# APPLICATION OF C++ TO SOME NUMERICAL METHODS

BY

<u>OLUFUYE</u>, JIMSON OWODUNNI

**M. Tech/051**

**BEING A PROJECT SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF A MASTER OF TECHNOLOGY (M.Tech) DEGREE IN MATHEMATICS IN THE DEPARTMENT OF MATHEMATICS & COMPUTER SCIENCE, SCHOOL OF SCIENCE & SCIENCE EDUCATION, FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA**

**SEPTEMBER 21, 1998**

## CERTIFICATION

This is to certify that this project work fulfils one of the requirements for the award of a Masters in Technology (M. Tech.) Degree in Mathematics/Computer Science.

**Professor K. R. Adeboye** **Date**
**Supervisor**

**Dr. S. A. Reju** **Date**
**HOD (Mathematics/Computer Science)**

**Professor Onumanyi** **Date**
**External Examiner**

## DEDICATION

It is indeed with awe I dedicate this project work to my Lord and Saviour Jesus Christ, to God the Father and to the Holy Spirit, the Spirit of truth and understanding!

## ACKNOWLEDGEMENT

For this project work to have been done successfully, a lot of people played very important role. I wish to thank Professor K. R. Adeboye, Dean of the School of Science & Science Education and my supervisor for his patience and academic guidance. My appreciation also goes to Dr. S. A. Reju Head of Department of Mathematics/Computer Science, for his encouragement to get the job done once and for all having appreciated my demanding business schedules. Also, my appreciation goes to members of staff of the Department of Mathematics/Computer Science for their very useful contributions. I want to really thank Mr Azu Francis who tirelessly typed and retyped the manuscript whenever data crash was experienced. Mr Biodun Fasasi and Mr Chidi Onwumere are appreciated for assisting with editing/keying-in of the programming codes. I want to specially appreciate my wife Mrs J. O. Olufuye for her fervent support and encouragement.

Above all, I want to thank the almighty God for His mercies and strength.

# TABLE OF CONTENT

## ABSTRACT

Several works have been done on the methods of obtaining 'reliable' solutions to mathematical problems. The extent of reliability of solutions obtained to any given mathematical problem is usually connected to the method, approach and tool utilized. In this work, a novel approach of obtaining reliable numerical solutions through a contemporary high level programing language, C++, is employed.

Bearing in mind that a lot of time is spent in computation manually or through the use of a pocket calculator, this project work is set out using C++ to reduce drastically processing time and effort.

A wide range of numerical analysis problems are considered. These include: root-finding, linear and non-linear equations and approximation of continuous functions by Least Square, Legendre and Chebyshev Polynomials. On discussion of several schemes available, the corresponding C++ program is constructed and tested, and results thus obtained are subjected to pairwise comparison and analysis. At the end, the C++ program definition and approach constructed can be applied to the construction of codes for other related schemes on numerical analysis.

# CHAPTER 1

# INTRODUCTION

## 1.1 NUMERICAL ANALYSIS

The subject of numerical analysis is concerned with the *derivation, analysis* and *implementation* of methods for obtaining *reliable* numerical solutions to mathematical problems. The adjective 'reliable' is used to indicate that it is essential to have confidence in any results obtained and an assessment of reliability can form part of the analysis of a method. In subsequent chapters we shall apply numerical methods to a variety of problems including finding some or all of the roots of an algebraic equation, solving a set of linear simultaneous equations and calculating the value of a definite integral. At this point a number of preliminary observations on the words 'derivation', 'analysis' and 'implementation' can be made.

The 'derivation' stage is concerned with deriving and describing the sequence of numerical steps which it is expected will eventually lead to the required numerical results. The complete description of these steps, perhaps written in some pseudo-programming language, is called an *algorithm*. This stage may or may not be easy and often intuition and experience will play an important role. As a single example of a derivation, we may cite the so-called trapezium rule in which we try to estimate the value of the definite integral

$$I = \int_a^b f(x)\,dx \qquad b>a \qquad (1.1)$$

In Fig. 1.1 (a), we plot the curve, y=f(x). Then, remembering the interpretation of a definite integral, if A is the point (a,f(a)) and B is the point (b,f(b)), we can say that an approximation to I is given by the shaded area.

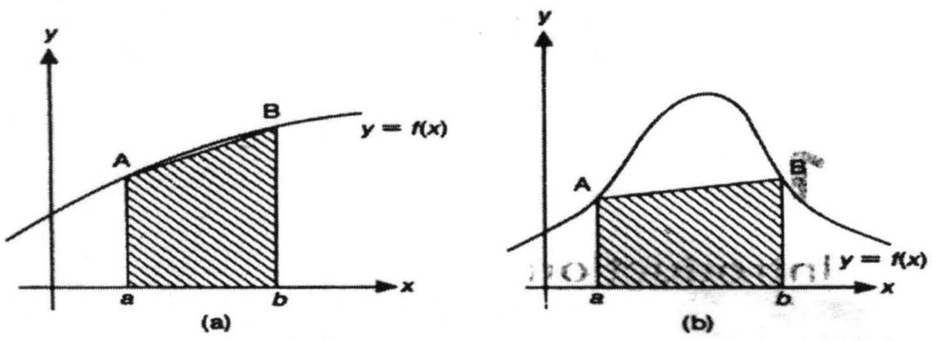


Fig. 1.1.

This corresponds to the area of the trapezium with base b-a and vertical sides f(a) and f(b); that is,

$$I \simeq \frac{b-a}{2}[f(a)+f(b)] \tag{1.2}$$

One now needs to ask immediately about the precision of the approximation (1.2). Trying to find an answer to this question forms a part of the 'analysis' stage to which we are making reference. For the present it can be observed that the error is represented by the unshaded part of the area under the curve. It may be 'small' in the sense that the trapezium rule of approximation is sufficiently accurate for the purpose at hand, or it may be unacceptably large as illustrated in Fig. 1.1(b). It is important to realise that, in general, it would be difficult to find this error. What one can do, however, is to try to find a bound for it; that is, if $E$ denotes the error, we try to find a positive number $M$ for which we can assert that $|E| \leq M$

Intuitively, one would expect that an improvement in precision can be obtained by dividing the interval of integration into a number of sub-intervals and then applying the trapezium rule to each sub-interval in turn.

While $C^{++}$ programming language will be used to implement several numerical methods, sufficient attention will be paid to efficiency; by this it means that time and storage requirements must not be excessive. Suppose that we wish to estimate (1.1) using the trapezium rule with $f(x)=2x^3-3x^2+4x+1$. Clearly, one needs to evaluate $2a^3-3a^2+4a+1$ (and, of course, $2b^3-3b^2+4b+1$) and this would appear to involve six multiplications, one subtraction and two additions. However, by expressing f(a) as $((2a-3)a+4)a+1$ we can reduce the number of multiplications to just three. Although this may seem a fairly small reduction in the number of operations (and hence the time taken to evaluate the expression) for high degree polynomials the savings achievable are very large indeed.

## 1.2 MATHEMATICAL FORMULAE

There are occasions when a numerical answer can be obtained satisfactorily using a known mathematical result. A simple example of this is furnished by the calculation of

$$I = \int_0^\alpha \cos(x)\,dx$$

where $\alpha$ is a given number. It can easily be verify that $I=\sin(\alpha)$ and, using a pocket calculator, say, *I* can be computed for any given value of $\alpha$ (to an accuracy which will depend on the particular calculator used). However, there are also occasions when a valid mathematical result is not always useful for computational purposes or is even useless. A simple example of the first situation is furnished by the problem of finding the roots of the quadratic equation

$$ax^2+bx+c = 0 \qquad a\neq 0$$

We can write these down formally as

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

and

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

To show that these results may not be satisfactory for computational purposes, we suppose a, b and c are such that $b^2$ is very much greater than 4ac. Then the calculation of $x_1$ (if $b>0$) or $x_2$(if $b<0$) involves forming the difference of two nearly equal numbers (since $\sqrt{(b^2-4ac)}\approx b$) with the consequence that there will be a loss of significant figures. As an example, the equation $x^2-200x+5=0$ has roots $100\pm\sqrt{9995}$ which, working to six significant figures, gives $100.000 \pm 99.9750$, that is 199.975 or 0.0250 and the second root is comparatively inaccurate. (This root may be obtained more accurately by observing that the product of the roots is 5 so that the second root can be estimated as 5/199.975.)

## 1.3 THE IDEA OF ITERATION

Iteration is an important numerical technique and we introduce the idea by deriving in a simple way a possible method for the estimation of the *m*th root of a given positive number a. Consider m=2, so that the square root is required, and suppose that an initial approximation, $x_o$, to $\sqrt{a}$ is available.
(For example, if a=50 we could take $x_o=7$.) Since $x_o$ is an approximation to $\sqrt{a}$ then so is $a/x_o$ and, moreover, if $x_o>\sqrt{a}$ then $a/x_o<\sqrt{a}$ and conversely. This suggests that a better approximation to a can be calculated as

$$x_1 = \frac{1}{2}(x_0 + \frac{a}{x_0})$$

We can repeat the argument with $x_1$ in place of $x_o$ and obtain a new approximation as

$$x_2 = \frac{1}{2}(x_1 + \frac{a}{x_1})$$

Proceeding in this way we can summarise the method as

$$x_n = \frac{1}{2}(x_{n-1} + \frac{a}{x_{n-1}}) \qquad n = 1, 2, \ldots$$

so that from $x_o$ we can generate a sequence of numbers $x_1, x_2, \ldots$ which can be easily written as $\{x_n\}$. This is another example of the derivation of a numerical method. The method is called iterative (that is, repetitive) and, among other things, such methods always require an initial approximation (in some methods, more than one) to the quantity we wish to find, in this case $\sqrt{a}$.

## 1.4 ERRORS

### 1.4.1 Sources of Error

We note the following sources of error, most if not all of which will be present in a computation of any size. (I exclude from detailed discussion the blunder due to human error in programming or implementation and the malfunction due to a computer error through hardware or through some failure in a software system that is supposed to be reliable).

(i) We have said that we are trying to obtain numerical answers to a mathematical problem. The mathematical problem will very probably have arisen as a part of modelling some situation in the physical, life or sciences.

(ii) Many numerical processes involve a truncation error. Examples of this situation include the use of a finite number of terms from an infinite series or the use of an approximation to estimate the value of a given definite integral (and the unshaded area under the curve in Fig. 1.1 can be regarded as the truncation error in applying the trapezium rule).

(iii) Some or all of the initial data of the problem may be subject to uncertainty. The measurement of length, weight, etc will produce data whose accuracy is limited by the calibration of the equipment being used.

(iv) The numbers we work with will be subject to rounding-off and we now spend some time examining some elementary ideas concerned with rounding-off, the representation of numbers in a binary computer, and the way in which a computer performs the basic arithmetic operations of addition, subtraction, multiplication and division.

### 1.4.2 Rounding-off Errors

Thinking of the decimal case for the time being, some rational numbers require an infinite number of digits after the decimal point. Examples are 5/3 and 22/7 where we have the fixed point representations 1.6666... and 3.1428571428571... In the second case it can be verified that the sequence of digits 142857 repeats indefinitely. In practice, we can work only with numbers containing a finite number of digits and we can work only with numbers containing a finite number of digits and we can approximate 22/7 by, say, the number 3.142857 with error 0.00000014... We say that 3.142857 is 22/7 *correct rounded to six decimal place accuracy* and 0.00000014... is known as the *rounding-off error*. Another way of describing the accuracy of 3.142857 is to say that it is 22/7 correct to seven *significant figures*. If we express a number in its normalised floating point format, the number of digits after the point gives the number of significant figures. Thus 0.00000014(=$0.14 \times 10^{-6}$) is 0.000000142857... correct to eight decimal places but only two significant figures. We note that rounding-off errors may occur even when it is possible to express a decimal number using a finite number of digits. Thus, an exact number in which we are interested might be 187.452963 but, because of limitations imposed by computing machinery, suppose we can work only with six digits available for the mantissa. The approximation to this number is $0.187453 \times 10^3$ and a rounding error of $0.37 \times 10^{-4}$ has been incurred.

# CHAPTER 2

# INTRODUCTION TO C$^{++}$

## 2.1 About C$^{++}$

The C$^{++}$ programming language was designed by B. Stroustrup and published in his book The C$^{++}$ Programming Language in 1986. C++ has been derived from the well-known programming language C. The original version of which we can write in a C program to increment a variable C. The original version of C was published by B. W. Kernighan and D. M. Ritchie in The C Programming Language in 1978. The second edition of their book, published in 1988, was a revised edition of the language, known as ANSI C. The languages C$^{++}$ and ANSI C are closely related, and both are successors to the original C language. Although C$^{++}$ is a much younger language than C, its use is already widespread, and its popularity will no doubt increase considerably as a result of the excellent quality of popular compilers such as Turbo C$^{++}$ from Borland.

One of the attractive aspects of C$^{++}$ is that it offers good facilities for object-oriented programming (OOP), but, as a hybrid language, it also permits the traditional programming style, so that programmers can shift to object-oriented programming if and when they feel the need to do so . In this regard, C$^{++}$ differs from some purely object-oriented languages, such as Smalltalk and Eiffel. Viewed from the angle of many C programmers, C$^{++}$ is simply 'a better C'. Besides the important class concept, essential to OOP, there are many other points in C++ that are not available to C programmers. To mention just a few, related to functions, we have function overloading, inline functions, default arguments, type-safe linkage, and the very simple requirement that functions be declared before they are used. In ANSI C, the old practice of using undeclared functions is still allowed in order to keep many exiting C programs valid; in C$^{++}$ it is not.

The point just mentioned and some others make C++ much 'safer' than C, but unlike Pascal, C$^{++}$ offers the same flexibility as C. This use of the word safe refers to what happens with incorrect programs. In this regard, assembly language is extremely 'unsafe', but this does not mean that programs written in assembly language cannot be perfectly correct and reliable. They can, and so can C programs. Most experienced programmers want as much control over the computer system as is possible and will therefore prefer C or C$^{++}$ to Pascal.

In general, realistic and easy-to-use programs are not always easy to read. It is very difficult not to make errors when complicated programs need to be modified. With C$^{++}$, this situation is much better than with some other languages in that we can define our own language extensions, known as classes. We write these in separate modules (called implementations), and simply use such language extensions in our main programs, or rather, in our application modules, which can then be kept much simpler than would be the case otherwise. For example, a program that performs arithmetic with complex numbers is likely to be complicated if it is to be programmed in a language that does not support type complex with its associated operators +,-,*,/. Although $C^{++}$ has no built-in facilities for complex arithmetic, we can define ourselves in such a way that application programs can be written easily as facilities were supported by the language itself. we can say that in this way we are extending the language. User-defined language extensions have the advantage of flexibility.

## 2.2 Our First $C^{++}$ Program

Let us begin with the $C^{++}$ program EXAMPLE1, which reads two integers a and b from the keyboard to compute both u =$(a+b)^2$ and v = $(a-b)^2$

```
/* EXAMPLE1: A program to compute the squares of the sum and the difference of two given imageries.
*/
#include <iostream .h>
main ()
{
cout << "Enter two integers: "; // Displays integer request
        int a, b:
        cin>> a >> b;              // Reads a and b
        int sum = a + b, diff = a - b,
                u = sum * sum, v = diff * diff;
        cout << "Square of sum        :" << u << endl:
        cout << "Square of difference: "<< v << endl;
}
```

After typing this program using a program editor, we save it as the file *EXAMPLE 1.CPP*. The filename extension .cpp distinguishes it from C programs. Turbo C++, for example, actually consists of both a C and a C++ compiler, and it depends on the filename extension which one is used. Since our program contains several elements that are specific to C++, it must be compiled with a C++ compiler: a compiler for plain C would display many error messages.

After compiling and linking. we can execute the program. Then the following text appears on the computer screen:

```
Enter two integers:
```

One can now enter, for example,

```
100 10
```

After pressing the Enter key, the following appears:

```
Square of sum      :12100
Square of difference: 8100
```

We can easily check these values: with a = 100 and b = 10, it follows that sum = 110 and diff = 90, and by squaring these we find 12100 and 8100 as the values of u and v, respectively.
We can write C++ programs only if we are familiar with some 'grammar rules'. Let us now briefly discuss these rules as far as they apply to our example. It is good practice to start any program with comment. This can be done in two ways. The notation also available in C is to let comment begin with the two characters

```
/*
```

and to let it end with the same characters in reverse order:

```
*/
```

These two character pairs may or may not be on the same program line. By contrast, there is a way of writing comment that is new in C++: we let comment begin with the two characters

```
//
```

The end of the line is then simply the end of the comment. As it can be seen, both ways of writing comment have been used in program EXAMPLE1.

After the final characters */ of the comment at the top of this program, we find the following include line:

```
#include <iostream.h>
```

We say that this line 'includes' the file iostream.h, which is a so-called header file (hence the file-name extension .h) for 'stream input and output'. The contents of this header file logically replaces this include line of their own. For example, you cannot write main() at the end of this line.

In this regard include lines form an exception to a general rule that says that, as far as the C++ compiler is concerned, program text may be split up over several lines as we please. For example, we can replace the line

*Int sum =a+b, diff=a-b;*

With the following two lines:

```
Int sum = a + b,
    diff = a-b;
```

We can even split these lines further, but that would obviously not improve readability. When splitting a line into two new lines, we say that we insert a new line character. Similar characters are the blank (that is, the space character) and the tab. Collectively, these three characters are called white-space characters. Every C++ program contains one or more function, one of which is called main. (Note that we do not use the term 'function' in the abstract, mathematical sense; instead, a function denotes a concrete program fragment, that is, a sequence of characters!) In our example, the main function is the only one. It has the form

```
Main ()
{...
...
}
```

Functions may or may not have parameter. We write them between parentheses, as is done here. The 'body of every function is surrounded by braces {}. It is good practice to write the two braces of a pair either on the same line, or in the same (vertical) column, with everything in between indented as shown in EXAMPLE1. After the open brace { of a function, we write so-called statements. As long as we are not using compound statements, every statement ends with a semicolon. You can count six semicolons in program EXAMPLE1, each is the end of a statement. We very often write precisely one statement on a line. However, there may be more than one statement on a line, and a statement may take more than one line, as this statement shows:

*Int a, b;*

Without being initialized immediately after this declaration their values are undefined. A completely different statement is

```
Cout << "Enter two integers:";
```

It is the typical C++ way of displaying some piece of text on the video screen. We say that cout is the standard output stream, to which we can send characters by means of the operator <<. Note that this operator is written as a character pair which looks like an arrow head that points to the left. It therefore suggests that the characters between the double quotes in "Enter two integers:" are sent to the steam cout. Instead of saying that we 'send characters to the output stream cout' (or to the video scene), we sometimes say that we print these characters. Analogously, the statement

```
Cin>> a >> b;
```

Reads two values from the standard input stream (that is, form the keyboard), and stores them into the variables a and b. The character pair >> may be associated with a arrow head pointing to the right, so the values go from cin to a and b. When executing this statement, the machine will be waiting for input, so we can now enter the two integers as requested.

After u and v have been computed, the values are printed as follows:

```
Cout << "Square of sum          : " << u << endl:
Cout << "Square of difference   : " << v << endl:
```

We use endl to indicate that we are at the end of the line. Instead of endl, we could also write \n , or \n". In the first of these lines, the output would have been

```
Square of sum   :12100Square of difference: 8100
```

In the actual output, shown in the following 'demonstration' of the program, you may notice that the numbers 12100 and 8100 are not properly aligned: since their first digits are in the same position. However, their final ones are not.

```
Enter two integers   : 100 10
```

```
Square of sum        : 12100
Square of difference: 8100
```

## 2.3 Definitions and Basic Illustrations

### 2.3.1 Objects and Instantiation

An object is an item declared to be of class type. For example, the following statements create two objects using the Gunk:

```
Class Gunk
    {
    //
    }:
Gunk gl, g2;
Int  i;
```

The variables gl and g2 are objects. However, i is not an object. Although the predefined data types have characteristics of classes, they do not support important object features. For example, you cannot derive a new class from the types float or long int. Generally, the term object refers to variables and constants that have class type. An object is also known as an instance of a class. C++ is an object-oriented programming language because it shifts the focus of the programmer from the functions a program performs to the objects the program creates.

The process of creating an object is called instantiation. A pointer or reference to an object provides indirect ways to access an object, but the pointer or reference does not cause an object to be created.

The instantiation of values with global and file scope occurs before the first statement of main is executed. Local values are instantiated upon their definition.

### 2.3.2 Data Members

A data declaration inside of a class, struct, or union is a data member of that structure. Data members cannot be declared as auto, register, or extern; they can be enums, bit fields, and other intrinsic or user-defined types. Data members can be objects in their own right. Moreover, only objects of previously declared or defined classes can be members. A class may not define data members which would be instances of itself; however, a class can contain references and pointers to instances of itself.

Each object has a unique set of instance variables that correspond in name and type to the data members defined by its class. A variable associated with an object is called an instance variable. The same syntax is used both to access an object's instance variables and to access the data members of a struct in C. Objects are self contained; the instance variables of one object have no effect on the instance variables of other objects.

