means, maximum and minimum
  values, range and the standard deviation of a set of numbers }
var
  i, n               : Integer;
  x                  : Real;
  Sumx, Sumxx        : Real;
  Product            : Real;
  ReciprocalSum      : Real;
  ArithmeticMean     : Real;
  GeometricMean      : Real;
  HarmonicMean       : Real;
  LargestSoFar       : Real;
  SmallestSoFar      : Real;
  Variance           : Real;
  StandardDeviation  : Real;
begin
  LargestSoFar :=-1.0E-20;
  SmallestSoFar:= 1.0E20;
  Sumx:=0.0; Sumxx:=0.0; Product:=1.0; ReciprocalSum:=0.0;
  write('How many values to be used ? '); readln(n);
  for i:=1 to n do begin
    write('x = '); readln(x);
    if x > LargestSoFar  then LargestSoFar :=x;
    if x < SmallestSoFar then SmallestSoFar:=x;
    Sumx          :=Sumx + x;
    Sumxx         :=Sumxx + Sqr(x);
    Product       :=Product*x;
    ReciprocalSum:=ReciprocalSum + 1/x;
  end;
  ArithmeticMean    :=Sumx/n;
  GeometricMean     :=Exp(Ln(Product)/n);
  HarmonicMean      :=n/ReciprocalSum;
  Variance          :=(Sumxx/n) - Sqr(ArithmeticMean);
  StandardDeviation:=Sqrt(Variance);
  writeln('Arithmetic Mean    = ', ArithmeticMean:12:8);
  writeln('Geometric  Mean    = ', GeometricMean :12:8);
  writeln('Harmonic   Mean    = ', HarmonicMean  :12:8);
  writeln('Minimum Value      = ', SmallestSoFar :12:8);
  writeln('Maximum Value      = ', LargestSoFar  :12:8);
  writeln('Range              = ', LargestSoFar-SmallestSoFar:12:8);
  writeln('Standard Deviation = ', StandardDeviation:12:8);
end.
```

```
Output

How many values to be used ? 5
x = 1
x = 2
x = 3
x = 4
x = 5
Arithmetic Mean     =   3.00000000
Geometric  Mean     =   2.60517108
Harmonic   Mean     =   2.18978102
Minimum Value       =   1.00000000
Maximum Value       =   5.00000000
Range               =   4.00000000
Standard Deviation  =   1.41421356
```

7.4 Monte-Carlo Methods

An interesting variation of the basic numerical integration method (i.e. adding up rectangles or the integrals of other simple functions) is the **Monte-Carlo** approach.

For finding areas under curves, the basic idea behind the class of strategies known collectively as Monte-Carlo methods is to first surround the domain whose area is being sought by a square (or rectangle) of suitable size. Then, random coordinate pairs are generated for points inside the square. For each point, it is then determined whether the point lies inside or outside the area of interest. If so then we increment a counter, otherwise do nothing. This process repeats until a large number of coordinate pairs have been generated. The ratio of the points "landing" inside the area to the total number of generated points is then equated to the ratio of corresponding areas, i.e.

$$\frac{\text{No of points in area required}}{\text{Total No of points}} = \frac{\text{Required Area}}{\text{Area of Square}}$$

Since the only unknown in this equation is the value of the area sought, the equation can easily be solved for the required area. Such techniques can be useful in cases where the function is difficult to describe by means of a formula, or even if no formula is known (perhaps only a set of points has been given to describe the function). However, this entire class of methods relies for its success on the following assumptions:

1. The coordinates generated must be random; i.e. each point in the square (or other regular shape whose area is known) must have an approximately equal chance of being generated as any other point.
2. It must be possible to determine whether any given point is inside or outside the area of interest.
3. A very large number of points is required in order to obtain any kind of reasonable accuracy.

As long as the above three constraints are kept in mind, the Monte-Carlo techniques can prove very useful in situations where more conventional methods are awkward to apply.

Example 1

As an example of the Monte-Carlo approach, we present a Pascal program to estimate the area of a unit circle (and thus the value of π) by enclosing a quadrant of it in a square of side 1 unit. Students should note the application of the distance formula in this example. Approximations to π are displayed after each 1000 iterations. A simple illustration is provided in Figure 7.1, which pretends that 22 values are generated which lie in Area A (inside the circle quadrant), and 6 values in area B (the part of the square outside the quadrant). Accordingly, we end up with an estimate of π equal to 22/7. This is a pure fudge for illustrative purposes only, and should not be taken as indicative of the accuracy of the method.

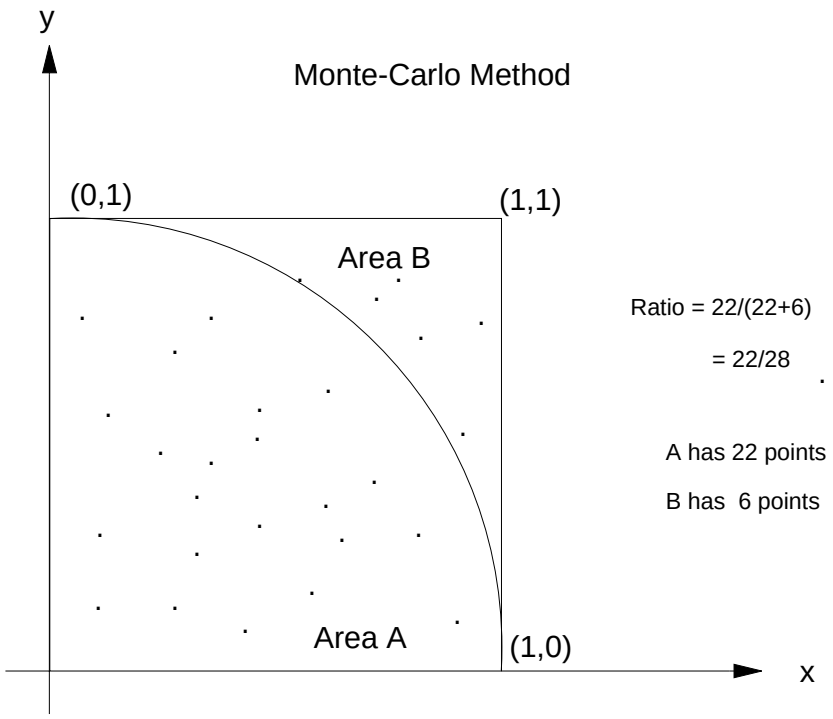


Figure 7.1 Monte-Carlo Method

```
program MonteCarlo;
{This program estimates the value of pi using the Monte-Carlo method}
var
  x, y, pi   : Real;
  i, n, hits : Integer;
begin
  n:=0;
  hits:=0;
  writeln(' No Pts     Pi Approx'); writeln;
  repeat
    for i:=1 to 1000 do begin
      x:=Random;
      y:=Random;
      if x*x + y*y < 1.0 then hits:=hits + 1;
    end;
    n:=n + 1000;
    pi:=4.0*hits/n;
    writeln(n:6, pi:16:8)
  until n=10000
end.
```

Output

```
No Pts     Pi Approx

 1000      3.22800000
 2000      3.21000000
 3000      3.18666667
 4000      3.20400000
 5000      3.18960000
 6000      3.16733333
 7000      3.17257143
 8000      3.17000000
 9000      3.17777778
10000      3.17080000
```

Remarks

1. Notice that the appearance of "convergence" to a value 3.126.... may well be deceptive (the true value of pi is 3.14159....). For one thing, the random number generator in Turbo Pascal may be biased (we consider a simple program later to check this). In any case, we should be cautious about inferring limiting behaviour of a sequence from just a few terms. We should also observe that the Monte-Carlo method does not give rise to spectacular accuracy. This becomes especially obvious when we compare it to a **power series** approximation which can produce pi to ten-decimal place accuracy with less than ten terms. This and other related matters are discussed in the chapter dealing with power series approximations.
2. It is difficult to decide how many points to take in order to get reasonable accuracy with this method. In fact, the method is really quite poor and should only be used when other methods are very awkward to apply. Statistical techniques to determine the minimum number of points to yield a certain confidence interval for the answer do exist, but consideration of these is quite beyond the scope of this book.

Example 2

Use the Monte-Carlo technique to estimate the value of ln 2. The integral of f(x) = 1/x on the interval [1,2] is a suitable integral which yields this value, i.e.

$$\int_1^2 \frac{1}{x}\,dx$$

= the area under the curve y = 1/x between x = 1 and x = 2
= ln 2.

A diagram to illustrate the area we are trying to find is given in Figure 7.2.

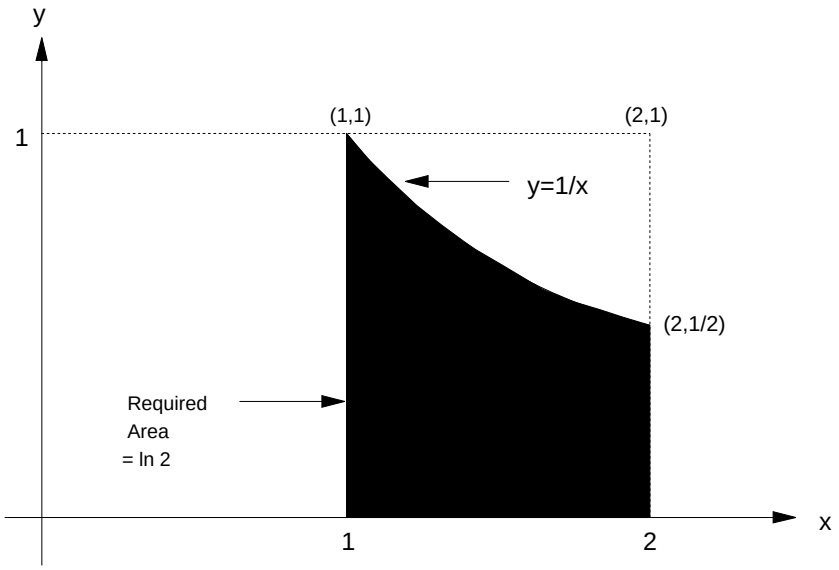


Figure 7.2 Area Under Curve y=1/x Between x=1 and x=2

```
program Ln2;
{This program obtains an approximation to ln(2) by
the Monte-Carlo method}
var
  x, y, ln2  : Real;
  i, n, hits : Integer;
begin
  n:=0;
  hits:=0;
  writeln(' No Pts     Ln2 Approx'); writeln;
  repeat
    for i:=1 to 1000 do begin
      x:=Random;
      y:=Random;
      x:=x+1;
      if y < 1/x then hits:=hits + 1;
    end;
    n:=n + 1000;
```

```
    ln2:=hits/n;
    writeln(n:6, ln2:16:8)
  until n=10000;
  writeln; writeln('Ln(2) correct to 7 d.p. = ',ln(2.0):8:7);
end.
```

```
Output

No Pts     Ln2 Approx

 1000      0.70100000
 2000      0.69750000
 3000      0.69500000
 4000      0.69025000
 5000      0.68740000
 6000      0.68766667
 7000      0.69042857
 8000      0.69112500
 9000      0.69155556
10000      0.68960000

Ln(2) correct to 7 d.p. = 0.6931472
```

Example 3

Write a program to test the randomness of the random number generator in your Pascal compiler. Simply generate a large number of random numbers between 0 and 1; compute the mean, and standard deviation, and examine them for bias. A solution is given below.

```
program RanTest;
{ This program tests the randomness of Turbo Random function }
var
  x, sum, sumsq, mean, v : Real;
  i, n                   : Integer;
begin
  n:=20000;
  sum:=0.0; sumsq:=0.0;
  for i:=1 to n do begin
    x:=Random;
    sum:=sum + x;
    sumsq:=sumsq + x*x;
  end;
  mean:=sum/n;
  v:=sumsq/n - sqr(mean);
  writeln('Mean     = ',mean:16:12);
  writeln('Variance = ',v   :16:12)
end.
```

```
Output

Mean     =   0.498494433850
Variance =   0.083712370454
```

Remark

Since we expect a rectangular distribution on [0,1], the mean should be very close to 0.5, and the variance is expected to be close to 1/12 or 0.0833. The results appear to be reasonable.

Writing Our Own Random Number Generator

In many applications of mathematics in science and engineering, it is useful to be able to generate sequences of **random numbers**. When we study mathematical statistics and probability theory, we meet the concept of **randomness.** When we say that a number, x, is randomly chosen from a set, S, we mean that any other member of S would have had an equal chance of being chosen.

For example, suppose S = {1,2,3,4,5,6}, and that a random value, x, is to be chosen from S. To have some degree of confidence that the value chosen **is** actually random (or approximately random), we must use for the selection process an algorithm which has this essential property of randomness. In the case of the example above, a reasonable approach is to roll a fair, balanced die and read the topmost face to obtain a member of S. Students of physics may argue that the final outcome is determined by the initial configuration and velocities and the laws of motion, and therefore the outcome is completely predetermined. This is essentially true, however we must be skilled enough and also have sufficient data to analyse in order to predict the final outcome with any degree of certainty.

Computer Generation Of Random Numbers

We learned in Chapter 1 that a digital computer is a completely deterministic device, that is, its output is 100% predictable if we know the inputs and the processing algorithms. So how can a finite, deterministic machine be used to generate random numbers? The answer is of course, that it can't! We have to be content with **pseudo-random** numbers. We rely on the fact that the user of the random numbers does **not** know the algorithm nor all of the data to be used in generating the numbers. For example, a large computer system may use the current time-of-day, the number of characters in the last file to be printed, and the last three digits of the most recent user to log on to the system as inputs to a complicated formula to generate our required random number. In general, it depends very much on the application to which the random numbers will be put, as to the acceptability or otherwise of a given random number generating algorithm.

We have just looked at the Monte-Carlo methods for estimating integrals, and we used the **intrinsic** (built-in) random number function of Turbo Pascal. Now we turn to consider a method of generating "pseudo-random" numbers from a very simple formula of our own. The method we consider is just one member of a whole family of algorithms known as **linear congruential sequence generators.** This fearsome-sounding name describes a method which is actually very simple, and we usually abbreviate the method to **LCG. LCG's** are a very well-known and simple-to-compute class of algorithms. Here we go with the definition and a simple example.

7.5 Random Number Generators

For two positive integers a and b, the value of **a mod b** is the remainder when a is divided by b. For example:

14 mod 3 = 2;
23 mod 5 = 3;
99 mod 10 = 9.

To generate a random sequence of integers between 0 and a natural number m, we use this idea to define a sequence of natural numbers as follows:

$$X_{n+1} = (aX_n + c) \bmod m, \quad for \quad n \geq 0$$

where m *is the modulus*; $m > 0$.

a *is the multiplier*; $0 \leq a < m$.

c *is the increment*; $0 \leq c < m$.

X_0 *is the starting value*; $0 \leq X_0 < m$.

All of these values are cardinal numbers (non-negative integers).

Example 1

Let's choose m = 7, a = 2, c = 3, and $X_0 = 1$. This leads to the sequence:

$$X_{n+1} = (2X_n + 3) \bmod 7; \qquad for \quad n = 0, 1, 2, \ldots$$

The sequence is thus: 1,5,6,1,5,6, ... etc. Not a very interesting sequence, and hardly "random". Our problem is that we have chosen poor values for the parameters a, m, and c. For a start, since the generated value is always a remainder after division by m, we can only have m values at most before the sequence starts to repeat. In our example, m=7, and so our sequence of values would repeat with a period of at most 7. In fact, it is clear that with the values of a and c that we have chosen, it repeats with a period of 3 terms (or cycle of length 3). The Pascal program below generates all numbers in "random" order between 0 and 102. The values chosen for m, a, c ensure that no values in this range are repeated or omitted. The reason can be found by looking at the theorem following the program.

Pascal Program To Calculate Linear Congruence

```
program LCG;
{ This is a Linear Congruential Generator program,
  i.e. it generates pseudo-random numbers }
var
  a, c, m, x, x0 : Integer;
  TermCount      : Integer;
begin
  x0:=1; m:=103; x:=x0;
  write('a = '); readln(a);
  write('c = '); readln(c);
  writeln;
  TermCount:=1;
  repeat              {write 10 numbers per line}
    write(x:4);
    if TermCount mod 10 = 0 then writeln;
    x:=(a*x+c) mod m;
    Termcount:=TermCount + 1
  until x = x0;
  writeln;
  writeln; writeln('No of Terms = ',TermCount - 1:4)
end.
```

```
Output
a = 53
c = 41

   1  94  79   5 100  88  70  43  54  19
  18  68  40 101  38  98  85  14  62  31
  36  95  29  33  39  48  10  56  22  74
  49  63  84  64  34  92  76  52  16  65
  87  17  15  12  59  78  55  72  46   7
   0  41  51  66  37  45  57  75 102  91
  23  24  77   2  44   4  47  60  28  83
  11   6  50  13   9   3  97  32  89  20
  71  96  82  61  81   8  53  69  93  26
  80  58  25  27  30  86  67  90  73  99
  35  42

No of Terms =  102
```

Remarks

From this example and the exercise we see that not just any old LCG will do. Some are in fact very poor at producing random numbers. What we need is a set of sufficient conditions on a, c, and m which will ensure that the sequence we generate has no repetitions until m terms have been generated. This is of course equivalent to requiring that the sequence has the maximum possible period of m.

Theorem (See for example, Knuth, Vol 2, p 16)

The linear congruential sequence defined by m, a, c and x_0 has period of length m if and only if:

(i) c is relatively prime to m;
(ii) b = a-1 is a multiple of p, for every p prime dividing m
(iii) b is a multiple of 4, if m is a multiple of 4.

Example 2

$X_1 = 2, \quad \text{and} \quad X_{n+1} = (aX_n + c) \bmod m, \quad for 0 \le n < 103$

$a = 53, \qquad c = 41, \qquad m = 103.$

With these values for the parameters a, c, m; it is guaranteed (see reference below) to generate every integer from 0 to m - 1 inclusive exactly once, in "random" order. Numbers are generated in the order shown in the table above (read row-wise, i.e. row 1 left to right, then row 2 left to right, etc.).

No prizes for finding two values the same, since there aren't any values the same! How could you use a computer to verify that no duplicates exist in this table? Discuss this matter with your teacher.

Reference

Knuth, D.E., "The Art Of Computer Programming", Vol 2, 2Ed., Seminumerical Algorithms, pp 9, 10, 16.

7.6 Frequency Distributions

Having already studied programs to compute simple statistics such as the various means, standard deviation etc., it should be a fairly straightforward matter to design a program to compile a frequency distribution of integer scores. Since we know how to generate random numbers, let's consider how we would write a program which generates 1000 integers between 0 and 9 inclusive, then compiles a frequency distribution of them. We have already foreshadowed the problem of compiling a frequency distribution and ideas central to its solution in a previous chapter.

To reiterate, our strategy in building the frequency distribution is simply to use an array, **Freq,** in which the **i**th element will ultimately contain the frequency count of value i. For example, Freq[4] will always contain the numbers of 4's encountered so far in the sample. We must of course remember to set each of the frequency counters to zero before we start counting.

```
program FreqDist;
{ This program produces a frequency distribution of a set
  integers between 0 and 9 generated randomly }
var
  x       : Real;
  i, n, d : Integer;
  Freq    : array[0..9] of Integer;
begin
  n:=1000;
  for i:=0 to 9 do Freq[i]:=0;
  for i:=1 to n do begin
    x:=Random;
    d:=trunc(10.0*x);
    Freq[d]:=Freq[d]+1;
  end;
  writeln('  Value     Frequency'); writeln;
  for i:=0 to 9 do writeln(i:5, Freq[i]:12)
end.
```

Output

```
  Value     Frequency

    0           87
    1           98
    2          102
    3          102
    4           81
    5          135
    6          110
    7           95
    8          101
    9           89
```

7.7 Historical Note

Pierre de Fermat (1601-1665)

Fermat was not a professional mathematician, but rather a lawyer serving in the parliament of Toulouse. Nevertheless, he has been referred to as the outstanding example of an amateur mathematician. His mathematical ability and achievements were of the highest order, and we use the adjective "amateur" here in the sense that he did not earn his living from his mathematical work. It was in fact his recreation.

This Frenchman co-discovered coordinate geometry along with his countryman René Descartes, and co-discovered probability theory with Blaise Pascal. He founded the branch of mathematics known as number theory, also worked in optics (Fermat's principle of least time), and in the 1630's developed methods for finding maxima, minima, tangents and areas under curves. Fermat was a contemporary of Newton and Leibniz.

He is most remembered for his extensive work in Number Theory. One of his results is the Fermat "little" Theorem, which states that $a^p - a$ is exactly divisible by p for all prime p and all integers a.

Perhaps his most famous theorem is his "last" theorem; really only a conjecture, since it has never been proved (nor disproved), despite the efforts of the most able mathematicians for centuries. Fermat claimed he had a proof for the conjecture when writing comments in a copy of **Diophantus' Arithmetica,** but "the margin is too narrow to contain it". Fermat's Last Theorem is very simple to state:

For every integer $n > 2$, *there exist no integers* x, y, z such that $x^n + y^n = z^n$ but evidently very difficult to prove (or disprove).

7.8 Summary

- Some commonly-computed statistics are - arithmetic, geometric, and harmonic means, standard deviation, variance, range and frequency distribution. All of these can be calculated in a single computer program.
- The advantage of the geometric over the arithmetic mean is that the former reduces the effect of a single large item on the average.
- The harmonic mean has a useful application when the average speed for a round-trip between say A and B is needed, given that the average speeds from A to B and B to A are not the same.
- The standard deviation is a useful indicator of the "spread" of the data.
- Monte-Carlo methods essentially are used to calculate unknown areas by enclosing domain whose area is being sought in a square and generating random points within the square. The area can be estimated using the formula:

$$\frac{\text{No of points in area required}}{\text{Total No of points}} = \frac{\text{Required Area}}{\text{Area of Square}}$$

- The success of the Monte-Carlo methods relies on the points generated being truly random. Accuracy can pose a problem in some applications.
- The Monte-Carlo method can be used to estimate the value of, for example, pi, log 2, and other mathematical constants.
- Random numbers can be generate using an algorithm from a family of algorithms known as linear congruential generators (LCG's).
- The LCG used to generate a sequence of numbers is given by:
  $X_{n+1} = (aX_n + c) \pmod{m}, \quad for \quad n \geq 0$
  Care needs to be taken in choosing the parameters a, m and c.
- Given a set of data, a frequency distribution is often desirable. The array is a useful data structure to use when compiling these distributions.

7.9 Exercises

*1. Using the programs **Means, Range and StdDev** find the mean, range and standard deviation of the following collection of numbers:

21 45 67 88 34 56 12 56 34 67 23 55 48 19 23
34 33 39 45 19 23 33 47 18 12 12 66 45 45 78

2. The following is a record of the percentage marks obtained by 100 students in an examination:

45 93 35 56 16 50 63 30 86 65 57 39 44 75 25 45 74
93 84 25 77 28 54 50 12 85 55 34 50 57 55 48 78 15
27 79 68 26 66 80 91 62 67 52 50 75 96 36 83 20 45

71 63 51 40 46 61 62 67 57 53 45 51 40 46 31 54 67
66 52 49 54 55 52 56 59 38 52 43 55 51 47 54 56 56
42 53 40 51 58 52 27 56 42 86 50 31 61 33 36

Modify the program **FreqDist** to produce a frequency distribution for the set of marks.

*3. Amend **FreqDist** in order that a frequency distribution based on the classes 0-9,10-19,20-29,....,90-99 is drawn up for the data below of the previous question.

4. (a) Find the geometric mean of 8 and 32.
   (b) Find the geometric mean of 4,6, and 9.
   (c) Find the geometric mean of 1,2,4,8, and 16.
   (d) A grocery item check takes a sample of 5 item and finds that they cost 112,120,104,240, and 116 per cent of what they they cost a year earlier. Find the geometric mean of these percentages.

5. The harmonic mean has limited usefulness, but it is appropriate in special situations. For instance, if a person drives 10 km on the freeway at 60 km/hr and the next 10 km off the freeway at 30 km/hr, he/she will **not** have averaged (60+30)/2 = 45 km/hr. The person will have driven 20 km in a total of 30 minutes, so that the average speed is 40 km/hr.

   (a) Verify that the harmonic mean of 60 and 30 is 40.
   (b) If a merchant spends $12 on novelty items costing 40 cents per dozen and another $12 on novelty items costing 60 cents per dozen, what is his average cost per dozen? Verify that the harmonic mean of 60 and 40 gives the correct answer.
   (c) If a restaurant buys $72 worth of butter at 60 cents per kg, $72 worth at 72 cents per kg and $72 at 90 cents per kg, calculate the average price per kg and verify that it is the harmonic mean of 60,72, and 90.

6. Use the programs supplied to find the arithmetic, geometric, and harmonic means for the French test data given in the beginning of this chapter. Comment on the variation on values with reference to data dispersion (standard deviation, range).

7. Improve the means program to handle the case where one of the x values is zero; it presently will "fall over" if x = 0 is entered for one of the values. What kind of error will occur? If you don't know, run it and see! How will the program behave if 0 is entered for n? Can you improve the program to reject such erroneous values? (A large part of correct program design is to protect the algorithm from unwanted values entered by users who either do not know any better or are just plain careless.)

8. Why is the geometric accumulator variable "product" initially set to 1.0, whereas those for the arithmetic and harmonic means are set to zero?

9. Why are SmallestSoFar and LargestSoFar in the program **Range** initialised to such large values?

10. Determine alternative values of a and c for the LCG example we considered with m=7. Try to find values which give a full maximum period of seven terms before repetition. Does the period depend on the starting value of x? Does the period depend on c? How many such pairs (a,c) give rise to full-length periods, i.e. periods of length 7?
    (Ans: a=c=1 works but is a trivial case).

*11. Generalise the LCG program to do the following:
    (a) Display a heading.
    (b) Display each term alongside its sequence index in two columns.
    (c) Display the period of the sequence at termination.

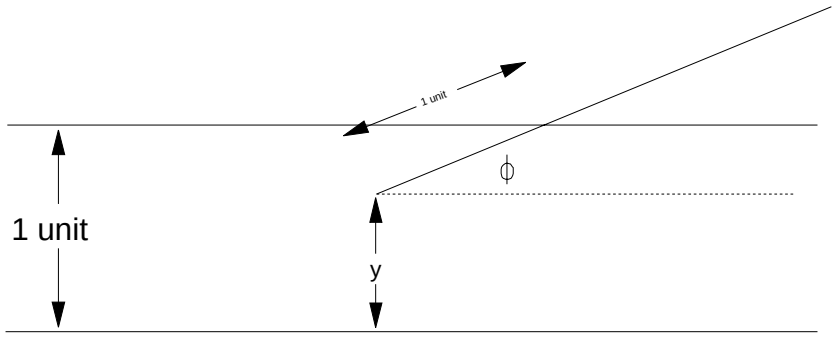
12. Why was the value 103 chosen for m in the LCG example in the text?

*13. Write a Pascal program to generate all numbers between 0 and 18 inclusive in "random" order using the LCG technique.

*14. Write a Pascal program which uses an LCG technique to generate numbers between 0 and 103, but which stops after generating the first 51 of these. The program should accumulate the mean and standard deviation of these 51 values and compare them with the expected values. For example, since we are assuming a rectangular distribution of values, the arithmetic mean should be close to (0 + 103)/2.

*15. Write a Pascal program which uses an LCG technique to generate "random" numbers between 0 and 1, each with four decimal places. Hint: Choose a prime close to 10,000 and use an appropriate LCG to get integers, which are then divided by 10,000. Use the theorem from Knuth to find appropriate values of a and c. Find the mean and standard deviation of the 1000 numbers generated in this way. Are the values reasonable?

16. Let's consider another Monte Carlo type method to calculate the value of π. We can find its value by repeatedly dropping a pin onto a piece of paper with two parallel lines. Suppose the lines are 1 unit apart and the length of the pin is also 1 unit as shown below:



We can simulate the tossing by randomly selecting the values of y between 0 and 1, and ϕ between 0 and π radians.

If the total number of tosses is recorded along with the total number of crossings, then the ratio can be used to find π.

The pin can of course land completely outside or completely inside the parallel lines and either of these would register as a non-crossing.

After studying the program **MonteCarlo** write one to estimate π using this method.

How many throws are needed to give 2,3,4,5 decimal place accuracy?

17. Show that the geometric mean of a set of positive numbers is always less than or equal to the arithmetic mean. When is equality true? Show further that the harmonic mean is always less or equal to the geometric mean.

8 FURTHER APPLICATIONS OF ARRAYS

8.1 Review Of Arrays

In this chapter, we use the **array** data structure that was introduced in Chapter 4 to illustrate some very fundamental algorithms for the processing of **polynomials.** Since the array is the most fundamental **data structure** in computer programming and is very important for numerical work on the computer, it will now be worthwhile to briefly revise and summarise the essential features of arrays. For our purposes, we may think of an array as corresponding to a mathematical **vector** (singly-subscripted array), or a **matrix** (doubly-subscripted array). In many areas of mathematics we need to do calculations on such collections of data. For example, we may be given a list of numbers and require to sort them into numerical order. In problems such as these, it is often required that all the values be in random-access memory simultaneously, and stored in such a way as to allow rapid access to arbitrary elements of the list, i.e. array or vector. For the sorting problem, we usually need to have the values in memory simultaneously so that we may compare values and shuffle them around if necessary. (Sorting is a very extensively-studied problem in computing, and many different algorithms are known for the sorting of arrays of data. We consider one simple sorting method in this chapter.)

An array is an ordered structure, consisting of elements of the same type, and whose contents can be **randomly accessed.** This means that each element can be stored or retrieved in the same time and in the same manner as any other element. The actual data items which comprise the array are called **elements** or **components.** Access to an arbitrary element is obtained by specifying its **index,** which is simply the sequential position of the element within the array. This corresponds to giving the **address** of the desired item in the computer's memory, although the programmer need know nothing about actual storage locations in order to access an array element. The memory of a typical modern computer can be regarded as one large **array of bytes;** the index being the addresses 0..65535 (for example).

Any collection of related data items of like type may be organised into an **array.** As stated above, in mathematical work, we usually refer to this as a vector, or perhaps a matrix. In business usage, we find the terminology **list,** or **table,** respectively. We shall not use the term **list** any further here as it has another technical meaning in computer science.

Let us suppose that we have a collection of ten numbers, which we call $x_1, x_2, \ldots, x_{10.}$ In Pascal, and other languages, we resort to what is essentially a **functional notation** for array subscripts (also called indices); Pascal using square brackets in order that the compiler may easily distinguish arrays from functions and procedures. Some earlier languages such as **FORTRAN** and older versions of **BASIC** use parentheses for functions, arrays, and in the case of **FORTRAN,** subroutines (procedures).

In Pascal and some modern versions of BASIC, we would refer to our x values as x[1], x[2], ..., x[10]. It is important to realise that we have a single **collective** name, x, for the entire ordered collection of values, and may refer to any individual member of the array by specifying its **index** (or subscript). Without the ability to group data items together into a composite **data structure** such as an array, we would need to invent a different variable name for each item of information. This would not only be awkward and inconvenient, but it would also conceptually separate objects which logically belong together. To gain access to an individual value (the particular one is unknown in advance), we would then need know its name, but with an array we just specify the collective name of the array followed by the index number of the element we require enclosed in square brackets. We may also use a variable or even an arithmetic expression for the index we need. Therefore we may refer to, for example, x[i], x[i+1], or even x[2*i+j-1] if an expression such as this is required, provided of course, that the variables i, j have been previously assigned values.

Examples Of Array Usage

(i) In an earlier chapter we considered the array **Grades,** which was declared as:

```
var Grades:array[1..7] of Integer;
```

Such an array could be used to store say the Mathematics grades of each of seven students. Alternatively, it could be used to store the grades for Mathematics, Science, English, History, Geography, French and German for just one student. It is entirely up to us when designing and using our array data structures to supply the necessary interpretation of the contents of each element.

(ii) Suppose that we have 240 students in Grade 12. We wish to compile a frequency distribution of their Mathematics grades. Each grade is a mark from 0 to 100 inclusive. In other words, we want to know how many scored 0%, how many scored 1%, and so on, up to the number who scored 100%. We can store these **frequencies** in an array as follows:

```
var Frequency:array[0..100] of Integer;
```

For example, the contents of the memory location reserved for Frequency[20] would contain the number of students who scored exactly 20 marks out of 100 for Mathematics; the value of Frequency[100] would tell us how many had perfect scores, and so on. A later chapter expands on this particular application of arrays.

(iii) Two vectors may be combined via the **dot product** operation. The dot product has many applications in Physics and Engineering, and is a concise way of expressing the operation of one vector (e.g. **force**) in the direction of another (e.g. **displacement**).

We define the dot product in Figure 8.1 and then look at an example.

DEFINITION OF DOT PRODUCT

The dot product *of two* $n-$*component vectors* $\underline{x}$ and $\underline{y}$ *is defined as follows* :

$$\underline{x}.\underline{y} = \sum_{i=1}^{n} x_i y_i$$

$$= x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

Geometrically, we have $\underline{x}.\underline{y} = xy\cos\theta$, *where*

x = magnitude of $\underline{x}$

y = magnitude of $\underline{y}$

θ = angle between $\underline{x}$ and $\underline{y}$

Figure 8.1 Definition Of Dot Product

Example Of Dot Product

Let $\underline{x} = (1, 0, 3)$, and $\underline{y} = (4, 2, -1)$.

Then $\underline{x}.\underline{y} = 1*4 + 0*2 + (3*(-1)) = 1.$

Also, $x = |\underline{x}| = \sqrt{1^2 + 0^2 + 3^2} = \sqrt{10}$

Also, $y = |\underline{y}| = \sqrt{4^2 + 2^2 + (-1)^2} = \sqrt{21}$

$$\therefore \quad \cos\theta = \frac{\underline{x}.\underline{y}}{|\underline{x}||\underline{y}|} = \frac{1}{\sqrt{10}\sqrt{21}} = \frac{1}{\sqrt{210}}$$

$$So, \quad \theta = \cos^{-1}\left(\frac{1}{\sqrt{210}}\right) \approx 1.5^R$$

Let's now consider a simple Pascal program to compute the scalar product (dot product) of two 3-element vectors.