### 2.3.3 Function Members

A function declared within the definition of a class is called a member function. The definition of a class contains prototypes for its member functions. Prototypes for member functions follow the same syntax as the prototypes used to declare nonmember functions. Generally, a programmer controls access to the internal data structure and operations of a class through the member functions.

Member function definitions have a slightly different header format than non-method functions. The name of the class with which a method is associated is added as a prefix to a method function's name. The scoping operator,::, separates the class name from the function name. Several classes may have functions of the same name, and this syntax indicates the class associated with the method-function definition.

The following program shows how this syntax is used to distinguish between methods of the same name that declared in different classes.

```
        #include <stio.h>
        class Cl
            {
            private:
                int i:
         Public:
             Void set(int x):
             Void print( ):
        };
    class c2
        {
        private
            int j;
        public:
            void set (int x);
            void print ( );
    };
void Cl: : set (int x)
    }
    j = x;
    }
void Cl : : print ( )
    {
    print f ( " % d " , j );
    }
     void C2 : : set ( int x)
    {
    j = x;
    }

void C2 : : print ( )
    }
    print f (" % x " , j );
    }

 int main ( )
    {
    Cl a:
    C2 b:
    a. set ( 100) :             // calls Cl : : set
    b. set ( 100 );             // calls C2 : : set
    a. print ( );               // calls Cl : : print
    b. print ( );               // calls C2 : : print
```

The classes C1 and C2 both define the methods set and print. Implementations of these methods include the name of the associated class in the header to identify the class that includes them.

Nonoperator methods are called using the same syntax you would use to access a data member of class. Member functions are called for a specific object by giving the name of the object, the member access operator (the period), and the name of the member function.

### 2.3.4 The Implicit Object

A call to a member function is associated with a specific object using the member access operator. The object for which a member function a member function is invoked is known as the implicit object. A pointer to the implicit object is passed as a "hidden" first argument in the call to a member function; it's automatically defined for any non-static member function.

For example, this class

```
Class line
    {
     public;
        Line (int len = 0);
        Line operator = (int len)
    Private:
        Int Length;
    }:
```

The members of the implicit object can be accessed in two ways. For example, Line : : operator = could be operated like this:

```
Line Line : : operator = ( int len )
    {
    Length = len;
    }
```

The assignments to length refer to the instance variables of names of those names that are part of the implicit object. No qualification is needed; the compiler assumes that unqualified instance-variable references are associated with the implicit object. This is known as inferred member access, because the programmer does not specify the object with which the instance variables are associated. The operator member function could be changed to reference the instance variables of the implicit objectusing the pointer. In that case, the definition would look something like this:

```
Line Line : : operator =  (int len )
    {
        Length = len;
        }
```

There is no practical difference between inferred and direct member access. In some cases, direct member access can be the clearer of the two, especially when there are many objects at work within a member function.

### 2.3.5 Class Scope

The term scope refers to the area in a program where a given identifier is accessible. The three most common types of scope in C are global, file, and local. Identifiers defined within a function or {} block are only accessible within that function or block and are said to have local scope. Identifiers declared outside of a ration to give them file scope. An identifier with global scope can be accessed from any where within a program; an identifier with file scope is visible only within the file in which it is declared.

The purpose of scope is to control access to identifiers. To control how members of classes are accessed, C++ has introduced the concept of class scope. All can reference any other member of the same class. This is part of encapsulation,

The member functions of a class have unrestricted access to the data members of that same class. Access to data and function members of a class outside of the class scope is controlled by the programmer. The idea is to encapsulate the data structure and functionality of a class so that access to the class's data structure from outside of the class's member functions is limited or unnecessary.

**Access Specifiers**

As discussed above, a class defines a group of members that can reference each other**.** Access specifiers are used to control the visibility of class members outside of the class scope. The access specifiers public and private are keywords that designate different kinds of external access to the members of a class. The public section lists those members that can be referenced from outside of the class's scope; the private section lists those members that can only be referenced inside the class scope. For example, the main function can call the four functions listed in the public section of the rational class definition; however, main cannot reference the two data members or the Reduce function that are listed in the private section. Using public and private, a programmer can control the visibility of an object's members.

A class may have multiple private and public sections. Each access specifier is in effect until the next access specifier (or the end of the class) is encountered. For example:

```
    Class government
```

```
{
Private:
      Float graft:          // graft is private
Public:
      Float taxes:          // taxes are public
Private:
    Float slushfund:    // slushfund is private
}:
```

Access specifiers are not required. Unless an access specifier indicates otherwise, the members of a struct are public, the same holds true for C++. The a and b of the members of this struct default to public access, while the c and d members are private:

```
Struct xyz
      {
      // these members default to public access
      int a. b:
       Private:
         Int c. d:
      }:
```

The only semantic difference between a struct and a class members default to private access. So. Changing the struct to class in the example above would change the access. Of the a and b members to private.

If class and struct are nearly identical. Why would anyone use class instead of struct? The choice is based on fitting the keyword to use. When encapsulating data and function. We are creating a class in object-oriented programming terms. Therefore, it seems logical to use the class syntax to accurately describe what our program is doing. When working strictly with data structures that don't have member functions, it's better to use the struct keyword.

In designing my classes. I've made all the data members private and all but one of the functional members public. By making the data members private, I have restricted access to those members to the member functions defined for the class. As a general rule, data members are always private. The member functions are then made public. So that they provide the interface to work with rational objects.

### 2.3.6 Constructors

When an object is created, space is allocated for it in memory. In many classes, the programmer wants to have values automatically assigned to an object's data members when that object is created. Furthermore, an object may have to perform other. More complex operations when it is instantiated.

A constructor is a special member function that literally builds objects. When an object is instantiated. A constructor is called to allocate space, assign values to data members, and perform other housekeeping tasks for a new object. Nearly every class you create will have one or more constructors. To choose which constructor to call, the compiler compares the arguments used in an object's declaration to the constructors' parameter lists. This process is identical to that used to chose between other overloaded functions.

A constructor is a member function with the same name as its class. It may have parameters like any other function, but it cannot have a return value. This restriction is imposed because constructors are usually called when defining a new object when there's no syntax for retrieving or examining a return value generated by a constructor.

Here's an example of a class that has constructors:

```
Class integer
    {
    Private:
        Int value:
    Public:
        Integer (int x):
        // other methods...
    }:
Integer::integer(int x)
    {
    Value = x:
    }
```

The integer class defines a single constructor that has an integer parameter. When an integer object is declared, it must have an argument following its name in order to provide parameters for the constructor. For example:

```
Integer il (10):  // correct declaration
Integer i2:        // error! Needs an argument
```

It's possible to make the declaration of i2 legal by defining a default argument for the constructor's x parameter, or by defining a default constructor (see below) with no parameters. In some cases, this eliminates the need to create multiple constructors.

Constructors for global and static objects are called before the main function is executed. Automatic objects are constructed when their declarations appear. For example:

```
Void function ( )
    {
    static integer i1 (8):
    integer i2 (2):
    // other statements
    }
```

The program constructs i/ before main is called. I2 is constructed each time function is called.

### 2.3.6 Destructors

A destructor is a member function with the same name as the class, and a leading tilde (~). A class has only one destructor function, which has no arguments and no return type. A destructor performs the opposite function of a constructor, cleaning up after an object is no longer needed (such as freeing up dynamic memory allocated by the object). Some objects may need to do some final housekeeping. For example, a display window object would probably erase itself from the screen when it is destroyed.

Here's an example of a class with a destructor:

```
Class chunk
    {
     private:
        void * p:
      public:
        chunk (unsigned int alloc):
        ~chunk ( ) : // destructor
        }
chunk::chunk (unsigned int alloc)
```

```
        {
        p = new char [alloc]:
        }

chunk::~chunk ( )
        {
         delete p:
        }
```

The destructor is called whenever an object is destroyed. Global, file scope, and static objects are destroyed at the end of a program; automatic objects are destroyed at the end of their scope. For example:

```
Chunk chg  (100):

Int main ( )
     {
      static integer is (10);
      integer ia (23);
      int  x = o;
       // some code
       if (/* some condition */)
          }
          chunk cha (50):
          // some more code
          return o:
          }
```

Before *main* is called, *chg* and *is* are constructed. *ia* is constructed when main is called. Cha is constructed only when the if statement is true; it is destroyed at the end of the block in which it is defined. When main terminates, ia is, and chg are destroyed. A call to the exit function also destroys any global or static objects; depending upon the compiler, a call to exit may or may not destroy existing automatic objects.

If a destructor is not defined for a class, the compiler generates a default destructor that does nothing. For many classes, such as rational, a do-nothing default destructor is all that is required.

### 2.3.7 Inline Member Functions

Like any other C++ function, a member function can be an inline function. There are two ways to make a member function inline: by applying the inline keyword to the function definition, or by defining it within the class definition. For example:

```
Class whatsis
   {
    private:
       int i:

    public:
       void set (int x)
            }
            i  = x:
            {

       int get ( ):
   }:

inline int whatsis::get ( )
   {
   return i :
   }
```

Both member function defines the function inside of the class definition, making it inline. The definition of get is qualified by the inline keyword.

### 2.3.8 Conversions

Intrinsic types have a predefined set of conversions. You can assign an int value to a long variable, or add a long value to a float. Conversions are either implicit or explicit. An implicit conversion is made by the compiler, such as when an int value is assigned to a long variable. Explicit conversions occur when a cast is used to force a specific conversion Explicit conversions are often used in function calls to pass arguments that have different types from the corresponding parameters.

The class types one define do not acquire conversions to other types; that is something that must be done personally. C++ provides ways to define both implicit and explicit conversions. An implicit conversion is defined by a conversion constructor, and an explicit conversion is defined by a conversion operator or cast operator.

**Conversion Constructors**

An implicit conversion is defined by providing a conversion constructor for a class. The conversion constructor changes an argument of the type being converted into an object of that class. For example, the following conversion constructor would convert an int value into a rational value:

```
Rational :: rational (int i)
    {
    Numerator = i :
    Denominator = 1:
    }
```

This is a one-way conversion that takes a value or object of one type and coverts it to an object of the class. Conversion constructors cannot be used to convert class objects to other types, and they can be used only in assignments and initializations.

**Conversion Operators**

However, conversion operators can be used to convert objects to other types, and they can also be used for purposes other than assignments and initializations. A conversion operator cannot have parameters or a return type. Its name is given in the following format:

```
Operator type ( ):
```

Type represents the name of the type to which an object will be converted. A conversion operator that converts a rational number to a floating-point value would be defined like this:

```
Rational :: operator float ( )
    {
     float result:
     result = float (Numerator) / float (Denominator):
      return result:
            }:
```

### 2.3.9 Member Objects

A class that has member objects is called an enclosing class. Member objects need to be constructed when an object of the class enclosing them is constructed. This is done by specifying a member-initialization list for object members in the definition of the constructor for the enclosing class. This example should clarify things:

```
Class foo
    {
    private:
        int i:
    public:
        foo ( ) {i = o: }
    }:
class bar
    {
    private:
        int  i:
    public:
        bar (int x) { i = x:}
    }:

class snafu
    {
    private:
        foo f:
        bar bl:
            bar b2:
    public:
        // constructor creating member objects
        snafu ( ) : bl (1), b2(2) {}
    }:
```

A colon separates the member-initialization list from the function header in the definition of snafu's constructor. The member-initialization list comes after the colon and before the actual function statements. Each object that requires initialization has its name listed along with a list of arguments for the object's constructor. If multiple member objects exist, their initialization can be listed in any order, separated by comas (as shown above).

When an object of class snafu is instantiated, constructors are called for its three member objects. Because the constructor class foo does not have arguments, there is no need for f to be in the member-initialization list. B1 andb2 both require an argument for their constructor. So they are listed. C++ does no guarantee the order in which member objects are constructed.

It's important to remember that an enclosing class must have a constructor if there are any object members that require constructor arguments. Although snafu's constructor contains no statember-initializations list can be provided. This means that a class such as snafu cannot rely upon a default constructor.

### 2.3.10 Static Members

A member of a class can be declared static. For a data member, the static designation means that there is only one instance of that member. A static data member is shared by all objects of that class and exists even if no objects of that class exist. For instance:

```
#include <stdio.h>

class pumplin
{
private:
    int weight:
     static int total-weight:
     static int total_number:

public:
      pumplin (int w)
         }
         weight = w:
         total_ weight += w:
                total_number ++:
                }

            ~pumpkin ( )
                {
                total_weight .= weight:
                total_ number..:
                }
        }:

//initialization of static members
int pumpkin::total_weight = o:
int pumplin::total_number = o:

int main ( )
        {
        pumpkin pl(15). P2 (20).  P3(12):

        p1. Display ( ):
        p2. Display ( ):
        p3. Display ( ):
        }
```

When a pumpkin object is created, its weight is added to total_weight, which is a static member of the pumpkin class. Another static member, total_number, is incremented as a count of the number of pumpkin objects in existence. Both static members of pumpkin are initialized outside of the class definition; C++ does not considers initialization to be in violation of their private status. Note that the name of the class and the are initialized.

When called, a member function declared with the static keyword is not associated with a specific object. Because an object is not required, a static member function does not have a pointer. The static member functions for a class can be called whether or not an object of that class has been instantiated. Static member functions are used to act globally on all objects of a class. To display the two static data members of the pumpkin class, the following function could be added to the class definition:

```
Static void pumpkin::total_display( )
    {
    printf ("%d pumpkins weigh %d pounds{"\n",
    }
```

**total_display** could be called with either of the following statements:

```
pumpkin::total_display ( ):
pl. total_display ( ):
```

In the second statement, total_display is called as if were a regular method. The first statement is considered the better form; remembering that static methods are not associated with a specific object, so using the first form is a more direct approach than using the second form.

### 2.3.11 Inheritance and Polymorphism

Two concepts are important to realizing the full power of object-oriented programming: inheritance and polymorphism. Inheritance allows classes to build upon existing classes. Polymorphism treats objects of related classes in a generic manner. We have already covered stand-alone classes, which provide for programmer designed data abstraction. Now let's look at how C++ implements inheritance and polymorphism.

**Inheritance**

First, a quick terminology refresher is in order. A class that inherits from another class is called a derived class. The class from which it inherits is known as a base class. Any class my be a base class; what's more, a class may be a base class for another class. It is through this mechanism that class hierarchies are built.

A derived class lists the name of its base class in its definition. It looks something like this:

```
Enum BugColor (Red, Green, Blue, Yellow, Black):

Class Bug
    {
    private:
        int legs:
            BugColor color:
        Public:
            Buh(int numLegs, BugColor c):


            Void Draw ( ):
        }:
    class HumBug : public Bug
        {
        private:
            int Frequency:
        public:
            HumBug( int numLegs. BugColor c. int Freq):


            Void Hum ( ):
        }:
```

The inclusion of public Bug in the definition of HumBug says that HumBug is derived from Bug. The public keyword indicates that all public members of Bug are also public members of HumBug.

Any HumBug objects will have the three data members: Legs, Color, and Frequency. While HumBug defines a method of its own, it also inherits the methods defined for Bug. Therefore the following function is valid:

```
Void func ( )
    {
    Bug b(6.Blue):
    Humbug h(10. Green. 1000):

    b.Draw ( ):
    h.Draw ( ):
    h.Hum ( ):
    }
```

# CHAPTER 3
# NON-LINEAR ALGEBRAIC EQUATIONS

## 3.1 INTRODUCTORY REMARKS

In this chapter we are concerned with the problems of finding one or more of the roots of the algebraic equation (while applying $C^{++)}$;

$$f(x) = 0 \qquad (3.1)$$

where f(x) is some given real-valued function. We shall assume that f(x) possesses all the necessary analytical properties (particularly with regard to continuity) for the methods we develop and analyse to be mathematically valid. We recall that a root of (3.1) is a number $\alpha$ such that $f(\alpha)=0$; in practice we shall aim to calculate an estimate of $\alpha$ correct to some prescribed precision.

**EXAMPLE 3.1**

(i) $f(x) \equiv e^{-x} - \sin(x) = 0$

(ii) $f(x) \equiv x^3 - 3x + 1 = 0$

(i) The problem of finding the roots of the equation is made easier when we express the equation as corresponding to the intersections of the graphs $y = e^{-x}$ and $y = \sin(x)$. The functions can thus be sketched without difficulty below

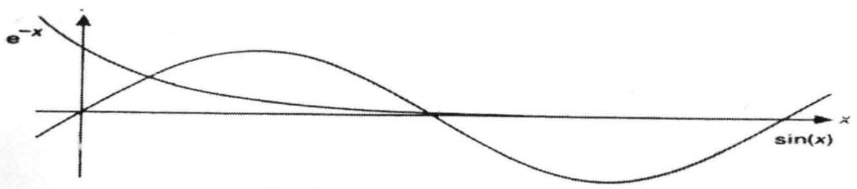


fig 3.1

The following deductions can be made:

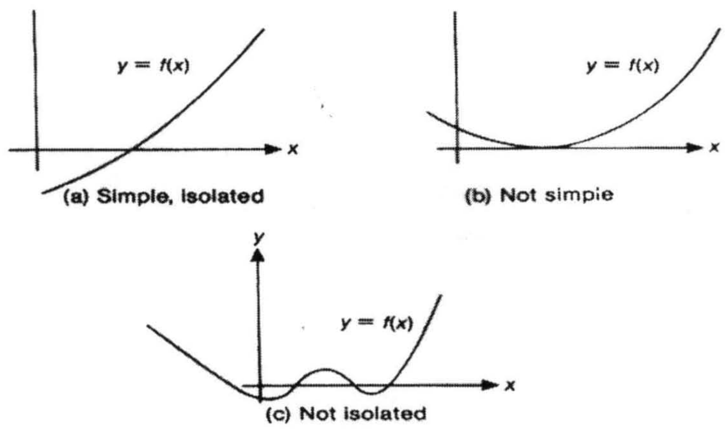
a) There are no negative roots of the equation since for $x<0$, $e^{-x}>1$ and $|\sin(x)| \leq 1$ and there can be no intersections.

b) There are infinite number of positive roots since for $x > 0$, $0 < e^{-x} < 1$ and $\sin(x)$ oscillates indefinitely between -1 and 1.

c) Since $e^{-x}$ decreases very rapidly ($e^{-\pi} \simeq 0.043$, $e^{-2\pi} \simeq 0.0019$) the root (apart from the first) get closer and closer to the zeros of $\sin(x)$; that is $x = n\pi$: $n=1,2,\ldots$ .Moreover, those for odd n are just less than $n\pi$ and those for even n just greater than $n\pi$.

(ii) The second equation is a (cubic) polynomial equation since $f(x)$ is a polynomial of degree three. It therefore follows that three roots are tenable albeit with diverse behaviours but satisfying $f(\alpha) = 0$ with $f'(\alpha) \neq 0$

The diagram below illustrate that while there exist an 'isolated' root, the others are separated in some sense.



Most of the methods of solution are iterative in nature. The idea of iteration was introduced in section 1.3 and, as presented there, it depends on the availability of an initial estimate of the value of the root. Some methods, however, will require the provision of two, or more, starting values. Using the initial information, we generate a sequence of values which hopefully will converge to the root we seek.

Before implementing an iterative method, the following four items need to be investigated.

(i) We must establish the conditions under which the sequence of iterates generated by the method will converge to the root.

(ii) If the iteration is convergent we need to know how and when to stop the iterative process. This, of course, depends on the accuracy we are demanding and we have to formulate a *termination criterion* which can be easily implemented.

(iii) There may be several iterative schemes which are equally acceptable in that they are all expected to converge to the root in question, starting from the same initial approximation. We need to determine whether one method is preferable to another and are led to the notion of trying to formulate some measure of ***computational efficiency.***

(iv) If the chosen method is to be programmed, *good programming practice* must be observed to ensure that the storage and time requirements are not excessive. In addition, we must take account of as many eventualities as possible when designing a piece of C++ code to ensure that it does not halt due to a run-time failure.

## 3.2 FUNCTIONAL ITERATION

### 3.2.1 Introduction

The starting point here is to rewrite (3.1) in the form

$$x = \varphi(x) \qquad (3.2)$$

so that if $\alpha$ is a root of (3.1) then $\alpha = \varphi(\alpha)$. There will usually be several obvious ways in which the given equation can be rewritten in the required form and, indeed, if we are prepared to modify slightly the problem, there will be an infinity of ways.

**EXAMPLE 3.2**

(i) $x^3-3x+1=0$ can be rewritten as $x=(x^3+1)/3$ or as $x=1/(3-x^2)$.

(ii) $x+\ln(x)=0$ can be rewritten as $x=-\ln(x)$ or as $x=e^{-x}$.

(iii) $x^3-3x+1=0$ can be rewritten as

$x=(1-k)x+k(x^3+1)/3$, where $k$ ($\neq 0$) is some chosen value.

Suppose we have an initial approximation, $x_o$, to the root $\alpha$ whose value we wish to determine. We can generate a sequence of iterates $\{x_n\}$ using the iterative process

$$x_n= \varphi(x_{n-1}) \quad n=1,2, ...$$

but before using it, we must try to establish the conditions under which the iterates converge to $\alpha$. Some insight into the question of convergence can be gained by graphical arguments (see Fig. 3.2). Using the same axes, we draw the graphs of $y=x$

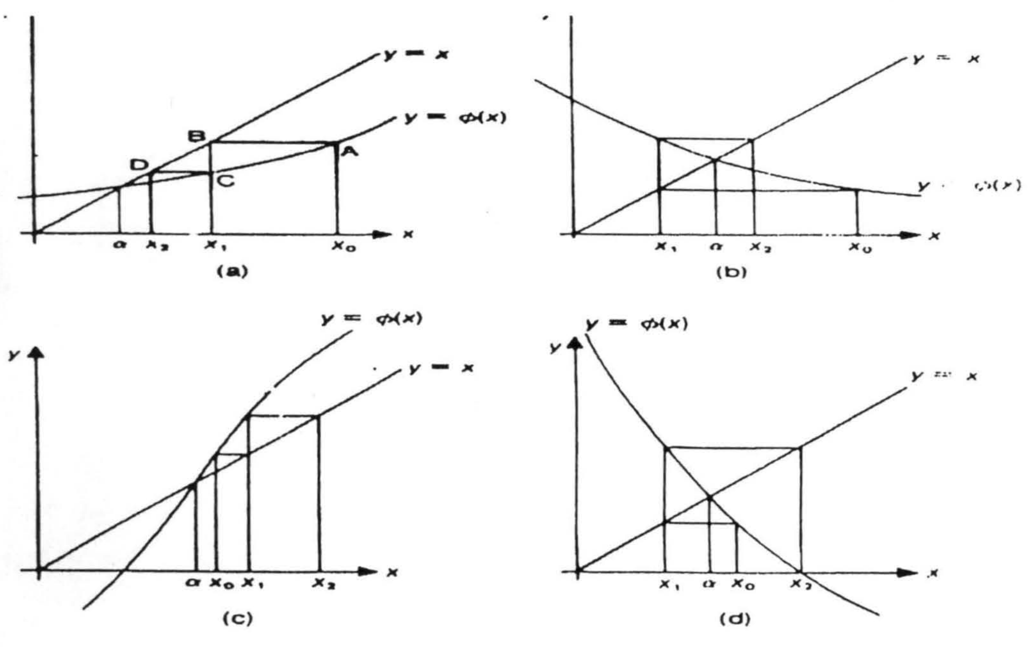


Fig. 3.2

and $y = \varphi(x)$. The points of intersection of these two curves give the roots of (3.1). We describe the progress of the iteration for Fig. 3.2(a) and by the same construction the other cases can be verified. Let $x_o$ be the chosen starting value. Then A is the point on the curve $y= \varphi(x)$ with coordinates $(x_0, \varphi(x_0))$ and B is the point on the straight line $y=x$ obtained by drawing a line through A parallel to the x-axis. The y coordinate of B is therefore $\phi(x_o)$ and, since B lies on $y=x$, the x coordinate of B is also $\phi(x_o)$. However, $x_1= \phi(x_o)$ and so we are able to mark the position of $x_1$ by drawing a line through B parallel to the y-axis. We now repeat the argument. C is the point on $y=\phi(x)$ with coordinates $(x_1, \phi(x_1))$ and D is the point on $y=x$ with coordinates $(\phi(x_1),\phi(x_1))$. Since $x_2 =\phi(x_1)$ we are able to mark the position of $x_2$ and the argument can be repeated once more. The diagram displayed suggests the iteration is converging and the same is true for case (b) although we note that the iterates are oscillating about the root. In (c) and (d) the iteration is diverging, that is, successive iterates are getting further away from the root. The key thing to note here is that, near the root, $|\phi'(x)|<1$ in (a) and (b) but in (c) and (d) $|\phi'(x)|>1$.

## 3.2.2 A CONVERGENCE THEOREM

**THEOREM 3.1**

Suppose the equation $x = \varphi(x)$ has a root $\alpha$ and that, on the interval $I$ defined by $\alpha$-a $\le x \le \alpha$+a, $\varphi'(x)$ satisfies the condition

$$|\varphi'(x)| \le L < 1.$$

Then, for any $x_0 \in I$

I) $x_n \in I \qquad n = 1,2,\ldots;$

Ii) $\lim_{n \to \infty} x_n = \alpha$; and

iii) $\alpha$ is the unique root of $x = \varphi(x)$ in $I$

Proof

i) Suppose $x_{n-1} \in I$.

Now $x_n = \varphi(x_{n-1})$ and $\alpha = \varphi(\alpha)$ so

$$x_n - \alpha = \varphi(x_{n-1}) - \varphi(\alpha)$$
$$= (x_{n-1} - \alpha)\varphi'(\xi_{n-1}) \qquad (3.3)$$

clearly $\xi_{n-1}$ lies between $\alpha$ and $x_{n-1}$.

Since $x_{n-1} \in I$ it follows that $\xi_{n-1} \in I$ and hence that

$$|x_n - \alpha| = |x_{n-1} - \alpha|\,|\varphi'(\xi_{n-1})| \leq L|x_{n-1} - \alpha|. \qquad (3.4)$$

Since $L < 1$, it follows that $x_n \in I$. By hypothesis, $x_0 \in I$ and so part(i) of the theorem is proved.

ii Here, we use the result (3.4) repeatedly. We have

$$|x_n - \alpha| \leq L|x_{n-1} - \alpha|$$
$$\leq L^2 |x_{n-2} - \alpha|$$
$$\leq \ldots \leq L^n |x_0 - \alpha|.$$

Since $L < 1$, $\lim_{n \to \infty} L^n = 0$ which implies that $\lim_{n \to \infty} |x_n - \alpha| = 0$ and hence that $\lim_{n \to \infty} x_n = \alpha$.

iii assume to the contrary and suppose the equation $x = \varphi(x)$ has another root $\beta(\neq \alpha)$ lying in $I$. Then

$$\alpha - \beta = \varphi(\alpha) - \varphi(\beta)$$
$$= (\alpha - \beta)\varphi'(\eta)$$

Clearly $\eta$ lies between $\alpha$ and $\beta$ and hence lies in $I$. We know that $|\varphi'(\eta)| < 1$ from the conditions of the theorem and so

$$|\alpha - \beta| = |\alpha - \beta|\,|\varphi'(\eta)| < |\alpha - \beta|$$

which gives a contradiction.

### 3.2.3 A Program for functional iteration

```
// a program to find the root of the equation x-exp(-x) = 0 using functional iteration
#include<stdio.h>
#include<math.h>
main()
{
int converged;
float xn,xnminus1,tol;
int iter,itermax;
scanf("%f %f %d ",&xn,&tol,&itermax);
printf("Starting value is  %f\n",xn);
printf("Accuracy Tolerance is  %f\n",tol);
printf("Maximum number of iterations allowed is  %d\n",itermax);
iter = 0;
do
{
iter = iter + 1;
xnminus1 = xn;
xn = exp(-xnminus1);
converged= abs(xn-xnminus1) < tol;
} while(converged || (iter == itermax));
        if( converged == 1 ) printf("Converged after: %d   iteration to: %f\n",iter,xn);else
printf("no convergence after:  %d\n",itermax);
}
```

**Sample data**

```
0.5      0.00005          20
```

sample output

```
starting value                              5.00000e -1
accuracy tolerance                          5.00000e -5
max number of iterations allowed               20
convergence to specified tolerance after 15 iterations to 5.67157e -1
```

## 3.3 NEWTON-RAPHSON ITERATION

### 3.3.1 Derivation

Suppose we have an approximation $x_{n-1}$ To a simple root $x = \alpha$ of the equation $f(x)=0$.

Let $\alpha = x_{n-1} + h$. Then, since $f(\alpha) = 0$ we have

$$0 = f(x_{n-1} + h) = f(x_{n-1}) + hf'(x_{n-1}) + \ldots$$

using Taylor's Theorem. Ignoring the rest of the terms we see that an approximation to h can be obtained as $-f(x_{n-1})/f'(x_{n-1})$ and so we take the next approximation to the root as

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

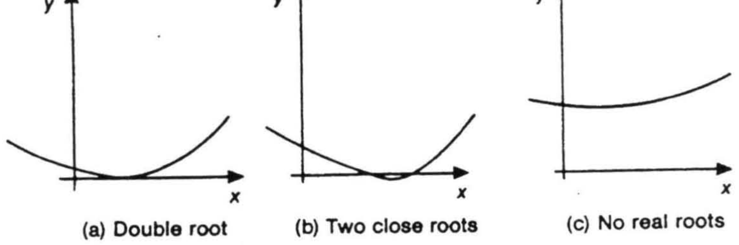
Using this for n = 1,2,... we have a process known as Netwon-Raphson iteration. We notice that each iteration involves the valuation of f(x) and its first derivative at the previous iterate and, since we are now using more information about the function than before, we might hope to obtain some benefits

### 3.3.2 Termination Criteria

In implementing Newton-Raphson iteration there are a number of important devices which should be included in a robust program. The termination criterion will usually monitor the absolute difference between the successive iterates, $| x_n - x_{n-1} |$, and also perhaps $|f( x_n)|$, continuing until either or both of these are sufficiently small to ensure that a satisfactory estimate of the root is found. If the magnitude of the numbers is not known in advance it is best to use a relative criterion in which $| x_n - x_{n-1} |/| x_n|$ and $|f( x_n)/F|$ are monitored, where $F$ is some estimate of the value of f(x) near the root which can be obtained during the iteration or preset before the calculation begins.
As usual, in implementing an iterative process the number of iterations should be counted and the process stopped if this number exceeds some pre-assigned value.

It is necessary here to observe that a computer implementation of Newton-Raphson iteration will result in the values $f(x_{n-1})$ and $f'(x_{n-1})$ being obtained to limited precision only. The values used in the termination criteria must take account of this restriction so that an accuracy which is impossible to satisfy is not specified.

In Newton-Raphson iteration it is usual also to monitor the behaviour of $|f'( x_n)|$. If the value of $f'( x_{n-1})$ becomes small in some sense, the program user should be made aware of this fact through an appropriate message although the iteration may be allowed to continue. A small value of $f'( x_{n-1})$ may be symptomatic of the presence of a double root, or a root of even higher multiplicity, or of the existence of two or more roots which are close together, or of the absence of a root. These situations are illustrated in Fig. 3.6



### 3.3.4 A program for Newton-Raphson iteration

```
// a program to find a root of the equation x-exp(-x)=0 using newton-raphson iteration
#include<iostream.h>
#include<math.h>
float f(float x)
{
float f= x-exp(-x);
 return f;
 }
 float fdash(float x)
 {
 float fdash = 1.0 + exp(-x);
 return fdash;
 }
 main()
        {int monitoriterations,converged,smallfdash;
         float xn,xnminus1,festimate,fxnminus1,fdashxnminus1,tol,twotol,changeiniterates;
         char charater;
         int iter,itermax;
         cout <<"Enter starting value,accuracy tolerance,maxnumber of iteration allowed,est.value of
function"<<endl;
         cin >> xn,tol,itermax,festimate;
         do
          { cin.get(charater);
          } while(charater!= '\n');
                 monitoriterations == 1;
                 cout<< "starting value = "<<xn<<endl;
                 cout<< "accuracy tolerance = " << tol<< endl;
                 cout<< "maximum number of itration allowed = "<< itermax << endl;
                 cout<< " estimeted value of the function at the root=" << festimate << endl;
                 smallfdash == 0 ;
                 twotol = tol+ tol;
                 if (monitoriterations) cout <<"     xn              f(xnminus) "<<endl;
                 iter =0;
        do
                { iter = iter + 1;
                xnminus1=xn;
                fxnminus1=f(xnminus1);
                fdashxnminus1= fdash(xnminus1);
                xn = xnminus1= fxnminus1/fdashxnminus1;
                if (monitoriterations) cout <<xn<<"    "<< fxnminus1<< endl;
                if (abs(xn)<tol) changeiniterates=abs(xn-xnminus1);else
                changeiniterates=abs((xn-xnminus1)/xn);
                converged = changeiniterates + abs(fxnminus1/festimate)<twotol;
                if (smallfdash!=0) smallfdash = abs(fdashxnminus1)<tol;
        } while (converged || (iter ==itermax));

        if(smallfdash) cout<<"Warning! the derivative of the function became less than "<<tol<<
    "at least one iteration point "<< endl;
        if (converged) cout <<"convergence to specified iteration after :"
                << iter<<"iterations"<<endl;else
    cout << "no convergence to the specified tolerance after "<<itermax<< "iterations "<<endl;
}
```

**Sample data**

```
0.5     0.00005         20      1.0     y
```

•

**sample output**

```
    starting value                              5.00000e -1
    accuracy tolerance                          5.00000e -5
    max number of iterations allowed               20
    estimated value of the function at the root         1.00000e    0
            xn                      f(xminus1)
    5.66311e  -1                    -1.06531e  -1
    5.67143e  -1                    -1.30466e  -3
    5.67143e  -1                    -2.55182e  -7
    convergence to specified tolerance after            3 iterations to 5.67143e -1
```

## 3.4 POLYNOMIAL EQUATIONS

### 3.4.1 Introduction

Here we are concerned with estimating some or all of the roots of the equation $f(x) = 0$ where $f(x)$ is a polynomial of degree $m$ in $x$, that is,

$$f(x) = a_m x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0 = \sum_{i=0}^{m} a_i x^i. \tag{3.5}$$

We assume that the coefficients $a_0, a_1, \ldots, a_m$ are all real. The polynomial equation has m roots, some or all of which may be complex. We recall that if complex roots are present they occur in conjugate pairs, that is, if $\alpha + i\beta$ is a root then so is $\alpha - i\beta$ .

Any of the methods which have been considered in this chapter may, in principle, be used to estimate the real roots of a polynomial equation and some of them may be adapted to deal with complex roots. Complex roots, however, are usually best determined using special techniques not discussed here.

The subject of root-finding in the case of polynomial equations is very large and attempt will not be made to give an exhaustive survey. Rather we concentrate on a number of basic ideas concerning the evaluation of a polynomial and its first derivative, the division of a polynomial by a polynomial of degree 2 and the extremely important notion of the conditioning of a polynomial equation.

### 3.4.2 Evaluation of a Polynomial and its Derivative

Suppose we wish to evaluate the polynomials f(x) and f '(x) for some real number  . (This would be required, for example, if we were employing Newton-Raphson iteration to find the root of f(x) = 0). The starting point is to use the nested form of a polynomial which, for (3.5) is

$$((\ldots((a_m x + a_{m-1})x + a_{m-2})x + \ldots)x + a_1)x + a_0. \tag{3.6}$$

Using (3.6) the evaluation of f(x) at the point x= can be described by the recursive algorithm

$$
\begin{aligned}
& b_{m-1} = a_m \\
& b_i = a_{i+1} + \lambda b_{i+1} \quad i = m-2, m-3, \ldots, 0 \qquad (3.7) \\
& f(\lambda) = a_0 + \lambda b_0
\end{aligned}
$$

which defines a sequence of intermediate values $b_{m-1}, b_{m-2}, \ldots, b_0$.

**EXAMPLE 3.3**

If $m = 3$ we have

$a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)x + a_1)x + a_0$

and hence, from (3.7)

$$
\begin{aligned}
& b_2 = a_3 \\
& b_1 = a_2 + \lambda b_2 (= a_2 + \lambda a_3) \\
& b_0 = a_1 + \lambda b_1 (= a_1 + \lambda(a_2 + \lambda a_3)) \\
& f(\lambda) = a_0 + \lambda b_0 (= a_0 + \lambda(a_1 + \lambda(a_2 + \lambda a_3)))
\end{aligned}
$$

For the purposes of hand computation, we can set out the coefficients of successive powers of $x$ in a row, supplying a zero entry if any power of $x$ is missing. The $b_i s$ and $f(\lambda)$ are then formed in a straightforward way.

**EXAMPLE 3.4**

To evaluate f(x) = $x^5 - 6x^3 + x^2 + 7x - 4$ at the point x = 2 we form the following table

| | ($X^5$) | ($X^4$) | ($X^3$) | ($X^2$) | ($X^1$) | ($X^0$) |
|---|---|---|---|---|---|---|
| | 1 | 0 | -6 | 1 | 7 | -4 |
| x=2 | 1 | 2 x 1 + 0 =2 | 2 x 2- 6 = -2 | 2 x (-2) +1 = -3 | 2 x (-3)+7 =1 | 2 x 1- 4 = -2 |
| | ($b^4$) | ($b^3$) | ($b^2$) | ($b^1$) | ($b^0$) | f(2) |

and deduce that f(2) = -2.

We now show that the $b_i$ s defined by (3.7) have a significance which is of interest. The following theorem defines a process which is known as *synthetic division.*

**THEOREM 3.2**

$$\frac{f(x)-f(\lambda)}{x-\lambda}=\sum_{i=0}^{m-1} b_i x^i \qquad (3.8)$$

Where the $b_i s$ is are defined by (3.7).

The proof of this theorem is straightforward and depends on multiplying the right-hand side of (3.8) by $x-\lambda$ and showing the result equals $f(x)-f(\lambda)$. Rearranging (3.8) further we have

$$f(x)=(x-\lambda)\sum_{i=0}^{m-1} b_i x^i + f(\lambda) \qquad (3.9)$$

so that if $x=\lambda=\alpha$, a root of the equation $f(x)=0$, the remaining roots may be found as the roots of the *deflated equation*

$$g(x)=\sum_{i=0}^{m-1} b_i x^i = 0 \qquad \textit{(3.10)}$$

This suggests that we can determine the real roots of a polynomial equation one at a time using the appropriate deflated equation after each root is found. The process can be quite unsatisfactory in practice, however. Suppose we have found the first root correct to some precision. Call this value $\bar{\alpha}$ . Since f( $\bar{\alpha}$ ) will not be exactly zero, the $b_i$ s in the deflated equation will not be exact and the root of this equation will also be calculated correct only to the required precision. The cumulative effect of the errors introduced at each deflation can be disastrous and this superficially attractive process can be recommended.

We now return to equation (3.9) and indicate how f '(x) can be evaluated at a point using the $b_i$ s which are generated in finding f( $\lambda$ ). Differentiating (3.9) formally we have

$$f'(x)=(x-\lambda)g'(x)+g(x)$$

where g(x) is defined by (3.10). Thus, f '( $\lambda$ )= g( $\lambda$ ) and we use the nested form of g(x) in order to find f '( $\lambda$ ).

**EXAMPLE 3.5**

To evaluate the derivative of $f(x)=x^5 - 6x^3 + x^2 + 7x - 4$ at the point x = 2 we evaluate the coefficients of the deflated equation as shown in Example 3.4 and form the following table

| | $(x^4)$ | $(x^3)$ | $(x^2)$ | $(x^1)$ | $(x^0)$ |
|---|---|---|---|---|---|
| | 1 | 2 | -2 | -3 | 1 |
| x=2 | 1 | 2 x 1+2 =4 | 2 x 4-2 =6 | 2 x 6-3 =9 | 2 x 9+1 =19 |
| | $(c_3)$ | $(c_2)$ | $(c_1)$ | $(c_0)$ | g(2) |

where the $c_i$s are the coefficients in the next deflated equation. We deduce that f '(2)=g(2)=19 so that if x=2 is a first estimate of a root of the equation f(x)=0 and Newton-Raphson iteration is employed, the next iterate is given by

$$x_2 = 2 - (-2)/19 = 2.105$$

to four significant figures.

### 3.4.3 A program for Polynomial evaluation

```
// a program to evaluate a polynomial and its derivative at a point
#include<iostream.h>
#include<iomanip.h>

void main(void)
 { double  alpha,b,c;
        int m;
        const int MAXN = 100;
        double a[MAXN];
        int i;
        cin >>m;
        for(i=0;i<=m;i++) cin >> a[i];
        cin >> alpha;
        cout << "the polynomial coefficient are (constant term first )" << endl;
        for (i=0; i<=m;i++) cout<<a[i]<<endl;
        cout <<" evaluation point is :" <<alpha<< endl;
        b= a[m];
        c=0.0;
        for(i= m-2;m-2>=i;i--)
        {
                c= (b+(alpha*c));
                b= (a[i+1]+(alpha*b));
        }
        cout <<" value of the polynomial at this point is "<< b<< endl;
        cout <<" value of the polynomial derivative at this point is "<< c << endl;
}
```

**Sample data**

```
5
-4.0    7.0     1.0     -6.0    0.0     1.0
2.0
```

**Sample Output**

```
the polynomial coefficients are ( constant term first )
-4.00000e 0
 7.00000e 0
 1.00000e 0
-6.00000e 0
 0.00000e 0
 1.00000e 0
evaluation point is 2.00000e 0
value of the polynomial at this point is                  -2.00000e 0
value of the polynomial derivative at this point is         1.90000e 1
```

This program avoids storing the coefficients of the first deflated equation explicitely. Each time a new $b_i$

is found it overwrites the value currently held in the real variable *b*, but not before the old value has itself been used to update *c*. The real variable *c* is used to store the coefficients of the second deflated equation in a similar manner.