```
program DotProduct;
{ This program determines the dot product of two vectors }
const
  n = 3;  { the dimension of the vectors i.e number of components }
var
  x,y : array[1..n] of Real;
  i   : Integer;
  Sum : Real;
begin
  Sum:=0.0;
  writeln('Type the corresponding components of both');
  writeln('vectors on the one line - each separated by a space');
  writeln;
```

```
  for i:=1 to n do begin
    readln(x[i], y[i]);
    Sum:=Sum + x[i]*y[i]
  end;
  writeln;
  writeln('The dot product equals ', Sum:12:8)
end.
```

```
Output

1 1
2 2
3 3

The dot product equals  14.00000000
```

8.2 Sorting

Any set on which an ordering relation is defined may be arranged in ascending order. Computers spend a lot of time just doing sorting of one form or another. Items sorted by computer could be just numbers representing, for example, people's ages, bank balances, measurements of temperature, force, pressure, etc., but perhaps more important is the textual or character information such as names and addresses which need to be put in **alphabetical** order by computer. Examples which quickly come to mind are telephone directories, student enrolment lists, and electoral rolls. In this course we study only numerical applications in Pascal, so that treatment of, for example, sorting of character strings is beyond the scope of this book.

We now look at one of the simplest known algorithms for sorting - BubbleSort. The idea is to scan the array, comparing adjacent elements and swapping them if necessary. Once this process is done once, the largest value is guaranteed to be at the bottom of the array, i.e. it will be the last element. The reason for this is the subject of an exercise. A second pass is now done, with the same comparison and possible interchange of adjacent elements, but the last two elements need not be compared. More passes are done until the numbers are all sorted. Students should note carefully that two nested loops are required for the algorithm. The outer loop counts the passes, while the inner loop runs through the array, comparing and exchanging, and in this way implements one single pass. These ideas are illustrated in Figure 8.2. After pass number **i,** the last **i** items will already be sorted, and. We emphasize this fact by showing the values in boldface.

20	-7	19	6	0	12	original data
-7	19	6	0	12	**20**	pass 1
-7	6	0	12	**19**	**20**	pass 2
-7	0	6	**12**	**19**	**20**	pass 3
-7	0	**6**	**12**	**19**	**20**	pass 4
-7	**0**	**6**	**12**	**19**	**20**	pass 5
-7	**0**	**6**	**12**	**19**	**20**	pass 6

Figure 8.2 BubbleSort

Notice that the array is actually correctly sorted after the third pass. Nevertheless, the algorithm we used will blindly carry on to do the full (n-1) passes even when the items may already be sorted. In fact, if given an already sorted array of n elements, it will still do (n-1) passes. Pretty stupid, isn't it! See if you can think of a way to improve the algorithm so it will detect when the array is sorted and then stop. Hint: What if one entire scan (pass) is made without a swap being made?

```
program BubbleSort;
{ This program uses the bubblesort algorithm to sort a given
set of integers }
const
  n = 10; { number of integers to sort }
var
  x             : array[1..n] of integer;
  temp, i, pass : integer;
begin
  writeln('Please enter 10 values; one per line'); writeln;
  for i:=1 to n do begin write('Value ',i:2,' ?  '); readln(x[i]) end;
  for pass:= 1 to n-1 do begin
    for i:=1 to n-1 do begin
      if x[i] > x[i+1] then begin
        temp  := x[i];
        x[i]  := x[i+1];
        x[i+1]:= temp
      end  {if}
    end  {for i}
  end; {for pass}
  writeln; writeln('Sorted Values are :'); writeln;
  for i:=1 to n do write(x[i]:6); writeln
end.
```

Output

```
Please enter 10 values; one per line
Value  1 ?  34
Value  2 ?  65
Value  3 ?  1
Value  4 ?  0
Value  5 ?  -22
Value  6 ?  17
Value  7 ?  99
Value  8 ?  -2
Value  9 ?  0
Value 10 ?  33

Sorted Values are :

   -22    -2     0     0     1    17    33    34    65    99
```

8.3 Evaluation of Polynomials

The algorithms we consider for processing of polynomials are concerned with:

1. Evaluation of a polynomial for a given value of x, and
2. Division of a polynomial by a linear factor.

Nested Evaluation Of Polynomials

Polynomials form the simplest class of functions that are used for analysis in mathematics and science. Much is known about their properties, since they have been studied by mathematicians for many centuries. Our purpose here is to devise an efficient algorithm, to be translated into Pascal, that will allow us to find the value of a given polynomial for some value of the indeterminate, x. Let us consider the polynomial of degree n:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \qquad (1)$$

For example, if n = 3, $a_3 = 2$; $a_2 = -1$; $a_1 = 0$; $a_0 = 5$, then we have:

$$P_3(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \qquad (2)$$

$$= 2x^3 - x^2 + 5$$

We call n the **degree** of the polynomial since it is the highest power of x appearing in the polynomial. The a's are called **coefficients** of the polynomial, and are just real numbers in any given case. The quantity x is given the rather strange-sounding name **indeterminate,** but we just think of it as the variable in which the polynomial is expressed. One reason for this is that a polynomial is really just a collection of coefficients - the powers of x are just there to hang the coefficients on. For example, the polynomial P of equation (2) could have been defined equally well by the formula:

$$P_3(t) = a_3t^3 + a_2t^2 + a_1t + a_0$$

$$= 2t^3 - t^2 + 5 \qquad (3)$$

Notice that the powers of the indeterminate that are allowed to appear in the definition of a polynomial are positive integers only. The exception is the term a_0 with no variable of equations (2) or (3). This is called the **constant** term.

If you have used the BASIC language, you may have used the **exponentiation operator** in BASIC. It allows us to raise a number to a power. Depending on the version of BASIC that you may be using, this operator may be ** or ^. For example,

```
2**3  = 8     or 2^3  = 8;
4**4  = 256   or 4^4  = 256;
10**2 = 100   or 10^2 = 100.
```

A smart operator might come up with the following BASIC program for evaluating the polynomial:

$$Ax^3 + Bx^2 + Cx + D$$

```
10 INPUT A, B, C, D
20 INPUT X
30 LET Y = A*(X**3) + B*(X**2) + C*X + D
40 PRINT Y
```

This program is in fact correct. It is also a very poor program. We need to remember that correctness, while being the single most important attribute of a program, is not the only important one. Many others are also very important. These include:

- Efficiency - how much memory does it need and how fast does it run?
- Generality - does it handle all possible cases?
- Extensibility - can it be easily extended to handle related problems?
- Maintainability - can it be debugged or upgraded with little effort?
- Comprehensibility by human readers - is it easily understood?

Our program here falls down badly on the first three of these properties. Its main problem is that it is not capable of being generalised, and that it uses the time-consuming ** operator completely unnecessarily. In order to rewrite the program we simply observe that the cubic can be expressed equivalently as follows:

((Ax + B)x + C)x + D.

You should verify this right now! This way of writing polynomials is called **nested** form, and was also discussed in Chapter 1. The Pascal language was designed by a mathematician and does not provide an exponentiation operator. This may appear paradoxical to some. Why the omission? We need to remember that for integer powers, we may use a more efficient iterative multiplication algorithm to compute powers, and for non-integer powers, the answer is defined in terms of the exponential and logarithm functions anyway. As for now, we simply use nested multiplication to evaluate our polynomial. Even in BASIC, it is much better to write:

```
10 INPUT A, B, C, D
20 INPUT X
30 LET Y = ((A*X + B)*X + C)*X + D
40 PRINT Y
```

Notice that the only operations used to evaluate this cubic now are addition and multiplication. Let's now look at an algorithm based on this idea, which will allow us to evaluate a polynomial of degree n. We shall use the form defined by equation (1) and code the algorithm in Pascal.

This algorithm is elegance itself, and infinitely superior to our first, primitive attempt in BASIC using the cubic as an example and the ** operator. As a general rule, in languages which have it, do not use the ** operator anyway. You can almost always do it better another way.

```
program NestPoly;
{ This program evaluates a given polynomial in nested form }
var
  i, n : Integer;
  p, x : Real;
  a    : array[0..9] of Real;
begin
  write('Degree Of Polynomial ? '); readln(n);
  for i:=0 to n do begin
    write('a[',i:1,'] = ? ');
    readln(a[i]);
  end;
  writeln; write('Value Of x = ? '); readln(x);
  p:=a[n];
  i:=n;
  repeat
    i:=i - 1;
    p:=p*x + a[i]
  until i=0;
  writeln('Value Of Polynomial = ',p:12:6)
end.
```

Output

```
Degree Of Polynomial ? 2
a[0] = ? 2
a[1] = ? -4
a[2] = ? 2
Value Of x = ? 5
Value Of Polynomial =    32.000000
```

8.4 Division Of A Polynomial By A Linear Factor

Suppose that we wish to divide the polynomial

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

by the linear polynomial (x-k). The quotient will be a polynomial of degree n-1, and the remainder will be a constant. In general, if we have:

$$P_n(x) = Q_{n-1}(x).(x-k) + r,$$

then Q is called the **quotient polynomial**, (x-k) is the **divisor polynomial,** and r is the **remainder polynomial.** In this case, r is a constant polynomial. Let us now write

$$Q_{n-1}(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \ldots + b_1 x + b_0 \qquad (4)$$

Example

Let $P_3(x) = 2x^3 + 5x^2 - 7x + 9,$ and

Let $k = 2$

Then we have $P_3(x) = 2x^3 + 5x^2 - 7x + 9 = Q_2(x).(x-2) + r.$

We can get Q and r by long division:

$$Q_2(x) = 2x^2 + 9x + 11 \quad \text{and} \quad r = 31.$$

Let's now consider the general case of relating the coefficients of Q to those of P. If we consider the coefficient of x to the power i on both sides of equation (4), we get:

$$a_i = b_{i-1} - kb_i, \quad for \quad i = 1, 2, \ldots, n-1$$

and

$$a_0 = -kb_0 + r, \quad for \quad i = 0 \qquad (5)$$

This means that:

$$a_1 = b_0 - kb_1$$
$$a_2 = b_1 - kb_2$$
$$a_3 = b_2 - kb_3$$
$$\vdots$$
$$a_{n-1} = b_{n-2} - kb_{n-1}$$

Now $b_{n-1} = a_n$, since they are coefficients of x^n on each side of the equation. Therefore, we may initialise $b_{n-1} = a_n$, using this fact, and then iterate according to:

$$b_{i-1} = a_i + kb_i, for i = n, n-1, n-2, \ldots, 2, 1$$

(in this reverse order!), then we have generated all the coefficients of the polynomial Q. Since the last one generated is b_0, we can now calculate the remainder r_0 from equation (5). Notice that we are counting backwards! This algorithm looks very similar to our nested polynomial evaluation algorithm. It is known as Horner's method, or **synthetic division.** We can code it in Pascal as follows:

```
program Horner;
{ In this program a polynomial is divided by a linear factor }
var
  i, n    : Integer;
  r, k, x : Real;
  a, b    : array[0..9] of Real;
begin
  write('Value  Of k ? '); readln(k);  { linear factor is (x-k) }
  write('Degree Of Polynomial ? '); readln(n);
  for i:=0 to n do begin
    write('a[',i:1,'] = ? ');
    readln(a[i]);
  end;
  b[n-1]:=a[n];
  i:=n - 1;
  repeat
    b[i-1]:=a[i] + k*b[i];
    i:=i - 1
  until i = 0;
  r:=a[0] + k*b[0];
  writeln; writeln('Remainder = ',r:7:5); writeln;
  for i:=0 to n-1 do writeln('b[',i:1,']=',b[i]:7:5);
end.
```

Output

```
Value  Of k ? 10
Degree Of Polynomial ? 3
a[0] = ? 1
a[1] = ? 2
a[2] = ? 3
a[3] = ? 4

Remainder = 4321.00000

b[0]=432.00000
b[1]=43.00000
b[2]=4.00000
```

The quotient polynomial is therefore:

$$Q_2(x) = 4x^2 + 43x + 432 \quad \text{and hence}$$

$$4x^3 + 3x^2 + 2x + 1 = (x - 10)(4x^2 + 43x + 432) + 4321$$

Do you notice any patterns here? Why do they appear? You should verify that if the coefficients of a polynomial are decimal digits (the natural numbers 0..9), then P(10) is simply the decimal integer whose digits are these coefficients.

8.5 Historical Note

Leonhard Euler (1707-1783)

The Swiss mathematician Euler is often considered to be the most prolific mathematician ever. He studied under Jean Bernoulli in Basel, but spent almost all of his later life outside his native Switzerland. His prodigious output is almost unbelievable, with extensive contributions to mathematical knowledge in every field of mathematics that existed at the time. His work continued unabated throughout his life despite losing an eye in 1735, and the second in 1766. A total of 886 books and papers were published, including some 241 manuscripts published posthumously, and another 115 discovered by a later researcher.

Euler wrote mathematical textbooks as well as papers on his own original discoveries. Many results of elementary calculus, geometry, infinite series are named after him. He greatly extended Newton's theory of mechanics to solid bodies and rotational dynamics. His works introduced many forms of notation which are still the accepted modern forms. The symbols pi,e,i (exp(i.pi)+1=0), the abbreviations for the trigonometric functions, the Greek sigma used for summation, and f(x) for function are all due to this outstanding Swiss mathematician.

The famous result $e^{ix} = \cos x + i \sin x$ is called Euler's formula. This formula appears on a Swiss postage stamp honouring the great mathematician. Euler also appears on the Swiss 10 franc note.

Another very famous result, also called Euler's formula, relates the number of faces, vertices and edges of any polyhedron:

V - E + F = 2.

The Euler constant is the limiting difference between the partial sums of the harmonic series and the natural logarithm of the number of terms; both of which increase without limit. Their difference does however have a limit - the Euler constant. More precisely:

$$\gamma = \lim_{n \to \infty} \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} - \log n \right)$$

Euler's constant to 10 decimal places is $\gamma \approx 0.57721\ 56649$.

8.6 Summary

- The **array** data structure is very commonly used in mathematical computing. One important use is to represent and process **polynomials.** In this class of applications, we use an array to store the **coefficients** of the polynomial. The powers of x in the polynomial need not be represented since they are implied by the respective coefficients. This means that zero coefficients must be stored as empty placeholders.
- Another application of arrays is to store the elements of **vectors** and **matrices.** Quantities such as **dot product** and products of matrices may then be computed.
- A further application of arrays is for **sorting.** One of the simplest methods of array sorting is commonly known as **BubbleSort.** In this method, adjacent values are exchanged if out of order, and this process repeated until the entire array is sorted.
- The best way to evaluate a polynomial for a given value of x is by **nested multiplication.** This method is sometimes referred to as **Horner's** method.
- A polynomial may be divided by a linear factor by the usual **long-division** process. This is most efficiently done on the computer by an iterative technique called **synthetic division,** in which no actual division is done. The coefficients of the quotient polynomial are generated by a linear recurrence formula involving the coefficients of the original polynomial and the constant term of the linear divisor polynomial.

8.7 Exercises

*1. By making modifications to the program **DotProduct,** find the angle between the following vectors:

(i) $2\hat{i}+3\hat{j}$ and $-\hat{i}+4\hat{j}$ (iii) $2\hat{i}+3\hat{j}+\hat{k}$ and $-3\hat{i}-4\hat{j}+3\hat{k}$

(ii) $(2,6)$ and $(4,-3)$ (iv) $(3,-8,4)$ and $(-2,5,-2)$

*2. Sort the array 44 55 12 42 94 18 6 67 using the program **BubbleSort.** Amend the program to show the state of the vector every time it is reformed.

3. If we use the bubble sort algorithm to sort 2,3,4,5,6,7,8,1 we need seven passes.
   (i) Verify this statement by writing the state of the array every time it is reformed.
   (ii) Develop a bubble sort which scans the numbers from right to left instead left to right. How many passes does it take to sort the above set of data?
   (iii) Develop a bubble sort which alternately sorts from left to right and right to left and discuss its merits (if any).

4. Another algorithm for sorting is the **insertion** sort. It is illustrated below:

Integer	Pass 1	Pass 2	Pass 3
3	2	1	1
2	3	2	2
6	6	3	3
1	1	6	5
5	5	5	6
9	9	9	9

5. Use the program **NestPoly** to evaluate f(-5), f(-10), f(3), f(8) for the following polynomial functions. Verify some of your answers by direct hand substitution.

(i) $f(x)=x^3-3x^2+7x-10$ (iii) $f(x)=2x^4+x^3+-2x^2+8$

(ii) $f(x)=7x^3-3x+5$ (iv) $f(x)=x^4-3x^2+9$

Write a program for this type of sort.

6. Using the program **Horner,** determine whether or not the given linear polynomials are factors of the polynomials:

(i) $f(x)=3x^3-3x^2+1$; $(x-1)$

(ii) $f(x)=x^5-3x^3+2x-7$; $(x+2)$

(iii) $f(x)=x^5-1$; $(x+1)$

Check each result using the factor theorem.

7. Using the program **Horner** find the quotient and remainder for the following divisions:

(i) $(x^3+5x^2-7x+13)\div(x+7)$

(ii) $(x^5-32)\div(x-2)$

(iii) $(3x^3-x^2+7x-9)\div(x-3)$

8. Determine whether which of the linear polynomials $(x-1)$; $(x+3)$; $(x-5)$; $(x+6)$; $(x-4)$ are factors of the given polynomials

(i) $f(x)=x^3-2x^2-5x+6$ (iv) $f(x)=x^4+3x^2+x$

(ii) $f(x)=x^3+2x^2-21x+18$ (v) $f(x)=2x^4-18x^3+43x^2-27x+60$

(iii) $f(x)=x^3-6x^2-13x+42$ (vi) $f(x)=x^5-x^4-43x^3+61x^2+342x-360$

9. Show that the largest element of the array is at the bottom after the first pass in the BubbleSort algorithm.

10. Verify Euler's formula V-E+F=2 for the five regular polyhedra (triangular pyramid (F=4), cube (F=6), octahedron (F=8), dodecahedron (F=12), icosahedron (F=20)).

9 INTEGRATION

9.1 Numerical Integration

Integral calculus is concerned with finding areas under curves, lengths of arcs and volumes of revolution. In ordinary English usage, integration of course means the "**binding together**" of parts or components to form a composite whole. In calculus we add a large number of small "parts" (e.g. rectangles under a curve) together in order to obtain some estimate of the area under the curve. This summation is shown in Figure 9.1.

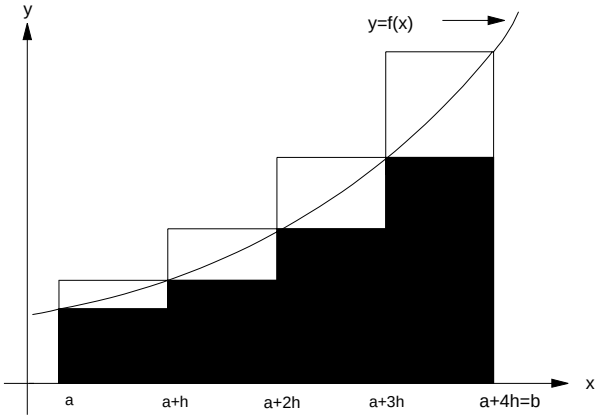


Figure 9.1 Numerical Integration

It is important to note that the term **integration** does **not** refer to the process of finding some **anti-derivative,** but to the process of finding **limits of sums.** It sometimes turns out that in our efforts to find limiting sums of small elementary areas (for example, rectangles in Figure 9.1), certain **anti-derivatives** can save us a lot of work. In such cases, we make use of the **Fundamental Theorem Of The Calculus** in order to avoid summation of "little parts"; but in general, we must remember that integration is concerned with the basic process of **summation**. The integral symbol itself is derived from an elongated "s" (for sum).

A Simple Program For Numerical Integration

As our first example, let's consider the function:

$$f(x) = x^2,$$

and obtain an estimate for the sum of rectangles between the arc of this curve, the ordinates at x=0 and x=1, and the x-axis. For each given number of rectangles in a subdivision of [0,1], the corresponding sum gives an estimate of:

$$\int_0^1 f(x)dx.$$

This notation stands for the (exact) area between the ordinates at x=0, x=1, the x-axis, and the curve y=f(x).

Considering **interior** rectangles, we have:

$$S = \sum_{i=0}^{n-1} A_i$$

$$= A_0 + A_1 + \ldots + A_n$$

where n is the number of rectangles, and A_i is the area of the i th rectangle. We have, of course, A_i = base x height, and the base of the i th rectangle = $\frac{1}{n}$ and height of the i th rectangle = $f\left(\frac{i}{n}\right)$.

Our sum then becomes:

$$S = \sum_{i=0}^{n-1} \frac{f\left(\frac{i}{n}\right)}{n} = \sum_{i=0}^{n-1} \frac{1}{n}\left(\frac{i}{n}\right)^2 = \frac{1}{n^3}\sum_{i=1}^{n-1} i^2, \qquad \text{since n is constant.}$$

We do not have space here (but see exercises), to show:

$$\sum_{i=1}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$$

Hence, $\frac{1}{n^3}\sum_{i=1}^{n-1} i^2 = \frac{(n-1)(2n-1)}{6n^2} = \frac{1}{3} - \frac{1}{2n} + \frac{1}{6n^2}$

We can see that as n approaches infinity, this quantity will approach 1/3.

Now that we have some theoretical background for comparison, and know what to expect, we make use of the computer as a tool for further insight into this very fundamental process of "adding up rectangles". We consider a simple Pascal program to evaluate these approximating sums for various values of n.

```
program NumInt1;
{ This program finds an approximation to the area under
the curve y=x*x by adding rectangles under the curve }
var
  i, n        : Integer;
  floati, sum : Real;
begin { main }
  n:=10;
  writeln('   n       sum'); writeln;
  repeat
    sum:=0;
    for i:=1 to n-1 do begin
      floati:=i;
      sum:=sum + sqr(floati);
    end;
    sum:=sum/n/n/n;
    writeln(n:4, sum:12:6);
    n:=n + 10;
  until n > 100
end. { main }
```

Output

```
 n     sum

10   0.285000
20   0.308750
30   0.316852
40   0.320937
50   0.323400
60   0.325046
70   0.326224
80   0.327109
90   0.327798
100  0.328350
```

Discussion of Output

We notice here that improvement in accuracy is very slow after n = 20. If **floating-point** (i.e. real) arithmetic on the computer were perfectly accurate, then we should expect to increase accuracy to an arbitrarily high level just by increasing the number of rectangles. However, with **finite-precision arithmetic,** the use of a large number of terms in a sum, each of which is subject to roundoff errors, will ultimately have a detrimental effect on the overall accuracy of the final result. We see therefore that there must be an optimal value of n, up to which accuracy increases, but beyond which accuracy begins to diminish.

Since we already know from calculus that the answer to our integral is 1/3, it may seem like a foolish waste of time to get only an approximation, and this at the expense of a lot of hard work, when we already have the exact answer at our fingertips! There are in fact at least three good reasons for looking at the numerical approach. These are:

(a) It reaffirms the basic definition of the integral as a **summation process** by simply using the computer as a calculating tool.

(b) To use the result (Fundamental Theorem of the Integral Calculus):

$$\int_a^b f(x)dx = F(b) - F(a),$$

where F is some function whose derivative is f; **one must be able to find such an F!** This is rarely easy, even for highly-skilled mathematicians. In fact, there are cases where it can be proved that no such function F exists, even though the integral exists and can be estimated perfectly well by numerical techniques. The fundamental theorem is therefore useless to us in these cases.

This of course means that to evaluate the integral, we have to go back to the definition and start "adding up little bits again". **In order to be prepared for this inevitable situation, we should have an appropriate, well-tested computer program at our disposal.** This brings us to the third point.

(c) It makes no sense to expect a computer program to work correctly on "hard" problems if it fails on the easy ones. **Always test your programs out first on simple test data for which you know the answer in advance, if at all possible.** In some cases, this will not be possible or feasible. However, in most cases, it is possible to devise test data for which the solution can be reliably predicted. For numerical integration programs, it is very easy to find such test cases. If our program does not come up with a reasonable approximation to, for example:

$$\int_2^4 x dx,$$

then how can we expect it to work for:

$$\int_0^\pi (\exp x \sin x + \log x cos x)dx \ ?$$

The moral is obvious: use simple test cases before moving on to the "hard" ones. But even if your program works for all test cases that you have tried, you are still not justified in claiming that it works for all possible cases. The only way to do this is to construct a mathematical proof of correctness for the algorithm and its implementation. Such considerations are far beyond the scope of this book.

Numerical Integration Algorithms

We now turn to look at classes of algorithms for numerical integration. We have already considered sums of interior rectangles, but have not yet formalised the basic method of summing rectangles for the general case of:

$$\int_a^b f(x)dx$$

We do this in the pages that follow; and after that progress to more accurate techniques.

All methods of numerical integration that we study in this chapter employ some form of **functional approximation** to the integrand f(x), and then go on to estimate the original integral by finding the integral of the approximating function. For example, the method we have considered so far uses a **step function,** i.e. piecewise-constant function to approximate the integrand f(x). The justification is that the integral of the approximating function will be close to the integral of the original function. This is indeed the case if the number of subdivisions is not too small. It is a simple matter to integrate constant functions (rectangles).

As a general point, it should be kept in mind that **numerical integration is by nature a very stable process,** and small errors or fluctuations tend to be "smoothed out". This is in direct contrast to **numerical differentiation,** which, as we have already noted, is **intrinsically unstable.** Great caution has to be exercised with any numerical approximation algorithm on a computer; differentiation being one of the most dangerous, while integration methods are generally very well-behaved.

Instead of using only **constant** functions (**i.e. constant polynomials**) to approximate our integrand, we might expect more accuracy if we used some higher-degree polynomials. This is certainly the case for polynomials of fairly low degree (e.g. linear, quadratic, cubic), and we now consider three such methods for numerical integration. These are referred to collectively by numerical analysts as **Newton-Cotes formulae**. They all involve approximating the integrand by a piecewise-polynomial function and then integration of the polynomial "parts" and adding the results. These methods are:

(a) The Interior/Exterior Rectangle Method (f is approximated by constant polynomials),
(b) The Trapezoidal Rule (f is approximated by linear polynomials), and
(c) Simpson's Rule (f is approximated by quadratic polynomials).

In all cases, the interval of integration is [a,b]; the integrand is f(x), and [a,b] is cut into n equal sub-intervals, where $n \in N$. We now need a few more preliminary definitions.

9.2 Integration - Rectangular Method

Regular Partitions

Given a closed interval [a,b], and a positive integer n, we define the set of points:

$$P_n(a,b) = \{a = x_0, x_1, x_2, \ldots, x_n = b\}, \text{ where } x_i = a + ih; \quad nh = b - a; \quad i = 0, 1, 2, \ldots, n.$$

to be a **regular partition** of the closed interval [a,b]. They form the **arithmetic progression:**

$$x_0 = a,$$

$$x_1 = a + h,$$

$$x_2 = a + 2h,$$

$$\ldots$$

$$x_n = a + nh = b.$$

Example of a Regular Partition

Let a = 0, b = 2, n = 4. Therefore, $P_4(0,2) = \{0.0, 0.5, 1.0, 1.5, 2.0\}$ and

$$a = x_0 = 0.0,$$

$$x_1 = 0.5,$$

$$x_2 = 1.0,$$

$$x_3 = 1.5,$$

$$b = x_4 = 2.0.$$

A regular partition is just the set of points formed by slicing up the interval [a,b] into equal parts (sub-intervals). These points are then used to compute function values for inclusion in sums which estimate the integral. Our formal definition of the regular partition may look a little abstract, but it is precisely these expressions that we must use in order to later write the Pascal code for our numerical integration programs. It seems like a good idea then to familiarize ourselves with the **compact** notation using the variable **i** for an index (or counter), such as we have in the definition of a regular partition. Not only are we then comfortable with this way of defining a regular partition, but, after a little practice, writing the Pascal code then becomes rather easy. A regular partition is illustrated in Figure 9.2.

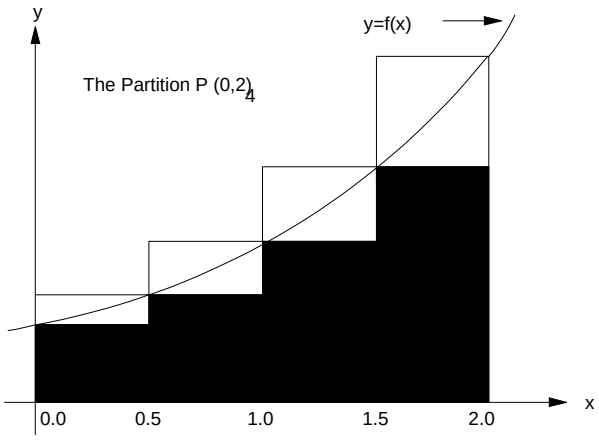


Figure 9.2 The Regular Partition $P_4(0,2)$

Our first attempt at estimating the area under a curve is just to construct two sequences of rectangles. These are called **Interior** and **Exterior** rectangles, and one of each is constructed for each pair of adjacent points of the partition. By adding up "base x height" for each of the interior and exterior sets of rectangles, we get lower and upper bounds for the integral of f(x) on [a,b], at least for those f which are non-negative on [a,b]. We define the **Interior Sum:**

$$I_n = \sum_{i=0}^{n-1} h f(x_i) = hf(a+0h) + hf(a+1h) + hf(a+2h) + \ldots + hf(a+(n-1)h)$$

and the **Exterior Sum:**

$$E_n = \sum_{i=1}^{n} h f(x_i) = hf(a+h) + hf(a+2h) + hf(a+3h) + \ldots + hf(a+nh)$$

We now consider a Pascal program for computing the interior sum for a given value of n. The function used as an example is f(x)=1/x.