## 3.4.2 Synthetic Division of a Polynomial by a Quadratic

Let $x^2 + px + q$ be the given quadratic. Then we wish to find the coefficients $c_i$ and remainders *R* and *S* in the identity

$$f(x) \equiv \sum_{i=0}^{m} a_i x^i \equiv \left(x^2 + px + q\right)\sum_{i=0}^{m-2} c_i x^i + Rx + S.$$

On equating coefficients of $x^m, x^{m-1}, \ldots$ in turn we obtain the following results

$$\begin{aligned} c_{m-2} &= a_m \\ c_{m-3} &= a_{m-1} - pc_{m-2} \\ c_i &= a_{i+2} - pc_{i+1} - qc_{i+2} \qquad i = m-4, m-5, \ldots, 0 \\ R &= a_1 - pc_0 - qc_1 \\ S &= a_0 - qc_0. \end{aligned}$$

If we introduce fictitious values $c_m = c_{m-1} = 0$ and let $c_{-1} = R$ , these results can be expressed compactly as

$$\begin{aligned} c_i &= a_{i+2} - pc_{i+1} - qc_{i+2} \\ s &= a_0 - qc_0 \end{aligned} \qquad i = m-2, m-3, \ldots, -1$$

The coefficients c and the remainders *R* and *S* are, of course, functions of *p* and *q*. If we can find values of *p* and *q* such that *R(p,q)* = *S(p,q)* = 0, then $x^2 + px + q$ will be a quadratic factor of $f(x)$ and on solving $x^2 + px + q = 0$ we will have two of the (possibly complex) roots of $f(x)$. The way in which appropriate values of p and q can be found is contained in Bairstow's method. The discussion has to be deferred until we have given an introduction to the problem of solving a pair of non-linear simultaneous equations.

### 3.4.4 Conditioning

Small changes in the initial data could produce relatively large changes in the solution and, in such a case, we say that the problem is 'ill-conditioned'. In the context of the polynomial equation

$$\sum_{i=0}^{m} a_i x^i = 0 \tag{3.11}$$

the initial data are the coefficients $a_0, a_1, \ldots, a_m$ . In general there will be a rounding error in the representation of each of these coefficients in a computer but there may be other uncertainties, for example, if they were the result of some previous computations which are not exact. We suppose that $a_i$ is in error by an amount $\delta a_i$: i = 0,1,...,m. If $\alpha$ is an exact root of (3.11) and $\alpha + \delta\alpha$ is the corresponding root of the perturbed equation

$$\sum_{i=0}^{m} (a_i + \delta a_i) x^i = 0$$

we would like to try to relate $\delta\alpha$ to the perturbations $\delta\alpha_i$ . Reference is made to Ralston and Rabinowitz (1978) for complete details and merely observe here that even small perturbations can have a dramatic effect on the zeros of polynomials of high degree.

**EXAMPLE 3.6**

The polynomial of degree 20

$$f(x)=(x+1)(x+2)...(x+20)$$

has zeros at - 1, - 2,..., - 20. The perturbed polynomial $f(x)+2^{-23}x^{19}$ which corresponds to a perturbation of about $10^{-7}$ in the coefficient of $x^{19}$ has zeros which apart from these corresponding to -1, -2,..., -5 are dramatically different from those of the original polynomial; indeed ten of the zeros become five conjugate complex pairs. This example, due originally to Wolkinson (1959), emphasises that there may be severe difficulties in estimating the roots of a polynomial equation of high degree.

# CHAPTER 4
# LINEAR SIMULTANEOUS EQUATIONS

## 4.1 INTRODUCTION

Usually, small set of linear simultaneous equations are solved by hand in which the unknowns are successfully eliminated until the equation is obtained where only one of the unknowns is involved. This unknown can then be determined and the remaining unknown found successively by a process of backward substitution.

In practice it is often necessary to solve large sets of equations in which tens, hundreds and possibly thousands of unknowns are involved. There are a number of reasons why it is important to analyse methods for the solution of large sets. One is that we must consider the computational effort involved; for example, in elimination methods the number of elementary arithmetic operations increases very rapidly with the number of unknowns and so an efficient algorithm must be used. Further, this very large number of arithmetic operations increases very rapidly with the number of unknowns and so an efficient algorithm must be used. Further, this very large number of arithmetic operations present a potentially serious problem: each operation in general will not be exact and, unless we organise the calculation properly, the cumulative effect of these rounding errors could lead to serious inaccuracies in the results. In other words, we may have a serious problem of numerical instability. In addition, the problem of ill-conditioning may be present. Here the initial data of the problem are the coefficients of the unknowns and the numbers on the right-hand side of each equation. These will often be subject to uncertainty and small changes in them may well cause relatively large changes in the computed solution. Similar effects can be produced by rounding-off errors introduced during the computation. It is therefore necessary to ask whether the reliability of a computed solution can be assessed in some way.

There are two different classes of method for the solution of linear simultaneous equations. Methods such as elimination followed by back-substitution are referred to as direct methods, the solution being obtained within the limits imposed by rounding-off errors in a predetermined number of stages. For a direct method it is, therefore, possible to state beforehand the number of elementary arithmetic operations required to obtain a solution as a function of n, the number of equations (or, equivalently, the number of unknowns). The other methods that we consider are iterative in nature. For such methods it is, of course, necessary to try to establish the conditions under which they will converge. A sequence of successive approximations to the solution will be produced from a given set of initial values. Hence it will not be possible to predict the number of arithmetic operations that will be involved; this will depend on the accuracy being demanded and on how good the initial approximation is.

## 4.2 NOTATION

A set of n linear simultaneous equations in the n unknowns $x_1, x_2, \ldots, x_n$ can be written in the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \cdot \quad \cdot \quad \quad \cdot \quad \cdot & \\ \cdot \quad \cdot \quad \quad \cdot \quad \cdot & \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{4.1}$$

so that $a_{ij}$ is the coefficient of x in the *i*th equation. The system (4.1) may be expressed more compactly as

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \qquad i=1,2\ldots,n \tag{4.2}$$

(so that if i=1, (4.2) gives the first equation in (4.1), and so on) or, making use of matrix notation, as

$$Ax=b$$

where A is an n x n square matrix containing the coefficients on the left-hand side of (4.1) and is defined by

$$A = \begin{bmatrix} a_{11}a_{12}\ldots a_{1n} \\ a_{21}a_{22}\ldots a_{2n} \\ \ldots\ldots\ldots \\ a_{n1}a_{n2}\ldots a_{nn} \end{bmatrix}$$

The right hand side vector

$b^T = (b_1, b_2, \ldots, b_n)$

Consists of the right-hand sides of the system (4.1) and

$x^T = (x_1, x_2, \ldots., x_n)$

is the vector of unknowns.

It is useful at this stage to identify a number of special matrices. The matrix A is said to be symmetric if $a_{ij} = a_{ji} : i,j = 1,2,\ldots,n$. It is said to be *tridiagonal* if there are at least some non-zero elements on the principal diagonal, on the co-diagonal immediately above and on the co-diagonal immediately below, but all other elements are zero. Thus, for a tridiagonal matrix $a_{ij} = 0$ for $|i-j|>1 : i,j=1,2,\ldots,n$. A is said to be *lower* (upper) *triangular* if $a_{ij} = 0$ for $j>i (j<i) : i,j=1,2,\ldots,n$. Thus, in a lower triangular matrix all the elements above the principal diagonal are zero; in an upper triangular matrix all the elements below the principal diagonal are zero. A matrix *A* for which the condition

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \qquad i = 1,2,\ldots,n$$

is satisfied is said to be *strictly diagonally dominant*. This definition means that for each row of the matrix, the magnitude of the element on the diagonal exceeds the sum of the magnitudes of the remaining elements in the row. If the condition

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \qquad i=1,2,\ldots,n$$

is satisfied, with inequality holding for at least one value of i, then A is said to be *diagonally dominant.*

**EXAMPLE 4.2**

The foregoing definitions are exemplified by the following 4 x 4 matrices

$$\begin{bmatrix}1&2&3&4\\2&5&6&7\\3&6&6&9\\4&7&9&10\end{bmatrix} \quad \begin{bmatrix}1&2&0&0\\3&4&5&0\\0&6&7&8\\0&0&9&10\end{bmatrix} \quad \begin{bmatrix}1&2&0&0\\2&3&4&0\\0&4&5&6\\0&0&6&7\end{bmatrix}$$

**(a) symmetric** **(b) Tridiagonal** **(c) Symmetric and tridiagonal**

$$\begin{bmatrix}1&0&0&0\\2&3&0&0\\4&5&6&0\\7&8&9&10\end{bmatrix} \quad \begin{bmatrix}1&2&3&4\\0&5&6&7\\0&0&8&9\\0&0&0&10\end{bmatrix} \quad \begin{bmatrix}10&-1&-2&4\\3&9&0&-5\\1&-1&-6&-2\\1&1&-2&5\end{bmatrix}$$

**(d) Lower triangular** **(e) Upper triangular** **(f) Strictly diagonally dominant**

$$\begin{bmatrix}10&-1&-2&4\\4&9&0&-5\\1&-1&-6&-2\\1&1&-2&5\end{bmatrix} \quad \begin{bmatrix}10&-2&0&0\\-2&-6&-1&0\\0&-1&3&1\\0&0&1&4\end{bmatrix}$$

**(g) Diagonally dominant** **(h) Symmetric, tridiagonal and strictly diagonally dominant**

A lower (upper) triangular matrix in which every element on the diagonal is equal to unity is said to be *unit lower (upper) triangular*. A lower (upper) triangular matrix in which every element on the diagonal is equal to zero is said to be *strictly lower (upper) triangular*. A matrix which has non-zero entries on the diagonal only is said to be *diagonal*. The diagonal matrix in which every non-zero entry is equal to unity is referred to as the identity (or unit) matrix and is denoted $I$. (Using the normal rules of matrix multiplication, we have that, for any square matrix $A$, $AI = IA = A$.)

Finally, we introduce two qualitative terms. We refer to a matrix as being *full* if it has at most a small number of zero elements. We refer to it as *sparse* if it has a large number of zero entries. These zero elements may occur in a regular pattern in the sense that there is a relation involving the row and column indices which defines the zero elements. Thus a 100x100 tridiagonal matrix (which contains at most 298 non-zero entries) would be described as a sparse matrix in which there is a regular pattern $\left(a_{ij} = 0 \textbf{ for } |i-j| > 1 : i,j = 1,2,\ldots,n\right)$.

## 4.3 GAUSS ELIMINATION AND BACK-SUBSTITUTION

### 4.3.1 Discussion of the basic method

*Gauss elimination* is a systematic way of solving the system (4.1) using the techniques employed in Example 4.1. We describe the method in the context of the set of four linear simultaneous equations

$$\sum_{j=1}^{4} a_{ij}x_j = b_i \qquad i = 1,2,3,4 \tag{4.3}$$

the generalisation being obvious. Writing the system (4.3) in its full form we have

$$\begin{aligned}
a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + a_{13}^{(1)}x_3 + a_{14}^{(1)}x_4 &= b_1^{(1)} \\
a_{21}^{(1)}x_1 + a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + a_{24}^{(1)}x_4 &= b_2^{(1)} \\
a_{31}^{(1)}x_1 + a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 + a_{34}^{(1)}x_4 &= b_3^{(1)} \\
a_{41}^{(1)}x_1 + a_{42}^{(1)}x_2 + a_{43}^{(1)}x_3 + a_{44}^{(1)}x_4 &= b_4^{(1)}
\end{aligned} \tag{4.4}$$

where we have added a superscript (1) to each coefficient $a_{ij}$ and each right-hand side $b_i$. This superscript is not an iteration number but its value will change as the elimination process proceeds. Its purpose will soon become apparent.

We begin by eliminating $x_1$ from the second, third and fourth equations. Assuming $a_{11}^{(1)} \neq 0$ we subtract the multiple $a_{21}^{(1)} / a_{11}^{(1)}$ Of the first equation from the second equation. Similarly, we take $a_{31}^{(1)} / a_{11}^{(1)}$ times the first equation from the third equation and $a_{41}^{(1)} / a_{11}^{(1)}$ times the first equation from the fourth equation. This transforms the system (4.4) to

$$a_{11}^{(1)} x_1 + a_{12}^{(1)} x_2 + a_{13}^{(1)} x_3 + a_{14}^{(1)} x_4 = b_1^{(1)}$$

$$\begin{aligned} a_{22}^{(2)} x_2 + a_{23}^{(2)} x_3 + a_{24}^{(2)} x_4 &= b_2^{(2)} \\ a_{32}^{(2)} x_2 + a_{33}^{(2)} x_3 + a_{34}^{(2)} x_4 &= b_3^{(2)} \\ a_{42}^{(2)} x_2 + a_{43}^{(2)} x_3 + a_{44}^{(2)} x_4 &= b_4^{(2)} \end{aligned} \qquad \textbf{(4.5)}$$

**where**

$$a_{ij}^{(2)} = a_{ij}^{(1)} - \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} a_{1j}^{(1)} \qquad \boldsymbol{i,j = 2,3,4} \qquad \textbf{(4.6)}$$

**and**

$$b_i^{(2)} = b_i^{(1)} - \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} b_{1j}^{(1)} \qquad \boldsymbol{i,j = 2,3,4} \qquad \textbf{(4.7)}$$

In the first stage of the elimination process the first equation in (4.4) remains unchanged. We now enter the second stage in which the first two equations of (4.5) are unaltered but $x_2$ is eliminated from the last two equations. This is achieved by subtracting $a_{i2}^{(2)} / a_{22}^{(2)}$ times the second equation from the ith equation for i=3,4 (provided that $a_{22}^{(2)} \neq 0$). This yields the system

$$
\begin{aligned}
a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + a_{13}^{(1)}x_3 + a_{14}^{(1)}x_4 &= b_1^{(1)} \\
a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= b_2^{(2)} \\
a_{33}^{(3)}x_3 + a_{34}^{(3)}x_4 &= b_3^{(3)} \\
a_{43}^{(3)}x_3 + a_{44}^{(3)}x_4 &= b_4^{(3)}
\end{aligned} \qquad \textbf{(4.8)}
$$

**where**

$$a_{ij}^{(3)} = a_{ij}^{(2)} - \frac{a_{i2}^{(2)}}{a_{22}^{(2)}} a_{2j}^{(2)} \qquad \boldsymbol{i,j{=}3,4} \qquad \textbf{(4.9)}$$

**and**

$$b_i^{(3)} = b_i^{(2)} - \frac{a_{i2}^{(2)}}{a_{22}^{(2)}} b_2^{(2)} \qquad \boldsymbol{i,j{=}3,4} \qquad \textbf{(4.10)}$$

Finally, we eliminate $x_4$ from the last equation in (3.8) by taking $a_{43}^{(3)} / a_{33}^{(3)}$ times the third equation away from it to give

$$
\begin{aligned}
a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + a_{13}^{(1)}x_3 + a_{14}^{(1)}x_4 &= b_1^{(1)} \\
a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= b_2^{(2)} \\
a_{33}^{(3)}x_3 + a_{34}^{(3)}x_4 &= b_3^{(3)} \\
a_{44}^{(4)}x_4 &= b_4^{(4)}
\end{aligned} \qquad (4.11)
$$

where

$$a_{44}^{(3)} = a_{44}^{(3)} - \frac{a_{43}^{(3)}}{a_{33}^{(3)}} a_{34}^{(3)} \qquad \textbf{(4.12)}$$

**and**

$$b_4^{(4)} = b_4^{(3)} - \frac{a_{43}^{(3)}}{a_{33}^{(3)}} b_3^{(3)} \qquad \textbf{(4.13)}$$

To summarise, the effect of the basic Gauss elimination process is to transform the original system (4.4) into the equivalent system (4.11) in which the matrix of coefficients is upper triangular. The solution is now obtained by a process of back-substitution: $x_4$ is obtained from the final equation in (4.11), $x_3$ from the last but one, and so on.

The equation which is used for elimination purposes at a particular stage is known as the pivotal equation. (Thus, in (4.4), the first equation, which is used to eliminate $x_1$ from the other equations, is the pivotal equation). In the pivotal equation, the coefficient of the term which is to be eliminated elsewhere is known as the pivot (so that in (4.4) $a_{11}^{(1)}$ is the pivot). The equations (4.6), (4.7), (4.10), (4.12) and (4.13) are referred to as the *updating formulae*. The generalisation of these formulae to the case of n linear simultaneous equations in n unknowns is quite straightforward. We have

$$\left.\begin{aligned} a_{ij}^{(k)} &= a_{ij}^{(k-1)} - \frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} a_{k-1,j}^{(k-1)} \qquad i,j = k,k+1,\ldots,n \\ b_i^{(k)} &= b_i^{(k-1)} - \frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} b_{k-1}^{(k-1)} \qquad i = k,k+1,\ldots,n \end{aligned}\right\} \begin{array}{l} (4.14) \\ k = 2,3,\ldots,n \\ (4.15) \end{array}$$

whilst

$$x_i = \left( b_i^{(i)} - \sum_{j=i+1}^{n} a_{ij}^{(i)} x_j \right) / a_{ii}^{(i)} \quad i = n,n-1,\ldots,1 \qquad (4.16)$$

is the general form for back-substitution.

**EXAMPLE 4.3**

Gauss elimination applied to the system

(i) $x_1 - x_2 - 2x_3 = -5$

(ii) $x_1 + 2x_2 + x_3 = 7$

(iii) $x_1 + 3x_2 - x_3 = 2$

Stage 1:

(I)' $x_1 - x_2 - 2x_3 = -5$ (= (i))

(ii)' $3x_2 + 3x_3 = 12$ (= (ii)-$\frac{1}{1}$ x (i))

(iii)' $4x_2 + x_3 = 7$ (=(ii)-$\frac{1}{1}$ x (i))

Stage 2:

$x_1 - x_2 - 2x_3 = -5$ (=(i)')

$3x_2 + 3x_3 = 12$ (=(ii)')

$-3x_3 = -9$ (=(iii)' $-$ $\frac{4}{3}$ (ii)')

Back-substitution:

$$x_3 = -9/\ -3 = 3$$
$$x_2 = \left(12 - 3x_3\right)/3 = 1$$
$$x_1 = \left(-5 + x_2 + 2x_3\right)/1 = 2$$

As mentioned in the introduction to this chapter, for a direct method it is possible to predict the total number of elementary arithmetic operations involved. It can be shown that the operations count for Gauss elimination is dominated by $n^3/3$ multiplications and additions/subtractions. Hence the effort in solving a linear system of equations by this method increased rapidly with n.

Although the numerical evaluation of a determinant is an operation which is rarely required, we make the observation that the value of the determinant of the matrix in the final system (4.11) is the same as that for the original coefficient matrix A. Moreover, it follows immediately that this value is just $a_{11}^{(1)}a_{22}^{(2)}a_{33}^{(3)}a_{44}^{(4)}$. The extension of this result to the general case is obvious.

At each stage of the elimination process we have made the assumption that the pivot $a_{ii}^{(i)}$ is non-zero. If $a_{11}^{(1)}$ =0 the process just described will break down. However, if this situation arises, at least one of $a_{21}^{(1)}, a_{31}^{(1)}$ and $a_{41}^{(1)}$ must be non-zero and one of these may be useful as the pivot instead of $a_{11}^{(1)}$ . This new pivot then defines a new pivotal equation. At the end of the first stage, $a_{22}^{(2)}$ may be zero but, if this is so, at least one of a $a_{32}^{(2)}$ and $a_{42}^{(2)}$ will be non-zero and so it will be possible to select a new non-zero pivot. Finally, if $a_{11}^{(1)} = 0, a_{43}^{3}$ ,which must be non-zero can be taken as the pivot. Even if none of the pivots is identically zero, it is possible that they may be very small in magnitude and consequently the multipliers $a_{i,k-1}^{(k-1)} / a_{k-1,k-1}^{(k-1)}$ may be large.

### 4.3.2 A program for Gauss Elimination

```
// a program for gauss elimination
# include<iostream.h>
#include<iomanip.h>
#define MAXN 50;
 main()
{
        int i,j,k,n;
        float a[50][50];
        float b[50];
  float x[50];
        float muit,sum;
 cin >>n;
                for(i=1; i<=n;i++)
                for(j=1; i<=n; j++) cin>>a[i][j];
                for(i=i; i<=n; i++) cin>>b[i];
                cout<<" the coefficient matrix is "<<endl;
                for(i=i; i<=n; i++)
{
        for(j=1; j<=n; j++)
        cout<< a[i][j];
        cout<<endl;
 }
        cout<<" the right hand side rec-tor is "<< endl;
        for(i=i; i<=n;i++) cout<< b[i];
        cout << endl;
        for(k=2; k<=n; k++)
        for(i=k; i <=n; i++)
        {
                muit = a[i][k-1]/a[k-1][k-1];
                b[i] = b[i] -muit*b[k-1];
                for (j=k ;j<=n; j++) a[i][j]=a[i][j]- muit*a[k-1][j];
        }
                for (i=n; n>=i;i--)
        {
                sum= b[i];
                for(j=i+1;i<=n; i++)
                        sum= sum-a[i][j]* x[j];
                        x[i]=sum/a[i][i];
        }
                        cout<<endl;
                        cout<<"the solution vector is "<< endl;
                        for(i=1;i<=n;i++) cout<<x[i];
                                        cout<<endl;
}
```

**Sample Data**

3

1.0 -1.0 -2.0

```
1.0     2.0      1.0
1.0     3.0     -1.0
-5.0    7.0      2.0
```

**Sample Output**

```
the coefficient matrix is
1.00000e 0       -1.00000e 0      -2.00000e 0
1.00000e 0        2.00000e 0       1.00000e 0
1.00000e 0        3.00000e 0      -1.00000e 0

the right hand side vector is
-5.00000e 0      7.00000e 0       2.00000e 0

the solution vector is
2.00000e 0       1.00000e 0       3.00000e 0
```

## 4.4 ITERATION

### 4.4.1 Derivation of the Jacobi and Gauss-Seidel Schemes

Before considering direct methods for the solution of the system (4.1) further we examine ways of setting up iterative processes which will, hopefully, converge to the required solution. These methods can be particularly useful if the coefficient matrix is sparse.

Suppose that we wish to solve the set of three linear simultaneous equations

$$\sum_{j=1}^{3} a_{ij}x_j = b_i \qquad i = 1,2,3 \qquad (4.17)$$

for the three unknowns $x_1, x_2$ and $x_3$. Let $x_1^{[0]}, x_2^{[0]}$ and $x_3^{[0]}$ be initial approximations to $x_1, x_2$ and $x_3$ respectively. Then, assuming that $a_{11}, a_{22}$ and $a_{33}$ are all non-zero, the system (4.17) may be rewritten as

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{aligned} \qquad (4.18)$$

and this suggests the iterative scheme (jacobi iteration)

$$\begin{aligned} x_1^{[k]} &= (b_1 - a_{12}x_2^{[k-1]} - a_{13}x_3^{[k-1]})/a_{11} \\ x_2^{[k]} &= (b_2 - a_{21}x_1^{[k-1]} - a_{23}x_3^{[k-1]})/a_{22} \qquad k = 1,2,\ldots \qquad (4.19) \\ x_3^{[k]} &= (b_3 - a_{31}x_1^{[k-1]} - a_{32}x_2^{[k-1]})/a_{33} \end{aligned}$$

where the subscript indicates an iteration number. Thus, a sequence of approximations $\{( x_1^{[k]}, x_2^{[k]}, x_3^{[k]} ) : k = 0,1, \ldots\}$ is generated and under suitable circumstances it will converge to the required solution.

**EXAMPLE 4.5**

The system

$$\begin{aligned} 4x_1 + x_2 - x_3 &= 12 \\ -x_1 + 3x_2 + x_3 &= 6 \\ 2x_1 + 2x_2 + 5x_3 &= 5 \end{aligned}$$

has the solution $x_1 = 2, x_2 = 3, x_3 = -1$. If the initial approximation is $x_1^{[0]} = x_2^{[0]} = x_3^{[0]} = 0,$

the first application of Jacobi iteration yields $x_1^{[1]} = 12/4 = 3, x_2^{[1]} = 6/3 = 2, x_3^{[1]} = 5/5 = 1.$

The next iteration gives

$$\begin{aligned} x_1^{[2]} &= (12 - 1 \times 2 - (-1) \times ) / 4 = 11/4 = 2.75000 \\ x_2^{[2]} &= (6 - (-1) \times )3 - 1 \times 1 \ ) / 3 = 8/3 = 2.66667 \\ x_3^{[2]} &= (5 - 2 \times 3 - 2 \times 2) / 5 = -5/5 = -1.00000 \end{aligned}$$

Further iterations are listed in Table 4.1 correct to six significant figures.

**Table 4.1**

| | Jacobi | | | k | Gauss-Seidel | | |
|---|---|---|---|---|---|---|---|
| k | $x_1^{[k]}$ | $x_2^{[k]}$ | $x_3^{[k]}$ | | $x_1^{[k]}$ | $x_2^{[k]}$ | $x_3^{[k]}$ |
| 0 | 0.00000 | 0.00000 | 0.00000 | | 0.00000 | 0.00000 | 0.00000 |
| 1 | 3.00000 | 2.00000 | 1.00000 | | 3.00000 | 3.00000 | -1.40000 |
| 2 | 2.75000 | 2.66667 | -1.00000 | | 1.90000 | 3.10000 | -1.00000 |
| 3 | 2.08333 | 3.25000 | -1.16667 | | 1.97500 | 2.99167 | -0.986667 |
| 4 | 1.89483 | 3.08333 | -1.13333 | | 2.00542 | 2.99736 | -1.00111 |
| 5 | 1.94583 | 3.00972 | -0.991667 | | 2.00038 | 3.00050 | -1.00035 |
| 6 | 1.99965 | 2.97917 | -0.982222 | | 1.99979 | 3.00005 | -0.999933 |
| 7 | 2.00965 | 2.97917 | -0.991528 | | 2.00000 | 2.99998 | -0.999994 |
| 8 | 2.00363 | 3.00039 | -1.00144 | | 2.00001 | 3.00000 | -1.00000 |
| 9 | 1.99954 | 3.00169 | -1.00161 | | 2.00000 | 3.00000 | -1.00000 |
| 10 | 1.99918 | 3.00038 | -1.00049 | | | | |
| 11 | 1.99978 | 2.99989 | -0.999823 | | | | |
| 12 | 2.00007 | 2.99987 | -0.999868 | | | | |
| 13 | 2.00007 | 2.99998 | -0.999976 | | | | |
| 14 | 2.00001 | 3.00001 | -1.00002 | | | | |
| 15 | 1.99999 | 3.00001 | -1.00001 | | | | |

An immediate extension of the scheme (4.19) is to make use of the new iterates as soon as they become available. This gives the process (Gauss-Seidel iteration)

$$\begin{aligned} x_1^{[k]} &= (b_1 - a_{12}x_2^{[k-1]} - a_{13}x_3^{[k-1]})/a_{11} \\ x_2^{[k]} &= (b_2 - a_{21}x_1^{[k-1]} - a_{23}x_3^{[k-1]})/a_{22} \qquad \boldsymbol{k = 1,2,....} \\ x_3^{[k]} &= (b_3 - a_{31}x_1^{[k-1]} - a_{32}x_2^{[k-1]})/a_{33} \end{aligned}$$

so that, for example, in the second equation of the system (4.22) the most recently computed estimate, $x_1^k$, of $x_1$ is used to find $x_2^k$ .

**EXAMPLE 4.6**

Consider the system of equations of Example 4.5 and again let $x_1^{[0]} = x_2^{[0]} = x_3^{[0]} = 0$. Then, as before, $x_1^{[1]} = 3$ but now

$$x_2^{[1]} = \left(6 - (-1) \times 3 - 1 \times 0\right) / 3 = 9/3 = 3$$

and

$$x_3^{[1]} = (5 - 2 \times 3 - 2 \times 3) / 5 = -7/5 = -1.4$$

Table 4.1 lists further iterations and we note that Gauss-Seidel iteration has obtained the solution (2.00000, 3.00000, -1.00000)after just 9 iterations whereas, at the same stage, Jacobi iteration stilll has some way to go. Further, after two iterations both the Jacobi and Gauss-Seidel schemes give $x_3^{[2]} = -1$ and then move away from this value before the converging to it ultimately. We conclude that when monitoring iterates it is important that we measure the overall convergence of the values $\left\{\left(x_1^{[k]}, x_2^{[k]}, x_3^{[k]}\right): k = 0;1, \ldots \right\}$ and we investigate ways of doing this in subsection 4.4.3.

Jacobi and Gauss-Seidel iteration can be readily extended to the case of n linear simultaneous equations in *n* unknowns. Jacobi iteration takes the form

$$x_1^{[k]} = \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{[k-1]} \right) / a_{ii} \qquad \begin{array}{l} i = 1,2,\ldots,n; \\ k = 1,2,\ldots \end{array} \qquad (4.20)$$

whilst Gauss-Seidel iteration is

$$x_1^{[k]} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{[k]} - \sum_{j=i+1}^{n} a_{ij} x_j^{[k-1]} \right) / a_{ii} \qquad \begin{array}{l} i = 1,2,\ldots,n; \\ k = 1,2,\ldots \end{array} \qquad (4.21)$$

In both cases we have assumed $a_{ii} \neq 0 : i = 1,2,\ldots,n.$

As in the iterative schemes discussed in Chapter 3, it is necessary to examine the conditions under which a suitable scheme will converge and, if convergent, to formulate a suitable termination criterion.

### 4.5.2 A Convergence Theorem

THEOREM 4.1

In the set of $n$ linear simultaneous equations

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \qquad i = 1,2,\ldots,n$$

suppose the matrix of coefficients is strictly diagonally dominant, that is,

$$|a_{ii}| > \sum_{\substack{i=1 \\ j \neq i}}^{n} |a_{ij}| \qquad i = 1,2,\ldots,n$$

Then Jacobi and Gauss-Seidel iteration converge from arbitrary initial estimates of the unknowns, that is, for any $x_1^{[0]}, x_2^{[0]}, \ldots, x_n^{[0]}$.

The proof of this theorem is not particularly difficult but, without some formal concepts from linear algebra, it is very lengthy (see Johnson and Riess (1982) for further details). The theorem shows the importance of the class of strictly diagonally dominant matrices. However, we note that strict diagonal dominance is a sufficient but not necessary condition for convergence and consequently these iterative schemes may converge for other sets of equations.

**EXAMPLE 4.7**

The coefficient matrix of Example 4.5 is strictly diagonally dominant and hence we would expect Jacobi and Gauss-Seidel iteration to converge for this particular problem. The results contained in Table 4.1 bear this out. In Example 4.1, the coefficient matrix is not strictly diagonally dominant and we can say there is no guarantee that Jacobi and Gauss-Seidel iteration will converge. We cannot guarantee that they will both diverge but, starting with $x_1^{[0]} = x_2^{[0]} = x_3^{[0]} = 0,$ Jacobi iteration gives $x_1^{[10]} = 302.125, x_2^{[10]} = -106.188, x_3^{[10]} = 217.375$ whilst Gauss-Seidel iteration gives

$x_1^{[10]} = 85011.3, x_2^{[10]} = -58568.9, x_3^{[10]} = -90697.4$ and both processes are clearly diverging from the true solution. In the system

$$4x_1 + 2x_2 + x_3 = 12$$
$$2x_1 + 3x_2 - x_3 = 7$$
$$2x_1 + 2x_2 + 2x_3 = 8$$

the coefficient matrix is again not strictly diagonally dominant. However, starting with $x_1^{[0]} = x_2^{[0]} = x_3^{[0]} = 0,$ Jacobi iteration gives $x_1^{[18]} = 3.00000, x_2^{[18]} = 1.00001, x_3^{[18]} = 2.00002$ and Gauss-Seidel iteration gives $x_1^{[15]} = 3.00000, x_2^{[15]} = 1.00000, x_3^{[15]} = 2.00000.$ (All figures here have been quoted correct to six significant figures). Since the true solution is $x_1 = 3, x_2 = 1, x_3 = 2$ we can say that both Jacobi and Gauss-Seidel iteration converge for this problem given the starting value $x^{[0]T} = (0,0,0)$.

### 4.4.3 Termination criteria

We first consider ways of determining the size of a vector y. Three possible measures are

(i) the sum of the absolute values of the components of y

$$\sum_{i=1}^{n} |y_i|, \qquad (4.22)$$

(ii) the square root of the sum of the squares of the components of y

$$\left(\sum_{i=1}^{n} y_i^2\right)^{1/2}, \qquad (4.23)$$

(iii) the maximum absolute component of y

$$\underset{1 \le i \le n}{\text{maximum}} |y_i| . \qquad (4.24)$$

The quantities (4.22), (4.23) and (4.24) are, in turn, referred to as the *$L_1$-norm, $L_2$-norm and $L\infty$ -norm of y*. Whilst these are not the only ways of quantifying the size of a vector, they are the ones most commonly used in numerical analysis.

Now, in Example 4.6, we noted that a convergence criterion for an iterative process must be based on the convergence of each of the iterates $x_i^{[k]} : k = 0,1,\ldots, i = 1,2,\ldots$. If we use (4.24) to test for convergence an iterative method such as Jacobi or Gauss-Seidel would be continued until

$$\max_{1 \le i \le n} \left| x_i^{[k]} - x_i^{[k-1]} \right| < \varepsilon \qquad (4.25)$$

where $\varepsilon$ is some chosen tolerance. It may be preferable to use a relative, as opposed to absolute, accuracy criterion.

If the chosen iteration is slowly convergent there are likely to be difficulties. It is noted that a convergence criterion based on ensuring that the absolute value of the difference between successive iterates is less than some tolerance does not guarantee that the root has been found to this accuracy. Here, if the iterative process is continued until (4.25) is satisfied, there is no guarantee that

$$\max_{1 \le i \le n} \left| x_i^{[k]} - x_i \right| < \varepsilon$$

holds. In such a case it may be preferable to examine the residual vector $r^{[k]}$ whose components are defined by

$$r_i^{[k]} = b_i - \sum_{j-1}^{n} a_{ij} x_j^{[k]} \qquad i = 1,2,\ldots,n$$

and formulate a criterion based on some measure of this vector. However, in the case of a badly conditioned set of equations it is clear that small residuals do not necessarily imply an accurate solution.

### 4.4.4 A program forGauss-Seidel Method

```
//a program for Gauss - Seidel Method
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
#define MAXN 50;
```

```
class matrixtype
{public:
          float matrixtype[50];
};
 class rhstype
 {public:
          float rhstype[50];
 };

  class itermaxtype
  {public:
          int itermaxtype;
          };

          main()
          { matrixtype a;
           rhstype x;
           rhstype b;
           float tol;
           int i,j,n;
           itermaxtype itermax;
           void gausseidel (int n,matrixtype a, rhstype x,rhstype b,float tol,itermaxtype itermaxl);
           cin>>n;
           for (i=1;i<=n;i++);
                   for (j=1;j<=n;j++);
                           cin>>a[i][j];
           for (i=1;i<=n;i++) cin>> b[i];
                   for (i=1;i<=n;i++) cin>> x[i];
                           cin>>tol>>itermax;
                           cout<<endl;
                           cout<<"the coefficient matrix is "<<endl'
                           for (i=1;i<=n;i++)
                           {
                                   for (j=1;j<=n;j++) cout<<a[i][j]<<endl;
```

```
        }
        cout<<endl;
        cout<<"the right hand side vector is "<<endl;
        for (i=1;i<=n;i++) cout<<b[i]<<endl<<endl;
        cout<<"the initial approximation is "<<endl;
        for (i=1;i<=n;i++) cout<<x[i]<<endl<<endl;
        cout<<"accuracy tolerance      "<<tol<<endl;
        cout<<"Maximum number of iteration allowed "<<itermax<<endl;
        gausseidel(n,a,b,x,tol,itermax);
        cout<<endl;
        cout<<"the solution vector is "<<endl;
        for ( i=1;i<=n;i++) cout<<x[i]<<endl;
}
void gausseidel (int n,matrixtype a, rhstype x,rhstype b,float tol,itermaxtype itermaxl);
        { rhstype xold;
                int converged;
                int iter;
                float sum;
                int i,j;
                iter =0;
                do
                 { iter = iter+1;
                        for (i=1;i<=n;i++) xold[i]= x[i];
                        for ( i=1;i<=n;i++)
                        {
                         sum = 0.0;
                         for (j=1;j<=n;j++)
                                if (i<>j) sum = sum + a[i][j] *x[j];
                         x[i[ = (b[i] -sum )/a[i][i];
                        }
                        i=0;
                        do
                        { i=i+1;
                                converged = abs(x[i] -xold[i]) < tol
```

```
                    } while ( (i=n) || (converged == 0));
                } while ((converged ) || (iter== itermax));

                cout<<endl;
                if ( converged == 0) cout<< "no convergence to specified tolerance
after"<<iter<<"iterations"<<endl;
    }
```

**Sample Data**

```
3
 4.0    -2.0     1.0
 2.0     3.0    -1.0
 2.0    -2.0     2.0
12.0     7.0     8.0
 0.0     0.0     0.0
0.00005
25
```

**Sample Output**

```
the coefficient matrix is
 4.00000e 0    -2.00000e 0     1.00000e 0
 2.00000e 0     3.00000e 0    -1.00000e 0
 2.00000e 0    -2.00000e 0     2.00000e 0

the right hand side vector is
 1.20000e 0    7.00000e 0     8.00000e 0

the initial approximation is
 0.00000e 0    0.00000e 0     0.00000e 0

accuracy tolerance                  5.00000e -5
maximum number of iterations allowed  25
convergence to specified tolerance after 13      iterations
the solution vector is
 3.00000e 0    9.99986e -1     1.99998e 0
```

### 4.4.5 Computational Efficiency (Complexity)

The number of elementary arithmetic operations involved in each iteration, whether Jacobi or Gauss-Seidel, is proportional to $n^2$ . The overall cost of each scheme is then determined by the number of iterations required to achieve convergence. Using the same initial approximation it is possible to compare the efficiency of these two schemes and, in this sense, it is usually the case that Gauss-Seidel iteration is more efficient than Jacobi (see Table 4.1). Other iterative schemes have been proposed which aim to keep the number of iterations required as low as possible.

Consider the generalised form of the system (4.18)

$$x_i = \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right) / a_{ii} \qquad i = 1,2,\ldots,n$$

Suppose that we choose some value p and multiply through by this number.

Adding $x_i$ to both sides and rearranging we have

$$x_i = (1-p)x_i + p\left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right) / a_{ii} \qquad i = 1,2,\ldots,n$$

and this suggests the iterative scheme.

$$x_i^{[k]} = (1-p)x_i^{[k-1]} + p\left( b_i - \sum_{j=1}^{i=1} a_{ij} x_j^{[k]} - \sum_{j=j+1}^{n} a_{ij} x_j^{[k-1]} \right) / a_{ii} \qquad (4.26)$$

$$i = 1,2,\ldots,n; \quad k = 1,2,\ldots$$

(Compare this with equation 4.21) known as *successive over-relaxation (SOR)*. The choice of the parameter p is a matter of advanced analysis and we can do no more than state here that, for certain classes of matrix, it is possible to find the optimum value of p in order to achieve rapid convergence, and that for other classes it may be possible to find a value for p which is near the optimum.

We conclude this subsection by observing that it is possible to formulate and use the Aitken acceleration scheme in the present context. Three successive iterates, $x^{[k]}, x^{[k+1]}$ and $x^{[k+2]}$ are required and the components $x_i^{[k]}, x_i^{[k+1]}$ and $x_i^{[k+2]}$ can be combined to give a new initial approximation which is used to generate two further iterates and so on.

The question can now be asked that how does programming in C++ improve the complexity?
From table 4.1 it is clear that Gauss-Seidel iteration converges quickly compared to Jacobi and for a longer array of variables in which solutions are required, manual computation using a desktop calculation might not suffice due error factor. Therefore, the advanced coding features of C++ which provide compact matrix operands, structured and object oriented programming engenders tremendous improvement in computational efficiency with regard to processing time, accuracy and convergence.

### 4.4.6 A Further Discussion of Iterative Methods

Before leaving iterative methods we give a very brief indication of the way in which a fuller analysis of some iterative schemes might be undertaken.