```
program Interior;
{ This program obtains an approximation to the area
  under y=1/x by summing interior rectangles }
var
  i, n              : Integer;
  a, b, h, xi, sum : Real;

function f(x : Real): Real;
begin
  f:=1.0/x
end;

begin  { main }
  n:=100; a:=1.0; b:=2.0; h:=(b - a)/n;
  writeln('   n        sum');
  sum:=0;
  for i:=0 to n-1 do begin
    xi:=a + i*h;
    sum:=sum + f(xi);
  end;
  sum:=sum*h;
  writeln(n:4, sum:12:6);
end. { main }
```

Output

```
   n        sum
 100    0.695653
```

Discussion of Output

Note that this answer is an estimate for ln 2. This area is shown in Figure 9.3.

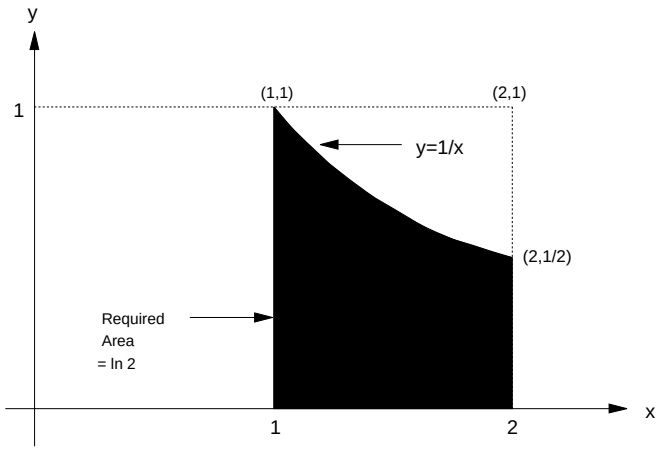


Figure 9.3 An Area Equal To Ln 2

The observant student will have also noticed that the formulae for the exterior and interior sums are almost identical. Furthermore, the arithmetic mean (ordinary average) of the two sums is a better approximation to the integral that either sum alone. Geometrically, this really amounts to joining the points on the curve by straight lines and summing trapezia, rather than just having interior or exterior rectangles. This leads us to the Trapezoidal Rule.

9.3 The Trapezoidal Rule

The Trapezoidal Rule formula is given in Figure 9.4.

THE TRAPEZOIDAL RULE

$$T_n = h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(a+ih)\right)$$

$$= h\left(\frac{f(a)+f(b)}{2} + f(a+h) + f(a+2h) + f(a+3h) + \ldots + f(a+(n-1)h)\right)$$

Figure 9.4 The Trapezoidal Rule

Can you see how this formula results from taking the average of the Interior and Exterior sums ? It is equivalent to joining the the ordinates on the curve at the partition points with direct straight lines. This is illustrated in Figure 9.4.

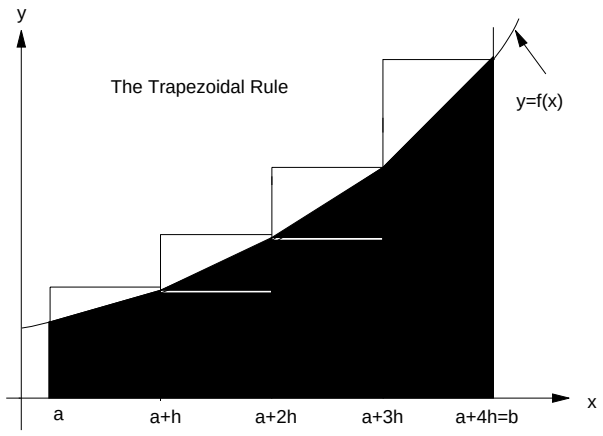


Figure 9.5 The Trapezoidal Rule

Exercise

Write a Pascal program for the trapezoidal rule. Test it out on some simple functions.

Solution

We present below a Pascal program for the trapezoidal rule, using the function y=1/x as an example.

```
program Trap;
{This program finds the area under a curve using the trapezoidal rule}
var
i, n             : Integer;
a, b, h, xi, sum : Real;

function f(x : Real): Real;
begin
  f:=1.0/x
end;

begin  { main }
  n:=100; a:=1.0; b:=2.0; h:=(b - a)/n;
  writeln('Trapezoidal Rule');
  writeln;
  writeln('   n       sum');
  sum:=(f(a) + f(b))/2.0;
  for i:=1 to n-1 do begin
```

```
    xi:=a + i*h;
    sum:=sum + f(xi);
  end;
  sum:=sum*h;
  writeln(n:4, sum:12:6);
end. { main }
```

```
Output

Trapezoidal Rule

   n       sum
 100    0.693153
```

Notes On The Trapezoidal Rule

In the exterior and interior rectangles methods, we approximate the function points by the simplest function possible: a **constant** (polynomial). In the trapezoidal rule, we improved on this by using a **linear function.** The linear function for each sub-interval of the partition joined the points on the curve by a straight line. A linear polynomial is the next simplest polynomial up from a constant. The trapezoidal rule simply estimates the value of the integral of the original function by the sum of the integrals of all the trapezia.

9.4 Simpson's Rule

A further gain in accuracy (for a given value of n) can be obtained by generalising the functions approximating f on each sub-interval by **quadratic polynomials.** In this method, known as **Simpson's Rule,** each arc of the curve of f between consecutive points of the partition is approximated to an **arc of a parabola.** We do not have space for the details here but refer the student to standard treatments of this subject in calculus texts. We simply give the formula in Figure 9.6, remarking that the number of strips (sub-divisions) must be even, so we call it 2n for convenience. We can then build a Pascal program for use with our test function f(x)=1/x and compare the results with those of the trapezoidal rule and the exact answer, which is ln 2.

SIMPSON'S RULE FOR 2N STRIPS

Simpson's approximation with 2n strips for $\int_a^b f(x)dx$ is given by:

$$S_{2n} = \frac{h}{3}\left(f(a) + f(b) + 4\sum_{i=1}^{n} f(x_{2i-1}) + 2\sum_{i=1}^{n-1} f(x_{2i})\right)$$

$$= \frac{h}{3}(f(a) + f(b) + 4(f(x_1) + f(x_3) + \ldots + f(x_{2n-1})) \quad + \quad 2(f(x_2) + f(x_4) + \ldots + f(x_{2n-2})))$$

Note: For n=1 (2 strips), we are to understand that the second sum (the even sum) is empty.

Figure 9.6 Formula For Simpson's Rule

A Pascal Program For Simpson's Rule

```
program Simpson;
{ This program finds the area under a curve using Simpson's rule }
var
i, n : Integer;
b, a, X, H, OddSum, EvenSum, Result : Real;

function f(x : Real) : Real;
begin
  f:=1/x
end;

begin { main }
  n:=6; a:=1.0; b:=2.0;
  h:=(b - a)/(2*n);
  OddSum:=f(b - h);
  EvenSum:=0.0;
  for i:=1 to n-1 do begin
    OddSum :=OddSum  + f(a + (2*i - 1)*h);
```

```
    EvenSum:=EvenSum + f(a + 2*i*h);
  end;
  Result:= h/3.0*(f(a) + f(b) + 4.0*OddSum + 2.0*EvenSum);
  writeln('Area under curve = ',Result:10:6)
end. { main }
```

Output

Area under curve = 0.693149

Arc Length Of A Curve

In the previous sections we looked at the estimation of areas bounded by curves using a large number of rectangles or other simply-computed areas. A similar approach, that is, the summation of a large number of small elements, may be used to obtain estimates of the length of arc of a given curve. The distance formula may be used to easily compute the straight-line distance between successive points along an arc, and, if the separation of points is sufficiently small, we should arrive by summation at a reasonable estimate of the arc length of a finite section of curve. Figure 9.7 shows a small section of arc and the straight-line segments whose lengths are added to approximate the arc length of the curve, i.e.

$$L = d_1 + d_2 + d_3 + \ldots + d_n, \quad \text{where} \quad d_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

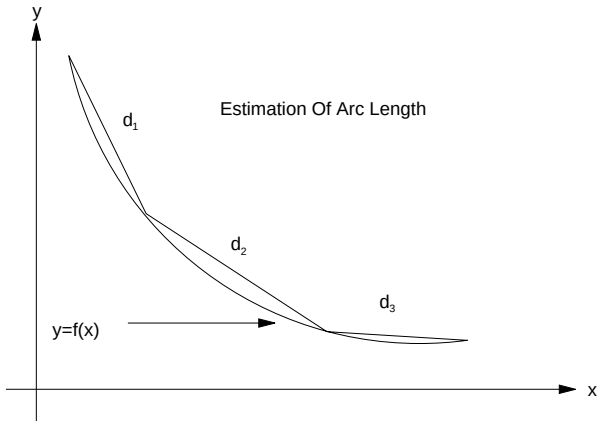


Figure 9.7 Arc Length Of A Curve

These ideas are illustrated in the following Pascal program, which estimates the length of a quarter-circle (quadrant) with radius unity. The exact answer is $\frac{\pi}{2.}$

```
program ArcLen1;
{ This program finds the arc length of a curve by adding up chords }
var
  a,b,h,xi,yi,prevx,prevy,Len : Real;
  i,n : Integer;

function f(x : Real) : Real;
begin
  f:=Sqrt(1.0 - x*x)
end;

function Dist(x1,y1,x2,y2 : Real) : Real;
begin
  Dist:=Sqrt(Sqr(x1 - x2) + Sqr(y1 - y2))
end;

begin  { main }
  a:=0.0; b:=1.0; n:=100;
  h:=(b - a)/n;
  Len:=0.0; prevx:=a; prevy:=f(a);
  for i:=1 to n do begin
    xi:=a + i*h; yi:=f(xi);
    Len:=Len + Dist(xi,yi,prevx,prevy);
```

```
    prevx:=xi; prevy:=yi                {for next time 'round}
  end;
  writeln('Computed length of quadrant   = ',Len:10:8);
  writeln('Correct value is pi/2; approx = ',pi/2.0:10:8)
end. { main }
```

Output

```
Computed length of quadrant   = 1.57064794
Correct value is pi/2; approx = 1.57079633
```

Use Of Simpson's Rule To Find Arc Length

Since the exact value of arc length may be expressed as an integral, and the integral in any given case may be estimated by one of the more accurate of the numerical integration formulae, e.g. Simpson's rule, we may obtain our arc length approximation in a much more efficient manner as follows. We write:

$$x_i - x_{i-1} = dx, \text{ and } dy = f'(x).dx$$

So our formula for arc length becomes:

$$L = \int_a^b \sqrt{1 + (f'(x))^2} \quad dx$$

Simpson's Rule may then be used to estimate this integral for a given f, normally with far fewer sub-divisions for a given accuracy. Since we already have a program for Simpson's Rule, this is easily done for the quadrant example above, and this is the subject of a problem at the end of the chapter.

9.5 Historical Note

Bernhard Riemann (1826-1866)

In Dirk Struik's interesting little book "**A Concise History Of Mathematics**", the author remarks that "With Bernhard Riemann,..... we reach the man who more than any other has influenced the course of modern mathematics."

Although dying at a relatively early age, this great man had an influence far beyond that which his contemporaries had been able to realize. He published only a small number of papers, but each made important contributions to mathematical learning. His doctoral thesis concerned functions of a complex variable, and today, students of fluid dynamics, elasticity, electromagnetic theory and related subjects learn the **Cauchy-Riemann equations,** which are the basis of mathematical analysis (complex function theory) in these fields.

The theory of integration that we studied in this chapter is known as **Riemann integration,** to distinguish it from more general theories of integration and measure theory due to Lebesgue, Perron, and others.

He gave in his lectures an example of a function continuous everywhere but having a derivative nowhere. Most mathematicians of the day initially refused to believe that such a function could exist! Can you imagine such a function ? It certainly strains the imagination and tests our intuitive concepts of just what is meant by the terms **continuity** and **differentiability.**

Riemann also worked on the classification of the various competing theories of geometry of the day, in particular **Non-Euclidean Geometries,** and was able to give a Euclidean model of a system in which Euclid fifth postulate (the **parallel** postulate) was altered.

The **Riemann Zeta Function** for complex argument z is defined as follows:

$$\zeta(z) = \sum_{n=1}^{\infty} \frac{1}{n^z}.$$

This function is important in number theory, in particular to the theory of the distribution of **prime numbers.** Riemann's famous conjecture (the **Riemann Hypothesis**) concerning this function is that all its zeroes have real part x=0.5. To this day, no one has been able to prove or disprove it.

9.6 Summary

- Integration is the process where the limit concept is used to infer values of areas under curves (integrals) from sums of rectangular areas. The same ideas can be extended to arc lengths, surface areas and volumes, though the latter two are not treated here.
- Approximations to integrals may be easily found by simple computer programs. This process is called **numerical integration.** The methods we study here are rectangular method, trapezoidal rule and Simpson's rule. They are members of a general class of numerical integration formulae called **Newton-Cotes** formulae, and estimate integrals by summing integrals of simple polynomials approximating the integrand.
- Some applications areas under a curve are the estimation of constants such as π and ln 2.

9.7 Exercises

1. From the definitions of the interior and exterior sums for a given f,n, and interval [a,b], show that:

   $$I_n + hf(a+nh) = E_n + hf(a+0h)$$

   Show further that this is equivalent to: $E_n = I_n + h(f(b) - f(a))$.

2. Verify that the output of the first simple integration program agrees with the algebraic result obtained for S_n for each value of n. The simplest way to do this is to extend the program to calculate:

   $$\frac{(n-1)(2n-1)}{6n^2} \text{ or } \frac{1}{3} - \frac{1}{2n} + \frac{1}{6n^2}$$

   and tabulate it alongside the existing columns.

3. Show that the trapezoidal rule results from taking the average of the interior and exterior sums formulae. In other words, show that:

   $$T_n = \frac{(I_n + E_n)}{2}$$

4. It can be shown that Simpson's rule with any (even) number of strips is **exact**, i.e. yields the correct answer, for all polynomials of degree 3 or less. Verify this result for the following integrals by comparing Simpson's approximation with 12 strips with the exact value obtained by using the Fundamental Theorem of the Calculus.

   (i) $\int_0^2 (x^2 - 2x + 1) \quad dx$

   (ii) $\int_2^4 x^3 \quad dx$

5. Write or use a program for the exterior sum estimate of ln 2. Calculate the average of the interior and exterior sums. Verify that this value is a better approximation to ln 2 than either the exterior or interior sum.

6. Use Simpson's Rule with 20 strips to estimate the integral for the length of one quadrant of the unit circle. Compare your answer both with the exact value of pi/2 (to 6 decimal places) and the estimate obtained in this chapter from the addition of small straight-line segments.

7. Use the trapezoidal and Simpson's rule programs to estimate the following integrals. Compare each value obtained using the Fundamental Theorem of the Calculus, and tabulate each alongside the absolute, relative, and percentage errors. Do each for 20, 40, 60, 80, 100 strips. Each table should therefore have seven columns and take the following form:

   No.Strips Trap Est Simp Est Exact AbsErr RelErr % Err

   (i) $\int_2^3 (x^2 + 6x - 5) \quad dx$

   (ii) $\int_0^{2\pi} \sin x \quad dx$

   (iii) $\int_0^{4\pi} e^{-x} \cos x \quad dx$

   $Hint: \frac{d}{dx}(e^{-x}(a\cos x + b\sin x)) = e^{-x}\cos x,$ for some constants a and b.

(*iv*) $\int_1^5 \ln x \quad dx \qquad Hint: \frac{d}{dx}(x\ln x) = 1 + \ln x$

8. Automate the procedure of the previous question by writing a Pascal program to do the calculations for 20,...,100 strips for trapezoidal and Simpson's rules and produce the tables as in the previous question. Compare the tables from both questions.

9. From your work in the previous two exercises, comment on the relative accuracies of the Trapezoidal Rule and Simpson's Rule.

10. Consider the sum:

$$s_n = \sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$$

This sum is a partial sum for the **Harmonic Series.** Write a Pascal program to print partial sums of this series for 10, 20, 30,.., 200 terms. Does it appear that a limit is being approached ? What is the real truth about this very simple-looking series ?

11. Concerning the Riemann zeta function defined in our historical section, show that:

$$\zeta(1) = \sum_{n=1}^{\infty} \frac{1}{n}$$

This the harmonic series once again. What have we learned about this series?
Show also that:

$$\zeta(2) = \sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots$$

Write a Pascal program to estimate $\zeta(2)$ by truncating this series after 10 or 20 terms. See if you can get a reasonable upper bound on the truncation error. A sketch of the program would be:

```
begin
  sum:=0;
  for n:=1 to 20 do sum:=sum + 1/sqr(n);
  writeln('zeta(2) = ',sum)
end.
```

(Exact value of $\zeta(2) = \frac{\pi^2}{6}$)

12. For our integral of x^2, we needed the sum:

$$\sum_{i=1}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$$

Derive this formula. Mathematical induction will do it, or you can assume that

$$\sum_{i=1}^{n-1} i^2 = An^3 + Bn^2 + Cn + D$$

for some constants A, B, C, D. Find these constants by substituting n=2,3,4,5 and solving the resulting set of simultaneous equations. You could do this by hand or use one of the programs in this book.

10 LINEAR EQUATIONS

10.1 Direct Methods

In modern mathematics, systems of two or more linear equations in two or more unknowns are very common. It is not unusual for scientists to have to solve a system of 50 equations in 50 unknowns. Of course a high speed electronic computer is essential to perform the necessary arithmetic operations. We usually express n simultaneous linear equations as:

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \ldots + a_{1j}x_j + \ldots + a_{1n}x_n = b_1 \\
&a_{21}x_1 + a_{22}x_2 + \ldots + a_{2j}x_j + \ldots + a_{2n}x_n = b_2 \\
&a_{31}x_1 + a_{32}x_2 + \ldots + a_{3j}x_j + \ldots + a_{3n}x_n = b_3 \\
&\quad\vdots \\
&a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{ij}x_j + \ldots + a_{in}x_n = b_i \\
&\quad\vdots \\
&a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nj}x_j + \ldots + a_{nn}x_n = b_n
\end{aligned}
$$

Alternatively, we may write this set of equations in the form:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \qquad i = 1, 2, 3, \ldots.., n \qquad (1)$$

In matrix form, we write **Ax=b i.e.**

$$
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\
a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\
a_{31} & a_{32} & a_{33} & \ldots & a_{3n} \\
. & . & . & & . \\
. & . & . & & . \\
. & . & . & & . \\
a_{m1} & a_{m2} & a_{m3} & \ldots & a_{mn}
\end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{pmatrix}
=
\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ . \\ . \\ . \\ b_n \end{pmatrix}
$$

This form is of course equivalent to the set of equations in (1).

There are many methods for solving simultaneous equations using computers. They essentially fall into two categories - **direct** and **indirect** methods. The particular method chosen depends on the efficiency, accuracy, reliability, and simplicity required. Also, a particular method will depend on the type of coefficient matrix. Coefficient matrices in problems of practical interest are most often one of two types:

(i) Most of the elements of the matrix are non-zero and the matrix is of order 30 or less.

(ii) Many elements are zero and the order can be 100 or more (sparse matrices).

Direct methods are generally superior for (i) and iterative methods better for (ii). In your mathematics studies you may have used a direct method (elimination) to solve 2 simultaneous equations in two unknowns, and possibly even 3 equations with 3 unknowns. In these cases, the coefficients were usually small integers. Here, we can lift this restriction since the computer can do all the tedious arithmetic work for us.

Direct Methods For Solving Simultaneous Equations

A direct method for the solution of simultaneous equations, is one which, if all computations were carried out without roundoff errors, would lead to an exact solution of the given system. Many direct methods involve some variation of the elimination procedure associated with the name of Gauss. The method you have probably used is the Gaussian elimination method, and this is illustrated in the example below.

Example 1

Solve the system of equations

```
2x - 5y = -6                    (1)
3x + 2y = 29                    (2)
```

Solution Method (a): Graphical

Clearly to obtain a "rough" solution, the graphs of the two equations could be plotted, and the point of intersection of the lines obtained. This is shown is Figure 10.1.

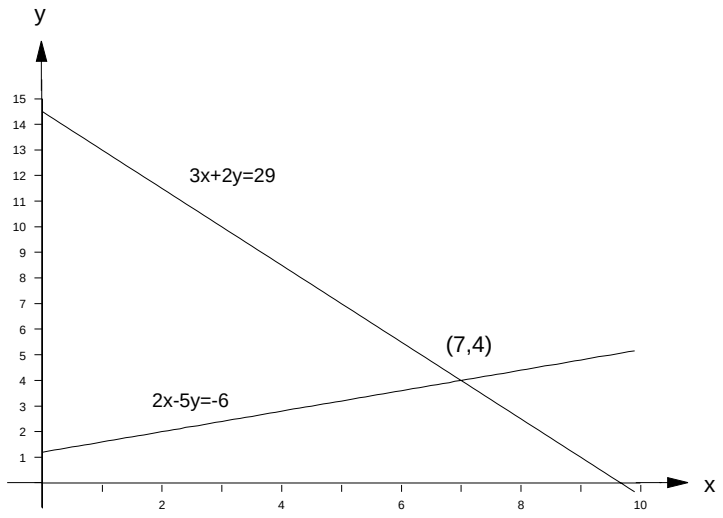


Figure 10.1 Solution Of Two Simultaneous Equations

Solution Method (b): Gaussian Elimination

Take equation (1) and multiply by (2), i.e.

```
4x - 10y = -12                    (3)
```

Take equation (2) and multiply by 5, i.e.

```
15x + 10y = 145                   (4)
```

Adding equations (3) and (4):

```
19x = 133  or  x = 7.             (5)
```

Substituting for x in (1) from (5), we arrive at:

```
2(7) - 5y = -6
     - 5y = -20  or  y = 4        (6)
```

Check by substituting for x and y from (5) and (6) into (1):

LHS of (1) = 2(7) - 5(4) = -6 = RHS of (1)

This shows that our solution (x,y)=(7,4) is correct. This verification step is very important. We even get the computer to do it when we write our programs for solving linear equations.

Solution Method (c): Matrix

Those students who have studied the unit on matrices and vectors could use the method outlined below. We first rewrite our equations in matrix form as shown below:

$$\begin{pmatrix} 2 & -5 \\ 3 & 2 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}=\begin{pmatrix} -6 \\ 29 \end{pmatrix}$$

Multiply both sides by the inverse of the coefficient matrix, i.e.

$$\frac{1}{19}\begin{pmatrix} 2 & 5 \\ -3 & 2 \end{pmatrix}\begin{pmatrix} 2 & -5 \\ 3 & 2 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}=\frac{1}{19}\begin{pmatrix} 2 & 5 \\ -3 & 2 \end{pmatrix}\begin{pmatrix} -6 \\ 29 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \end{pmatrix}=\frac{1}{19}\begin{pmatrix} 133 \\ 76 \end{pmatrix}=\begin{pmatrix} 7 \\ 4 \end{pmatrix}$$

Substituting for x=7 and y=4 in equation (1) gives 2(7) - 5(4) = -6, so our solution x=7, y=4 is correct.

Clearly, to use this matrix multiplication method, you need to be able to find the inverse of the coefficient matrix. In your other mathematics courses you probably will only be able to find an inverse for a 2x2 matrix. Therefore this method will be of very limited use to you.

In addition, we should remember that not all systems of simultaneous equations have solutions. Systems of equations which have no solution are said to be **inconsistent.** The coefficient matrix of such systems has no inverse. Actually, no matter how many equations or unknowns there are, only three cases are possible. Since these general groupings of cases are not dependent on the number of unknowns, we can illustrate them quite adequately for the case of two or three unknowns, for which cartesian graphs may be drawn.

Classification Of Solutions

Linear systems may be classified into those having:

(i) unique solution
(ii) no solution (overdetermined)
(iii) infinitely many solutions (underdetermined)

We are normally looking for a **unique** solution, but sometimes it is useful to look for an entire **family** (set) of solutions. But notice carefully here that we if we don't have a unique solution to a linear system, then it is either "all or nothing". By this we mean that, in the absence of a unique solution, we either have none or an infinite number of solutions. We cannot for example, have exactly two distinct solutions to a system of linear equations. It's simply impossible. If it's not unique, then either it doesn't exist or it is accompanied by infinitely many others. A graphical representation of the classification of solutions for two equations in two unknowns is given in Figure 10.2.

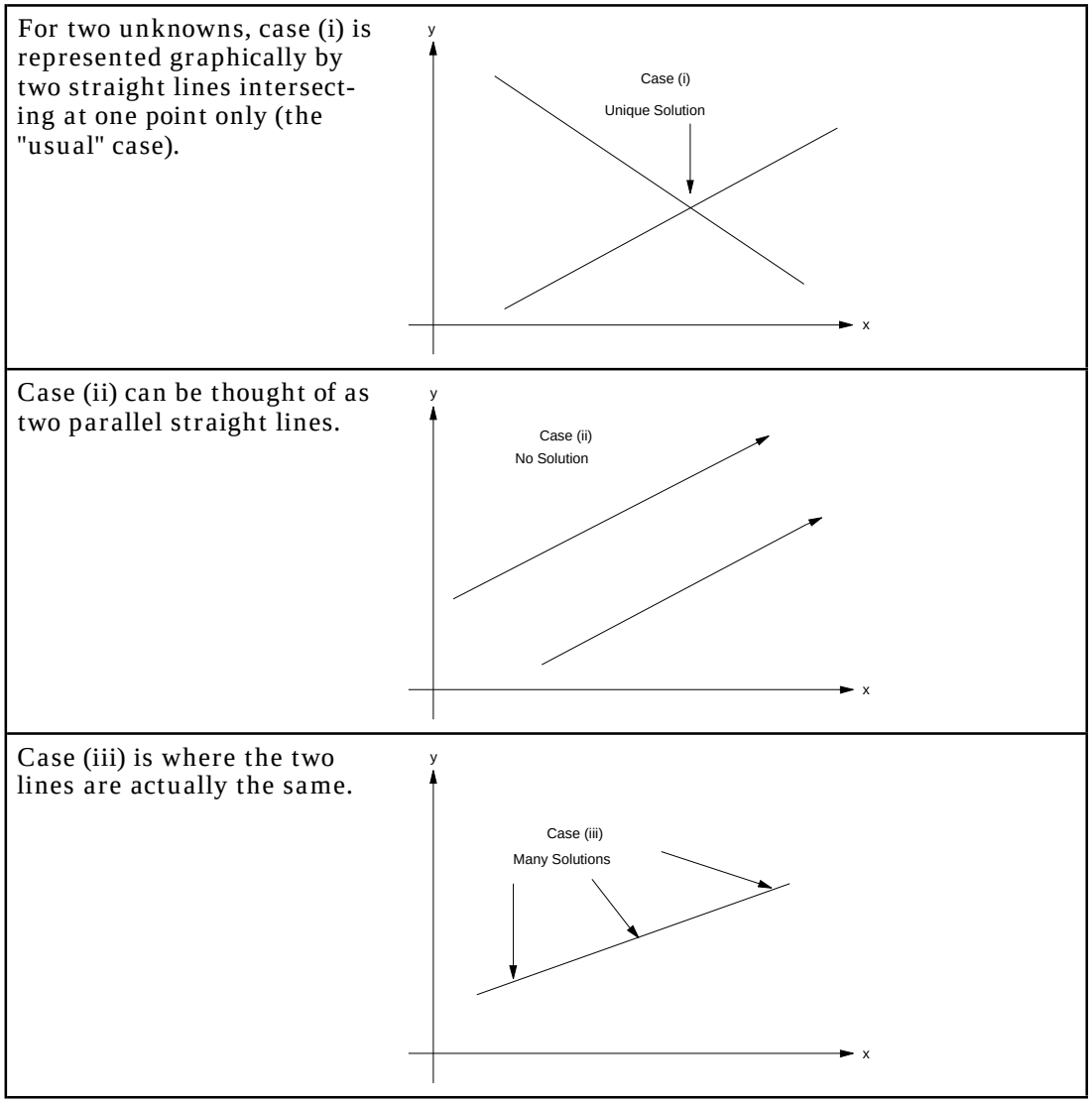


Figure 10.2 The Three Cases For Linear Equations

For systems of simultaneous equations in three or more unknowns (and particularly if the coefficients are not simple integers) the solution is a tedious and error-prone task if done by hand. If you don't believe this, then try your hand at solving the following simple-looking 3x3 system:

```
1x + 2y + 1z = 3
-x + 5y + 1z = -4
3x + 4y - 7z = -19.
```

Your answer should be x=2, y=-1, z=3.

If you really want a challenge, then you are invited to write a program to solve any linear system in three unknowns. See if you can extend the program to deal with n equations in n unknowns. Be assured that this will not be a trivial task.

So we see that, except for very simple systems of linear equations in two or three unknowns, it would be very useful to have some computer program to carry out the solution process. In fact, there are a large number of programs for this very purpose. The following program is a naive implementation of the Gaussian reduction algorithm. It does not handle cases where zero elements appear on the diagonal of the coefficient matrix.