Consider the system of equations (4.1) and assume that $a_{ii} \neq 0 : i = 1,2,\ldots,n.$ The coefficient matrix *A* may be split as

$$A = L + D + U$$

where D is a diagonal matrix with elements $a_{11}, a_{22}, \ldots, a_{nn}$ on the diagonal and L and U are strictly lower and upper triangular matrices, respectively, defined by

$$L = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ a_{21} & \mathbf{0} & \cdots & \mathbf{0} \\ \cdots & & & \\ a_{n1} & a_{n2} & \cdots & \mathbf{0} \end{bmatrix}, \quad U = \begin{bmatrix} \mathbf{0} & a_{12} & \cdots & a_{1n} \\ \mathbf{0} & \mathbf{0} & \cdots & a_{2n} \\ \cdots & & & \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$$

This means that the original system may be expressed as

$$(L + D + U)\times = b$$

and this can be rewritten in a number of ways to provide different iterative schemes. In particular, we have

$$Dx = -(L+U)x + b$$

which suggests the scheme

$$x^{[k]} = -D^{-1}(L+U)x^{[k-1]} + D^{-1}b$$

and this is Jacobi iteration. It follows immediately that Gauss-Seidel iteration takes the form

$$x^{[k]} = -D^{-1}Lx^{[k]} - D^{-1}Ux^{[k-1]} + D^{-1}b. \qquad (4.27)$$

However, on premultiplying (3.27) by D and rearranging, we have

$$(D+L)x^{[k]} = -Ux^{[k-1]} + b$$

and so Gauss-Seidel iteration may also be written as

$$x^{[k]} = -(D+L)^{-1}Ux^{[k-1]} + (D+L)^{-1}b$$

Now, consider the system (3.26). Multiplying through by a, these equations may be written, in matrix form as

$$Dx^{[k]} = (1-p)Dx^{[k-1]} + pb - pLx^{[k]} - pUx^{[k-1]}$$

Rearranging, we have

$$(D+pL)x^{[k]} = ((1-p)D - pU)x^{[k-1]} + pb$$

or

$$x^{[k]} = (D+pL)^{-1}((1-p)D - pU)x^{[k-1]} + p(D+pL)^{-1}b$$

Hence, it is possible to express each of the iterative schemes that we have considered in the form

$$x^{[k]} = Mx^{[k-1]} + g \qquad (4.28)$$

where

(i) for Jacobi iteration $M = -D^{-1}(L+U)$ and $g = D^{-1}b$;

(ii) for Gauss-Seidel iteration $M = -(D+L)^{-1}U$ and $g = (D+L)^{-1}b$;

(iii) for SOR $M = (D+pL)^{-1}((1-p)D - pU)$ and

$$g = p(D + pL)^{-1} b.$$

Equation (4.28) allows us to study the convergence of all of our iterative methods within a general framework. Let $\varepsilon_k$ denote the error in $x^{[k]}$, so that

$$\varepsilon_k = x - x^{[k]}$$

where x is the solution to the original system (4.1). Then from (4.28) we must have

$$x = Mx + g \qquad (4.29)$$

so that subtracting (4.28) from (4.29) we find that

$$\varepsilon_k = M\varepsilon_{k-1}$$

Continuing the process

$$\varepsilon_k = M^2 \varepsilon_{k-2} = \ldots = M^k \varepsilon_0$$

where $\varepsilon_0$ is the error in the initial approximation $x^{[0]}$. Hence, the iterative process will converge if $M^k$ tends to the zero matrix (that is, the matrix whose entries are all zero) as $k$ increases.

## 4.5 FACTORISATION METHODS

### 4.5.1 Introduction

In this section we return to the development of direct methods for the solution of a system of linear simultaneous equations. We shall need to use matrix notation and some elementary results of matrix algebra.

We begin by considering the solution of three equations in three unknowns for which the coefficient matrix, $A$, is given by

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Let $L, D$ and $U$ be the three matrices

$$L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \quad D = \begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & d_3 \end{bmatrix} \quad U = \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

that is, *L* and *U* are unit lower and upper triangular respectively and *D* is diagonal. We now investigate whether it is possible to find values for the components of *L, D* and *U* such that

$$LDU = A. \tag{4.30}$$

If we perform the matrix multiplications on the left-hand side of (4.33) and then equate corresponding elements row-by-row, we obtain the following results.

Row 1: $d_1 = a_{11}; d_1 u_{12} = a_{12}; d_1 u_{13} = a_{13}$

Row 2: $l_{21} d_1 = a_{21}; l_{21} d_1 u_{12} + d_2 = a_{22}; l_{21} d_1 u_{13} + d_2 u_{23} = a_{23}$

Row 3: $l_{31} d_1 = a_{31}; l_{31} d_1 u_{12} + l_{32} d_2 = a_{32}; l_{31} d_1 u_{13} + l_{32} d_2 u_{23} + d_3 = a_{33}$

The results for row 1 enable $d_1, u_{12}$ and $u_{13}$ to be found in turn; those for row 2 enable $l_{21}, d_2$ and $u_{23}$ to be found in turn and those for row 3 enable $l_{31}, l_{32}$ and $d_3$ to be found in turn.

**EXAMPLE 4.8**

Consider the system of equations of Example 4.1. Then the coefficient matrix for this problem is

$$A = \begin{bmatrix} 1 & -1 & -2 \\ 1 & 2 & 1 \\ 1 & 3 & -1 \end{bmatrix}$$

so that

$$d_1 = a_{11} = 1$$

$$u_{12} = a_{12} / d_1 = -1/1 = -1$$
$$u_{13} = a_{13} / d_1 = -2/1 = -2$$
$$l_{21} = a_{21} / d_1 = 1/1 = 1$$
$$d_2 = a_{22} - l_{21}d_1u_{12} = 2 - 1 \times 1 \times (-1) = 3$$
$$u_{23} = (a_{23} - l_{21}d_1u_{13}) / d_2 = (1 - 1 \times 1 \times (-2)) / 3 = 1$$
$$l_{31} = a_{31} / d_1 = 1/1 = 1$$
$$l_{32} = \left(a_{32} - l_{31}d_1u_{12}\right) / d_2 = \left(3 - 1 \times 1 \times (-1)\right) / 3 = 4/3$$
$$d_3 = a_{33} - l_{31}d_1u_{13} - l_{32}d_2u_{23} = -1 - 1 \times 1 \times (-2) - 4/3 \times 3 \times 1 = -3$$

Hence

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 4/3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & -1 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

This result should now be checked by first forming $Z = DU$ and then $A = LZ$.

If $a_{11} = 0$ then $d_1 = 0$ which means $u_{12}, u_{13}, l_{21}$ and $l_{31}$ cannot be found and the process breaks down. If $a_{11} \neq 0$ but $d_2 = 0$, it will not be possible to compute $u_{23}$ and $l_{32}$. Now, $d_2 = a_{22} - l_{21}d_1u_{12} = a_{22} - \left(a_{21}/d_1\right)d_1\left(a_{12} / d_1\right) = a_{22} - a_{21}\, a_{12}/a_{11}$. Hence the condition $d_2 \neq 0$ is the same as $a_{11}a_{22} - a_{21}a_{12} \neq 0$. But this means that the determinant of the minor

of *A* must be non-zero. Extending these results further we have the following theorem.

**THEOREM 4.2 (without proof)**

In the $n \times n$ matrix $A$ suppose that the determinant of each sub-matrix $A_1, A_2, \ldots, A_n$ is non-zero where

$$A_1 = \left(a_{11}\right), \qquad A_2 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad A_3 = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

and so on. Then there exists a unique factorisation $A = LDU$, where $L$ and $U$ are respectively unit lower and upper triangular and $D$ is diagonal with non-zero diagonal elements.

However, two of its consequences shall be examined. The first is that the product $DU$ is an upper triangular matrix so a unique factorisation of $A$ into a unit lower triangular matrix and an upper triangular matrix exists. Similarly, the product $LD$ is a lower triangular matrix so a unique factorisation into a lower triangular matrix and a unit upper triangular matrix exists.

### 4.5.2 The Methods of Doolittle, Crout and Choleski

In the previous subsection we observed that if the matrix $A$ satisfies the conditions of Theorem 4.2 a unique resolution $A=LU$ exists where $L$ is unit lower triangular and $U$ is upper triangular. If $A$ has three rows and three columns we have

$$\begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ l_{21} & \mathbf{1} & \mathbf{0} \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{22} \\ 0 & \mathbf{0} & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad \textbf{(4. 31)}$$

The entries in $L$ and $U$ can either be found by forming the $LDU$ decomposition of $A$ (where, here, $U$ is unit upper triangular) and computing $DU$ or directly by multiplying out the left-hand side of (4.31) and equating corresponding elements in the equation to find $u_{11}, u_{12}, u_{13}, l_{21}, u_{22}, u_{23}, l_{31}, l_{32}$ and $u_{33}$ in that order.

Given that the $LU$ decomposition of a matrix $A$ exists, the system of equations (4.1) may be written as

$LU$x=b

or (4.32)

$L$y=b

where y satisfies the equation

Ux=y (4.33)

Hence, the solution vector x may be found by factorising $A$ into the product $LU$ and then

(i) forming the intermediate vector y from (4.32) using forward substitution,

(ii) forming x from (4.33) using back-substitution.

In detail, for a system of three equations in three unknowns we have, for stage (i)

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{31} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

so that

$$\begin{aligned} y_1 &= b_1 \\ l_{21}y_1 + y_2 &= b_2 \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_3 \end{aligned}$$

and $y_1, y_2$ and $y_3$ can be found in turn. For stage (ii), we have

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

giving

$$
\begin{aligned}
u_{33}x_3 &= y_3 \\
u_{22}x_2 + u_{23}x_3 &= y_2 \\
u_{11}x_1 + u_{12}x_2 + u_{13}x_3 &= y_1
\end{aligned}
$$

and now $x_3, x_2 \text{and } x_1$ can be found in turn.

The process based on the *LU* factorisation of *A* and the subsequent solution of equations (4.32) and (4.33) is known as *Doolittle's* method. Formally, for a system of *n* equations in n unknowns we have

$$
\begin{aligned}
&u_{ij} = a_{1j} && j = 1,2,\dots,n \\
&l_{i1} = a_{i1} / u_{11} && i = 2,3,\dots,n \\
&\left.\begin{aligned}
&u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} && j = i, i+1,\dots,n \\
&l_{ji} = \left( a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki} \right) / u_{ii} && j = i+1, i+2,\dots,n
\end{aligned}\right\} \quad i = 2,3,\dots,n
\end{aligned}
$$

and then

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j \qquad i = 1,2,\dots,n \qquad \textbf{(4. 34)}$$

$$x_i = \left( y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii} \qquad i = n, n-1,\dots,1. \quad \textbf{(4. 35)}$$

**EXAMPLE 4.8**

The coefficient matrix of Example 4.1 may be expressed as

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

Hence, if $\mathrm{b}^{\mathrm{T}} = (-5,7,2)$

$$\begin{aligned} y_1 &= -5 \\ y_2 &= 7 - 1 \times (-5) = 12 \\ y_3 &= 2 - 1 \times (-5) - 4/3 \times 12 = -9 \end{aligned}$$

and

$$\begin{aligned} x_3 &= -9/-3 = 3 \\ x_2 &= (12 - 3 \times 3)/3 = 1 \\ x_1 &= (-5 - (-2) \times 3 - (-1) \times 1)/1 \end{aligned}$$

There are three observations which must be made. The first is that there is a close connection between the Doolittle method as we have described it and the basic Gauss Elimination process described in subsection 4.3.1. The upper triangular matrix $U$ is the matrix of coefficients defined by (4.14) and the off-diagonal components of the matrix $L$ are the multipliers appearing in the same equation. In fact, we have

$$l_{i,k-1} = \frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} \qquad i = k, k+1, \ldots, n; k = 2,3,\ldots,n.$$

The intermediate vector is the vector defined by equation (4.15). It is not difficult to verify this association in the case of three equations in three unknowns but the general proof is lengthy and we shall not pursue the matter further. The second observation is that it will be necessary to consider some reformulation of the method so that pivoting strategies can be included. Finally, we note that once we have the $LU$ factorisation of a Matrix $A$ any problem containing this coefficient matrix may be solved by making use of equation (4.34) and (4.35) only and we refer back to the process of the iterative refinement of an approximate solution described in subsection 4.4.2.

The method of *Crout* can be developed in a similar manner to that of Doolittle. It depends on the resolution *A=LU,* where *L* is lower triangular and *U* is unit upper triangular. *Choleski's* method makes use of the resolution $A = LL^{\mathrm{T}}$ for a symmetric matrix, which *L* is lower triangular.

Factorisation methods can be of special value when the matrix A has a special form and we give an example of this by considering the case of a tridiagonal matrix.

### 4.5.3 Factorisation of a Tridiagonal Matrix

Let *A* be the tridiagonal matrix

$$A = \begin{bmatrix} a_1 & c_1 & & & 0 \\ d_2 & a_2 & c_2 & & \\ & \cdots & \cdots & \cdots & \\ & & d_{n-1} & a_{n-1} & c_{n-1} \\ 0 & & & d_n & a_n \end{bmatrix}$$

with the $a_i s$ on the diagonal, the $c_i s$ on the co-diagonal immediately above and the $d_i s$ on the co-diagonal immediately below. All other entries are zero. We seek a resolution of *A* of the form *A =LU* where

$$L = \begin{bmatrix} \alpha_1 & & & 0 \\ d_2 & \alpha_2 & & \\ \cdots & \cdots & & \\ & d_{n-1} & \alpha_{n-1} & \\ 0 & & d_n & \alpha_n \end{bmatrix}, \quad U = \begin{bmatrix} 1 & \gamma_1 & & & 0 \\ & 1 & \gamma_2 & & \\ & & \cdots & \cdots & \\ & & & 1 & \gamma_{n-1} \\ 0 & & & & 1 \end{bmatrix}$$

In this definition of the lower triangular matrix *L*, the zeros indicate that all entries except those on the diagonal and the co-diagonal immediately below are identically zero. Similarly, the upper triangular matrix *U* has non-zero entries only on the diagonal an the co-diagonal immediately above.

On multiplying out and equating elements we find that

$$\begin{aligned} &\alpha_1 = a_1 \\ &\left.\begin{aligned} \alpha_{i-1}\gamma_{i-1} &= c_{i-1} \\ d_i\gamma_{i-1} + \alpha_i &= a_i \end{aligned}\right\} \quad i = 2,3,\ldots,n \end{aligned} \qquad \textbf{(4.36)}$$

From this system we can find $\alpha_1, \gamma_1, \alpha_2, \gamma_2, \ldots$ and so on in turn provided that no $\alpha_i = 0{:}i = 1,2,\ldots,n$ (and this condition is satisfied if *A* is positive definite). Now, let *A* be the coefficient matrix in the system (4.1). The solution vector x may be obtained by first solving

*L*y=b

for the immediate vector y and then solving

*U*x=y.

The forward substitution process for y is simply

$$\begin{aligned} y_1 &= b_1/\alpha_1 \\ y_i &= (b_i - d_i y_{i-1})/\alpha_i \qquad i = 2,3,\ldots,n \end{aligned} \qquad (4.37)$$

whilst the back-substitution for x is

$$\begin{aligned} x_n &= y_n \\ x_i &= y_i - \gamma_i x_{i+1} \qquad i = n-1, n-2, \ldots, 1. \end{aligned} \qquad \textbf{(4.38)}$$

We note that the number of multiplications involved in each of the stages (4.36), (4.37) and (4.38) is proportional to $n$. This compares very favourably with the basic Gauss elimination process where the operations count is proportional to $n^3$. In addition, if the tridiagonal matrix is diagonally dominant the method can be shown to be numerically stable, so that no pivoting strategy is required.

If the matrix A is symmetric as well as tridiagonal, that is

$$A = \begin{bmatrix} a_1 & c_1 & & & 0 \\ c_1 & a_2 & c_2 & & \\ & \cdots & \cdots & \cdots & \\ & & c_{n-2} & a_{n-1} & c_{n-1} \\ 0 & & & c_{n-1} & a_n \end{bmatrix}$$

and the conditions of Theorem 4.2 are satisfied, the factorisation $A = LL^{\mathrm{T}}$ exists where

$$L = \begin{bmatrix} p_1 & & & 0 \\ q_1 & p_2 & & \\ & \cdots & \cdots & \\ & & q_{n-2} & p_{n-1} & \\ 0 & & & q_{n-1} & p_n \end{bmatrix}$$

and

$$p_1 = \sqrt{a_1}$$
$$\left.\begin{aligned} q_{i-1} &= c_{i-1} / p_{i-1} \\ p_i &= \sqrt{\left(a_i - q_{i-1}^2\right)} \end{aligned}\right\} \quad i = 2,3,\ldots n.$$

If A is positive definite the values $p_i : i = 1,2,\ldots,n$ are guaranteed to be real, and not complex, numbers.

A tridiagonal matrix is an example of a band matrix. For such matrices, $a_{ij} = 0 : |i - j| > k$ for some *k*.

The value $2k+1$ is known as the band width of the matrix. Clearly the ideas introduced here may be readily extended to the case $k > 1$.

### 4.5.4 A program for the factorisation of Tridiagonal Coefficient Matrix

```
//a program for the solution of a system of linear equations in which the coefficient matrix is tridiagonal
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
#define MAXN 50;
class indextype
 {public:
                int indextype;
 };
 class vector
 {public:
         float vector[50];
         };
main()
{ vector a,b,c,d;
        indextype i,n;
        void readtriding (indextype n,vector a,c,d);
        void printtridiag  (indextype n, vector a,c,d);
        void factorise (indextype n, vector a,c,d);
        void forwardsub (indextype n,vector a,d,b);
        void backsub (indextype n, vector b,c);
        cin>>n;
        readtridiag (n,a,c,d);
        for (i=1;i<=n;i++) cin>> b[i] ; cout<<endl;
        cout<<"the coefficient matrix is "<<endl;
        printtridiag (n,a,c,d);
        cout<<endl;
        cout<<"the right hand side vector is "<<endl;
        for (i=1;i<=n;i++) cout<<b[i];
        cout<<endl;
        factorise (n,a,d,b);
        forwardsub(n,a,d,b);
        backsub (n,b,c);
 coout<<endl;
 cout<<"the solution vector is "<<endl;
 for (i=1;i<=n;i++) cout<<b[i];
 cout<<endl;
}
void readtridiag(indextype n,vector a,c,d);
{
 for (i=1;i<=n;i++)
  {
          if (i<>1) cin>>d[i];
          cin>>a[i];
          }
}
void readtridiag(indextype n,vector a,c,d);
{
  for (i=1;i<=n;i++)
  {
          for(j=1;j<=i-2;j++) cout<<0.0;
          if (i<>1) cout<<d[i];
          cout<<a[i];
          if (i<>n) cout<<c[i];
          for (j=i+2,j<=n;j++) cout<<0.0;
          cout<<endl;
  }
}
void factorise (indextype n,vector a,c,d);
{  indextype i;
          for (i=2;i<=n;i++)
          {
                    c[i-1]=c[i-1]/a[i-1];
                    a[i] = a[i]-d[i]*c[i-1]
          }
 }
 void forwardsub (indextype n, vector a,d,b);
```

```
{  indextype i;
        b[1]=b[1]/a[1];
        for (i=2;i<=n;i++) b[i]=(b[i] -d[i]*b[i-1])/a[i]
}
void backsub (indextype n,vector b,c);
        { indextype i;
                for (i=n-1;i>=1;i--) b[i] = b[i] - c[i]* b[i+1}}
        }
```

**Sample Data**

```
3
 2.0     2.0
-4.0    -3.0    2.0
         1.0    3.0
 2.0     3.0    11.0
```

**Sample Output**

```
the coefficient matrix is
 2.00000e 0     2.00000e 0     0.00000e 0
-4.00000e 0    -3.00000e 0     2.00000e 0
 0.00000e 0     1.00000e 0     3.00000e 0

the right hand side vector is
 2.00000e 0     3.00000e 0     1.10000e 0

the solution vector is
 2.00000e 0    -1.00000e 0     4.00000e 0
```

# CHAPTER 5

# APPROXIMATION OF CONTINUOUS FUNCTIONS

## 5.1 INTRODUCTION

This Chapter looks at the approximation of a continuous function over finite and infinite intervals. There are often very good reasons for wishing to do this, not least of which is that the function may be difficult, and hence expensive, to evaluate. If it were possible to replace such a function accurately with, say, a low degree polynomial then the situation would be somewhat improved.

There are two distinct ways of approaching approximation. First, the function could be evaluated at a set of points and the problem is then transformed to fitting an approximating function to a discrete set of data values. Second, an overall approximation could be sought and it is this class of problems that concerns us here.

Quite unashamedly most of the worked examples in this chapter are concerned with the approximation of the exponential function $e^x$, usually on the interval [0,1]. This choice is governed by the fact that the exponential function possesses a number of desirable properties which make it an ideal candidate for illustrating the main features of the methods discussed. In particular, the mathematics involved (for example, differentiation) can be kept reasonably simple.

## 5.2 TAYLOR SERIES APPROXIMATION

Perhaps the simplest way of obtaining an approximation to a function $f(x)$ is provided by its Taylor Series expansion. Suppose that we are interested in the behaviour of $f(x)$ over the finite interval $[a,b]$ and let $x_0 \in [a,b]$ be some chosen point. Then Taylor's theorem (Theorem 1.1) gives

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2} f''(x_0) + \dots + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0) + \frac{(x - x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi) \qquad (5.1)$$

where $\xi$ is some interior point of the open interval $]x_0, x[(\textbf{if } x > x_0)$ **or** $]x, x_0[(\textbf{if } x < x_0)$.. If this series is convergent then, provided that $n$ is large enough, the polynomial

$$p_n(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2} f''(x_0) + \ldots + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0) \qquad (5.2)$$

which is of degree at most $n\left(f^{(n)}(x_0) \textbf{ may be zero}\right)$ can be used as an approximation to *f(x).*

EXAMPLE 5.1

Consider the function $f(x) = e^x$ and let [0,1] be the interval of interest. Since each derivative of $e^x$ **is** $e^x$ itself, the Taylor series for this function expanded about the point $x = 0$ gives

$$e^x = e^0 + xe^0 + \frac{x^2}{2}e^0 + \frac{x^3}{6}e^0 + \frac{x^4}{24}e^\xi \quad \xi \in ]0, x[.$$

Hence

$$p_3(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$$

gives a polynomial approximation of degree 3 to e .