```
program Gauss;
{ Naïve Gaussian Elimination: assumes no diagonal element
is zero at any stage }
const
  nmax=10;
var
  i,j,k,n : Integer;
  m,t     : Real;
  A       : array[1..nmax,1..nmax] of Real;
  b,x     : array[1..nmax] of Real;
  InFile  : Text;

procedure ReadData;
begin
  Assign(InFile,'GAUSS.DAT'); {open data file}
  Reset(InFile);
  i:=0; readln(InFile,n);  {first line must have value of n}
  for i:=1 to n do begin
    for j:=1 to n do read(InFile,A[i,j]);
    readln(InFile,b[i]);
  end;
  Close(InFile)
end;

procedure DisplayData;
begin
  writeln('Input Data:'); writeln;
  writeln(n:4);
  for i:=1 to n do begin
    for j:=1 to n do write(A[i,j]:8:3);
    writeln(b[i]:8:3);
  end;
  writeln
end;

procedure Eliminate;
begin
  for k:=1 to n do begin
    for i:=k+1 to n do begin
      m:=A[i,k]/A[k,k];
      for j:=k+1 to n do A[i,j]:=A[i,j]-m*A[k,j];
      b[i]:=b[i]-m*b[k]
    end
  end;
end;

procedure BackSubstitute;
begin
  x[n]:=b[n]/a[n,n];
  for k:=n-1 downto 1 do begin
    t:=b[k];
    for j:=k+1 to n do t:=t-A[k,j]*x[j];
    x[k]:=t/a[k,k]
  end
end;
```

```
procedure WriteSolution;
begin
  writeln('Solution Vector:'); writeln;
  for k:=1 to n do writeln('x[',k:1,'] = ',x[k]:12:8)
end;

begin { main }
  ReadData;
  DisplayData;
  Eliminate;
  BackSubstitute;
  WriteSolution
end. { main }
```

```
Output

Input Data:
   3
   1.000   2.000   1.000   3.000
  -1.000   5.000   1.000  -4.000
   3.000   4.000  -7.000 -19.000

Solution Vector:

x[1] =   2.00000000
x[2] =  -1.00000000
x[3] =   3.00000000
```

10.2 Cramer's Rule

One very well-known direct method for solving linear equations is **Cramer's Rule.** If the solution is unique, then this technique gives us an explicit formula for calculation of it in terms of the coefficient matrix **A** and the constant vector, **b**. Unfortunately, the method is based on the evaluation of rather complicated recursive functions known as **determinants,** and is of no practical use for computing solutions when n is greater than about 3 or 4.

Cramer's Rule expresses the solution for each unknown as the ratio of two determinants. Determinants have a number of remarkable properties that allow many shortcuts in their evaluation. These properties are essentially equivalent to the operations that we would use on our augmented matrix in one of the Gauss or Gauss-related reduction methods. A determinant of a real matrix of order n is just a real number. Its value is defined in terms of n smaller determinants of order one less (n-1). Some thought may convince you that to blindly evaluate such a quantity directly from the definition would therefore take a number of arithmetic operations proportional to n(n-1)(n-2)....4.3.2.1, i.e. n! The value of n! increases worse than exponentially as n increases. As we have said, this method is absolutely useless for machine computation unless n is very small. Nevertheless, it is a very useful theoretical result, and we can still use it for n=2 and n=3.

We now look at a Pascal program to solve a 2x2 system of equations by Cramer's Rule.

```
ax + by = e
cx + dy = f
```

```
program Cramer2;
{ Cramer's Rule for 2x2 system ax+bc=e
                               cx+dy=f }
var
  a,b,c,d,e,f : Real;
  x,y,Delta   : Real;
begin
  writeln('Enter values of a,b,e on one line');
  readln(a,b,e);
  writeln('Enter values of c,d,f on one line');
  readln(c,d,f);
  Delta:=a*d-b*c;
  x:=(e*d-b*f)/Delta;
  y:=(a*f-e*c)/Delta;
  writeln; writeln('Solution is:');
  writeln('x = ',x:12:8);
  writeln('y = ',y:12:8);
end.
```

Output

```
Enter values of a,b,e on one line
2 -5 -6
Enter values of c,d,f on one line
3 2 29

Solution is:
x =   7.00000000
y =   4.00000000
```

For the 3x3 system:

```
ax + by + cz = j
dx + ey + fz = k
gx + hy + iz = l
```

the following program uses Cramer's rule once again. This program is presented mainly to show you how complicated this formula becomes even for n=3. This complexity grows rapidly with n, and it is just not feasible to use the Cramer method for n greater than 3 or 4. Also, this program can be a useful one to use for comparison with the others given in this chapter (Gauss and Jacobi). As for the Gauss method, our implementation of Cramer's rule is rather simple-minded, since it will fall on its ear if the determinant (Delta in the program) is zero.

```
program Cramer3;
{ Cramer's Rule for 3x3 system -  ax+by+cz=j
                                  dx+ey+fz=k
                                  gx+hy+iz=l
and anything beyond this is madness! }
var
  a,b,c,d,e,f,g,h,i,j,k,l : Real;
  x,y,z,Delta             : Real;
  Deltax,Deltay,Deltaz    : Real;
begin
  writeln('Enter values of a,b,c,j on one line');
  readln(a,b,c,j);
  writeln('Enter values of d,e,f,k on one line');
  readln(d,e,f,k);
  writeln('Enter values of g,h,i,l on one line');
  readln(g,h,i,l);
  Delta :=a*(e*i-h*f)-b*(d*i-g*f)+c*(d*h-g*e);
  Deltax:=j*(e*i-h*f)-b*(k*i-l*f)+c*(k*h-l*e);
  Deltay:=a*(k*i-l*f)-j*(d*i-g*f)+c*(d*l-g*k);
  Deltaz:=a*(e*l-h*k)-b*(d*l-g*k)+j*(d*h-g*e);
  x:=Deltax/Delta;
  y:=Deltay/Delta;
  z:=Deltaz/Delta;
  writeln; writeln('Solution is:');
  writeln('x = ',x:12:8);
  writeln('y = ',y:12:8);
  writeln('z = ',z:12:8);
end.
```

Output

```
Enter values of a,b,c,j on one line
1 2 1 3
Enter values of d,e,f,k on one line
-1 5 1 -4
Enter values of g,h,i,l on one line
 3 4 -7 -19

Solution is:
x =   2.00000000
y =  -1.00000000
z =   3.00000000
```

10.3 An Indirect Method - Jacobi's Algorithm

With systems of linear equations, we find that some cases benefit from the application of iterative methods. These may lead to acceptable results faster than direct methods. An iterative method which generates **a sequence of approximations** is termed an **indirect method.** Jacobi's method is one of the simplest of such methods, and we now describe its operation.

Suppose that a system of n equations in n unknowns is given by:

Ax = **b**

The first step is to write the matrix **A** in the form **A**=**L**+**D**+**U** where **D** is a diagonal matrix, and **L** and **U** are lower and upper triangular matrices respectively, with zeroes on the diagonal. This operation is quite simple; we just split the original coefficient matrix **A**, into its diagonal, lower and upper "component matrices", whose remaining entries are just filled out with zeroes. An example will serve to illustrate these points.

$$\text{If A} = \begin{pmatrix} 4 & 16 & -17 \\ 8 & 22 & 19 \\ 14 & 6 & 5 \end{pmatrix}, \textit{ then}$$

$$\text{D} = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 22 & 0 \\ 0 & 0 & 5 \end{pmatrix} \quad \text{L} = \begin{pmatrix} 0 & 0 & 0 \\ 8 & 0 & 0 \\ 14 & 6 & 0 \end{pmatrix} \quad \text{U} = \begin{pmatrix} 0 & 16 & -17 \\ 0 & 0 & 19 \\ 0 & 0 & 0 \end{pmatrix}$$

Using the equation **A**=**D**+**L**+**U**, we can write our system **Ax**=**b** as

$(D+L+U)x = b$, or

$Dx + (L+U)x = b$

Rearranging, we have:

$Dx = -(L+U)x + b.$

This suggests the iterative sequence:

$x_{i+1} = -D^{-1}(L+U)x_i + D^{-1}b.$

To simplify our notation, we now write

$P = -d^{-1}(L+U)$, and $c = D^{-1}b$

We can now write our iteration as:

$x_{i+1} = Px_i + c.$

x_{i+1} is just the next estimate after x_i. The initial approximation x_0 could be set to the zero vector, or in fact any known reasonable approximation. This is the Jacobi iteration.

Example

We use the Jacobi method by hand to solve the system:

```
+4u - 1v + 0w = 2
-1u + 4v - 1w = 6
 0u - 1v + 4w = 2
```

Use x=0 as the first approximation.

Solution

Writing the system of equations in the form Ax=b, we have:

$$\begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix}\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix}$$

The matrices, L,D,U are then:

$$\text{D} = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix} \quad \text{L} = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix} \quad \text{U} = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}$$

Therefore, we can find the Jacobi iteration maxtrix **P**, and vector **c**:

$$P = \frac{1}{4}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \text{ and } c = \frac{1}{4}\begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix}$$

Our iteration is therefore:

$$x_{i+1} = \frac{1}{4}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} x_i + \frac{1}{4}\begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix}$$

This is a recurrence formula (or iteration sequence). It tells us how to get the next approximation from the current one. If we start with:

$$x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \text{ then clearly we have } \; x_1 = \frac{1}{4}\begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix}$$

Next, we get:

$$x_2 = \frac{1}{4}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix} + \frac{1}{4}\begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix} = \begin{pmatrix} 0.875 \\ 1.75 \\ 0.875 \end{pmatrix}$$

Check these calculations, and also see if you can work out x_3. The answers are given in the output table of the Jacobi program.

Convergence Properties Of The Jacobi Method

One might think that roundoff errors would accumulate in a method such as Jacobi's, however this is not the case. Since we are using an iterative method, any small errors such as these are normally very small compared to the increase in accuracy of our solution estimate from iteration to iteration. Roundoff errors just get swallowed up in the convergence process. Nevertheless, there is a limit to which we can iterate and expect any gain in accuracy, and this will usually be obvious by inspection of the computer output.

It is important also to realise that the Jacobi iteration is **not guaranteed to converge.** Convergence is likely if the coefficient matrix A is **strictly diagonally dominant.** This means that each diagonal element A[i,i] is in magnitude the largest in its row and column (i.e. |A[i,i]| is the largest in row i and column i).

With Jacobi's method, all we need to be able to do is add and multiply matrices after first forming the iteration matrix **P** and vector **c**. One method then of coding this technique in Pascal is to first write procedures for the matrix multiplication and addition operations. We haven't done this explicitly here, but it should be fairly clear from the code that the Jacobi iteration has been faithfully implemented.

```
program Jacobi;
{ This program uses Jacobi Iteration method to solve
a set of linear equations }
const
  n             = 3;
  MaxIterations = 20;
var
  a, p             : array[1..n,1..n] of Real;
  xold, xnew, b, c : array[1..n] of Real;
  i, j, ItNum      : Integer;
  InFile           : Text;

procedure ReadData;
begin
  Assign(InFile,'JACOBI.DAT');
  Reset(InFile);
  for i:=1 to n do begin
    for j:=1 to n do read(InFile,a[i,j]);
    readln(InFile,b[i])
  end;
  Close(InFile)
end;

procedure EchoData;
begin
  writeln; writeln('a,b');
  for i:=1 to n do begin
    for j:=1 to n do write(a[i,j]:8:3);
```

```
    writeln(b[i]:8:3)
  end
end;

procedure FormIterationMatrix;
begin
  for i:=1 to n do begin
    for j:=1 to n do p[i,j]:=-a[i,j]/a[i,i];
    c[i]:=b[i]/a[i,i];
    p[i,i]:=0
  end;
end;

procedure DumpIterationMatrix;
begin
  writeln; writeln('p,c');
  for i:=1 to n do begin
    for j:=1 to n do write(p[i,j]:8:3);
    writeln(c[i]:8:3)
  end
end;

procedure UpdateSolutionVector;
begin
  for i:=1 to n do begin
    xnew[i]:=c[i];
    for j:=1 to n do xnew[i]:=xnew[i]+p[i,j]*xold[j]
  end;
end;

procedure DumpSolutionVector;
begin
  for i:=1 to n do begin
    write(xnew[i]:16:8);
    xold[i]:=xnew[i]; {for next time around}
  end;
  writeln;
end;

begin { main }
  ReadData;
  EchoData;
  FormIterationMatrix;
  DumpIterationMatrix;
  for i:=1 to n do xold[i]:=0; writeln;
  for ItNum:=1 to MaxIterations do begin
    write(ItNum:4);
    UpdateSolutionVector;
    DumpSolutionVector
  end
end. { main }
```

```
 Output

a,b
  4.000  -1.000   0.000   2.000
 -1.000   4.000  -1.000   6.000
  0.000  -1.000   4.000   2.000
p,c
  0.000   0.250   0.000   0.500
  0.250   0.000   0.250   1.500
  0.000   0.250   0.000   0.500

  1      0.50000000      1.50000000      0.50000000
  2      0.87500000      1.75000000      0.87500000
  3      0.93750000      1.93750000      0.93750000
  4      0.98437500      1.96875000      0.98437500
  5      0.99218750      1.99218750      0.99218750
  6      0.99804688      1.99609375      0.99804688
  7      0.99902344      1.99902344      0.99902344
  8      0.99975586      1.99951172      0.99975586
  9      0.99987793      1.99987793      0.99987793
 10      0.99996948      1.99993896      0.99996948
 11      0.99998474      1.99998474      0.99998474
 12      0.99999619      1.99999237      0.99999619
 13      0.99999809      1.99999809      0.99999809
 14      0.99999952      1.99999905      0.99999952
 15      0.99999976      1.99999976      0.99999976
 16      0.99999994      1.99999988      0.99999994
 17      0.99999997      1.99999997      0.99999997
 18      0.99999999      1.99999999      0.99999999
 19      1.00000000      2.00000000      1.00000000
 20      1.00000000      2.00000000      1.00000000
```

10.4 Historical Note

Godfrey Harold Hardy (1877-1947)& Srinivasa Ramanujan (1887-1920)

G.H.Hardy is probably the most famous English mathematician of the 20th century, and his short but fruitful collaboration with the Indian Ramanujan is one of the most fascinating anecdotes in the long history of mathematical discovery. Hardy's field was pure mathematics in general, and number theory in particular. (Number theory is concerned, among other things, with the analysis and distribution of prime numbers throughout the set of natural numbers, and with other investigations into the set of natural numbers - often with the aid of higher calculus and special functions).

Ramanujan was born at Erode in the Tanjore district of southern India on December 22, 1887. He had studied mathematics only at high school, but on being given a mathematics text was able to teach himself the basics of functional analysis. He pursued his mathematics privately, and at age 23, he had made some original discoveries in number theory. He then decided to send some of his best work to the Englishman Hardy, who at the time was well-known as probably the greatest living mathematician - certainly in England. Try to picture the reaction of the great Hardy, as he read through the papers in this unsolicited collection from an unknown, untutored Indian man, who was really little more than a boy. We quote from Hardy's book entitled simply **Ramanujan.** Notice the transparent honesty of the man, as he is confronted by by the startling work of an unknown man ten years his junior:

> *"...It soon became obvious that Ramanujan must possess much more general theorems and was keeping a great deal up his sleeve ... [Some formulae] defeated me completely; I had never seen anything in the least like them before. A single look at them is enough to show that they could only be written down by a mathematician of the highest class. They must be true because, if they were not true, no one would have had the imagination to invent them. Finally ... the writer must be completely honest, because great mathematicians are commoner than thieves or humbugs of such incredible skill."*

Hardy arranged for a scholarship for Ramanujan to come to England in 1913, where the two collaborated until Ramanujan's untimely death in 1920 from tuberculosis. This union resulted in profound contributions to number theory.

Ramanujan had an amazing "friendship with the integers", as his colleague in England, J.E.Littlewood expressed his amazing intuitive skills in number theory. A very famous story has been passed down over the years in mathematics classrooms all around the world. Hardy relates it for us:

> *"I remember going to see him when he was lying ill at Putney. I had ridden in taxi-cab No. 1729, and remarked that the number seemed to me rather a dull one, and that I hoped it was not an unfavourable omen. 'No,' he replied, 'it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.'"*

Challenge Question - What are the two pairs of cubes that Ramanujan was referring to? (Since $10^3 + 10^3$ is greater than 1729, both numbers in each pair must be less than 10.)

10.5 Summary

- Systems of linear equations are very common in science, engineering and business problems.
- Methods for solving simultaneous equations using may be divided into two broad categories - **direct** and **indirect** methods.
- One of the most common direct methods is that of Gaussian elimination.
- Linear systems may be classified into those having:
  (i) unique solution
  (ii) no solution (overdetermined)
  (iii) infinitely many solutions (underdetermined)
- Another well-known direct method for solving linear equations is **Cramer's Rule.** Cramer's Rule expresses the solution for each unknown as the ratio of two determinants. It is not practical to use for more than 3 or 4 equations.
- One indirect method is that of Jacobi. In this method, the coefficient matrix is split into its lower and upper triangles and its diagonal. An **iteration matrix** is then formed and, after choosing an initial approximation, subsequent vectors are produced by matrix multiplication. Provided that the original coefficient matrix is suitable, this sequence of iterates will **converge** to the solution of the linear system.

10.6 Exercises

1. Solve the following sets of equations

(a)
$$\begin{aligned} 2x - 5y &= 7 \\ x &= 8 - 3y \end{aligned}$$

(b)
$$\begin{aligned} y &= 5 - 2x \\ 3y &= -x - 1 \end{aligned}$$

(e)
$$\begin{aligned} x + 2y &= 5 \\ x - y &= -1 \end{aligned}$$

(f)
$$\begin{aligned} 2.8x - 3.2 &= 1.6y \\ 3.3y - 4.2x &= 0.9y - 4.8 \end{aligned}$$

using the methods below:

(i) graphing,
(ii) Gaussian reduction by hand,
(iii) matrix by hand,
(iv) Gaussian reduction by computer,
(v) matrix by computer.

Make sure that you verify your answers.

2. Solve the following systems of equations by the following methods:

(i) Gaussian reduction by hand,
(ii) Jacobi's iteration by computer

Be sure to verify your answers.

(a)
$$\begin{aligned} 3x - 2y + z &= 6 \\ x - 8y + 2z &= -1 \\ -2x - 5y - z &= 5 \end{aligned}$$

(b)
$$\begin{aligned} x + 2y + z &= 3 \\ -x + y + z &= 0 \\ 3x + 4y - z &= -1 \end{aligned}$$

(c)
$$\begin{aligned} 4u - 3v - w &= 3 \\ 7u - 4v + 3w &= 1 \\ -u + 2v + 5w &= -4 \end{aligned}$$

(d)
$$\begin{aligned} x + y - z &= 3 \\ -2x + 7y - 4z &= -6 \\ -x + 2y - z &= 1 \end{aligned}$$

(e)
$$\begin{aligned} 0.5x + 2y &= 2.5 + 3z \\ 1.5x &= 0.5 + 2z \\ -y + 0.5z &= -1 - 0.5x \end{aligned}$$

(f)
$$\begin{aligned} 3x - 1 &= 2(y - z) + x \\ 4(z - 3y) &= y + 2x \\ (x + 2y)/2 &= 8(y+2) \end{aligned}$$

3. In the problems below form the system of equations in two or three unknowns and solve the system using any computer method.

   (a) Three times one number is nine times more than a second number, and the second is 31 more than the first. Find the numbers.

   (b) Regular ticket prices for a rock concert cost $7, but a radio station gave away $1 discount slips. If 3,000 people paid admission to the concert and a total of $20,079 was collected at the gate, how many people paid the full price and how many used discount slips? Ignore sneak-ins.

   (c) A man who has two bank accounts with a combined value of $6,000 plans to reinvest the money in the first account at 7% interest and to leave the money in the second account, where it is earning 8%. If he will receive $460 interest the first year, how much money does he have in each account?

   (d) Two small hamburgers and a small order of chips costs $2.62, and one hamburger and two orders of chips cost $2.18. How much would one hamburger and an order of chips cost?

   (e) A solution that is 10% alcohol by volume is to be mixed with a solution that is 20% alcohol by volume in order to make 16 ounces of of an 18% solution. How many ounces of each solution should be used?

   (f) A local manufacturer of bicycle frames estimates that the demand equation is 20n + p = 400 and that the supply equation is 20n = p - 80, where n is the number sold at price $p. How many frames should be made and how should they be priced to achieve equilibrium (supply = demand)?

   (g) The sum of three numbers is 2. The second number minus 3 is equal to the first number, and the sum of the first and third numbers is -5. Find the three numbers.

   (h) A man plans to invest $10,000. Part of this will be in tax-free city council bonds that pay 7% annually, part in taxable corporate bonds that pay 8% annually, and the rest in a taxable savings account that pays 6% annually. How much should he put into each type of investment if he wants to receive $360 interest per year from the taxable investments and $590 interest per year from the investments in bonds?

   (i) The height h, in metres, of a projectile above the ground t seconds after it is fired is given by an equation of the form $h = at^2 + bt + c$. Observations taken at 1, 2, and 3 seconds after firing showed that the projectile had heights of 4, 56, and 36 metres respectively. Find the values of a, b, and c in the equation for the projectile's height.

4. Solve the following systems of equations on the computer using Jacobi's iterative method. Be sure to verify your answers.

   (a)
   $$\begin{aligned} 4x + 0y + 1z + 1w &= 1 \\ 0x + 1y + 0z + 1w &= 2 \\ 0x + 1y + 4z + 0w &= 3 \\ 0x + 0y + 0z + 4w &= 4 \end{aligned}$$

   (b)
   $$\begin{aligned} 1a + 2b + 0c - 1d + 3e &= 1 \\ 4a + 3b + 5c - 6d - 1e &= -19 \\ 1a + 1b - 1c - 1d + 4e &= 0 \\ 2a + 4b + 1c - 8d + 0e &= -23 \\ 1a + 1b - 1c + 1d + 0e &= 8 \end{aligned}$$

   (c)
   $$\begin{aligned} 2a + 1b - 4c + 1d - 1e + 4f &= 15 \\ 1a - 3b - 1c + 2d - 1e + 1f &= -6 \\ 3a + 1b - 6c + 4d - 3e + 3f &= 11 \\ 1a + 0b + 1c - 1d - 1e + 3f &= -6 \\ 5a - 1b + 3c - 5d - 2e + 2f &= -14 \\ 1a - 1b + 1c + 1d + 1e + 1f &= 2 \end{aligned}$$

11 GEOMETRY & TRIGONOMETRY

11.1 Triangles

Euclidean geometry has traditionally been very widely taught in primary and secondary schools not only because of its practical use in the world at large, but also for its proven worth as a vehicle to teach the development of logical reasoning in the minds of young people.

We consider here problems of Euclidean geometry which are amenable to solution on a computer, and are of an immediate "practical" nature. These include the solution of plane triangles, the computation of perimeters and areas of common plane figures such as triangles and polygons, and related problems. In the course of this work, we shall be required to make use of standard elementary results such as Pythagoras' Theorem, and of elementary trigonometry. Students should therefore be reasonably familiar with the definitions of the trigonometric ratios, and results such as the sine and cosine rules.

Plane triangles are the simplest kinds of **polygons,** and their analysis, i.e. computation of sides, angles, and areas forms a very important part of elementary geometry and trigonometry. Indeed, the word **trigonometry** means quite literally "triangle measurement". In many areas of science and mathematics it is required to be able analyse triangles. Applications which come to mind immediately are Surveying, Astronomy, Navigation, and Civil Engineering.

When we study triangles, we usually classify them into acute, or obtuse, equilateral, isosceles, scalene, right-angled or non-right-angled, an so on. We look at such things as congruence and similarity, and geometrical results such as sum of angles equals 180 degrees. The logical progression from here is to consider how we can calculate the angles and sides of a triangle. In order to do this, we need some algebraic relationship between angles and sides. This leads naturally to the study of **trigonometry**, where we use the fact that the ratios of sides are constant in similar triangles, that is, each ratio does not depend on the size of the triangle, but merely its "shape" (the angles). Since they are easiest to analyse, right-angled triangles are used to define the trigonometric ratios. The trigonometric ratios give us exactly what we need, namely an algebraic relationship between the sides and angles of a right-triangle. If the triangle we wish to analyse is not a right-angled triangle, then we need not worry; we simply divide it into two triangles which are.

We now turn to consider some simple algorithms for the analysis of triangles. They will be based on simple trigonometry and results from coordinate geometry such as the distance formula.

11.2 Perimeter Of A Triangle

Suppose we are given the cartesian coordinates of the three vertices of a plane triangle. The distance formula may be used for each side and the results added to obtain the perimeter. We present a Pascal program to do this. This is quite a simple program so no further comment is really necessary regarding the prime purpose of the program. We remark however that a test is required to check that the points are not collinear, since no triangle is formed in this case. In fact, this is by far the hardest part of the program! This is not an unusual phenomenon in computing; very often the mere validation of the input data and "pretty" presentation of the output will consume more time than even a very careful and skilful design of the underlying algorithm for the problem at hand.

It is a symptom of poor algorithm design when a program fails because of some error condition not anticipated by its designer. We have at least realized that the collinearity problem must be catered for in our program, but its resolution may be trickier than we think!

How do we test if three points A, B, C are collinear? We can try to find the gradients of say, the line segments AB, and AC. If these differ then we can conclude that the points are not on the same straight line - or can we? What if A=(1,1); B=(1,3); C=(1,2)? Suddenly we find that AB has no slope, but the point are certainly collinear (you should check this!). This is a real nuisance! We thought we had a good idea but a nasty counterexample has arisen. (In fact, counterexamples are of great benefit to us and we should make the most of them. They often save from following false paths of reasoning and wasting a lot of time.) To make the most of a seemingly negative result from a counterexample, we should generally ask why this particular instance contradicts the proposition (statement) we had perhaps thought to be true. In this case it is not too hard, since no slope will exist for points with the same x-values (abscissae). OK, then we shall just have to be a bit more cunning. Suppose we had the equations of the (infinite) straight-lines containing the segments AB and AC. Is this a help to us? Yes, since the points A, B, and C are collinear if and only if the two (infinite) straight-lines coincide. We must be careful however not to use division, unless we are absolutely certain we are not attempting to divide by zero.

Suppose our points are $A = (x_1, y_1), B = (x_2, y_2), C = (x_3, y_3).$

The equation of the straight-line containing the segment AB can be written (notice we do not use division):

$$(y - y_1)(x_2 - x_1) = (y_2 - y_1)(x - x_1).$$

If C (x_3, y_3) satisfies this equation then we have collinearity. Thus, the condition for collinearity is obtained by substituting the coordinates of C for x and y, i.e.

$$(y_3 - y_1)(x_2 - x_1) = (y_2 - y_1)(x_3 - x_1).$$

Another, perhaps less tricky method is to check if one of the sides of the triangle is the sum of the other two. This is a limiting case of the triangle inequality which we examine in connection with our algorithm for finding the angles of a triangle given the sides.

```
program TriPerim;
{This program finds the perimeter of a triangle given its vertices}
var
  i, n       : Integer;
  x1, x2, x3 : Real;
  y1, y2, y3 : Real;
  Perimeter  : Real;

function Dist(x1,x2,y1,y2 : Real): Real;
begin
  Dist:=Sqrt(Sqr(x2 - x1) + Sqr(y2 - y1))
end;

function Collinear(x1,x2,x3,y1,y2,y3 : Real) : Boolean;
begin
  Collinear:=((y3 - y1)*(x2 - x1) = (y2 - y1)*(x3 - x1))
end;

begin  { main }
  writeln('Type two real numbers/line, separated by spaces');
  write('Coordinates of A = '); readln(x1,y1);
  write('Coordinates of B = '); readln(x2,y2);
  write('Coordinates of C = '); readln(x3,y3);
  writeln;
  if not (Collinear(x1,x2,x3,y1,y2,y3)) then begin
    Perimeter:=Dist(x1,x2,y1,y2)+Dist(x2,x3,y2,y3)+Dist(x3,x1,y3,y1);
    writeln('Perimeter = ',Perimeter:10:5)
  end
  else writeln('Points are collinear; no triangle is formed')
end. { main }
```

Output

```
Type two real numbers/line, separated by spaces
Coordinates of A = 0 0
Coordinates of B = 3 0
Coordinates of C = 3 4

Perimeter =   12.00000
```

11.3 Solution Of Triangles

Right Angled Triangles

If we are given the sides of a right-angled triangle, it is a fairly simple matter to calculate the angles of the triangle. We already know that one is 90 degrees, and the other may be determined from simple trigonometry. For example, if we have our old friend the 3,4,5 triangle, then the smaller acute angle, α, is given by:

$$\tan\alpha = \frac{3}{4} = 0.75,$$

from which we may obtain α in degrees or radians from tables or a scientific calculator. We may use the computer as a scientific calculator by writing a Pascal program just to evaluate the inverse tangent function. This function returns the answer in radians, so if we want degrees, we must multiply by $180/\pi$. Once we have this angle, we simply subtract it from 90 degrees, i.e. find its complement, to obtain the other acute angle of the right triangle. An almost trivial program to find this angle in radians, given its tangent is shown below.

```
program ArcTangent;
begin
  write('Inverse tangent of 0.75 = ');
  writeln(arctan(0.75):10:8)
end.
```

```
Output

Inverse tangent of 0.75 = 0.64350111
```

Non-Right Triangles

It is an unfortunate fact of life that most triangles encountered in practical problems do not contain a right-angle! This being the case, we must develop an algorithm which allows us to find the angles of arbitrary triangles once we are given the sides. The strategy is simply to drop a perpendicular from one vertex (corner) to the opposite side. In this manner, two right triangles are obtained at the base of the perpendicular, and we may solve each of the two right-triangles created by the method given just above. This construction is in fact the standard one which is used to derive the so-called **Cosine Rule** and, it is precisely the cosine rule that we use here to solve our triangle for the three angles.

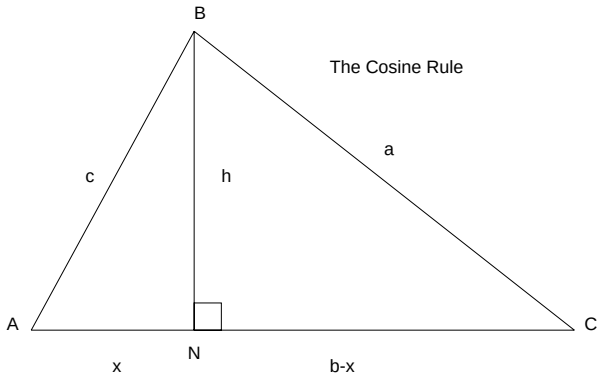


Figure 11.1 The Cosine Rule

With reference to Figure 11.1, we apply the Theorem of Pythagoras to the right-triangles **ABN** and **CBN** to get:

$a^2 = h^2 + (b-x)^2$ and $c^2 = h^2 + x^2$ respectively.

Eliminating h^2 from these equations, we obtain:

$$a^2 = c^2 - x^2 + (b-x)^2$$

$$= c^2 + b^2 - 2bx$$

Since x = c cos A, from triangle ABN, we finally arrive at:

$$a^2 = b^2 + c^2 - 2bc\cos A$$

This last equation is the **Cosine Rule.**

Using The Cosine Rule To Solve A Triangle

The program that we shall develop will accept three positive real numbers, which purport to be the lengths of the sides of a plane triangle. The first thing we must do is to ensure that this is in fact the case. Can any three positive real numbers represent the sides of a plane triangle? If you think that the answer to this question is **yes**, then consider the triple 1,1,1000. Is it possible to have a triangle with these sides? Of course not. What property of a triangle is it that prevents this? It is

the simple fact that **any side of any triangle must be shorter than the sum of the other two.** This is called the **triangle inequality.** Algebraically, if the sides have length a, b, c then **all three** of the following conditions must be satisfied:

$a < b + c$,
$b < c + a$, and
$c < a + b$.

Notice that if a=1, b=1, c=1000, as above, the first two of these triangle inequalities are satisfied, but the third fails. We cannot shortcut by requiring only that 1 or 2 of them be true - all three are always required. What about the condition that all numbers are positive? It can be shown that these seemingly extra conditions are in fact consequences of the three triangle inequalities.

Our strategy then is to first check that the three sides satisfy the three triangle inequalities, and then to calculate the cosines of each angle. On the surface of it, we could then find each angle by using the inverse cosine function, but there are a couple of snags to prevent us from using this approach. First, a very pragmatic consideration is that the common Pascal compilers, including Turbo v3.0, do not include an inverse cosine function. We could write our own using the methods of our chapter on sequences and series, or do a similar thing for inverse sine, however there are also some more fundamental objections to this technique. The simple fact is that one angle of the triangle may be greater than a right-angle. Since the sine function is not one-to-one on $(0,\pi)$ and the inverse cosine function is conventionally defined on (-1,1) mapping onto $(-\pi/2,\pi/2)$, things quickly get messy. In this case it is much simpler to obtain both sine and cosine for each angle, and apply the inverse tangent function to the ratio. One further snag concerns the case where one angle is 90 degrees (or close to 90 degrees), since then cosine will be zero, and we will be once again confronted with the possibility of attempted division by zero. We take care of this possibility by a simple test. A program to solve a triangle, given the lengths of its sides is given below.

```
program Triangle;
{ The area and angles of a triangle are determined given the
lengths of the sides.  a, b, and c are the lengths of the sides
of a plane triangle in any convenient units.  We test for a+b>c
& b+c>a & c+a>b  (triangle inequalities).  We also require that
a>0 & b>0 & c>0.  However the last three are redundant since
they are implied by the first three. }
var
  a, b, c                  : Real;
  AngleA, AngleB, AngleC   : Real;
  s, Area, Sum             : Real;

function Angle(OppSide,Adj1,Adj2 : Real) : Real;
const
  Pi=3.14159265;
var
  CosAngle, SinAngle, Temp : Real;
begin
{Computes in degrees the angle of triangle associated with OppSide}
  CosAngle:=(Adj1*Adj1 + Adj2*Adj2 - OppSide*OppSide)/Adj1/Adj2/2.0;
  SinAngle:=Sqrt(1.0 - CosAngle*CosAngle);
  if CosAngle=0 then Angle:=90.0
    else begin
      Temp:=Arctan(SinAngle/CosAngle)*180.0/Pi;
      if Temp<0.0 then Angle:=Temp + 180.0 else Angle:=Temp
    end
end;

begin { main }
  write('Length of side a  ?  '); readln(a);
  write('Length of side b  ?  '); readln(b);
  write('Length of side c  ?  '); readln(c);
  if ((a+b>c) and (b+c>a) and (c+a>b)) then begin
    s:=(a+b+c)/2;
    Area:=Sqrt(s*(s-a)*(s-b)*(s-c));
    writeln;
    writeln('The area is ', Area:10:4, ' square units');
    AngleA:=Angle(a,b,c);
    AngleB:=Angle(b,c,a);
    AngleC:=Angle(c,a,b);
    writeln('Calculated angles are :  ');
    writeln('Angle A = ',AngleA:10:6,' degrees');
    writeln('Angle B = ',AngleB:10:6,' degrees');
```

```
        writeln('Angle C = ',AngleC:10:6,' degrees');
        Sum := AngleA + AngleB + AngleC;
        writeln('Sum of calculated angles = ',Sum:10:6,' degrees');
        if Abs(Sum-180.0)>1.0E-5 then writeln('Angular discrepancy to
5D');
      end
      else
        writeln('Values input do not form a triangle')
    end. { main }
```

Output

```
Length of side a ?  1.0
Length of side b ?  1.0
Length of side c ?  1.0

The area is     0.4330 square units

Calculated angles are:

Angle A =  60.000000 degrees
Angle B =  60.000000 degrees
Angle C =  60.000000 degrees
Sum of calculated angles = 180.000000 degrees
```

11.4 Area Of A Triangle

The area of a general plane triangle is given by Heron's formula:

$$Area = \sqrt{S(S-a)(S-b)(S-c)},$$

where a, b, c are the sides, and S is the semi-perimeter, i.e.

$$2S = (a+b+c).$$

We have included Pascal statements in the previous program illustrating the application of Heron's formula.

11.5 Polygons

A triangle is the simplest kind of **polygon,** (an n-sided plane figure) in that it has the smallest possible number of sides (n=3). We may find perimeters and areas of polygons in a similar manner to the methods that we used for triangles, however we first need to iron out a few matters with respect to how we are going to describe these plane figures. We must find an (unambiguous) way to describe a polygon to the computer so that it "knows" just exactly what we mean. The basic idea is to give a set of ordered pairs which are to represent the vertices of the polygon we are interested in. We have a problem however if n>4, since just listing a set of vertex coordinate pairs unfortunately does not uniquely define a polygon. For example, consider the set:

S = { A(0,2), B(2,0), C(8,0), D(10,2), E(4,1) } .

If we plot these five points on graph paper, we can see that a pentagon (5-sided polygon) ABCDEA can be formed, but so can another pentagon ABECDA. You should verify this fact on paper before proceeding. Are any others possible? Figures 11.2 and 11.3 illustrate these two cases we have mentioned. Which pentagon does the set define? Strictly speaking, the set of pairs in fact does not define **any** pentagon, because of the ambiguity just discussed. We have to tighten up our means of defining polygons a little, so as to make the definition unambiguous. A computer will seem to do "funny" things when presented with ambiguities, but its behaviour is always predictable since it is a deterministic machine. It just may not appear to us that it is behaving properly when it is doing what we **told** it to do. The fact that it may not be what we **wanted** it do is of no concern to the machine. The machine may solve one interpretation of the problem, or it may encounter a run-time error such as division by zero - it all depends on the algorithm used to solve the problem and the precise input data.

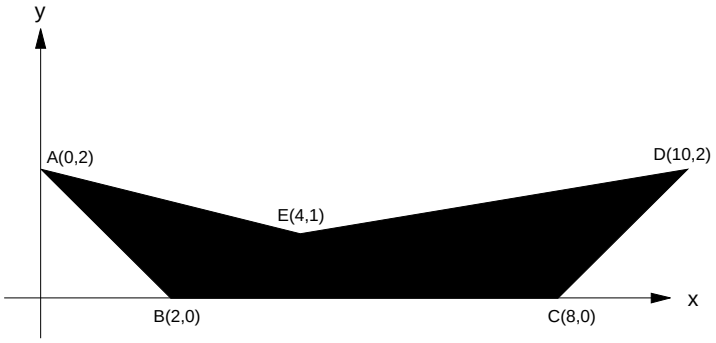


Figure 11.2 Pentagon ABCDEA

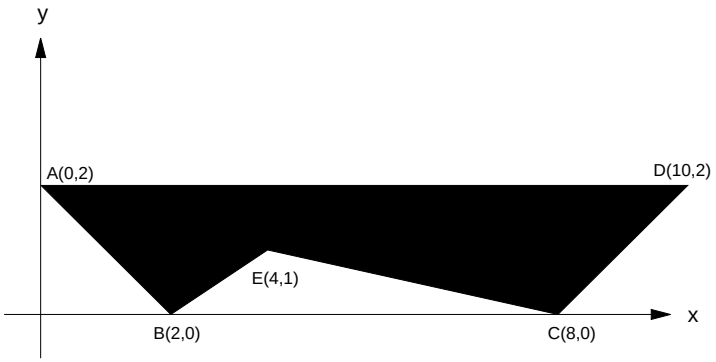


Figure 11.3 Pentagon ABECDA

If we insist that the points are to be interpreted as the coordinates of vertices in the order presented, then our problem disappears. We now have a **sequence** of pairs rather than a **set**. Our program to find the perimeter of the polygon defined in this way will simply assume that the sequence of coordinate pairs presented to it does in fact represent the set of vertices of a valid polygon. The algorithm for verification of this is too intricate to be discussed here. A program to determine the perimeter of a polygon, given the coordinates of its vertices is shown below. Notice the application of the distance formula to this problem.

```
     program PolPerim;
     { This program determines the perimeter of a polygon given its ver-
tices }
     var
       i, n  : Integer;
       x, y  : array[1..10] of Real;
       Perim : Real;

     function Dist(x1,x2,y1,y2 : Real): Real;
     begin
       Dist:=Sqrt(Sqr(x2 - x1) + Sqr(y2 - y1))
     end;

     begin  { main }
       repeat
         write('How many sides in polygon (3 or more) ? ');
         readln(n);
         if n<3 then writeln(Chr(7),'3 sides or more please !');
       until n>=3;
       writeln('Please enter vertex coordinates 2 per line');
       for i:=1 to n do readln(x[i],y[i]);
       Perim:=Dist(x[n],x[1],y[n],y[1]);
       for i:=1 to n-1 do Perim:=Perim + Dist(x[i],x[i+1],y[i],y[i+1]);
       writeln('Perimeter = ',Perim:10:5)
     end. { main }
```

```
Output

How many sides in polygon (3 or more) ? 5
Please enter vertex coordinates 2 per line
0 2
2 0
8 0
10 2
4 1
Perimeter =   21.86272
```

Area of a Polygon

To calculate the area of a convex polygon, our strategy will be to divide it up into triangles, calculate the area of each using Heron's formula, then add the results (a convex polygon is one in which the straight-line segment joining any two interior points remains entirely inside the polygon). To construct the triangles, we need to choose a convenient point, radiate lines from this point to all vertices, and process the triangles that we obtain as described above. Since we are assuming the polygon to be convex, we may choose our point to be the first vertex, and radiate to all other vertices. Convexity then guarantees that no triangles will overlap, and that the sum of their areas will yield the area of the n-gon. More precisely:

Calling the points $(x_1, y_1) \ldots (x_n, y_n)$, *we have*

$$\text{Total Area} = \sum_{2}^{n-1} a_i$$

where $a_i =$ area of triangle with vertices $(x_1, y_1), (x_i, y_i), (x_{i+1}, y_{i+1})$.

Notice that this sum contains one term for n=3 (triangle), as expected. A program to calculate the area of a polygon is given below. Notice once again our use of the distance formula in this problem.

```
program PolyArea;
{ This program determines the area of a polygon given its vertices }
var
  i, n                    : Integer;
  x, y                    : array[1..10] of Real;
  TriangleArea, PolyArea  : Real;
  a, b, c, s              : Real;

function Dist(x1,x2,y1,y2 : Real): Real;
begin
  Dist:=Sqrt(Sqr(x2 - x1) + Sqr(y2 - y1))
end;

begin { main }
  repeat
    write('How many sides in polygon (3 or more) ? ');
    readln(n);
    if n<3 then writeln(Chr(7),'3 sides or more please !');
  until n>=3;
  writeln('Please enter vertex coordinates 2 per line');
  for i:=1 to n do readln(x[i],y[i]);
  PolyArea:=0;
  for i:=2 to n-1 do begin
    a:=Dist(x[1],x[i],y[1],y[i]);
    b:=Dist(x[i],x[i+1],y[i],y[i+1]);
    c:=Dist(x[i+1],x[1],y[i+1],y[1]);
    s:=(a + b + c)/2.0;
    TriangleArea:=Sqrt(s*(s - a)*(s - b)*(s - c));
    PolyArea:=PolyArea + TriangleArea;
  end;
  writeln('Area of Polygon = ',PolyArea:10:5)
end.  { main }
```

```
Output

How many sides in polygon (3 or more) ? 5
Please enter vertex coordinates 2 per line
 0 2
 2 0
 8 0
10 2
 4 1

Area of Polygon =   21.00000
```

11.6 Historical Note

John Thompson & Walter Feit

Early this century the English mathematician Burnside postulated that all groups of odd order are solvable. It was not until 1963 that the two American mathematicians John Thompson and Walter Feit finally proved this result, which of course up until that time, was only conjecture.

For their proof, which required over 250 pages, they received the Cole Prize of the American Mathematical Society. John Thompson also was recipient of the Fields Medal at the International Congress of Mathematicians in Nice in 1970. As there is no Nobel Prize in mathematics, this medal is generally acknowledged as its equivalent.

An interesting story surrounds the non-existence of a Nobel Prize for mathematics. It is said that Alfred Nobel, because of a personal dislike of the Swedish mathematician Mittag-Leffler, insisted that there should not be a Nobel Prize for mathematics. Apparently he was concerned that if there were one, then Mittag-Leffler might have won it.

11.7 Summary

- Geometry and trigonometry have considerable practical use in the world at large. For example, calculations involving perimeters, areas, and volumes are very common in many walks of life. Analysis of triangles is a particularly frequent task in many professional areas.
- Polygons are plane figures which, for the calculation of areas, can be considered as compositions of triangles.
- In finding the perimeter of a triangle using the computer, the hardest task in fact is to test for the existence of a triangle, i.e. three points may be collinear and either parallel or not parallel to the axes. The best way of doing this is to determine the equation of the line segment formed by the first two points, and then see if the other point satisfies this equation. If it does, then the points are collinear.
- In solving triangles, it should be remembered that Turbo Pascal does not have the inverse sine and cosine functions. This limitation can be overcome by using the inverse tangent function.
- In using the cosine rule on a computer, care must be taken to ensure that input data for lengths of sides and angles are legitimate, i.e. the triangle inequality rule must be obeyed.
- The area of a triangle may be calculated using Heron's rule, i.e.

  $$Area = \sqrt{S(S-a)(S-b)(S-c)},$$

  where a, b, c are the sides, and S is the semi-perimeter. All three triangle inequalities must be obeyed.
- To define a particular polygon, not only must a set of points (ordered pairs) for the vertices be given, but also the order in which they are joined.

11.8 Exercises

1. Show that if the three triangle inequalities are true, then a>0, b>0, and c>0. *Hint:* assume that a<0 and get a contradiction by comparing the last two triangle inequalities.
2. Test the triangle perimeter program out for an equilateral triangle, and check the results. Remember that a program should always be tested first with data for which you know the answer (if possible).

3. Test the triangle area program with the following data sets, which represent lengths of sides:

   (i) 3, 4, 5 (ii) 1, 1, 1 (iii) 5, 12, 13

   Verify that the answers are correct in each case.

4. Test the polygon perimeter program for a triangle and a square and check the results.

5. Test the polygon area program for a triangle and a square and check the results.

6. Show that the sequence of pairs (0,0), (1,0), (0,1), (1,1) does **not** represent a valid quadrilateral when traversed in the order presented. If we rearrange them, of course, we get a rectangle. Feed this data to the POLPERIM program and check if the answer is correct. Can these results be generalised?

7. Extend the program ArcTangent to find the angle in degrees instead of radians.

8. Extend the program ArcTangent further to allow the user to input the tangent value at run time, and then compute and display the angle in degrees for him.

12 COMPUTER GRAPHICS

12.1 What Is Computer Graphics ?

Modern microcomputers on which students will be studying computer mathematics will generally have a reasonably good graphics capability. By this we mean that it will be possible to display non-textual images of reasonable quality on the video screen with appropriate programming. These images can be "pictures" of everyday objects such as trees or houses, and may even be animated if the computer processor and screen display are fast enough to rewrite images so as to give the impression of continuous movement. Computer games generally involve some very sophisticated programming techniques in computer graphics in order to take advantage of the computer's fast processor and high-resolution screen.

In a mathematics course such as this one, we can take advantage of the graphical capability of our computers to plot the graphs of mathematical functions and relations. This may not seem quite as interesting as computer games, however there is really no reason why keen students cannot extend their programming skills to simple computer games involving animation. Some mathematical ability will certainly be required to do this, and a lot of hard work. We have not attempted to cover such advanced topics in this introductory text, but interested students may obtain pointers from their teachers for further references on this fascinating subject.

Let's begin now with some of the basics of computer graphics. The first program that we shall use is one which is supplied on the program diskette with this textbook. It allows us to plot a vast range of functions and is called FUNPLOT.PAS. Another program called PARAPLOT.PAS is provided to enable us to plot functions and relations represented by parametric equations.

Function Plotting

Before using either of these programs, we need to discuss the general principles of computer function-plotting so that you may later be able to modify these programs, or even write your own from the beginning. We have written the programs with the assumption that they will be running under Turbo Pascal on an IBM PC or compatible microcomputer. This enables us to ensure that the programs work before you try them out.

Step	PRINCIPLES OF FUNCTION PLOTTING
Step 1.	The x-values should be scaled by a factor equal to the ratio of the screen width (640) to the length of the desired interval (b-a).
Step 2.	Once the x scale factor has been determined, we should step through each of the 640 values of x that will map onto [a,b] and calculate the function value at each of these points. In this way we can determine the extreme values of the desired function on the interval [a,b]. These are simply the minimum and maximum values attained by the function on the interval [a,b].
Step 3.	The difference of the max and min values gives us the range of the function on [a,b].
Step 4.	Since the y range is now known, a scale factor for the y values equal to the screen height (200) divided by the range can be computed.
Step 5.	The position of the origin on the screen may now be found and the coordinate axes drawn on the screen.
Step 6.	Re-compute the function values as in step 2 and plot them on the screen.

Table 12.1 Principles Of Function Plotting

On the IBM PC, high-resolution graphics mode gives us 640 pixels (points on the screen) horizontally, by 200 pixels vertically. The top left-hand corner of the screen has screen coordinates (xs,ys)=(0,0); where x increases horizontally to the right, and y increases vertically downward. Thus, the bottom right-hand corner of the screen has screen coordinates (xs,ys)=(639,199). Turbo Pascal provides a procedure (Plot) to plot an individual point (pixel) at any of the 640*200=128,000 points on the screen. There is also a procedure (Draw) to draw a straight-line between any two of these points. This is all fine, except that before we try to plot an arbitrary function on our computer screen, we must map the set of ordered pairs with which we wish to represent our function onto the set of pixels on the screen. It is no good asking the machine to plot the screen point (300,300), since that point does not exist! The general procedure to be followed when one wishes to plot a function y=f(x) over an interval [a,b] is given in Table 12.1.

These steps have been followed in the programs below. Pascal procedures have been written for the major tasks in each program, and the student should study each program carefully.

Program To Plot y=f(x)

```
program FunPlot;
{This program plots the curve of a function given in cartesian form}
const
  xsmax = 640;                {max screen x coordinate (hor) }
  ysmax = 200;                {max screen y coordinate (vert)}
var
  a, b           : Real;      {min and max of x              }
  x, y           : Real;      {original coordinates          }
  xs, ys         : Integer;   {screen coords; top left =(0,0)}
  ymax, ymin     : Real;      {min and max of f(x) on [a,b]  }
  xscale, yscale : Real;      {scale factors for (x,y) to (xs,ys)}
  xs0, ys0       : Integer;   {screen coords of (x,y) origin  }

procedure Wait;
begin    {Wait}
  gotoxy(57,25);
  write('Press A Key To Continue');
  repeat until keypressed
end;     {Wait}

function f(x : Real) : Real;
begin
  f:=sin(3.0*x)*exp(-x/5.0)
end;

procedure Initialize;
begin    {Initialize}
  HiRes;
  write('  Value of a = '); read(a);   {get endpts of interval }
  write('  Value of b = '); read(b);
  xscale:=xsmax/(b-a);                 {scale factor for x axis}
end;     {Initialize}                   {reqd first in FindExtrema}

procedure FindExtrema;                 {find ymax and ymin on [a,b]}
begin    {FindExtrema}
  writeln('  Searching For Max and Min');
  xs:=0; ymin:=1.0E20; ymax:=-ymin;
  while xs<=xsmax do begin
    gotoxy(77,1); write(xs:3);
    x:=a+xs/xscale;
    y:=f(x);
    if y>ymax then ymax:=y;
    if y<ymin then ymin:=y;
    xs:=xs+1
  end;
  yscale:=ysmax/(ymax-ymin);           {compute y scale factor}
  {write('  YMIN = ',ymin:10:6);
  write('  YMAX = ',ymax:10:6);}
end;     {FindExtrema}
```

```
procedure DrawAxes;
begin    {DrawAxes}
  xs0:=trunc(-a*xsmax/(b-a));          {screen x coord of origin}
  ys0:=trunc(ymax*ysmax/(ymax-ymin));  {screen y coord of origin}
  draw(0,ys0,xsmax,ys0,1);             {draw x - axis           }
  draw(xs0,0,xs0,ysmax,1);             {draw y - axis           }
end;     {DrawAxes}

procedure PlotFunction;
begin    {PlotFunction}
  xs:=0;
  while xs<=xsmax do begin
    x:=a+xs/xscale;
    y:=f(x);
    ys:=trunc((ymax*yscale)-y*yscale);
    plot(xs,ys,1);
    xs:=xs+1
  end;
end;     {PlotFunction}

begin  { main }
  Initialize;
  FindExtrema;
  DrawAxes;
  PlotFunction;
  Wait
end.  { main }
```

12.2 Plotting Of Parametric Equations

The equations of many curves of interest are very awkward to express in the standard **cartesian form y=f(x)** that we have used in the previous section. It turns out that a **parametric** form is usually adequate to express most of the classical polar curves, such as lemniscates, spirals, cardioid etc., and also a number of the more "modern" graphs that are used in various branches of science and engineering. If we regard the real variable t as a parameter, then the pair of equations:

$$x = f(t),$$

$$y = g(t), \text{ for } t \in [t1, t2]$$

defines a path (a relation between x and y; the path is generally **not a function),** since different values of t could produce the same value of x but different values of y. This path or **locus** may be plotted either by hand or by machine, simply by substituting a sequence of values of t (usually an arithmetic progression in the case of a computer plot) into the two functional equations and then plotting.

Example Of Parametric Curve

A good example of a curve which is easily expressible in both cartesian and parametric forms is the **circle.** The cartesian and polar (parametric) equations for the circle are respectively:

$$x^2 + y^2 = a^2 \quad (1)$$

$$r = a \quad (2)$$

Notice that the polar form of the equation to a circle of radius **a** is particularly simple (r=a). Since the equations relating cartesian and polar coordinates are:

$$x = r\cos\theta; \qquad y = r\sin\theta \quad (3)$$

we can show that the two forms for the circle are equivalent by substituting r=a from equation (2) into the pair of equations labelled (3). This yields:

$$x = a\cos\theta; \qquad y = a\sin\theta \quad (4)$$

Squaring each of the equations in (4) and adding yields:

$$x^2 + y^2 = a^2\cos^2\theta + a^2\sin^2\theta$$

$$= a^2(\cos^2\theta + \sin^2\theta)$$

$$= a^2, \quad \text{since} \quad \cos^2\theta + \sin^2\theta = 1.$$

The following program (PARAPLOT.PAS) contains parametric equations to plot a Lissajous figure (shown in Figure 12.1), which should be recognised by the majority of Australian students; at least those whose family has a television set!

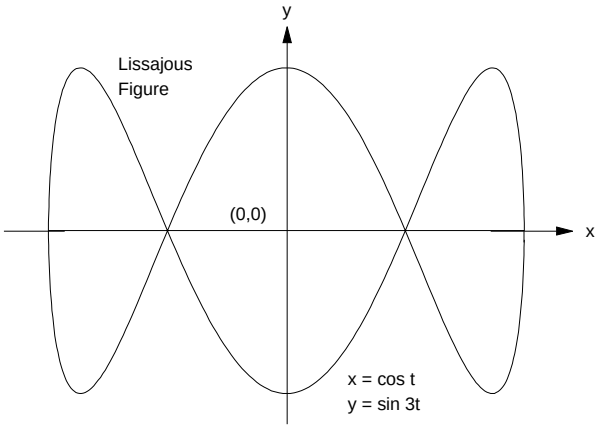


Figure 12.1 The Graph Of x=cos t, y=sin 3t - A Lissajous Figure

Program To Plot x=f(t), y=g(t)

```
program ParaPlot;
{This program plots the curve of a function given in parametric form}
const
  xsmax = 640;                {max screen x coordinate (hor) }
  ysmax = 200;                {max screen y coordinate (vert)}
var
  t1, t2          : Real;     {min and max of parameter t    }
  t, x, y         : Real;     {original coordinates          }
  xs, ys          : Integer;  {screen coords; top left =(0,0)}
  ymax, ymin      : Real;     {min and max of f(x) on [t1,t2]}
  xmax, xmin      : Real;     {min and max of   x  on [t1,t2]}
  xscale, yscale : Real;      {scale factors for (x,y) to (xs,ys)}
  xs0, ys0        : Integer;  {screen coords of (x,y) origin }
  deltat          : Real;
  n               : Integer;

procedure Wait;
begin    {Wait}
  gotoxy(57,25);
  write(Chr(7),'Press A Key To Continue');
  repeat until keypressed
end;     {Wait}

function f(t : Real) : Real; { x=f(t), y=g(t) parametric equations}
begin
  f:=cos(t)
end;

function g(t : Real) : Real;
begin
  g:=sin(3.0*t)
end;

procedure Initialize;
begin    {Initialize}
  ClrScr;
  write('t1= '); read(t1);  {initial & final vals for param}
  write('  t2= '); read(t2);
  write('  No. of points = '); read(n);
  deltat:=(t2-t1)/n;
end;     {Initialize}

procedure FindXExtrema;       {xmax and xmin for t in [t1,t2]}
begin    {FindXExtrema}
  gotoxy(1,25); write(' Searching For XMax & XMin');
  xmin:=1.0E20; xmax:=-xmin;
  t:=t1;
  while t<=t2 do begin
    gotoxy(40,1); write('t = ',t:6:2);
    x:=f(t);
```