Having obtained a polynomial approximation we would like to know how good it is. From equations (5.1) and (5.2) we have that the error $E_n(x)$ in the approximation of $f(x)$ **by** $p_n(x)$ is given by

$$E_n(x) = f(x) - p_n(x) = \frac{(x - x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi) \qquad (5.3)$$

On its own, the error function defined by (5.3) does not provide us with much information since we do not know the value of $\xi$. However, in principle, provided that successive derivatives of *f(x)* do not grow at too fast a rate then, by increasing n, a point will be reached where the denominator in (5.3) will begin to dominate. (But there can be severe computational problems, see Example 5.3). This does mean that we need to be able to compute the appropriate derivative values which is likely to be a tedious operation (although it is possible that the derivatives will satisfy a recurrence relation). In addition algebraic errors can easily creep in when forming derivatives.

In function approximation we often look not for the actual error in approximation, which is usually not computable, but for an upper bound on this quantity. We shall see in Example 5.2 that such bounds are often over-pessimistic and should be treated with some caution. Nevertheless they do allow us to use the underlying numerical methods with some degree of confidence. Here, suppose that we can bound the *n*+1'st derivative of *f(x)* on the interval [a,b], that is, we can determine a value $M_{n+1}$ such that

$$\left|f^{(n+1)}(x)\right| < M_{n+1} \qquad x \in [a,b]$$

Then $|x - x_0| \leq b - a$ so that

$$|E_n(x)| < \frac{(b-a)^{n+1}}{(n+1)!} M_{n+1}$$

provides a bound on the error in the approximation of $f(x)$ **by** $p_n(x)$.

**EXAMPLE 5.2**

For the approximation of $e^x$ by the polynomial $1 + x + (x^2/2) + (x^3/6)$

(Example 5.1) we have

$$E_3(x) = \frac{x^4}{24} e^{\xi}$$

which is everywhere positive in [0,1]. We can therefore draw the qualitative conclusion that the polynomial approximation will be less than $e^x$ for all $x \in [0,1]$. Further,

$$\max_{0 \leq x \leq 1} \left|e^x\right| = e \approx 2.71828$$

and so

$$\left|E_3(x)\right| \leq \frac{1}{24} e \approx 0.11326. \qquad \textbf{(5.4)}$$

In Table 5.1 we list some actual errors which support the above conclusions. Clearly, for small values of $x$ the error bound (5.4) is a gross overestimate of the actual error incurred but it does least guarantee that the error will be no greater than this quantity.

It is possible to use Taylor series approximations in a computer program to evaluate standard functions. We add successive terms until a point is reached at which all subsequent terms are small enough to be ignored. However, care must be taken as the next example shows

**Table 5.1**

| $x$ | $f(x) = e^x$ | $p_3(x) = 1 + x + x^2/2 + x^3/6$ | $E_3(x) = f(x) - p_3(x)$ |
|---|---|---|---|
| 0 | **1.000 00** | **1.000 00** | **0.000 00** |
| $\frac{1}{4}$ | **1.284 03** | **1.283 85** | **0.000 17** |
| $\frac{1}{2}$ | **1.648 72** | **1.645 83** | **0.002 89** |
| $\frac{3}{4}$ | **2.117 00** | **2.101 56** | **0.015 44** |
| 1 | **2.718 28** | **2.666 67** | **0.051 62** |

**EXAMPLE 5.3**

Suppose that we wish to determine an approximation to $e^{-x}$ valid on $[0,10]$. Expanding about the point 0 we have

$$e^{-x} = 1 - x + \frac{x^2}{2} - \frac{x^3}{6} + \frac{x^4}{24} - \ldots \qquad (5.5)$$

which is convergent power series for all $x$. Now

$$e^{-10} = 4.53999 \times 10^{-5}$$

to six significant figures. Let us see what happens if we try to evaluate the Taylor series for $x$=10, accumulating terms until the first one whose absolute value is less than $5 \times 10^{-8}$, say, is encountered. If we use $T_i$ to denote the $i$th term then

$$T_i = \frac{x^i}{i!}(-1)^i \qquad i = 0,1,\ldots \qquad \mathbf{(5.6)}$$

and we note that the recurrence relation

$$T_i = -\frac{x}{i}T_{i-1} \qquad i = 1,2,\ldots \qquad (5.7)$$

holds with $T_0 = 1$. Using (5.7) to compute the individual terms instead of (5.6) will help to keep the operations count down. As soon as $i$>10 the terms start to decrease in value. However, in order to reach such a stage we need to compute terms such as

$$T_3 = -\frac{10^3}{6} = -\frac{1000}{6}$$

and, to six significant figures, $T_3 = -166.667$. The rounding-off error here $0.3 \times 10^{-3}$, is greater than the value we are trying to compute and hence if we work to this accuracy the result obtained will be completely meaningless. (If it were possible to work to greater precision we would find that the very large terms which occur early on in the series cancel each other out.) These difficulties may be overcome by either

(a) writing $e^{-10}$ **as** $\left(e^{-1}\right)^{10}$, evaluation (5.5) for $x = 1$ and raising the result to the tenth power, or

(b) writing $e^{-10}$ **as** $1/e^{10}$, using the Taylor series expansion for $e^x$ with $x$=10 and then forming the reciprocal of the result.

Before leaving this section we observe that with Taylor series approximations we are effectively solving the problem of finding the polynomial of degree at most *n* whose first *n* derivatives agree with those of *f(x)* at the point $x = x_0$ . For example, from (5.2) we have

$$\frac{\mathbf{d}^2}{\mathbf{d}x^2} p_n(x) = f''(x_0)f'''(x_0)+\ldots+\frac{(x-x_0)^{n-2}}{(n-2)!} f^{(n)}(x_0)$$

and hence

$$\frac{\mathbf{d}^2}{\mathbf{d}x^2} p_n(x)|_{x=x_0} = f''(x_0).$$

We shall make use of this fact in section 5.8 when considering certain types of rational approximation.

## 5.3 LEAST SQUARES APPROXIMATION

### 5.3.1 Introduction to best approximation

One of the most important concepts that we are concerned with in this chapter is the idea of a best approximation, that is, an approximating function which is, in some sense, the best of all those in its class. For example, there are infinite number of straight lines which we might choose to take as an approximation to a given function and we need some criterion for deciding which to accept. Let *f(x)* be a continuous function on the finite interval [*a,b*]. Then we wish to determine values of $\alpha_1$ (the intercept along the y-axis) and $\alpha_2$ (the slope) which ensure that

$$p_1(x) = \alpha_1 + \alpha_2 x$$

is a good approximation to *f(x)* everywhere in $[a,b]$. We need, therefore, to formulate a condition for deciding which are the optimal values, $\alpha_1^*$ **and** $\alpha_2^*$, of the parameters $\alpha_1$ **and** $\alpha_2$ that is, we need to define a *measure* such that, with respect to it,

$$p_1^*(x) = \alpha_1^* + \alpha_2^* x \qquad (5.8)$$

is the best possible straight line approximation to *f(x)*.

Consider the *residual function*

$$r(x) = f(x) - p_1(x).$$

Then, as a measure of the error in the approximation of $f(x)$ **by** $p_1(x)$ we could use

$$\underset{a \le x \le b}{\textbf{maximum}} \; |r(x)| \qquad (5.9)$$

that is, the maximum absolute discrepancy between *f(x)* and $p_1(x)$ **in** $[a,b]$.. The best approximation (5.8) is therefore defined to be that which gives the minimum value of (5.9); that is, we aim to minimise (5.9) with respect to the intercept and slope. Clearly, the optimal values $\alpha_1^*$ **and** $\alpha_2^*$ will be such that

$$\underset{a \le x \le b}{\textbf{maximum}} \; |f(x) - \alpha_1^* - \alpha_2^* x| \le \underset{a \le x \le b}{\textbf{maximum}} \; |f(x) - \alpha_1 - \alpha_2 x|$$

for all values of $\alpha_1$ **and** $\alpha_2$. (There is a direct relationship between the measure (5.9)and the $\infty$ - *norm* of a vector (equation (4.24)). We do not pursue the details here but remark that (5.9) is known as the $\infty$ - norm of the residual function.

The measure (5.9) is not the only one that can be used and an alternative is

$$\int_a^b (r(x))^2 \, \mathbf{d}x \qquad (5.10)$$

The optimal values of $\alpha_1$ **and** $\alpha_2$ which minimise (5.10) define the *least squares* straight line approximation to *f(x)*. Throughout this and the next two sections it is (5.10) which provides the measure on which our numerical methods are based. The use of (5.9) and other measures is discussed in section 5.7.

### 5.3.2 Least Squares

In the previous subsection we considered the approximation of a continuous function *f(x)* on a finite interval $[a,b]$ by a straight line. In the general case we seek values of the coefficients $\alpha_1, \alpha_2, \ldots, \alpha_n$ which ensure that

$$\begin{aligned} p_{n-1}(x) &= \alpha_1 + \alpha_2 x + \ldots + \alpha_n x^{n-1} \\ &= \sum_{j=1}^{n} \alpha_j x^{j-1} \end{aligned}$$

is the best polynomial approximation of degree $n$ - 1. Using the least squares criterion this means that we aim to find values $\alpha_1^*, \alpha_2^*, \ldots, \alpha_n^*$, which give

$$\underset{\alpha_1, \alpha_2, \ldots, \alpha_n}{\textbf{minimum}} \int_a^b \left(f(x) - p_{n-1}(x)\right)^2 \mathbf{d}x.$$

Let $\alpha^{\mathrm{T}} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and define

$$I(\alpha) = \int_a^b \left(f(x) - p_{n-1}(x)\right)^2 \mathbf{d}x. \qquad (5.11)$$

Then we wish to minimise $I(\alpha)$ with respect to the parameters $\alpha_1, \alpha_2, \ldots, \alpha_n$. We recall that for a function of one real variable, at a stationary point (that is, a maximum or minimum) the tangent (that is, the derivative) is equal to zero. For a function of more than one real variable the situation is similar except that now we have to talk in terms of partial, instead of full, derivatives. The necessary conditions are

$$\frac{\partial}{\partial \alpha_i} I(\alpha) = 0 \qquad i = 1,2,\ldots,n. \qquad (5.12)$$

The solution $\alpha^*$ of (5.12) gives a turning point of $I(\alpha)$ and it is fairly easy to show (we need to look at second derivatives also) that we actually get a minimum. For simplicity, from here on, we drop the asterisk attached to the best approximation.

In order to see what the equations defining the least squares polynomial approximation looks like, we return to the approximation of *f(x)* by a straight line. Here $n = 2$ and we need to find $\alpha_1$ **and** $\alpha_2$ such that

$$\frac{\partial}{\partial \alpha_1} \int_a^b \left(f(x) - (\alpha_1 + \alpha_2 x)\right)^2 \mathbf{d}x = 0$$
$$\frac{\partial}{\partial \alpha_2} \int_a^b \left(f(x) - (\alpha_1 + \alpha_2 x)\right)^2 \mathbf{d}x = 0$$

that is,

$$-2\int_a^b \left(f(x) - (\alpha_1 + \alpha_2 x)\right) \mathbf{d}x = 0$$

$$-2\int_a^b \left(f(x) - (\alpha_1 + \alpha_2 x)\right) x \, \mathbf{d}x = 0$$

Rearranging these equations and dividing by 2 we obtain

$$\alpha_1 \int_a^b 1 \, \mathbf{d}x + \alpha_2 \int_a^b x \, \mathbf{d}x = \int_a^b f(x) \, \mathbf{d}x$$

$$\alpha_1 \int_a^b 1 \, \mathbf{d}x + \alpha_2 \int_a^b x^2 \, \mathbf{d}x = \int_a^b x f(x) \, \mathbf{d}x$$

which is a system of two linear simultaneous equations in the two unknowns $\alpha_1$ **and** $\alpha_2$

In the general case we need to solve the system of equations

$$0 = \frac{\partial}{\partial \alpha_1} I(\alpha)$$

$$= -2\int_a^b \left[ f(x) - \sum_{j=1}^{n} \alpha_j x^{j-1} \right] x^{i-1} \, \mathbf{d}x \qquad i = 1,2,\dots,n$$

and so, rearranging, we get

$$\sum_{j=1}^{n} \alpha_j \int_a^b x^{i-1} x^{j-1} \, \mathbf{d}x = \int_a^b x^{i-1} f(x) \, \mathbf{d}x \qquad i = 1,2,\dots,n \qquad (5.13)$$

or, in matrix form,

$$\text{Aa=b} \qquad (5.14)$$

where

$$b_i = \int_a^b x^{i-1} f(x) \, \mathbf{d}x \qquad i = 1,2,\dots,n \qquad (5.15)$$

$$A_{ij} = \int_a^b x^{i-1} x^{j-1} \, \mathbf{d}x \qquad i,j = 1,2,\dots,n. \qquad (5.16)$$

Usually it will be possible to evaluate the integrals (5.15) using integration by parts. The entries (5.16) in the coefficient matrix can be easily found since

$$A_{ij} = \left[\frac{x^{i+j-1}}{i+j-1}\right]_a^b = \frac{b^{i+j-1} - a^{i+j-1}}{i+j-1} \qquad i,j = 1,2,\ldots,n. \qquad (5.17)$$

The equations (5.13) are known as the normal equations. Having set up the matrix equation the solution to the least squares approximation problem may be obtained using, for example, Gauss elimination (although, as we shall see shortly, the situation is not quite as straightforward as it appears at first sight). We observe that the coefficient matrix A is symmetric and this property should be borne in mind if a factorisation method is used (see section 5.6).

**EXAMPLE 5.4**

In this example we consider the approximation of the function $e^x$ on the interval [0,1] by a quadratic using the least squares criterion. To find the optimal values of the coefficients in

$$p_2(x) = \alpha_1 + \alpha_3 x + \alpha_3 x^2$$

we have to set up a system of three equations in three unknowns in which the coefficient matrix is, from (5.17), given by

$$A_{ij} = \frac{1}{i+j-1} \qquad i,j = 1,2,3 \qquad (5.18)$$

(since $a = 0$ **and** $b = 1$). To obtain the elements of the right-hand side vector we note that

$$b_i = \int_0^1 e^x \mathbf{d}x = e - 1 \qquad (5.19)$$

and

$$b_2 = \int_0^1 xe^x \mathbf{d}x = \left[xe^x\right]_0^1 - \int_0^1 e^x \mathbf{d}x = e - b_1 = 1 \qquad (5.20)$$

$$b_3 = \int_0^1 x^2 e^x \mathbf{d}x = \left[x^2 e^x\right]_0^1 - 2\int_0^1 xe^x \mathbf{d}x = e - 2b_2 = e - 2 \quad (5.21)$$

Hence, the system of normal equations for the unknowns $\alpha_1, \alpha_2$ **and** $\alpha_3$ takes the form

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} e-1 \\ 1 \\ e-2 \end{bmatrix}$$

Eliminating $\alpha_1$ from the second and third equations gives

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 0 & 1/12 & 1/12 \\ 0 & 1/12 & 4/45 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} e-1 \\ \frac{1}{2}(-e+3) \\ \frac{1}{3}(2e-5) \end{bmatrix}$$

Finally, eliminating $\alpha_2$ from the third equation, we have

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 0 & 1/12 & 1/12 \\ 0 & 0 & 1/180 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} e-1 \\ \frac{1}{2}(-e+3) \\ \frac{1}{6}(7e-19) \end{bmatrix}$$

Now, using back-substitution, we obtain

$$\begin{aligned} \alpha_3 &= \frac{180}{6}(7e-19) = 30(7e-19) \approx 0.83918 \\ \alpha_2 &= 12\left(\tfrac{1}{2}(-e+3) - \tfrac{1}{12}\alpha_3\right) = -12(18e-49) \approx 0.85113 \\ \alpha_1 &= e - 1 - \tfrac{1}{2}\alpha_2 - \tfrac{1}{3}\alpha_3 = 3(13e-35) \approx 1.01299 \end{aligned}$$

so that, to five decimal place accuracy, the least squares quadratic approximation to $e^x$ on [0,1] is

$$p_2(x) \approx 1.01299 + 0.85113x + 0.83918x^2.$$

### 5.3.3 A program for Least Squares Approximation

```
//a program for the least squares approximation of exp(x) on [0,1] by a polynomial
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
#define MAXN 50;
#define M 5;
class rhsindextype
{public:
int rhsindextype;
};
class matrixindextype
{public:
int matrixindextype;
};
class rhstype
{public:
float rhstype[50][5];
};
class matrixtype
{public:
float matrixtype[50][50];
};

class rownumtype
{public:
matrixtype rownumtype[50];
};
main()
{matrixindextype i,j,n;
rhsindextype p;
```

```
matrixtype a,savea;
rhstype alpha, b;
rownumtype rownum;
float e, residual;
void gausselim (matrixindextype n, matrixtype a, rhsindextype m,rhstype b,rownumtype rownum);
void backsub (matrixindextype n, matrixtype a, rhsindextype m, rhstype x,b, rownumtype rownum);
cin>>n;
b[1][1]=1.7;
b[1][2]=1.72;
b[1][3]=1.718;
b[1][4]=1.7183;
b[1][5]=1.71828;
e=exp(1.0);
for (i=2;i<=n;i++)
        for (p=1;p<=m;p++)
        b[i][p]=e-(i-1.0)*b[i-1][p];
        for (i=1;i<=n;i++)
        {
        a[i][i]=1.0/(2*i-1.0);
        for (j=i+1;j<=n;j++)
        {
        a[i][j]=1.0 /(i+j-1.0);
        a[j][i]=a[i][j]
        }
        }
        for (i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                savea[i][j]=a[i][j];
                saveb=b;
                gausselim (n,a,m,b,rownum);
                backsub (n,a,m,alpha,b,rownum);
                for (p=1;p<=m;p++);
                {
                        cout<<endl;
```

```
                    cout<<"using "<<"decimal place accuracy "<<endl;
                    cout<<"the right handside vector is"<<endl;
                    for (i=1;i<=n;i++) cout<< saveb[i][p]<<endl<<endl;
                    cout<<"and the solution vector is "<<endl;
                    for (i=1;i<=n;i++) cout<<alpha[i][p]<<endl<<endl;
                    cout<<"the residual is "<<endl;
                    for (i=1;i<=n;i++)
                    {
                         residual = saveb[i][p];
                         for(j=1;j<=n;j++)
                              residual = residual - savea[i][j]*alpha[j][p];
                         cout<<residual;
                         }
                         cout<<endl;
                         cout<<endl;
                    }
               }

void gausselim (matrixindextype n, matrixtype a, rhsindextype m,rhstype b,rownumtype rownum);
        {

        }

        void backsub (matrixindextype n, matrixtype a, rhsindextype m, rhstype x,b, rownumtype rownum);
         {

         }
```

**Sample Data**

```
3
```

**Sample Output**

```
using    1        decimal place accuracy
the right hand side vector is
```

```
            1.70000e 0       1.01828e 0       6.81718e -1
and the solution vector is
            -9.06606e -1     1.16009e 1       -9.58148e 0
the residual is
            1.19209e -6      7.15256e -7      5.96046e -7

using    2         decimal place accuracy
the right hand side vector is
            1.72000e 0       9.98282e -1      7.21718e -1
and the solution vector is
            1.19341e 0       -1.59217e -1     1.81860e 0
the residual is
            4.76837e -7      2.38419e -7      2.38419e -7

using    3         decimal place accuracy
the right hand side vector is
            1.71800e 0       1.00028e 0       7.17718e -1
and the solution vector is
            9.83398e -1      1.01685e 0       6.78536e -1
the residual is
            2.38419e -7      2.38419e -7      2.38419e -7

using    4         decimal place accuracy
the right hand side vector is
            1.71830e 0       9.99981e -1      7.18318e -1
and the solution vector is
            1.01492e 0       8.40348e -1      8.49630e -1
the residual is
            4.76837e -7      2.38419e -7      2.38419 -7

using    5         decimal place accuracy
the right hand side vector is
            1.71828e 0       1.00000e 0       7.18278e -1
and the solution vector is
```

```
        1.01281e 0     8.52125e -1    8.38215e -1
the residual is
        4.76837e -7    4.76837e -7    3.57628e -7
```

### 5.3.4 Ill-Conditioning of the Normal Equations

We now look at Example 5.4 and the program of the previous subsection in more detail. For polynomial approximation on [0,1] the general form of the coefficient matrix is given by (5.22). This matrix is the *n*th principal minor of the infinite Hilbert matrix and it is well know that an equation of the form (5.14) involving this matrix is likely to be very ill-conditioned.

If we look at the sample output of program 5.3.3 we see that if the accuracy to which $b_1$ is computed is decreased by just one decimal place the solution vector changes by a much greater factor. The residual vector indicates that the elimination and back-substitution processes have worked quite well; the observed behaviour of the solution vector is due to the problem itself, not the method of solution. Investigating the matter further, the program of 5.3.3 was executed for values of $n = 3,4,...,7$ and Table 5.2 lists the results for the case in which b is rounded to five decimal places.

n=3 or n=4

For $n$ equal to three and four the polynomial coefficients start to look something like the first few terms in the Taylor series expansion of $e^x$ about the point $x$=0 (see Example 5.1). However, from n=5 onwards there is no consistency at all in the results and, in fact, they appear to be getting significantly worse. What we are in effect saying is that the ill-conditioning property of the problem becomes more pronounced as n increases.