```
    if x>xmax then xmax:=x;
    if x<xmin then xmin:=x;
    t:=t+deltat
  end;
  xscale:=xsmax/(xmax-xmin);     {compute the x scale factor  }
  {write('  XMIN = ',xmin:10:6);
  write('  XMAX = ',xmax:10:6);}
end;     {FindXExtrema}

procedure FindYExtrema;     {ymax and ymin on for t in [t1,t2]}
begin    {FindYExtrema}
  gotoxy(1,25); write(' Searching For YMax & YMin');
  ymin:=1.0E20; ymax:=-ymin;
  t:=t1;
  while t<=t2 do begin
    gotoxy(40,1); write('t = ',t:6:2);
    y:=g(t);
    if y>ymax then ymax:=y;
    if y<ymin then ymin:=y;
    t:=t+deltat
  end;
  yscale:=ysmax/(ymax-ymin);           {compute the y scale factor  }
  {write('  YMIN = ',ymin:10:6);
  write('  YMAX = ',ymax:10:6);}
end;     {FindYExtrema}

procedure DrawAxes;
begin    {DrawAxes}
  xs0:=trunc(-xmin*xsmax/(xmax-xmin)); {get screen x coord of origin}
  ys0:=trunc(ymax*ysmax/(ymax-ymin));  {get screen y coord of origin}
  draw(0,ys0,xsmax,ys0,1);             {draw x - axis               }
  draw(xs0,0,xs0,ysmax,1);             {draw y - axis               }
end;     {DrawAxes}

procedure PlotFunction;
begin    {PlotFunction}
  t:=t1;
  while t<=t2 do begin
    x:=f(t);
    y:=g(t);
    xs:=trunc((xmax*xscale)-x*xscale);
    ys:=trunc((ymax*yscale)-y*yscale);
    plot(xs,ys,1);
    t:=t+deltat
  end;
end;     {PlotFunction}

begin  { main }
  Initialize;
  FindXExtrema;
  FindYExtrema;
  Wait;
  HiRes;
  DrawAxes;
  PlotFunction;
  Wait
end.  { main }
```

12.3 Plotting Of Polar Equations

A particularly useful case of parametric equations occurs when an implicit relation between the cartesian variables x and y is given by a pair of **polar equations.** As already mentioned the standard relationship between polar and cartesian coordinates is given by the equations:

$$x = r\cos\theta \tag{1}$$

$$y = r\sin\theta \tag{2}$$

This relationship is illustrated geometrically in Figure 12.2.

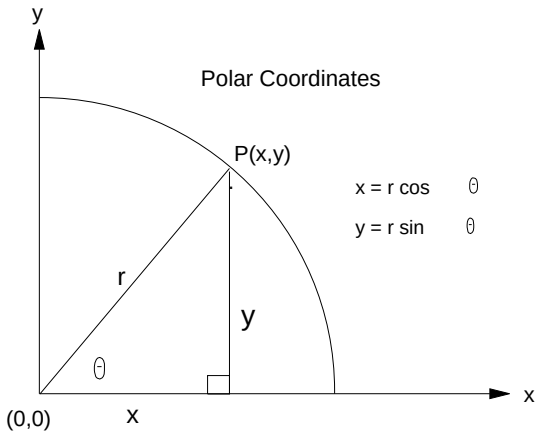


Figure 12.2 Polar Coordinates

If we compare this pair of equations to the pair x=f(t), y=g(t) pair that we considered in the previous section, we can see a similarity of form. The only difference is that now we have two parameters (r and θ), whereas before we had one (t). However the present pair of equations x=f(t), y=g(t) is used to describe a general relationship between two coordinate systems, and does not determine a relation between and y, and therefore does not define a graph. We may nevertheless plot a curve given as:

$$r = H(\theta), \text{ or } \theta = K(r),$$

just by eliminating one of the polar variables by substitution of the polar equation into both equations (1) and (2). If this is done, we end up with x=f(t), y=g(t); where t is either r or θ. An example should serve to clarify these points.

Example: Converting Polar Equation To (x,y) Parametric Form

A well-known polar curve is the **cardioid**. It is so named because of its heart shape. The canonical (standard) equation for a cardioid in polar form is:

$$r = 2a(1+\cos\theta), \quad \textit{where} \quad \theta \quad \textit{ranges} \quad \textit{from} \quad 0 \quad \textit{to} \quad 2\pi.$$

Substituting for r from this equation into equations (1) and (2) yields the following pair of parametric equations defining a relation between x and y:

$$x = 2a(1+\cos\theta).\cos\theta$$

$$y = 2a(1+\cos\theta).\sin\theta$$

It is usually (but not always) the case that r is the variable eliminated when the parametric equations for the cartesian variables x and y are derived. The cardioid is shown in Figure 12.3.

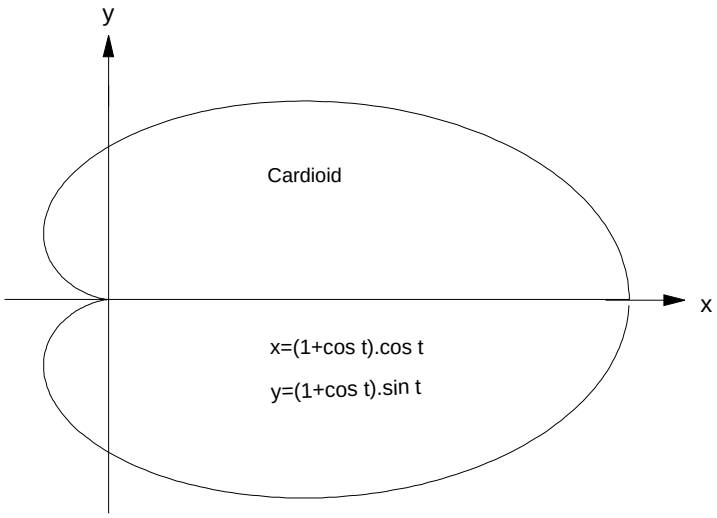


Figure 12.3 Example of Polar Plot - The Cardioid

Envelopes

12.4 Line Patterns

Most computers with graphics capability have inbuilt commands ("intrinsics") which allow us to draw straight line segments quite rapidly, that is, without having to tell the machine to plot every single point. Since a line segment is defined by its end points, in principle all we need to specify to such a routine are the coordinates of these points, and perhaps the colour or intensity of the line to be drawn. As we have mentioned, Turbo Pascal on the IBM PC has such a facility - a procedure called **Draw.** Some interesting patterns can be made just by drawing a large number straight-line segments. The parameters to **Draw** are as follows:

Draw(x0,y0,x1,y1,c)

where (x0,y0) are the (integer) screen coordinates of the starting point in graphics mode; (x1,y1) the coordinates of the final point, and c is the colour to use.

If we allow our usual angle theta (or t), to go from 0 to 360 degrees 0 to 2π radians), we can calculate x as cos t and y as sin t (just polar coordinates with r=1), but then scale x by half a screen (320) and similarly for y (200). Choosing our origin at (320,100), i.e. the centre of the screen, allows us to draw 361 straight-line segments, which form an ellipse. If your school has a colour IBM or compatible, you may like to experiment with different colours for the lines. For example, you could make each quadrant a different colour. A Pascal program to generate the ellipse in Figure 12.4 is given below. *Make sure that you run this program from your disk to see how this ellipse is formed.*

```
program Ellipse;
{ Plots an ellipse from its polar equations }
var
  x, y, t : Integer;
  k       : Real;
begin
  HiRes;
  k:=3.14159/180.0;
  for t:=0 to 360 do begin
    x:=320+trunc(320*cos(k*t));
    y:=100+trunc(100*sin(k*t));
    Draw(320,100,x,y,1);
  end;
  repeat until keypressed
end.
```

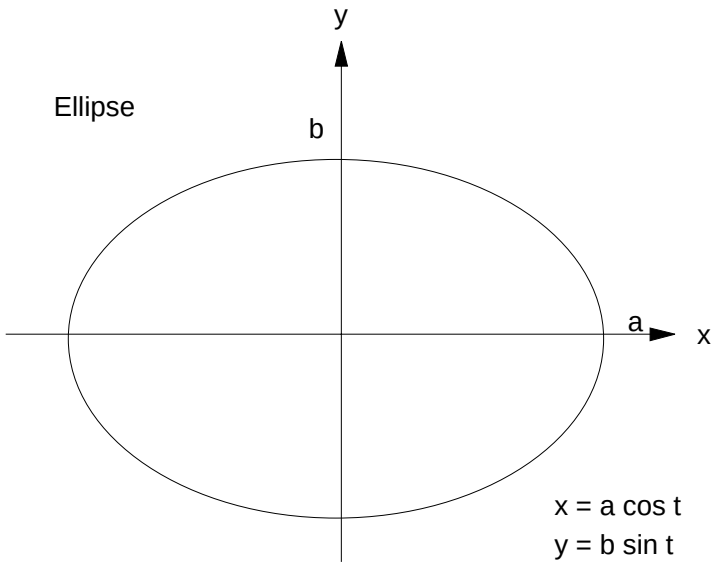


Figure 12.4 Ellipse In Parametric Form

A useful application of using straight-line-segment graphs to define a curve is in the study of **envelopes**. An envelope of a family of straight lines is a curve E, whose tangent at every point is a member of the straight-line family, and every member of the family of straight lines is a tangent to E. In other words, we have an infinite collection of straight-lines which form the entire set of tangents to a certain curve, E. Let's look at a simple example. We consider the canonical parabola whose equation is:

$x^2 = 4ay$.

To find the equation of the tangent to this curve at $P(t) = (2at, at^2)$, we note that the slope is t, and therefore we have:

$$y - at^2 = t(x - 2at).$$

This reduces to:

$$y = tx - at^2 = t(x - at) \qquad (3)$$

It should be fairly clear that equation (3) represents a family of straight lines with parameter t. That is, for each real value of t, there corresponds a straight-line given by equation (3), having slope t and which is tangent to the parabola at P(t). Conversely, for each point on the parabola, there exists a value of t such that equation (3) gives the equation for the tangent to the parabola at that point. For example, simple substitution of some values of t gives:

Value of t	**Equation of Line**
t=1	y=x-a
t=2	y=2x-4a
t=3	y=3x-9a

Figure 12.5 shows a few members of the set of tangents, i.e. **envelope** to a parabola.

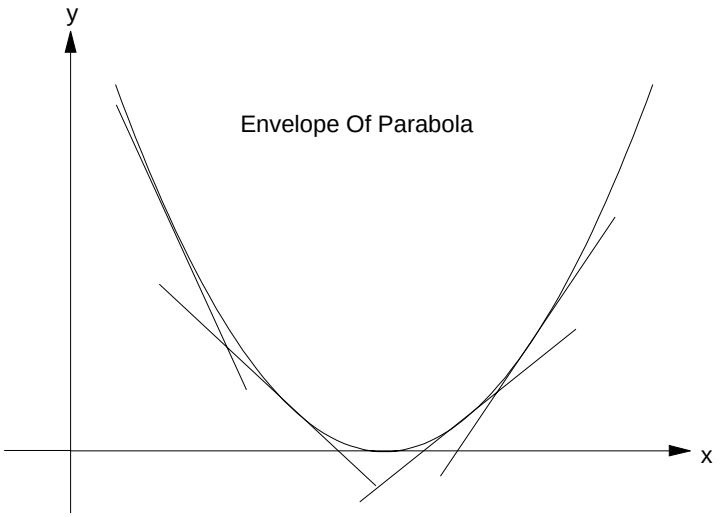


Figure 12.5 Envelope Of Parabola

Plotting Of Envelopes

We now turn to investigate how we might use the graphical capabilities of our the computer to illuminate some of these ideas. It is a simple matter to plot a straight-lines in Turbo Pascal; we considered it in the previous section. To see our envelope now take shape on the screen, we need only plot a succession of straight-line segments for t taking some reasonable sequence of values. The tricky part now however is to determine the endpoints of our line segments to feed to the **Draw** procedure. The information we have for each line is just its equation, and the point of tangency. To make life easier, let us simply take for each line 50 pixels (points on the screen) either side of the point of tangency. We shall centre the parabola on the screen, that is, place the vertex at screen coordinates (320,100).

Apart from stepping through a loop from -50 to 50 for each line, the only other thing to remember is that we must scale everything to fit the screen. The parabola in Figure 12.5 was generated using the program below. Make sure that you run this program to see how this parabola is formed.

```
program Envelope;
{ Plots a family of tangents to form the envelope of a parabola }
var
  x0, x1, xs, ys, xs0, xs1, y0, y1 : Integer;
  a                                : Real;
```

```
function z(xs : Integer; a : Real) : Integer;
var
  t : Real;
begin
  t:=(1.0*(xs - 320))/2/a;
  z:=199-Trunc(a*t*t);
end;

begin { main }
  HiRes;
  xs0:=0; xs1:=639;
  a:=200.0;
  for xs:=xs0 to xs1 do begin
    ys:=z(xs,a);
    x0:=xs-300; y0:=z(x0,a);
    x1:=xs+300; y1:=z(x1,a);
    gotoxy(1,1); writeln(xs);
    Draw(x0,y0,x1,y1,1);
  end;
  repeat until keypressed
end. { main }
```

12.5 Historical Note

René Descartes (1596-1650)

Perhaps the most famous mathematician/philosopher of the 17th century, René Descartes was born in La Haye, France, in 1596. In common with many famous mathematicians, he showed a preference for mathematics at an early age. His most famous contribution to mathematics is his coordinate geometry, which is used throughout modern mathematics and is taught to all high-school students. We call it **cartesian geometry** after Descartes, although his countryman, **Pierre de Fermat** actually co-discovered it independently.

Descartes received his degree in Law from the University of Poitier in 1616, and although French, most of his published works were written in Holland, in which he travelled extensively.

The work for which he is most famous is "**Discourse On Method",** to which was appended "**La Géométrie**". He also had great interest in theology. Probably most famous of his theological statements is his proof that God exists, which although very interesting as a chain of reasoning, would be regarded by modern philosophers as rather naïve. We quote from Kline's book, "Mathematics In Western Culture":

> *"From the one reliable source that his doubts left unscathed - his consciousness of self - he extracted the building blocks of his philosophy: (a) I think, therefore I am; (b) each phenomenon must have a cause; (c) an effect cannot be greater than the cause; and (d) the ideas of perfection, space, time, and motion are innate to the mind."*
>
> *"Since man doubts so much and knows so little he is not a perfect being. Yet, according to axiom (d), his mind does possess the idea of perfection and, in particular, of an omniscient, omnipotent, eternal, and perfect being. How do these ideas come about? In view of axiom (c) the idea of a perfect being could not be derived from or created by the imperfect mind of man. Hence, it could be obtained only from the existence of a perfect being, who is God. Therefore God exists."*

In October 1649, Descartes was invited by Queen Christina of Sweden to tutor her in mathematics and philosophy. This invitation he accepted, but learned that the Queen required her lessons at the strange hour of 5 a.m. each day. Poor Descartes, who had always been rather frail, was subjected to a very cold Swedish winter and died on 11th February 1650.

12.6 Summary

- Programs to plot functions in either cartesian or parametric forms can be written with the aid of a range of procedures supplied with Turbo Pascal.
- The tools to plot a function (no matter how complicated) allow us to experiment with the mathematical concepts of relations and functions, and hopefully lead to an improved understanding.

- The cartesian and polar equations of a circle are:
  $x^2 + y^2 = a^2$, and $r = a$ respectively
  Since the equations relating cartesian and polar coordinates are:
  $x = r\cos\theta$, $y = r\sin\theta$,
  we can readily convert from one form to another.
- A number of interesting figures can be obtained by plotting parametric equations. For example, cardioid, ellipse, Lissajous figures, cycloid, and many others.
- An envelope of a curve is a family of straight-lines, each of which is a tangent to the curve. Since plotting straight-lines on the computer is a simple matter, we readily create envelope plots.

12.7 Exercises

1. Show that the cartesian and polar equations for the circle given in this chapter are equivalent. To do this, you must show that the polar equations yield the cartesian equations when the substitutions:

   $x = r\cos\theta$; $y = r\sin\theta$ are made.

2. Use FUNPLOT to plot the following function on the intervals indicated. You will of course have to alter the formula for the function f and recompile before trying to run the program each time.

   (a) $f(x) = x^2$, *on* $(-2.0, 2.0)$,

   (b) $f(x) = \tan x$, *on* $\left(-\frac{\pi}{2.0} + 0.1,\ \frac{\pi}{2.0} - 0.1\right)$,

   (c) $f(x) = \cos^2 x$, *on* $(-\pi, \pi)$

   (d) $f(x) = \sin^2 x$, *on* $(-\pi, \pi)$

   (e) $f(x) = \sin^2 x + \cos^2 x$, *on* $(-\pi, \pi)$

   For the function of (e), the program misbehaves. What is wrong? Hint: "de-comment" the write statements in the FindExtrema procedure and then determine why the DrawAxes procedure comes to grief. What is the solution to this problem?

3. Use PARAPLOT to plot the following parametric relations. The parameter t ranges from 0 to 2π in each case.

(a) x = cos t; y = sin t;	(circle)
(b) x = a(t-sin t); y = a(1-cos t);	(cycloid)
(c) r = a cos 2t;	(quadrifolium)
(d) r = t;	(spiral)
(e) x = 2cost+cos(2t); y = 2sint-sin(2t);	(tricuspid)
(f) x = 3cost-cos(3t); y = 3sint-sin(3t);	(nephroid)

4. Can you suggest some improvements to the function plotting programs? The error which occurs in Question 2(e) is a very common one in mathematical computing, and its occurrence should be anticipated in advance and catered for. In other words, we simply do not divide if the divisor would be zero. Depending on the context of the problem, certain other actions are desirable for special cases such as this. Can you anticipate any other errors which could occur in the plotting programs? Hint: In Q2(b), why didn't we plot tan x on $(-\pi/2, \pi/2)$?

5. If the computer you are using has a colour monitor, modify the ELLIPSE program to make each of the quadrants a different colour.

6. Show that the curve:
   $x^2 = 4ay$
   has gradient t at the point $(2at, at^2)$.

7. Criticize Descartes' proof of the existence of God.

13 CURVE FITTING

13.1 Introduction

The analysis of data from scientific experiments commonly requires plotting of the empirical points as (x,y) coordinates. A smooth curve which somehow approximates these observed data points can then be drawn. The object of this is to graphically determine a function (or relation) between x and y. We simply start with a set of n ordered pairs, plot them on graph paper, and then "fill in the gaps" with a curve. This curve may be drawn freehand, or devices such as flexible curves may be used to give a more polished end-product. In either case, the underlying assumption is that a functional relationship holds between x and y, and some amount of guesswork is involved as to what the function is between the points which we actually have. This process of "filling the gaps" between known points on a curve is called **interpolation,** or curve-fitting.

It is important to realize that any empirical data are subject to experimental error, and therefore it is not absolutely essential that the curve we fit to a set of points must pass precisely through each point. Indeed, some points may be bad observations and this is more likely to become obvious after plotting on graph paper. At the option of the person doing the analysis, such points may be discarded before any kind of curve-fitting is attempted. Figure 13.1 shows some experimental data that has been plotted and a "best-fit" curve has been drawn. It can be seen that some points do not fit the chosen curve very well. These may correspond to relatively large experimental errors, and normally would be ignored.

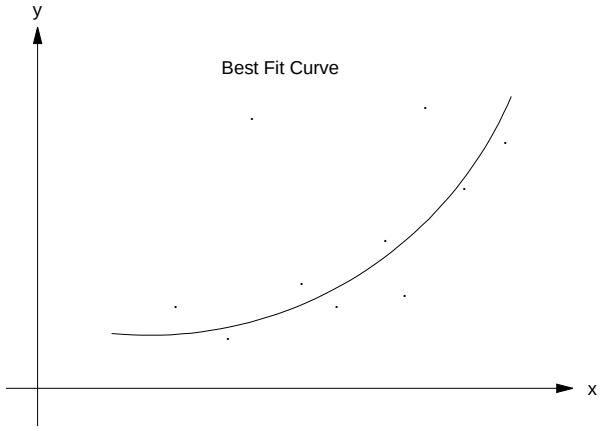


Figure 13.1 A Curve Of "Best Fit"

We now turn to consider some **numerical techniques** for curve-fitting. Graphical techniques are useful as far as they go, but generally only give very rough quantitative estimates of the accuracy of the interpolating function. Many numerical methods exist which allow us to generate functions which in some sense, "best" approximate the data we wish to interpolate. We need to realize first of all that there is no single sense in which a set of points is best approximated (interpolated) by a given function. Interpolation or curve-fitting may be regarded as searching for, or choosing from an infinite set of functions, one which satisfies some chosen condition of "best fit" to our (finite) set of data points. Two of the commonest ways that we can specify an approximating function to be optimal are:

(i) Linear least squares
(ii) Strictly interpolating polynomial.

Let's now briefly consider and contrast these two approaches. A more detailed treatment will be found in the sections devoted to each method below.

Criterion (i) specifies that the approximating linear function (i.e. a straight-line) be such that the sum of the squares of the y-deviations for all the data points be minimised. Notice that we do not require that the graph of the function pass through all of the points. This is in fact generally impossible anyway, since three or more distinct points are normally not collinear.

Criterion (ii) requires that a polynomial P(x) of lowest possible degree be found such that the points lie **exactly** on the graph of P(x). Suppose that we write:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x^1 + a_0$$

If we have N points (x_i, y_i) then by simple substitution of each of the points into the equation for the polynomial, we see that N simultaneous equations are obtained for (n+1) unknowns - the coefficients $a_n, a_{n-1}, \ldots, a_2, a_1, a_0$.

If we choose n=N-1, then we have N-1 simultaneous equations in N-1 unknowns. This does not mean that we have to use some method of solving linear equations to solve the interpolation problem, but it does tell us that to fit N data points to a polynomial of lowest degree, we generally need a polynomial of degree N-1. It can also be shown that this polynomial always exists and is unique, that is, we can always find a polynomial to interpolate all the points, and once we have found it, by whatever method, then we know that it is the correct one.

A Note On Existence & Uniqueness

Existence and uniqueness of solutions to any mathematical problem are very valuable items of information to have, especially when they are simultaneously true. Of course problems exist where one or other of these properties of the solution does not hold. For example, real solutions do not exist for all quadratic equations. If we have somehow proved existence and uniqueness though, then once we have obtained a solution, whether by fair means or foul (and any proposed interpolating polynomial solution can be verified by simply substituting the points to see if they satisfy the equation), then we know that we have **the** solution. We may have obtained it by mathematically dubious means, but once we've got it, verification by substitution coupled with existence and uniqueness make it just as correct as if we had obtained it by the most laborious, rigorous, and time-consuming process imaginable.

13.2 Least Squares

Let's suppose that we are given a set of n ordered pairs, which we have plotted on graph paper. These data points have been obtained from a chemistry experiment and we have reason to believe that a linear relationship exists between x and y. From our graph ("scatter diagram") it appears as though a reasonable straight-line could be drawn to pass very close to most of these points. We now wish to confirm this idea by applying an analytical technique to the data. As outlined in the introduction to this chapter, we wish to find the straight-line which minimises the sum of squares of ordinate (y value) deviations between any given straight-line and the observed ordinate at each of the points. These ideas are illustrated in Figure 13.2.

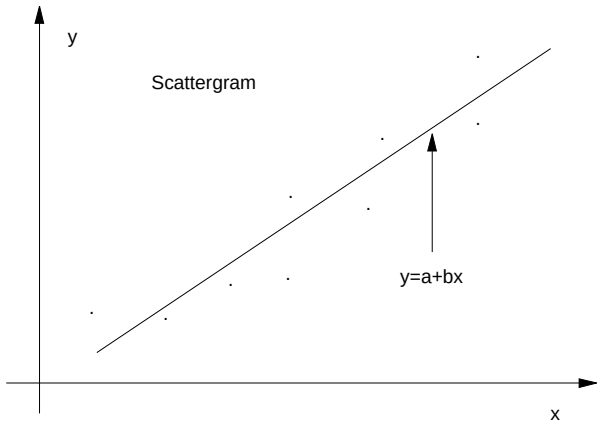


Figure 13.2 The Method Of Least Squares

Let our line be y = a+bx and let our set of points be $S = \{(x_i, y_i)\}$, for $i \in \{1, 2, \ldots, n\}$.

In symbols, we wish to determine a and b such that the sum $Z(a,b) = \sum(y_i - (a + bx_i))^2$

is a minimum. In this expression for Z (the sum of the squares of the deviations), it is important to realise that, contrary to their usual roles, the x's and y's are **constants**, while a and b are the **variables**. After all, it is the values of a and b that we must determine in order to get the equation of the straight-line we are seeking.

To find the minimum of Z(a,b), we equate the **partial derivatives** of Z with respect to both a and b to zero. Partial derivatives are just like ordinary derivatives, except that when differentiating, we pretend that the other variable is **constant.** So for example, if Z(a,b) is a function of a and b, then the (partial) derivative of Z with respect to **a** is obtained just by differentiating with respect to **a** the formula defining Z in terms of a and b, while treating b as if it were a constant. When this is done, the following two simultaneous equations for a and b are obtained:

$$na + (\sum x_i)b = \sum y_i$$

$$(\sum x_i)a + (\sum x_i)b = \sum x_i.y_i$$

Some straightforward algebra yields the formulae for a and b:

$$a = \frac{\sum y.\sum x^2 - \sum x.\sum xy}{n\sum x^2 - (\sum x)^2} \qquad b = \frac{n\sum xy - \sum x.\sum y}{n\sum x^2 - (\sum x)^2}$$

Just a quick point: when we divide, we must always be certain that we are not dividing by zero. You may gloss over this in your algebra or calculus studies, but you won't get very far here if you try to ask the computer to divide by zero. It won't accept any such nonsense. One of the exercises asks you to show that the denominator in these expressions for a and b can never be zero (under the assumptions of the least squares method). Now that we've settled this point, let's build a Pascal program to do the required calculations for us.

```
program LeastSqr;
{ This program determines the line of best fit based on
the least square method }
var
  i, n                      : Integer;
  x, y                      : array[1..10] of Real;
  a, b, Denom               : Real;
  Sumx, Sumy, Sumxx, Sumxy  : Real;
  Ex  , Ey  , Exx  , Exy    : Real;
begin
  repeat
    write('How many points (3 or more) ? ');
    readln(n);
    if n<3 then writeln(Chr(7),'3 points or more please !');
  until n>=3;
  writeln('Please enter (x,y) coordinates 2 per line');
  for i:=1 to n do readln(x[i],y[i]);
  Sumx:=0; Sumxx:=0; Sumxy:=0; Sumy:=0;
  for i:=1 to n do begin
    Sumx:=Sumx + x[i];
    Sumy:=Sumy + y[i];
    Sumxx:=Sumxx + Sqr(x[i]);
    Sumxy:=Sumxy + x[i]*y[i];
  end;
  Ex:=Sumx/n;
  Ey:=Sumy/n;
  Exx:=Sumxx/n;
  Exy:=Sumxy/n;
  writeln('Expected value of x  = ',Ex:10:5);
  writeln('Expected value of y  = ',Ey:10:5);
  writeln('Expected value of xx = ',Exx:10:5);
  writeln('Expected value of xy = ',Exy:10:5);
  writeln;
  Denom:=Exx - Ex*Ex;
  a:=(Ey*Exx - Ex*Exy)/Denom;
  b:=(Exy    - Ex*Ey )/Denom;
  writeln('Line Of Best Fit is  y = ',a:10:5,' + ',b:10:5,'x')
end.
```

Output

```
How many points (3 or more) ? 3
Please enter (x,y) coordinates 2 per line
1 1
2 2
3 3
Expected value of x  =    2.00000
Expected value of y  =    2.00000
Expected value of xx =    4.66667
Expected value of xy =    4.66667

Line Of Best Fit is  y =    0.00000 +    1.00000x
```

13.3 Polynomial Interpolation

In the introduction we stated that a polynomial P(x) of lowest possible degree can be found such that the points belonging to the set:

$$\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

lie **exactly** on the graph of P(x). We saw that n distinct points can in general be interpolated by a polynomial of degree (n-1). If we consider the set of all degree (n-1) polynomials with real coefficients, then we want to find one which passes exactly though each of our points. Any polynomial of degree (n-1) can be written as:

$$P_{n-1}(x) = a_{n-1}x^{n-1} + \ldots + a_2x^2 + a_1x^1 + a_0$$

For the problem to make sense, we must have all the x values different. Why is this? Can any one polynomial pass through both, say (0,1) and (0,2)? No.

$$P_{n-1}(x) = a_{n-1}x^{n-1} + \ldots + a_2x^2 + a_1x^1 + a_0$$

Suppose we take our n points (x_i, y_i). By simple substitution of each of these points into the equation for the polynomial, we see that n simultaneous equations are obtained for n unknowns.

As we remarked in the introduction, it can be shown that the interpolating polynomial always exists and is unique, that is, we can always find a polynomial to interpolate all the points, and once we have found it, by whatever method, then we know that it is the correct one - there is no other. All we need now is a method to find it! We just said that we can arrive at a system of n simultaneous equations in n unknowns (the coefficients a[i]). The methods of an earlier chapter could be used to solve such a system on the computer. We do not have space for the details, but it turns out that, because of the special nature of the problem (it is not just some arbitrary set of linear equations, but a set obtained from substitution in a polynomial), it is not necessary to go to these lengths.

The method we now consider for obtaining the polynomial is due to **Lagrange,** and we simply state it and check that it works. No further proof is required, since, as we have already said, the polynomial is unique, and once we have it, it can be easily checked by substitution of each of the points. If the substitutions work, then we must have the right polynomial, because there **is** only one! Existence and uniqueness are very powerful and useful allies to have.

Lagrange's Form Of The Interpolating Polynomial

Although unique, the interpolating polynomial for a given set of points, can be written in a number of different forms. The two common forms are due to the Lagrange and Newton. We do not consider Newton's form here. A definition of Lagrange's form is given in Figure 13.3.

LAGRANGE'S FORM OF THE INTERPOLATING POLYNOMIAL

Let there be given a set of n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in which no two of the x values are equal.

Then there exists a unique polynomial $P(x)$ of degree at most n-1 such that $P(x_i) = y_i$ for $i \in \{1, 2, \ldots, n\}$

The Lagrange form of this polynomial is:

$$P(x) = y = \sum_{k=1}^{n} L_k(x) y_k, \text{ where } L_k(x) = \frac{(x - x_1)(x - x_2)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n)}{(x_k - x_1)(x_k - x_2)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)}$$

Figure 13.3 Lagrange's Form Of The Interpolating Polynomial

Example Of Lagrange Interpolating Polynomial

Suppose we have the three points (0,1), (1,8), (2,13). We have chosen these points on the curve of:

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

This allows us to verify our working. Since we have three points with no two x-values equal, we can expect that a unique polynomial of degree at most 3-1=2 (i.e. a quadratic or perhaps a linear function) can be found to pass exactly through each of the points. Let's use the Lagrange formula to find this quadratic, and then verify that each point lies on the parabola. We have:

$(x_1, y_1) = (0, 1)$

$(x_2, y_2) = (1, 8)$

$(x_3, y_3) = (2, 13)$

We can now find the polynomials $L_1(x), L_2(x), L_3(x)$ by using Lagrange's formula for $L_k(x)$ for k=1, 2, 3.

$$L_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} = \frac{(x-1)(x-2)}{(0-1)(0-2)} = \frac{(x-1)(x-2)}{2}$$

$$L_2(x) = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} = \frac{(x-0)(x-2)}{(1-0)(1-2)} = -x(x-2)$$

$$L_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} = \frac{(x-0)(x-2)}{(2-0)(2-1)} = \frac{x(x-2)}{2}$$

The final step is to substitute these three quadratics into the formula for P(x). The result is:

$$P(x) = y_1L_1(x) + y_2L_2(x) + y_3L_3(x)$$

$$P(x) = 5L_1(x) + 8L_2(x) + 13L_3(x)$$

$$P(x) = \frac{5(x-1)(x-2)}{2} - 8x(x-2) + \frac{13x(x-2)}{2}$$

$$P(x) = x^2 + 2x + 5.$$

Pascal Program For The Interpolating Polynomial

We now present a Pascal program to build the Lagrange form of the interpolating polynomial, and then allow the user to calculate y-values (ordinates) at any given value of x. The polynomial is in fact rebuilt every time the user asks for a calculation. This is not the most efficient way to solve the problem, but here we are primarily concerned with understanding the problem and one relatively simple means for its solution on a computer. Students should study the code carefully, particularly the section that builds the L(x,k) polynomials defined just above.

```
program Lagrange;
{ Builds Lagrange Interpolating Polynomial and then allows
  the user to find other points on the curve }
const
  nmax = 10;
type
  Vector = array[1..nmax] of Real;
var
  xdata, ydata : Vector;
  Ch           : Char;
  i, n, k      : Integer;
  newx, newy   : Real;

function L(x : Real; k, n : Integer) : Real;
{ Computes L(k,x) for k and x vectors - xdata and ydata are global }
var
  P : Real;
  i : Integer;
begin
  P:=1.0;
  for i:=1 to k-1 do begin
    P:=P*(x-xdata[i])/(xdata[k]-xdata[i]);
    {writeln(i,P)}
  end;
  for i:=k+1 to n do begin
    P:=P*(x-xdata[i])/(xdata[k]-xdata[i]);
    {writeln(i,P)}
  end;
  L:=P
end;
```

```
begin { main }
  write('How many points?  ');
  readln(n);
  writeln('Input one x,y pair/line; separated by space');
  for i:=1 to n do readln(xdata[i],ydata[i]);
  writeln('Data are:'); writeln;
  for i:=1 to n do writeln('(',xdata[i]:6:2,',',ydata[i]:6:2,')');
  repeat
    writeln; write('Type C for Calculate; E for End  ');
    repeat
      read(kbd,Ch);
      Ch:=UpCase(Ch);
      if not (Ch in ['C','E']) then write(#7)
    until Ch in ['C','E'];
    if Ch='C' then begin
      writeln; writeln('Value of x to compute function at? ');
      readln(newx);
      newy:=0.0;
      for k:=1 to n do newy:=newy+L(newx,k,n)*ydata[k];
      writeln('Function value at x= ',newx:6:3,' is ',newy:8:4)
    end
  until Ch='E'
end. { main }
```

Output

```
How many points?  3
Input one x,y pair/line; separated by space
1 3
2 7
3 13

Data are:
(  1.00,  3.00)
(  2.00,  7.00)
(  3.00, 13.00)

Type C to Calculate function value; E to Exit
Value of x to compute function at?
4
Function value at x=  4.000 is  21.0000

Type C to Calculate function value; E to Exit
Value of x to compute function at?
5
Function value at x=  5.000 is  31.0000

Type C to Calculate function value; E to Exit
Value of x to compute function at?
-3
Function value at x= -3.000 is   7.0000

Type C to Calculate function value; E to Exit
```

13.4 Historical Note

Comte Joseph Louis de Lagrange (1736-1813)

Lagrange was born in Turin, Italy, of French parentage and at the age of 19 became Professor of Mathematics in the artillery school of Turin. In 1766 he succeeded Euler at the Berlin Academy, having been summoned by Frederick the Great with a note which said in part: "it is necessary that the greatest geometer of Europe should live near the greatest of kings".

In 1770 he published a method for solving algebraic equations using the **resolvent** equation. The degree of the resolvent was usually lower than that of the original equation. Quadratic, cubic, and quartic equations could then be solved by this technique. However the resolvent equation for the quintic (polynomial of degree 5) had degree 6. Later it was proved by Galois that the general polynomial could be solved in terms of its coefficients if and only if the degree was 4 or less.

Early in his career, Lagrange made great contributions to the calculus of variations, which is essentially concerned with the determination of optimal **functions** in certain minimization or maximization problems. For example, a classic example of a problem in the calculus of variations is the **Brachistochrone problem,** in which one must determine the curve (i.e. function) whose shape is such that a body sliding down it will reach a certain point in the minimum time. This curve turns out to be a **cycloid**, which is also the curve generated by a point on the circumference of a rolling wheel.

With Lagrange we find the genesis of the theory of functions of a real variable, which was later to be greatly extended by men such as Weierstrass, Cauchy, Riemann and others. Lagrange started with the Taylor series and showed that a function f(x) could then be expanded into a series of powers of x, and with coefficients derived from the derivatives $f'(x)$, $f''(x)$. Lagrange invented the notation $f'(x)$, $f''(x)$.

Lagrange also did much work in the area of analytical mechanics. His most famous work is probably **Mécanique Analytique,** in which he applies analytical methods to the motions of point masses and rigid bodies. He developed very general methods of analysis, including his generalized coordinates and Lagrangian forms of equations of motion (relating energy to force). These concepts are very powerful and are still in widespread use today. Although Newton virtually singlehandedly invented the whole theory of motion, gravitation and analysis by calculus, his methods were largely geometrical, and therefore the work of Lagrange was a milestone in the sense that it succeeded by purely analytical methods alone. In fact, Lagrange emphasised in the preface to **Mécanique Analytique:** "**on ne trouvera point de figures dans cet ouvrage, seulement des opérations algébriques**" (no diagrams will be found in this treatment, only algebraic operations).

13.5 Summary

- Interpolation (or curve-fitting) is the process of "filling in the holes" between data points obtained from experiments. Its aim is to find a functional relationship between the variables.
- Two common numerical techniques for curve-fitting are linear least-squares and strictly interpolating polynomial.
- The linear least-squares approach minimises the sum of squares of the ordinate deviations from the straight-line fitted to the data points.
- The polynomial interpolation method uses a polynomial P(x) of lowest degree such that the data points lie exactly on the graph of P(x). One method used to obtain P(x) is due to Lagrange. The polynomial formed using this method can of course be used to predict the y-value for any value of x.

13.6 Exercises

1. Why can't any polynomial pass through both (x,p) and (x,q), where p is different from q?
2. Show that the denominator in the least squares formulae for a and b cannot be zero unless all the x's are the same.
3. Run the Least Squares program with all x's the same and see what happens. Improve the program to handle this case a little more gracefully, i.e. detect the error condition as soon as possible and terminate with an error message instead of asking the computer to do the impossible!
4. Show that the least squares formulae for a and b result from solving the two simultaneous linear equations for a and b.
5. In an experiment designed to test the effect of weightlessness on an astronaut's reaction time, an astronaut was required to to respond to a flashing light by depressing a button. The time that elapsed between the light flash and the depression of the button was denoted by y and the length of time that the astronaut was weightless was denoted by x. The resulting data is given below:

x (hours)	y (secs)
5	0.70
6	1.08
7	1.02
8	1.26
9	1.62
10	1.42
11	1.47
12	1.85
13	2.02
14	1.93

(a) Find the equation of the line of best fit for this data.
(b) Use the equation in (a) to predict the astronaut's reaction time after 20 hours of weightlessness.

6. A department store reports the following sales data over a six year period:

Year	Yearly Sales ($00,000)
1981	55.7
1982	68.7
1983	71.3
1984	81.1
1985	82.0
1986	89.0

Let y denote the yearly sales (in hundred of thousands of dollars) and let the years 1981, 1982,.....,1986 be coded as 0,1,2,.....,5. That is, x=0 represents 1981, x=1 represents 1982, and so on.

(a) Find the line of best fit relating x and y.
(b) Estimate the yearly sales in 1982.
(c) Why is it convenient to code the years?

7. In each of the following cases, use the Lagrange interpolation program to interpolate at the known points, and then predict the ordinates at the points where these are unknown.

(a) (-2,-10), (0,2), (-1,-3), (4,?)
(b) (3,18), (-1,-2), (4,48), (0,0), (2,?).

14 SEQUENCES & SERIES

14.1 Ways Of Defining Functions

Sequences and series form a very important part of our mathematical education. We learn of arithmetic and geometric progressions, and other elementary sequences. A sequence is just a succession of items (usually numbers), to each of which is assigned a counter value, or **index**. The index is just its position in the sequence, and therefore takes values 1, 2, 3, 4...; i.e. a natural number starting at 1. We may define a sequence then, as a mapping (or function) whose domain is the set of natural numbers.

Since a sequence is really a function whose domain is the natural numbers, it will be helpful to "spell-out" the two fundamental ways in which functions may be defined. If a function is defined on a **finite** set, there are basically two ways to define it:

1. The first is by a **formula.** This can be either a direct formula in terms of the index n, or a recurrence formula defining F_n as a function of previous terms of the sequence. Our definition of the Fibonacci sequence in Table 14.2 is an example of a recurrence formula.
2. The second way to define a function on a finite set is simply to list all the ordered pairs (n, F_n). This process is called **exhaustive enumeration.** Our table of pairs in Figure 14.2 is an example of this, although it does **not** of course define the entire (infinite) Fibonacci sequence, since the domain of this sequence is all of the natural numbers.

An illustration of the kinds of definitions using the three examples of sequences in this chapter is given in Table 14.2.

14.2 The Fibonacci Sequence

Let's define the sequence $\{F_n\}$ as follows:

$$F_1 = 1,$$

$$F_2 = 1,$$

$$F_{n+2} = F_{n+1} + F_n, \text{ for } n \in \{1, 2, 3, 4\ldots\}$$

This sequence is known as the **Fibonacci** sequence. The first ten terms of this sequence are given in Table 14.1.

n	1	2	3	4	5	6	7	8	9	10
F_n	1	1	2	3	5	8	13	21	34	55

Table 14.1 The Fibonacci Sequence

You should verify these values. Each term after the first two (which must be separately defined) is simply the sum of the previous two terms. Just substitute into the formula and check that each value is as claimed. This sequence was first investigated in the 13th century by Leonardo of Pisa (1170-1230), otherwise known as Fibonacci (literally "son of Bonaccio"), in connection with the breeding of rabbits. According to the ideal model, the terms of the Fibonacci sequence represent the number of breeding rabbit pairs at the end of successive breeding periods. The sequence has since found a number of other applications, which include other biological models, music, and search/sort techniques in computer science.

A Direct Formula For the Fibonacci Sequence

Since we already have a recurrence formula which adequately defines each Fibonacci number in terms of its two immediate predecessors, it is natural to enquire whether a formula exists which defines F(n) directly as a function of n. Such a formula does indeed exist. If we assume that there exists a formula of the form:

$$F_n = Ax^n$$

for the Fibonacci sequence, where A and x are constants, we get after substitution:

$$Ax^{n+2} = Ax^{n+1} + Ax^n.$$

EXAMPLE	Direct Formula	Recurrence Formula	Enumeration
Fibonacci Sequence	$F_n = \frac{p^n - q^n}{\sqrt{5}}$ where $p = \frac{1+\sqrt{5}}{2}$, $q = \frac{1-\sqrt{5}}{2}$	$F_1 = 1,$ $F_2 = 1,$ $F_{n+2} = F_{n+1} + F_n$	$F_1 = 1,$ $F_2 = 1,$ $F_3 = 2,$ $F_4 = 3,$ $F_5 = 5,$ $F_6 = 8,$ etc.
Arithmetic Progression	$t_n = a + (n-1)d$	$t_n = t_{n-1} + d$	$t_1 = a,$ $t_2 = a + d,$ $t_3 = a + 2d,$ $t_4 = a + 3d,$ $t_5 = a + 4d,$ $t_6 = a + 5d,$ etc.
Geometric Progression	$t_n = ar^{n-1}$	$t_n = rt_{n-1}$	$t_1 = a,$ $t_2 = ar,$ $t_3 = ar^2$ $t_4 = ar^3$ $t_5 = ar^4$ $t_6 = ar^5,$ etc.

Table 14.2 Different Ways Of Defining Sequences

After division throughout by Ax^n, we get $x^2 = x + 1$. This is equivalent to:

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

Solving this quadratic equation using the quadratic formula gives:

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

Let's call these two values p and q, since it gets a bit awkward to write them in equations otherwise. We have:

$$p = \frac{1+\sqrt{5}}{2} \text{ and } q = \frac{1-\sqrt{5}}{2}$$

Therefore:

$$F_n = Ap^n \text{ and } F_n = Aq^n$$

will satisfy the Fibonacci recurrence formula. Since A is any constant, a different one may used for the second possible formula. We may write the two possible formulae then as:

$$F_n = Ap^n \text{ and } F_n = Bq^n$$

Rewriting the Fibonacci recurrence as:

$$F_{n+2} - F_{n+1} - F_n = 0,$$

and substituting these two solutions, we obtain:

$Ap^{n+2} - Ap^{n+1} - Ap^n = 0,$ and

$Bq^{n+2} - Bq^{n+1} - Bq^n = 0.$

Adding these last two equations, we arrive at:

$$(Ap^{n+2} + Bq^{n+2}) - (Ap^{n+1} + Bq^{n+1}) - (Ap^n + Bq^n) = 0.$$

Inspection of the last three equations shows that if

$F_n = Ap^n$ and $F_n = Bq^n$

are solutions, then so is $F_n = Ap^n + Bq^n$ for any constants A and B.

You should check that this equation satisfies the Fibonacci recurrence formula for the values of p and q given, and for arbitrary A and B. Appropriate values for the constants A and B can be found by forcing the first two terms of the sequence to be 1, as in the original definition of the Fibonacci sequence by recurrence formula:

$F_1 = F_2 = 1.$

Substitution gives:

$F_1 = 1 = Ap + Bq,$ and

$F_2 = 1 = Ap^2 + Bq^2 = A(p+1) + B(q+1)$

(since $p^2 = p + 1$ and $q^2 = q + 1$)

Subtracting these two equations gives A+B=0. Solving for A and B, we get:

$$A = \frac{1}{p-q} = -B = \frac{1}{\sqrt{5}}.$$

We have therefore:

$$F_n = \frac{p^n - q^n}{\sqrt{5}}.$$

Let's now look at a simple Pascal program to compute the first 20 Fibonacci numbers by both formulae, and compare the two. The program uses an array to store the numbers generated by the recurrence formula, and although this is not strictly necessary, it simplifies the coding. It is left as an exercise to rewrite the program without using an array. With floating-point arithmetic used to compute the terms via the direct formula, it is to be expected that some loss of accuracy will result. In this case, we are fortunate to have the exact answer for comparison; this is a rare luxury indeed. It is very instructive to note the discrepancies starting to creep in after a dozen or so terms. The rule is to never trust floating-point ("real") arithmetic to be exact, but always accept it with reservations about its accuracy. Further, always be on the lookout for some means of independent confirmation of such computer-generated results. As we have noted, we are rather lucky in this example to have the opportunity of comparison with the exact answers.

```
program Leonardo;
{ This program finds the Fibonacci numbers by recurrence
and direct formulae }
var
  F                    : array[1..20] of Integer;
  p, q, A, B, V, W, G : Real;
  n                    : Integer;
begin
  p:=(1 + sqrt(5.0))/2.0;
  q:=(1 - sqrt(5.0))/2.0;
  A:=1.0/(p - q); B:=-A;
  V:=A*(p + 1); W:=A*(q + 1);
  F[1]:=1; F[2]:=1;
  writeln('  n    Recurrence  Power Formula'); writeln;
  for n:=3 to 20 do begin
    F[n]:=F[n - 1] + F[n - 2];
    V:=V*p; W:=W*q;
    G:=V - W;
    writeln(n:4,F[n]:10,'  ', G:16:8);
  end;
end.
```

```
Output

n    Recurrence    Direct

 3          2          2.00000000
 4          3          3.00000000
 5          5          5.00000000
 6          8          8.00000000
 7         13         13.00000000
 8         21         21.00000000
 9         34         34.00000000
10         55         55.00000000
11         89         89.00000000
12        144        144.00000000
13        233        233.00000000
14        377        376.99999999
15        610        609.99999999
16        987        986.99999998
17       1597       1597.00000000
18       2584       2583.99999990
19       4181       4180.99999990
20       6765       6764.99999980
```

14.3 Arithmetic & Geometric Progressions

The Fibonacci sequence has a very simple recurrence formula in which each term is obtained as the sum of the two previous terms. In fact the simplest type of sequence possible is one in which each term is obtained by adding a **constant** to the previous term. Such a sequence is called an **Arithmetic Progression (AP).** The general formula for the nth term of an AP is:

$$t_n = a + (n-1)d.$$

The constants a and d are the first term, and common difference respectively. The sum of the first n terms of this sequence is:

$$s_n = \frac{n(a + (n-1)d)}{2}$$

and the proof is left as an exercise.

Example of AP.

The sequence given below is an arithmetic progression.

2, 5, 8, 11,.... 3n-1,...

The first term is clearly a=2, and each term is obtained by adding 3 to the previous term (common difference d=3). The general term is:

$t_n = 3n - 1 = a + (n-1)d$, where $a = 2$ and $d = 3$

To find the tenth term, we just substitute n=10 in the formula for t_n:

$$t_{10} = 2 + (10-1)3 = 2 + 27 = 29$$

To find the sum of the first ten terms, substitute into the S_n formula:

$$S_{10} = \frac{10(2 + (10-1)3)}{2} = 145$$

As we saw in Table 14.2, an A.P. can also be defined by the following recurrence formula:

$$t_1 = a$$

$$t_n = t_{n-1} + d$$

Pascal Program To Generate Terms Of An AP.

It is a simple matter to write a Pascal program to generate terms of an arithmetic progression. In fact, if the language allowed an increment other than 1 or -1 in the **for** loop, the whole program would be just a **for** loop containing a writeln statement! As it is, we just have to include an extra statement to add in the common difference.

We use two different methods to arrive at the value for each term of the AP in the program. One is the standard AP formula given above, and the other is the recurrence formula (which is actually the discrete derivative of the AP formula). You should be familiar with both forms.

```
program AP;
{ This program generates an AP given the first term
and common difference }
var
  a, v, u, n, i, d, max : Integer;
begin
  a:=1; d:=2; max:=10;
  v:=a;
  writeln('   Direct   Recurrence');
  writeln('   Formula  Formula'); writeln;
  for n:=1 to max do begin
    u:=a + (n - 1)*d;
    writeln(u:6,'     ',v:6);
    v:=v + d;  { get next term using recurrence formula }
  end;
end.
```

Output

```
Direct     Recurrence
Formula    Formula

   1            1
   3            3
   5            5
   7            7
   9            9
  11           11
  13           13
  15           15
  17           17
  19           19
```

A **Geometric Progression (GP)** is similar to an Arithmetic Progression, but instead of each term being obtained from the previous by **addition of a constant,** the terms of a GP are derived by **multiplying the previous term by a constant.** This constant is called the **common ratio.** The general formula for the nth term of an AP is:

$$t_n = ar^{n-1}.$$

and the proof is left as an exercise.

The constants a and r are the first term, and common ratio respectively. The sum of the first n terms of this sequence is:

$$s_n = \frac{a(1-r^n)}{1-r}.$$

Example Of GP.

The sequence given below is a GP.

$$3, 6, 12, 24, \ldots, 3 \text{ x } 2^{n-1}, \ldots$$

The first term is clearly a=3, and each term is obtained by the previous term by multiplying by 2 (common ratio r=2). The general term is:

$t_n = 3 \text{ x } 2^{n-1} = ar^{n-1}$, where $a = 3$ and $r = 2$

To find the tenth term, we just substitute n=10 in the formula for t_n:

$$t_{10} = 3 \text{ x } 2^{10-1} = 3(512) = 1536$$

To find the sum of the first ten terms, substitute into the formula for S_n:

$$S_{10} = \frac{3(1-2^{10})}{1-2} = 3(1023) = 3069$$

Pascal Program To Generate Terms Of A GP.

The program we present for generating terms of a GP is very similar to the previous one for AP's, but notice carefully that we make no explicit use of a **power function** (none exists in Pascal). Each term is simply obtained from the definition by multiplying the previous one by the common ratio. As we saw in Table 14.2, a G.P. can also be defined by the following recurrence formula:

$t_1 = a,$

$t_n = r t_{n-1}$

```
program GP;
{ This program works out the terms of a GP given the first term
  and common ratio }
var
  a, v, u, n, i, r, max : Integer;
begin
  a:=1; r:=2; max:=10;
  u:=a;
  writeln('   Index   Recurrence'); writeln;
  for n:=1 to max do begin
    u:=u*r;
    writeln(n:6,'    ',u:6);
  end;
end.
```

```
Output

Index    Recurrence

    1             2
    2             4
    3             8
    4            16
    5            32
    6            64
    7           128
    8           256
    9           512
   10          1024
```

14.4 Euler's Constant

In an earlier chapter, we mentioned some of the achievements of the great Swiss mathematician **Leonhard Euler.** A mathematical constant is named after him - the **Euler** constant (γ), and it is the asymptotic difference between the **Harmonic Series** and the **natural logarithm** function (both divergent as n approaches infinity), i.e.

$$\gamma = \lim_{n \to \infty} \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} - \log n \right)$$

Euler's constant to 10 decimal places is $\gamma \approx 0.57721\ 56649$.

The harmonic series is a very simple-looking series. It is one in which the terms are monotone decreasing, and its partial sums actually appear to be approaching a limiting sum. In fact, this series is **divergent.** In this case, this means that its partial sums can be made to exceed **any preassigned real number**, simply by taking enough terms. In other words, as the number of terms tends to infinity, so do the partial sums. For a series to converge, the general term must tend to zero. But this condition is only necessary - it is **not sufficient.** We can state this another way: **all convergent series have general terms which tend to zero, but not all series with such general terms converge.** The harmonic series is a good example.

Let's first write a Pascal program to accumulate the partial sums of the Harmonic Series. As we have already seen, floating-point arithmetic is not perfectly accurate, and we must be very careful when adding hundreds or even thousands of terms, especially with a series that is known to diverge. This series is notorious for its accumulation of **roundoff errors.** We can then extend the program to use the Pascal intrinsic **ln** function, and by subtraction, give us an estimate of Euler's constant. The first program finds partial sums of the harmonic series by adding forwards and backwards.

Inspection of the output from the program RoundOff below clearly demonstrates the finite accuracy of real arithmetic, and in particular, highlights the facts that this arithmetic is **neither commutative nor associative.** In other words, with floating-point arithmetic on a computer, it is **not** always true that:

```
a + b = b + a,     and
a + (b + c) = (a + b) + c
```

The first program may take several seconds before any output is generated, so be patient. See if you can extend the program to give a progress report by displaying the iteration counter, i.

```
program RoundOff;
{ This program illustrates roundoff errors with the harmonic seies }
var
  i, k, n     : Integer;
  ForwardSum  : Real;
  BackwardSum : Real;
begin
  n:=10000;
  ForwardSum:=0.0;
  for i:=1 to n do ForwardSum:=ForwardSum + 1/i;
  write('After ',n:6,' terms, forward  sum = ');
  writeln(ForwardSum:16:8);
  BackwardSum:=0.0;
  for i:=n downto 1 do BackwardSum:=BackwardSum + 1/i;
  write('After ',n:6,' terms, backward sum = ');
  writeln(BackwardSum:16:8);
end.
```

Output

```
After  10000 terms, forward  sum =       9.78760597
After  10000 terms, backward sum =       9.78760603
```

Remarks

There is agreement here to six significant figures (5 decimal places), but after that the rest is *noise*. Notice also that the sum is almost 10, after 10,000 terms. Can you estimate how many terms would be necessary (assuming of course, the computer is accurate enough) for the sum to reach 100, or even 1000? Will the sum *ever* get to these values? Remember that we would be adding terms like

$$\frac{1}{1,000,000} + \frac{1}{1,000,001} + \cdots$$

These values are very small, and getting smaller! We have already said that this series (harmonic series) **diverges**. There is only one way that a series of positive terms can diverge. It must tend to infinity. By definition, this means that we can make the partial sums **as large as we please!**; certainly as large as 100, or 1000. Can we prove this? The proof is very instructive, so we'll sketch it here, and then ask you to fill-in the details as an exercise.

Proof That The Harmonic Series Diverges

First, we note the following pattern:

$$S_1 = \frac{1}{1}$$

$$S_2 = S_1 + \frac{1}{2}$$

$$S_4 = S_2 + \left(\frac{1}{3} + \frac{1}{4}\right)$$

$$S_8 = S_4 + \left(\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}\right)$$

$$S_{16} = S_8 + \left(\frac{1}{9} + \frac{1}{10} + \frac{1}{11} + \frac{1}{12} + \frac{1}{13} + \frac{1}{14} + \frac{1}{15} + \frac{1}{16}\right)$$

We have:

$$S_1 = 1$$

$$S_2 = S_1 + \frac{1}{2} = \frac{3}{2}$$

$$S_4 = S_2 + \left(\frac{1}{3} + \frac{1}{4}\right) > S_2 + \frac{1}{4} + \frac{1}{4}$$

i.e., $S_4 > S_2 + \frac{1}{2}$

Similarly,

$$S_8 = S_4 + \left(\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}\right) > S_4 + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right)$$

i.e., $S_8 > S_4 + \frac{4}{8}$ or $S_8 > S_4 + \frac{1}{2}$

In general:

$$S_{2^{n+1}} > S_{2^n} + \frac{1}{2}$$

This means that:

$$S_4 > \frac{4}{2}, \quad S_8 > \frac{5}{2}, \quad S_{16} > \frac{6}{2}, \quad S_{32} > \frac{7}{2}, \quad S_{64} > \frac{8}{2}, \qquad \text{etc.}$$

In general:

$$S_{2^n} > \frac{n+2}{2}$$

How many terms N, would then guarantee that $S_N > 1000$?

A Program For Euler's Constant (Gamma)

We now extend our program to find estimates for the Euler constant.

```
program Euler;
{ Euler's constant : correct to 10 decimal places is 0.5772156649.
  This program gives very poor performance }
var
  i, n : Integer;
  s    : Real;
begin
  n:=100;
  writeln('    n     Approx to Euler constant'); writeln;
  while n<=1000 do begin
    s:=0.0;
    for i:=1 to n do s:=s+1.0/i;
    writeln(n:6,s-ln(n):16:8);
    n:=n+100;
  end
end.
```

Output

```
  n    Approx to Euler constant

 100     0.58220733
 200     0.57971358
 300     0.57888140
 400     0.57846514
 500     0.57821533
 600     0.57804876
 700     0.57792978
 800     0.57784053
 900     0.57777111
1000     0.57771558
```

The accuracy is not spectacular, is it?

14.5 Power Series

Many scientific functions available on pocket calculators and in computer languages such as Pascal are expressible as **power series** (a formal definition is given in the next section). Some common functions are the trigonometric functions sine, cosine, tangent and their inverses, along with the exponential and logarithm functions. The algorithms to compute approximations to functions such as these are generally based on polynomials or rational functions (a rational function is just one polynomial divided by another). In this section we look at only polynomial approximations to functions. Let's first look at an example to illustrate these ideas.

Example

For any natural number n, we may approximate the function $e^x = \exp(x)$ by the polynomial:

$$1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots+\frac{x^n}{n!}$$

That is, we have a **sequence of polynomials,** which progressively better approximate the exponential function:

$$E_0(x) = 1$$

$$E_1(x) = 1+\frac{x}{1!}$$

$$E_2(x) = 1+\frac{x}{1!}+\frac{x^2}{2!}$$

$$E_3(x) = 1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}$$

etc.

Using this information, we can get a good approximation to for example, exp(1) = e ≈ 2.7182818 by taking $E_k(1)$ (i.e. k + 1 terms and with x=1), where k is suitably large.

$$1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{k!}$$

A Pascal program to add up terms until the first (neglected) term is very small is shown below:

```
program Taylor1;
{ This program uses a power series expansion to obtain
approximate values for exp(x) }
var
  x, E : Real;
  i    : Integer;

function Exponential(x : Real) : Real;
const
  Tolerance = 0.5E-8;
var
  Sum, Term : Real;
  i         : Integer;
begin
  Sum:=1.0; Term:=1.0; i:=1;
  while abs(Term) > Tolerance do begin
    Term:=Term*x/i;
    Sum:=Sum + Term;
    i:=i + 1;
    {writeln(i:6,Term,Sum)}
  end;
  {writeln(i,' Terms of Power Series required');}
  Exponential:=Sum
end;

begin  { main }
  writeln('Table Of Exponential Function');
  writeln;
  writeln('    x         Our exp(x)     Pascal exp(x)    Difference');
  writeln;
  for i:=-10 to 10 do begin
    x:=i; x:=x/10.0;
    E:=Exponential(x);
```

```
    writeln(x:8:4,E:16:8,exp(x):16:8,E-exp(x):18:12)
  end
end. { main }
```

Output

```
   x          Our exp(x)      Pascal exp(x)     Difference

 -1.0000      0.36787944       0.36787944      0.00000000015
 -0.9000      0.40656966       0.40656966      0.00000000004
 -0.8000      0.44932896       0.44932896     -0.00000000014
 -0.7000      0.49658530       0.49658530     -0.00000000003
 -0.6000      0.54881164       0.54881164      0.00000000009
 -0.5000      0.60653066       0.60653066      0.00000000001
 -0.4000      0.67032005       0.67032005     -0.00000000003
 -0.3000      0.74081822       0.74081822      0.00000000006
 -0.2000      0.81873075       0.81873075     -0.00000000006
 -0.1000      0.90483742       0.90483742      0.00000000002
  0.0000      1.00000000       1.00000000      0.00000000000
  0.1000      1.10517092       1.10517092     -0.00000000003
  0.2000      1.22140276       1.22140276     -0.00000000007
  0.3000      1.34985881       1.34985881     -0.00000000007
  0.4000      1.49182470       1.49182470     -0.00000000004
  0.5000      1.64872127       1.64872127     -0.00000000002
  0.6000      1.82211880       1.82211880     -0.00000000011
  0.7000      2.01375271       2.01375271     -0.00000000004
  0.8000      2.22554093       2.22554093     -0.00000000017
  0.9000      2.45960311       2.45960311     -0.00000000006
  1.0000      2.71828183       2.71828183     -0.00000000020
```

Notice how we generate each term by simple operations such as multiplication and division from the previous term. For the exponential power series, this **recurrence relation** is a particularly simple one. There is no need to calculate large powers such as x^7 or x^8 from scratch each time, since if we already have x^6, then x^7 is obtained simply by multiplying by x. The Pascal statement that generates each term for us is:

```
Term:=Term*x/i;
```

Notice also how the current value of Term is used on the right-hand side of the assignment to build the next Term by multiplying by x and dividing by i. This is really just making use of the **recurrence relation** for this particular series. Since we have:

$$t_i = \frac{x^i}{i!} \text{ and } t_{i-1} = \frac{x^{i-1}}{(i-1)!},$$

we have the ratio of successive terms for this series:

$$\frac{t_i}{t_{i-1}} = \frac{x^i(i-1)!}{x^{i-1}i!} = \frac{x}{i}$$

Hence, $t_i = \frac{t_{i-1}x}{i}$

Translating this to our program, we end up with Term:=Term*x/i, since we have used Term for t_i on the left and also for t_{i-1} on the right of the assignment statement. We make use of these ideas more fully in the examples to follow on general power series.

Another very important point to note from the program output is that, while the two functions appear to be exactly equal, an inspection of their difference reveals that they may well differ in the tenth or eleventh decimal places. Since floating-point (real) arithmetic is only accurate to eight decimal places at most on Turbo Pascal version 3 on IBM compatibles, it may be concluded that our *Exponential* function agrees with the intrinsic exp(x) function.

Figure 14.3 illustrates the partial sums of the power series used, which are seen to approach exp(x) more closely as n increases. Observe that even $E_3(x)$ is very close according to our graph, on the interval [0,1]. More terms are required however as if x>1 to obtain any kind of reasonable approximation to the exponential function. In actual practice, rational functions are generally used (the so-called **Padé approximations)** because of their greater economy.

Power Series

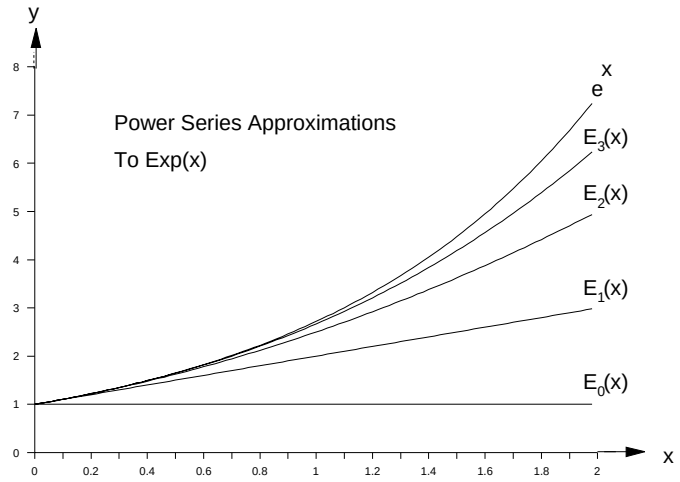


Figure 14.3 Power Series Approximation To Exponential Function

The example we have just considered is an example of a **power series approximation.** The polynomial defined for each value of n above can be regarded as a partial sum for a power series. Speaking somewhat loosely, a power series can be thought of as a polynomial of infinite degree. If we take the sequence of partial sums, each one is a polynomial. A power series is in fact an infinite series. The formal definition is given in Figure 14.4.

DEFINITION OF POWER SERIES

Given a sequence of real numbers $a_0, a_1, a_2, \ldots, a_n, \ldots$ a Power Series (in x) is defined to be:

$$P(x) = \sum_{0}^{\infty} a_n x^n$$

$$= a_0 x^0 + a_1 x + a_2 x^2 + \ldots + a_n x^n + \ldots$$

The coefficients $a_0, a_1, a_2, \ldots$ are real numbers.

Figure 14.4 Definition Of Power Series

For example, our approximating polynomials for exp(x) are all partial sums of the power series:

$$1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots. = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Convergence, Limits, Roundoff Errors

Since power series are infinite series, the question of convergence arises. For a given value of x, the power series is just an infinite series of real numbers. We are interested in whether this infinite series converges to a limit. Convergence of infinite series is a similar concept to the limit concept in calculus.

For a given value of x, the term $a_n x^n$ can be written u_n, and we have a series $\sum u_n$. For any series $\sum u_n$ we may define the **partial sums.**

Definition

The partial sums of a series are defined to be:

$$S_0 = u_0 = \sum_{i=0}^{0} u_i$$

$$S_1 = u_0 + u_1 = \sum_{i=0}^{1} u_i$$

$$S_2 = u_0 + u_1 + u_2 = \sum_{i=0}^{2} u_i$$

$$S_3 = u_0 + u_1 + u_2 + u_3 = \sum_{i=0}^{3} u_i$$

and, in general :

$$S_n = u_0 + u_1 + u_2 + u_3 + \ldots + u_n = \sum_{i=0}^{n} u_i$$

For the series u_n to converge to a limit L, we have the following definition in Figure 14.5.

DEFINITION OF LIMIT OF A SERIES

Given any $\varepsilon > 0$, if there exists an N such that $|s_n - L| < \varepsilon$ whenever $n > N$, then we say that the series has limit L.

Figure 14.5 Definition Of Limit Of Series

This means that the partial sums can be made to approach the limit L as closely as we please, just by taking n large enough. From a computing point of view, this apparently means that we can get our approximation as accurate as we please by taking enough terms and adding them up. This is not quite true on a digital computer since floating-point arithmetic is of limited accuracy. After a certain number of terms, the accumulation of roundoff errors renders any attempts at further accuracy futile.

Error Bounds

For some power series, it is not too hard to place an upper bound on the error committed by terminating the series after a finite number of terms. This is usually called **truncation of power series.** It is of course the same as taking a partial sum, i.e. a polynomial to approximate our power series for a given value of x. To do this we really need the general mathematical theorem which gives rise to many of the power series - Taylor's Theorem. This result is very important, both theoretically and for practical applications of computation such as ours at present. Taylor's Theorem says in essence that any function which is sufficiently smooth (has continuous derivatives of many orders) can be approximated by a power series whose coefficients are proportional to the various derivatives of the function. Let's first state the theorem and then look at some examples. If we first notice that the coefficients of a polynomial function are related to its derivatives of various orders, then Taylor's Theorem should not really come as a surprise.

For if we consider the polynomial:

$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$, we have :

$$a_i = \frac{f^{(i)}(0)}{i!}, \text{ for } 0 \le i \le n.$$

The value of a_i is obtained by differentiating the polynomial expression for f(x) **i** times and putting x=0. For example:

$$f(0) = a_0 = \frac{f^{(0)}(0)}{0!}, \text{ since } 0! = 1$$

Also, $f'(x) = a_1 + 2a_2 x + 3a_3 x^2 + \ldots + na_n x^{n-1}$

Therefore:

$$f'(0) = a_1 = \frac{f^{(2)}(0)}{1!}$$

We leave the rest of these as an exercise. Mathematical induction is a good way establish the general result.

If we assume the result here, then after substituting for the a's, we find for any polynomial f of degree n:

$$f(x) = f(0) + x\,f'(0) + \frac{x^2}{2!}f''(0) + \ldots + \frac{x^n}{n!}f^{(n)}(0).$$

An exercise at the end of this chapter asks you to derive this result.

More generally, if we write x=a+h, where a is fixed, we have for the polynomial f:

$$f(a+h) = f(a) + h\,f'(a) + \frac{h^2}{2!}f''(a) + \ldots + \frac{h^n}{n!}f^{(n)}(a).$$

Our motivation for Taylor's Theorem is that the foregoing results will be at least approximately true even for functions which are not polynomials. Since the derivatives of order up to n are required, then clearly the function in question must be smooth enough to possess such derivatives. In fact, the requirement is that they also be continuous. This will be the case for all functions that we study. The Taylor theorem is stated in Figure 14.6.

TAYLOR'S THEOREM

Suppose that *f(x)* is a real function which has continuous derivatives of all orders up to *n-1* for $a \le x \le a+h$, and $f^{(n)}$ exists for $a < x < a+h$. Then:

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2!}f''(a) + \ldots + \frac{h^n}{n!}f^{(n)}(a) + \frac{h^n}{n!}f^{(n)}(\beta), \quad \text{where} \quad a < \beta < a+h.$$

Figure 14.6 Statement Of Taylor's Theorem

The last term in the formula for Taylor's Theorem is called the remainder term. It is a measure of the extent to which f(x) differs from the polynomial which forms the approximation. Notice that the expression is exact, that is, f(x) is exactly equal to the polynomial of degree n plus the remainder term. The whole idea of having an approximation is that it is in some sense reasonably close to the quantity being approximated. This will be the case of course if our remainder term R_n is small.

$$R_n = \frac{h^n}{n!}f^{(n)}(\beta), \text{ where } a < \beta < a+h.$$

If $f^{(n)}(\beta)$ is not too large then we are ok if n is large enough, especially if $|h| < 1$.

Let's consider the size of this remainder term for the exponential example earlier. (By the way, this power series is just the Taylor series for exp(x) with a=0, and h replaced by x.) We use h=1, so that:

$$R_n = \frac{h^n}{n!}f^{(n)}(\beta)$$

$$\le \frac{h^n}{n!}e, \text{ since } \beta < 1$$

$$= \frac{e}{n!}, \text{ since h = 1}$$

For ten terms (n=10), we have: $\frac{e}{10!} \approx 7.3 \times 10^{-7}$

In the absence of roundoff error, we should then expect accuracy to approximately six decimal places with our estimate of e.

Taylor's Theorem is a powerful tool for computing approximations to functions. Once we write a Pascal program for one function, it becomes a straightforward matter to adapt it for a different function. For the approximation to be useful, however, we must have some reasonable upper bound on our error term (remainder term of Taylor's Theorem). For the common functions, exp, sin, cos, ln, it is easy to get a useful error bound. For practice, our next task is to do a Taylor analysis of the sine function and write a program to approximate it on the interval $(0, \pi/4)$.

Example Of The Use Of Taylor's Theorem.

Let f(x)=sin x. To apply Taylor's Theorem, we need to differentiate f(x) many times. We make the convention that the zeroth derivative is the function itself.

$$f^{(0)}(x) = +\sin x$$
$$f^{(1)}(x) = +\cos x$$
$$f^{(2)}(x) = -\sin x$$
$$f^{(3)}(x) = -\cos x$$
$$f^{(4)}(x) = +\sin x$$
$$f^{(5)}(x) = +\cos x$$
$$f^{(6)}(x) = -\sin x \quad \text{etc.}$$

We can see that the derivatives of the sine function repeat themselves after every four differentiations. They are periodic - is this a surprise when we already know that sine is a periodic function? It is a different type of periodicity, but we can take advantage of it for our present purpose. Since the even-numbered derivatives are proportional to sin x, then they will all be zero if we use a=0 in Taylor's Theorem. This is called "expanding about x=0". Some books take this special case (expansion about 0 and give it a completely different name - Maclaurin's Theorem).

The odd-numbered derivatives are all $\pm\cos x$ and will therefore evaluate to ± 1 at a=0. We end up with:

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

This can be written more compactly as:

$$\sin x = \sum_{i=1}^{\infty} \frac{(-1)^{i-1} x^{2i-1}}{(2i-1)!}$$

For the remainder term we have:

$$|R_n| = \left|\frac{h^n}{n!} f^{(n)}(\beta)\right|$$

$$\leq \frac{h^n}{n!}, \text{ since } |\sin\beta| < 1 \text{ and } |\cos\beta| < 1$$

$$< \frac{1}{n!}, \text{ since } h \leq \frac{\pi}{4} \leq 1$$

A Pascal program for this example follows.

```
program Taylor2;
{ This program uses the power series to determine approx.
values of sine x }
var
  x : Real;
  i : Integer;

function Sine(x : Real) : Real;
const
  Tolerance = 0.5E-8;
var
  Sum, Term, xsq : Real;
  i, NumOfTerms  : Integer;
begin
  Sum:=x; Term:=x; xsq:=x*x; i:=2;
  while abs(Term) > Tolerance do begin
    Term:=-Term*xsq/i/(i + 1);
    Sum:=Sum + Term;
    i:=i + 2;
    {writeln(i div 2:6,Term,Sum)}
  end;
  {NumOfTerms:=i div 2;
  writeln(NumOfTerms,' Terms of Power Series required');}
  Sine:=Sum
end;
```

```
begin  { main }
  writeln('Table Of Sine Function');
  writeln;
  writeln('    x         Our sin(x)     Pascal sin(x)');
  writeln;
  for i:=-10 to 10 do begin
    x:=i; x:=x/10.0;
    writeln(x:8:4,Sine(x):16:8,sin(x):16:8)
  end
end.  { main }
```

Output

```
 Table Of Sine Function

     x         Our sin(x)     Pascal sin(x)

  -1.0000     -0.84147098     -0.84147098
  -0.9000     -0.78332691     -0.78332691
  -0.8000     -0.71735609     -0.71735609
  -0.7000     -0.64421769     -0.64421769
  -0.6000     -0.56464247     -0.56464247
  -0.5000     -0.47942554     -0.47942554
  -0.4000     -0.38941834     -0.38941834
  -0.3000     -0.29552021     -0.29552021
  -0.2000     -0.19866933     -0.19866933
  -0.1000     -0.09983342     -0.09983342
   0.0000      0.00000000      0.00000000
   0.1000      0.09983342      0.09983342
   0.2000      0.19866933      0.19866933
   0.3000      0.29552021      0.29552021
   0.4000      0.38941834      0.38941834
   0.5000      0.47942554      0.47942554
   0.6000      0.56464247      0.56464247
   0.7000      0.64421769      0.64421769
   0.8000      0.71735609      0.71735609
   0.9000      0.78332691      0.78332691
   1.0000      0.84147098      0.84147098
```

Discussion Of Output

It is evident that our algorithm for estimating sin(x) is in excellent agreement with that of our Pascal compiler. See if you can find other values of x where the two estimations for the sine function differ. Hint: try larger values of x.

14.6 Historical Note

Karl Friedrich Gauss (1777-1855)

Karl Friedrich Gauss was born in Brunswick, Germany and was early recognized as an child prodigy, obtaining his doctorate at Göttingen. Like Euler, there was virtually no area of mathematics that remained untouched by Gauss. The benefits of his work can be seen in all branches of pure and applied mathematics, and he also made gigantic strides in theoretical physics and astronomy. We really only have space for a brief selection of his discoveries:

- fundamental theorem of algebra (every polynomial has a zero),
- many results in number theory,
- proof of the possibility of construction of 17-sided polygon by ruler and compass (done before he was 20)
- proof that the regular polygon of n sides is constructable by ruler and compass alone if and only if n is a prime of the form:
  $n = 2^p + 1$, where $p = 2^k$, and k = 0,1,2,3,...
- the method of least squares
- terrestrial magnetism, inverse-square forces, electrostatics, potential theory

Gauss was often reluctant to publish his results, and in a number of cases, including the important field of **Non-Euclidean Geometry,** his memoirs indicated that he was in possession of knowledge thought to be originally discovered by his mathematical successors many years after his death. A monument to the great man remains in Braunschweig and has a base in the form of a polygon of 17 sides. At no time during his life did he leave his native Germany.

The story of Newton's inspiration for the theory of gravitation by watching an apple fall from a tree is said by Gauss to be Newton's way of dismissing stupid persons who asked him how he formulated his theory.

The following very interesting quotation is taken from Morris Kline's excellent book "**Mathematics In Western Culture**", and gives some indication of the enormous ability but great humility of Gauss:

> *"Of these three the greatest, and one who ranks with Newton and Archimedes, was Karl Friedrich Gauss. Karl showed unbelievable precocity in many fields and a particular predilection for mathematics. When as a young man he proved that the regular polygon of 17 sides could be constructed with straightedge and compass, he was so delighted that he abandoned his intention of becoming a philologist is order to study mathematics. He soon contributed masterful work to many branches of the subject and also achieved note as an inventor and experimentalist. Though his contributions were no less numerous than those of other mathematicians, Gauss was extremely modest. He said, 'If others would but reflect on mathematical truths as deeply and as continuously as I have, they would make my discoveries.' Those who believe that genius is 99 per cent perspiration as well as those who despair of their mathematical abilities may find comfort in Gauss's statement."*

14.7 Summary

- Functions on a finite set may be defined using a formula (direct or recurrence) or by exhaustive enumeration.
- The Fibonacci sequence, AP, and GP are probably best remembered using the recurrence formula, since the relationship between successive terms is readily apparent.
- Programs to generate the terms and partial sums of the Fibonacci sequence, AP, GP are easily written. The normal precaution of guarding against roundoff errors needs to be observed. Floating-point arithmetic cannot generally be assumed to be commutative or associative.
- The harmonic series
  $$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}$$
  although appearing to be convergent is in fact divergent.
- Many scientific functions can be approximated using a power series. For example:
  $$e^x \approx 1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots+\frac{x^n}{n!}$$
  To evaluate e, we let x=1. Clearly, the number of terms taken affects the accuracy of the approximation.
- The Taylor theorem allows us to express any sufficiently smooth (has continuous derivatives of many orders) function as a power series approximation.
- An example of an application of Taylor's theorem is the power series expansion for sin x about x=0, i.e.
  $$\sin x = x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\ldots$$

14.8 Exercises

1. The sequence of partial sums S_n, for the Fibonacci sequence is defined as for any sequence:
   $$S_n = F_1+F_2+\ldots+F_n = \sum_{i=1}^{n} F_i.$$
   See if you can derive a more compact formula for S_n, in terms of just one of the terms of the original Fibonacci sequence, i.e. F_k, for some k (depending on n, of course).

Hint: Tabulate the original terms, the partial sums alongside each other and look for a pattern. Then formulate a likely relationship and try to prove it. Some clever algebra should do the trick, or if stuck, you can try **mathematical induction.**

2. Write a Pascal program to generate the table of the previous question.

3. For the Fibonacci sequence $\{F_n\}$ verify that:

$$F_3^2 + F_4^2 = F_7$$
$$F_4^2 + F_5^2 = F_9$$
$$F_5^2 + F_6^2 = F_{11}$$
$$F_6^2 + F_7^2 = F_{13}$$

What is the general relationship here? Can you prove it? Write a program to verify your conjecture for n=1 to 18.

4. (a) Show that:

$$\frac{F_{n+1}}{F_n} < 2$$

for all natural numbers n.

(b) Write a Pascal program to compute the ratio:

$$\frac{F_{n+1}}{F_n}$$

for n=1 to 20. Does this ratio seem to approach a limit? Look at your explicit formula for the nth term of the Fibonacci sequence and substitute into this ratio. See if you can prove what the limit will be. (Remember that a **proof** is required, since counterexamples of "false convergence" can easily be found. In other words, it is easy to be fooled by computer output which may seem to show apparent convergence. The harmonic series is an excellent example of such a series).

5. **Challenge Problem:** Show that every fifth Fibonacci number is divisible by five. *Hint:* Suppose the 5n th Fibonacci number were not divisible by 5 for some n. Let F be the first Fibonacci number where this happens. Let A, B, C, D, E be the five preceding Fibonacci numbers. Show that A must be divisible by 5, and the use the Fibonacci recurrence formula to propagate this information up the sequence to get information about F. Establish a contradiction and you have the result.

6. Write a Pascal program to compute every fifth Fibonacci number and print it out. Let it go on indefinitely and check that each one is divisible by 5. What finally happens? Explain.

7. Extend the AP program to also output a column which contains the sequence of partial sums of the AP. Do the same for the GP program.

8. **Challenge Problem:** Find the missing term in the following sequence:

10, 11, 12, 13, 14, 15, 16, 17, 20, 22, 24, 31, 100, ?, 10000, 1111111111111111.

Hint: Regard the terms of the sequence as representations of the same number in different bases.

9. Write a Pascal program to compute the product:

$$\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdots\frac{2n}{2n-1}\cdot\frac{2n}{2n+1}$$

for values of n from 1 to 100. The limit of this product as n tends to infinity is $\pi/2$ and is known as **Wallis' product.** Verify that your program produces reasonable results. ($\pi/2 \approx 1.570796$.) Does this method of computing pi/2 give very accurate results? What happens as n approaches 100? Explain this seemingly strange behaviour. You may get a clue by writing out more details each time around the loop. Just exactly what is it that we are multiplying each time? Is this the same phenomenon as the roundoff-errors we had with the Euler constant example?

10. For the sequence of polynomials $E_n(x)$ discussed in this chapter for approximating the exponential function, fill out the following table, where each entry pertains to $E_n(1)$, so that the true value being approximated is e.

n	Absolute Error	Relative Error	Percent Error
0			
1			
2			
3			

11. Write a Pascal program to produce the table above with correct contents.

12. Determine Taylor series for the following functions about x=0:
    (a) $f(x) = \cos x$
    (b) $f(x) = x^3 + x^2$
    (c) $f(x) = \ln(1+x)$

13. Let y = arctan x. We wish to estimate π by substituting x=1 and then multiplying by 4. Differentiating once, we have:

$$\frac{dy}{dx} = \frac{1}{1+x^2}.$$

To generate a Taylor series for arctan x (expanded about x=0), we need to have some general expression for the nth derivative of this function at x=0. These are not hard to derive directly with the aid of **Leibniz's Theorem;** a result which allows us to write the nth derivative of a product as a kind of **binomial expansion** (it's really just the product rule for derivatives applied n times). Since this result is a little beyond the scope of this book, we obtain a power series for arctan x is a different manner; but note that it is the same one that would be obtained from Taylor's Theorem. We first consider the **geometric progression** with first term a=1, and common ratio $r = -x^2$:

$$1 - x^2 + x^4 - x^6 + \ldots$$

Since the sum to infinity of a GP is just a/(1-r), provided | r | <1, we have:

$$1 - x^2 + x^4 - x^6 + \ldots = \frac{1}{1+x^2} \text{ for } |x| < 1$$

Assuming that we can integrate both sides of this equation we have:

$$x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots = arctan\, x \text{ for } |x| < 1$$

Use this series to estimate π as $\pi \approx 4\, arctan(1.0)$. Write a Pascal program to estimate π in this manner by substituting x=1.

14. The series for arctan(x) of the previous question can be used in a much more cunning way to estimate π. The problem with just substituting x=1 in this series is that the terms decrease too slowly (we are right on the edge of the **interval of convergence** of this series : x=1). We first need some simple trig identities:

(a) Since $\tan(a+b) = \frac{\tan a + \tan b}{1 - \tan a . \tan b}$, by substituting $A = \tan a$, $B = \tan b$, show that:

$$arctan A + arctan B = arctan\left(\frac{A+B}{1-AB}\right)$$

(b) We wish to find A, B such that $\frac{A+B}{1-AB} = 1$

This leads to A+B+AB = 1. The simplest case is A=1/2, B=1/3. Use these values to substitute in our series for arctan x to get a much more rapidly converging series for arctan(1) than in the previous question.

(c) Try for an even faster series by getting A, B both small (the smaller, the better). Hint: try A=1/4. In each case, use your Pascal program of the previous question to do all the arithmetic for you. Comment on the relative accuracies for each of the values of A and B (last question corresponds to A=1, B=0.) and the number of iterations required to achieve the accuracy.

15. Show by successive differentiation that for any polynomial, f(x), of degree n or less:

$$f(x) = f(0) + x\, f'(0) + \frac{x^2}{2!} f''(0) + \ldots + \frac{x^n}{n!} f^{(n)}(0).$$

16. Fill out the details of the proof that the harmonic series is divergent. Use the outline given in this chapter to show that: given any number M, S_n can be made larger than M, just by taking n large enough, i.e. enough terms. Do this by explicitly giving a formula for n in terms of M. In particular, show that:

$$S_{2^n} > \frac{n+2}{2}$$

17. Show that the Taylor Series for sin(x), i.e.

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

can be written more compactly as:

$$\sin x = \sum_{i=1}^{\infty} \frac{(-1)^{i-1} x^{2i-1}}{(2i-1)!}$$

18. The Fibonacci numbers (terms of the sequence) have many interesting interrelationships other than the ones we have considered in this chapter. For example, let's look briefly at the two numbers, p and q, that were the solutions of our quadratic equation $x^2 - x - 1 = 0$. Remembering that for the quadratic equation $Ax^2 + Bx + C$ we have:

Sum of Roots $= \frac{-b}{a}$, and Product of Roots $= \frac{c}{a}$,

then p+q=1, and pq=-1, since p and q are roots of this quadratic. This means that p and q are numbers such that their sum plus their product equals zero. Are there any other numbers with this property? If so, then find some.

Index

Table of Contents

Table of Figures

List of Pascal Programs

1) Level: Warning
Message: Invalid date/time string
Location: Global Page Settings

2) Level: Format Error
Message: Page break required with Keep enabled
Location: Document Body
Page: 1

3) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 21 Distance from TOF: 7.762in
Level: 2 Section: 2.3 Block: Text #7 Column: 1
Table 2.1 Algorithms & Processes

4) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 22 Distance from TOF: 5.392in
Level: 2 Section: 2.3 Block: Text #21 Column: 1
Table 2.2 Algorithmic Primitive Operations

5) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 26 Distance from TOF: 8.508in
Level: 2 Section: 2.4 Block: Text #15 Column: 1
Figure 2.2 Making a Cup Of Coffee - Second Attempt

6) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 27 Distance from TOF: 3.350in
Level: 2 Section: 2.4 Block: Text #24 Column: 1
Figure 2.3 Making a Cup of Coffee - Third Attempt

7) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 29 Distance from TOF: 5.424in
Level: 2 Section: 2.7 Block: Text #22 Column: 1
Your algorithm should handle the usual case of two different, real roots as well as the special

8) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 31 Distance from TOF: 3.568in
Level: 2 Section: 3.2 Block: Text #13 Column: 1
Figure 3.2 Structure Chart - Second Attempt

9) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 31 Distance from TOF: 6.077in
Level: 2 Section: 3.2 Block: Text #20 Column: 1
Figure 3.3 Structure Chart - Third Attempt

10) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 34 Distance from TOF: 2.281in
Level: 2 Section: 3.3 Block: Text #30 Column: 1
Table 3.2 Pascal Arithmetic Operators

11) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 34 Distance from TOF: 5.896in
Level: 2 Section: 3.3 Block: Text #36 Column: 1
Table 3.3 Pascal Relational Operators

12) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 34 Distance from TOF: 9.173in
Level: 2 Section: 3.3 Block: Text #43 Column: 1
Table 3.4 Valid & Invalid Real Numbers in Pascal

13) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 35 Distance from TOF: 3.642in
Level: 2 Section: 3.3 Block: Text #50 Column: 1
Table 3.5 Real Output Formatting In Pascal

14) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 36 Distance from TOF: 9.744in
Level: 2 Section: 3.4 Block: Text #18 Column: 1
Table 3.6 Algebraic Expressions In Pascal

15) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 39 Distance from TOF: 6.147in
Level: 2 Section: 3.6 Block: Text #2 Column: 1
```
then DIFFERENCE := FIRST - SECOND
```

16) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 39 Distance from TOF: 6.300in
Level: 2 Section: 3.6 Block: Text #2 Column: 1
else DIFFERENCE := SECOND - FIRST

17) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 39 Distance from TOF: 9.008in
Level: 2 Section: 3.6 Block: Text #6 Column: 1
then alternative 1

18) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 39 Distance from TOF: 9.161in
Level: 2 Section: 3.6 Block: Text #6 Column: 1
else alternative 2

19) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 46 Distance from TOF: 3.253in
Level: 2 Section: 3.8 Block: Text #5 Column: 1

Table 3.9 Definite & Indefinite Iteration

20) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 49 Distance from TOF: 7.795in
Level: 2 Section: 3.11 Block: Text #3 Column: 1
case

21) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 49 Distance from TOF: 7.948in
Level: 2 Section: 3.11 Block: Text #3 Column: 1
if .... then .... else

22) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 49 Distance from TOF: 8.101in
Level: 2 Section: 3.11 Block: Text #3 Column: 1
while ..... do

23) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 49 Distance from TOF: 8.254in
Level: 2 Section: 3.11 Block: Text #3 Column: 1
`for .....to/downto. ....do`

24) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 63 Distance from TOF: 4.115in
Level: 2 Section: 4.6 Block: Text #1 Column: 1
1. An important characteristic of arrays is:

25) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 65 Distance from TOF: 1.377in
Level: 2 Section: 4.6 Block: Text #39 Column: 1
Compare your result with the value given by the SIN function. Remember that π radians =

26) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 73 Distance from TOF: 4.673in
Level: 2 Section: 5.3 Block: Text #9 Column: 1
Figure 5.7 The Graph of f(x)=(x-4)(x+2)(x-6).

27) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 1.573in
Level: 2 Section: 8.1 Block: Text #8 Column: 1
var Frequency:**array**[0..100] **of** Integer;

28) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 1.573in
Level: 2 Section: 8.1 Block: Text #8 Column: 1
var Frequency:**array**[0..100] **of** Integer;

29) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 3.916in
Level: 2 Section: 8.1 Block: Text #13 Column: 1

30) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 4.104in
Level: 2 Section: 8.1 Block: Text #13 Column: 1

31) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 4.313in
Level: 2 Section: 8.1 Block: Text #13 Column: 1

32) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 5.124in
Level: 2 Section: 8.1 Block: Text #13 Column: 1

33) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 5.332in
Level: 2 Section: 8.1 Block: Text #13 Column: 1

34) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 5.521in
Level: 2 Section: 8.1 Block: Text #13 Column: 1

35) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 105 Distance from TOF: 5.729in
Level: 2 Section: 8.1 Block: Text #13 Column: 1

36) Level: Warning
Message: Picture is too large; will be automatically scaled
Location: Document Body: [EQUATION]
Page: 112 Distance from TOF: 8.588in
Level: 2 Section: 8.7 Block: Text #13 Column: 1

37) Level: Warning
Message: Picture is too large; will be automatically scaled
Location: Document Body: [EQUATION]
Page: 112 Distance from TOF: 8.894in
Level: 2 Section: 8.7 Block: Text #13 Column: 1

38) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 130 Distance from TOF: 3.684in
Level: 2 Section: 10.3 Block: Text #6 Column: 1

39) Level: Warning
Message: Tab error (tab character occurs after line wraps)
Location: Document Body
Page: 135 Distance from TOF: 7.115in
Level: 2 Section: 10.6 Block: Text #10 Column: 1
an equation of the form Observations taken at 1, 2, and 3 seconds after firing

40) Level: Warning
Message: Missing opening or closing bracket ([])
Location: Document Body: [EQUATION]
Page: 172 Distance from TOF: 7.528in
Level: 2 Section: 14.5 Block: Text #17 Column: 1

41) Level: Warning
Message: Picture is too large; will be automatically scaled
Location: Document Body: [EQUATION]
Page: 174 Distance from TOF: 3.674in
Level: 2 Section: 14.5 Block: Text #40 Column: 1

42) Level: Format Error
Message: Tab error (a tab character occurs after the last tab setting)
Location: Document Body
Page: 174 Distance from TOF: 3.674in
Level: 2 Section: 14.5 Block: Text #40 Column: 1