Table 5.2

| | $n$ | | | | |
|---|---|---|---|---|---|
| | 3 | 4 | 5 | 6 | 7 |
| $\alpha_1$ | **1.012 79** | **0.996 249** | **0.949 770** | **- 0.188 766** | **27.732 4** |
| $\alpha_2$ | **0.852 254** | **1.050 74** | **1.980 15** | **36.000 4** | **- 1009.63** |
| $\alpha_3$ | **0.838 089** | **0.341 889** | **- 3.840 13** | **- 241.345** | **9354.44** |
| $\alpha_4$ | | **0.330 798** | **6.835 84** | **638.958** | **- 35252.9** |
| $\alpha_5$ | | | **- 3.252 41** | **- 713.348** | **63042.8** |
| $\alpha_6$ | | | | **283.712** | **- 53376.4** |
| $\alpha_7$ | | | | | **17229.8** |

To overcome these difficulties we must reformulate the problem and we now consider ways of doing this.

## 5.4 USE OF ORTHOGONAL POLYNOMIALS

### 5.4.1 Legendre Polynomials

In the previous section we saw that polynomial approximation of a function on the finite interval $[a,b]$ leads to the derivation of a system of equations which may well be ill-conditioned. We now look at ways of getting round this difficulty and observe first of all that the situation would be considerably healthier if the coefficient matrix were not full but diagonal. The solution vector could then be read off directly. Now only would we avoid the pitfalls of ill-conditioning but we would also reduce considerably the operations count involved in solving the system of equations. We shall see shortly that such a situation can be readily achieved by employing a fairly simple transformation of the problem.

In order to proceed further we introduce a sequence of polynomials, $P_n(x): n = 0,1,\ldots,$ known as the *Legendre* polynomials. These polynomials can be defined via the *three term recurrence relation*

$$P_n(x) = \frac{2n-1}{n} x\, P_{n-1}(x) - \frac{n-1}{n} P_{n-2}(x) \qquad n = 2,3,\ldots \quad (5.24)$$

with the starting values

$$P_0(x) = 1; \;\; P_1(x) = x \qquad\qquad (5.25)$$

and it is fairly easy to see that $P_n(x)$ is a polynomial of degree *n*. Legendre polynomials possess a property which is very important in the present context, namely they are *orthogonal* on the interval [-1,1) of the product of two Legendre polynomials is zero unless they happen to be the same. Formally, we have

$$\int_{-1}^{1} P_n(x)\, P_m(x)\, \mathbf{d}x = 0 \qquad n \neq m$$

and also that

$$\int_{-1}^{1} \left(P_n(x)\right)^2 \; \mathbf{d}x = \frac{2}{2n+1} \qquad n = 0,1,\ldots \qquad (5.26)$$

This definition of the orthogonality of functions should be compared with the corresponding definition for vectors. Recall that two vectors are orthogonal if their scalar product is zero. Here, the scalar, or inner, product is the integral of the product of two polynomials and we write

$$\left(P_n(x), P_m(x)\right) = \int_{-1}^{1} P_n(x)\, P_m(x)\, \mathbf{d}x \qquad n,m = 0,1,\ldots \;.$$

**EXAMPLE 5.5**

From (5.25) we have that

$$\int_{-1}^{1} P_0(x)\, P_1(x)\, \mathbf{d}x = \int_{-1}^{1} x\, \mathbf{d}x = \left[\frac{x^2}{2}\right]_{-1}^{1} = \tfrac{1}{2} - \tfrac{1}{2} = 0$$

and so $P_0(x)$ **and** $P_1(x)$ are orthogonal. From (5.24)

$$P_2(x) = \frac{3}{2}x.x - \frac{1}{2} = \frac{1}{2}\left(3x^2 - 1\right).$$

**Now**

$$\int_{-1}^{1} P_0(x)\, P_2(x)\, \mathbf{d}x = \tfrac{1}{2} \int_{-1}^{1} \left(3x^2 - 1\right)\mathbf{d}x = \tfrac{1}{2}\left[x^3 - x\right]_{-1}^{1} = \mathbf{0}$$

and

$$P_1(x)\, P_2(x)\, \mathbf{d}x = \tfrac{1}{2} \int_{-1}^{1} \left(3x^3 - 1\right)\mathbf{d}x = \tfrac{1}{2}\left[\frac{3x^4}{4} - \frac{x^2}{2}\right]_{-1}^{1} = \mathbf{0}$$

and hence $P_2(x)$ is orthogonal to both $P_0(x)$ **and** $P_1(x)$.

Now, suppose that we wish to find the least squares approximation of the form

$$p_2(x) = \alpha_1 + \alpha_2 x + \alpha_3 x^2$$

to the continuous function f(x) on the interval [-1,1]. Then rearranging (5.27) we have

$$x^2 = \frac{1}{3}\left(2p_2(x) + 1\right) = \frac{1}{3}\left(2p_2(x) + p_0(x)\right)$$

so that

$$\begin{aligned} p_2(x) &= \alpha_1 p_0(x) + \alpha_2 p_1(x) + \alpha_3 \tfrac{1}{3}\left(2p_2(x) + p_0(x)\right) \\ &= \left(\alpha_1 + \tfrac{1}{3}\alpha_3\right)p_0(x) + \alpha_2 p_1(x) + \tfrac{2}{3}\alpha_3 p_2(x) \qquad (5.28) \\ &= \beta_1 p_0(x) + \beta_2 p_1(x) + \beta_3 p_2(x) \end{aligned}$$

where

$$\beta_1 = \alpha_1 + \tfrac{1}{3}\alpha_3, \;\; \beta_2 = \alpha_2, \;\; \beta_3 = \tfrac{2}{3}\alpha_3.$$

We have, therefore, transformed $p_2(x)$ from a linear combination of the monomials $\{1, x, x^2\}$ to a linear combination of the first three Legendre polynomials.

From the material of section 5.3, the least squares quadratic approximation of the form (5.28) to the function $f(x)$ on the interval [-1,1] is given as the solution to a minimisation problem. We need to find the values of $\beta_1, \beta_2$, **and** $\beta_3$ which minimise the quantity.

$$\int_{-1}^{1}\left(f(x)-\sum_{j=1}^{3}\beta_j p_{j-1}(x)\right)^2 \mathbf{d}x.$$

Proceeding in exactly the same way as before we end up with the matrix equation

$$A\beta = \mathbf{b} \qquad (5.29)$$

where

$$A_{ij} = \int_{-1}^{1} P_{i-1}(x)P_{j-1}(x)\,\mathbf{d}x = \left(P_{i-1}(x), P_{j-1}(x)\right) \qquad i,j = 1,2,3$$

$$b_i = \int_{-1}^{1} f(x)P_{i-1}(x)\,\mathbf{d}x = \left(f(x), P_{i-1}(x)\right) \qquad i = 1,2,3$$

But, from the orthogonality property of the Legendre polynomials, we have that the coefficient matrix has non-zero entries only on the diagonal so that, using the result (5.26),

$$\beta_i = \frac{\left(f(x), P_{i-1}(x)\right)}{\left(P_{i-1}(x), P_{i-1}(x)\right)} = \frac{2i-1}{2}\left(f(x), P_{i-1}(x)\right) \qquad i = 1,2,3.$$

The solution to our last squares approximation problem is now in terms of $P_0(x), P_1(x)$ **and** $P_2(x)$.. In order to find out what the solution looks like as a straightforward quadratic we need to solve the system

$$\begin{aligned}\alpha_1 \qquad\qquad &+\frac{1}{3}\alpha_3 = \beta_1\\ \alpha_2 \qquad &\qquad\;\; = \beta_2\\ &\frac{2}{3}\alpha_3 = \beta_3\end{aligned}$$

which gives

$$\alpha_3 = \frac{3}{2}\beta_3, \alpha_2 = \beta_2, \alpha_1 = \beta_1 - \frac{1}{2}\beta_3$$

**EXAMPLE 5.6**

Suppose that we wish to find the least squares quadratic approximation to $e^x$ on [-1,1]. Then the polynomial coefficients in $p_2(x) = \beta_1 P_0(x) + \beta_2 P_1(x) + \beta_3 P_2(x)$ are given by

$$\beta_1 = \tfrac{1}{2}\int_{-1}^{1} e^x P_0(x)\,\mathbf{d}x = \tfrac{1}{2}\int_{-1}^{1} e^x dx = \tfrac{1}{2}\left(e - e^{-1}\right)$$
$$\beta_2 = \tfrac{3}{2}\int_{-1}^{1} e^x P_1(x)\,\mathbf{d}x = \tfrac{3}{2}\int_{-1}^{1} e^x x\,\mathbf{d}x = 3e^{-1}$$
$$\beta_3 = \tfrac{5}{2}\int_{-1}^{1} e^x P_2(x)\,\mathbf{d}x = \tfrac{5}{2}\int_{-1}^{1} e^x \tfrac{1}{2}\left(3x^2 - 1\right)dx = \tfrac{5}{2}\left(e - 7e^{-1}\right)$$

using integration by parts. Hence the quadratic approximation is

$$p_2(x) = \tfrac{1}{2}\left(e - e^{-1}\right) + 3e^{-1}x + \tfrac{5}{2}\left(e - 7e^{-1}\right)\tfrac{1}{2}\left(3x^2 - 1\right).$$

The coefficients in $p_2(x) = \alpha_1 + \alpha_2 x + \alpha_3 x^2$ are given by

$$\alpha_3 = \tfrac{3}{2}\beta_3 = \tfrac{15}{4}$$
$$\alpha_2 = \beta_2 = 3e$$
$$\alpha_1 = \beta_2 - \tfrac{1}{3}\beta_3 = \tfrac{1}{2}\left(e - e^{-1}\right) - \tfrac{5}{4}\left(e - 7e^{-1}\right) = \tfrac{1}{4}\left(-3e + 33e^{-1}\right)$$

so that

$$p_2(x) = \tfrac{1}{4}\left(-3e + 33e^{-1}\right) + 3e^{-1}x + \tfrac{15}{4}\left(e - 7e^{-1}\right)x^2.$$

The generalisation of the above is obvious and it is clearly advantageous to be able to express an approximating polynomial in terms of orthogonal polynomials. We no longer need to compute all of the entries in the coefficient matrix but just those on the diagonal. This has important consequences for the storage requirements of a least squares approximation program; the two-dimensional $n \mathbf{X} n$ array of coefficients is no longer required. Although in Example 5.6 we converted the polynomial to standard form this is not really necessary and in subsection 5.4.3 we give code for evaluating a polynomial expressed as a linear combination of Legendre polynomials at a point.

Unless we are very fortunate the domain of interest is unlikely to be [-1,1] and we need to take account of this. One simple way is to map the given range $[a,b]$, say, onto [-1,1] using the mapping

$$z = \frac{2x - a - b}{b - a}$$

Alternatively we could use polynomials which are orthogonal on the interval $[a,b]$. This does nto mean that we need to know a large set of orthogonal polynomials; it is possible to generate from the monomials a sequence of polynomials which are orthogonal on a given interval using a technique known as the *Gram-Schmidt orthonormalisation process.*

### 5.4.2 Polynomial Orthonormalisation

Suppose that we wish to generate the sequence of polynomials $\left\{Q_i(x) : i = 0,1,2\right\}$ with $Q_i(x)$ of degree $i$ such that the $Q_i(x)s$ are orthogonal on $[a,b]$. Define

$$\begin{aligned} Q_0(x) &= c_{00} \\ Q_1(x) &= c_{10} + c_{11}x \\ Q_2(x) &= c_{20} + c_{21}x + c_{22}x^2 \end{aligned}$$

Now, orthogonality implies that

$$\begin{aligned} \left(Q_0(x), Q_1(x)\right) &= \int_a^b Q_0(x)Q_1(x)\,\mathbf{d}x = 0 \\ \left(Q_0(x), Q_2(x)\right) &= \int_a^b Q_0(x)Q_2(x)\,\mathbf{d}x = 0 \\ \left(Q_1(x), Q_2(x)\right) &= \int_a^b Q_1(x)Q_2(x)\,\mathbf{d}x = 0 \end{aligned}$$

and this gives a system of three equations for the six unknowns $\left\{c_{ij}\right\}$. For the time being we choose $c_{00} = c_{10} = c_{20} = 1$ arbitrarily and then the system (5.30) defines $c_{11}, c_{21}$ **and** $c_{22}$ uniquely.

**EXAMPLE 5.7**

To find a sequence of three polynomials which are orthogonal on $[0,1]$ we have

$$0=\left(Q_0(x),Q_1(x)\right)=\int_0^1\left(1+c_{11}x\right)\,\mathbf{d}x=\left[x+c_{11}\tfrac{x^2}{2}\right]_0^1=1+\tfrac{c_{11}}{2}$$

so that $c_{11}=-2$ . Then

$$0=\left(Q_0(x),Q_2(x)\right)=\int_0^1\left[1+c_{21}x+c_{22}x^2\right]\,\mathbf{d}x$$

$$=\left[x+c_{21}\frac{x^2}{2}+c_{22}\frac{x^3}{3}\right]_0^1=1+\frac{c_{21}}{2}+\frac{c_{22}}{3}$$

and

$$0=\left(Q_0(x),Q_2(x)\right)=\int_0^1(1-2x)\left[1+c_{21}x+c_{22}x^2\right]\,\mathbf{d}x$$

$$=\int_0^1\left(1+\left(c_{21}-2\right)x+\left(c_{22}-2c_{21}\right)x^2-2c_{22}x^3\right)\,\mathbf{d}x$$

$$=\left[x+\left(c_{21}-2\right)\frac{x^2}{2}+\left(c_{22}-2c_{21}\right)\frac{x^3}{3}-c_{22}\frac{x^4}{2}\right]_0^1$$

$$=1+\frac{1}{2}\left(c_{21}-2\right)+\frac{1}{3}\left(c_{22}-2c_{21}\right)-\frac{1}{2}c_{22}$$

$$=-\frac{1}{6}c_{21}-\frac{1}{6}c_{22}\,.$$

Hence $c_{21}=-c_{22}$ and therefore, from (5.31) $c_{22}=6$ **and** $c_{21}=-6$ . We conclude that the polynomials are

$$Q_0(x) = 1$$
$$Q_1(x) = 1 - 2x$$
$$Q_2(x) = 1 - 6x + 6x^2 .$$

Having generated a sequence of polynomials which are orthogonal on a given interval the solution to the corresponding least squares approximation problem may be obtained using a straightforward extension of the material of the previous subsection. All that we need to do is to replace the Legendre polynomials with the orthogonal polynomials we have just generated and integrate over $[a,b]$ instead of [-1,1]. In the matrix equation (5.29) which defines the coefficients in the polynomial $p_2(x) = \beta_1 + \beta_2 x + \beta_3 x^2$ the matrix $A$ has components

$$A_{ij} = \left(Q_{i-1}(x), Q_{j-1}(x)\right) = \int_a^b Q_{i-1}(x) Q_{j-1}(x)\,\mathbf{d}x \qquad i,j = 1,2,3$$

and is diagonal. The entries in the right-hand side vector are given by

$$b_i = \left(f(x), Q_{i-1}(x)\right) = \int_a^b f(x) Q_{i-1}(x)\,\mathbf{d}x \qquad i = 1,2,3$$

and hence the solution vector may be found directly since its components are

$$\beta_i = \frac{\left(f(x),\ Q_{i-1}(x)\right)}{\left(Q_{i-1}(x),\ Q_{i-1}(x)\right)} \qquad i = 1,2,3. \qquad (4.32)$$

Note that in the last three equations the definition of the inner product has been modified to take account of the fact that the domain of interst is now the general interval [a,b].

**EXAMPLE 5.8**

From Example 5.7 we know that the polynomials $Q_0(x) = 1, Q_1(x) = 1 - 2x$ **and** $Q_2(x) = 1 - 6x + 6x^2$ are orthogonal on [0,1]. Therefore the coefficients in the least squares polynomial approximation of the form $p_2(x) = \beta_1 Q_0(x) + \beta_2 Q_1(x) + \beta_3 Q_2(x)$ **to** $e^x$ on [0,1] are given by

$$\beta_1 = \int_0^1 e^x dx \, / \int_0^1 dx = e - 1$$
$$\beta_2 = \int_0^1 e^x (1-2x) dx \, / \int_0^1 (1-2x)^2 dx = 3(e-3)$$
$$\beta_3 = \int_0^1 e^x (1-6x+6x^2) dx \, / \int_0^1 (1-6x+6x^2)^2 dx = 5(7e-19).$$
$$Q_0(x), Q_1(x), \ldots, Q_{n-2}(x)$$

Hence

$$p_2(x) = e - 1 + 3(e-3)(1-2x) + 5(7e-19)(1-6x+6x^2).$$

This solution should now be compared with that of Example 5.4.

The generation of higher order polynomials which are orthogonal on $[a,b]$ follows in an obvious manner. Suppose that we have already obtained the polynomials $Q_0(x), Q_1(x), \ldots, Q_{n-2}(x)$ which are such that

$$(Q_i(x), Q_j(x)) = 0 \qquad i,j = 0,1,\ldots,n-2.$$

Then to ensure that $Q_{n-1}(x)$ is orthogonal to we let

$$Q_{n-1}(x) = c_{n-1,0} + c_{n-1,1}x + \ldots + c_{n-1,n-1}x^{n-1}$$

with $c_{n-1.0} = 1$ and find values for $c_{n-1,1}, c_{n-1,2}, \ldots, c_{n-1,n-1}$ such that

$$(Q_i(x), Q_{n-1}(x)) = 0 \qquad i = 0,1,\ldots,n-2.$$

Equation (5.32) is now valid for all values of i and hence the last squares polynomial approximation to the function *f(x)* on [a,b] may be found immediately.

At the start of this subsection we chose to set the first term of each orthogonal polynomial to be one. We could set the coefficient of $x^i$ **in** $Q_i(x)$ to be one instead and then find the remaining terms. Other variations of the basic method are of course possible. An alternative approach however is to impose the additional constraints

$$\left(Q_i(x), Q_i(x)\right) = 1 \qquad i = 0,1,\ldots \,. \qquad \mathbf{(5.33)}$$

The original orthogonality conditions plus the normalising conditions (5.33) give a system of equations which uniquely defines the coefficients of the $Q_i(x)$**s.** The polynomials so generated are now said to be orthonormal on the interval [a,b].

**EXAMPLE 5.9**

Using (5.30) and (5.33) it can be shown that the polynomials

$$Q_0(x) = 1$$
$$Q_1(x) = \sqrt{3} - 2\sqrt{3x}$$
$$Q_2(x) = \sqrt{5} - 6\sqrt{5x} + 6\sqrt{5x^2}$$

are orthonormal on $[0,1]$. We note that these polynomials are the same as those derived in Example 5.7 apart from the presence of a multiplying factor in $Q_1(x)$ **and** $Q_2(x)$. Indeed a straightforward way of generating orthonormal polynomials is to generate a sequence of orthogonal polynomials $\{Q_i(x)\}$ and then divide $Q_i(x)$ by the normalising factor $\left(Q_i(x), Q_i(x)\right)^{1/2}$. One can now check that the polynomial $Q_1(x)$ **and** $Q_2(x)$ of Example 5.7 satisfy $\left(Q_1(x), Q_1(x)\right) = \frac{1}{3}$ **and** $\left(Q_2(x), Q_2(x)\right) = \frac{1}{5}$ (these values were used in Example 5.8).

If we express a least squares approximation polynomial in terms of orthonormal polynomials the matrix in the equation defining the coefficients is now the identity matrix and so the solution vector is just the right hand side vector.

### 5.4.3 A program for Polynomial evaluation

```
//a program to evaluate a polynomial expressed as a linear combination of legendre polynomial at a point
//using the three term recurrence relation
```

```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
#define MAXN 50;

class indextype
{public:
        int indextype;
        };


        class coeffsvector
        {public:
                float coeffsvector[50];
                };
 float evaluate(float x, indextype n,coeffsvector coeffs)
 {
        float pn,pnminus1,pnminus2,sum;
        indextype i;


        pn=x;
        pnminus1=1.0;
        sum = coeffs[0] + coeffs[1]*x;
        for (i=2;i<=n;i++)
        {
                pnminus2 = pnminus1;
                pnminus1= pn;
                pn = ((2.0*i-1.0)*x*pnminus1-(i-1.0)*pnminus2)/i;
                sum = sum + coeffs[i]*pn
        }
        evaluate = sum
        return evaluate;
}
main()
{
        indextype i,n;
        coeffsvector beta;
        float x;
        cin>>n;
        for (i=0;i<=n;i++) cin>> beta[i];
        cin>>x;
        cout<<endl;
        cout<<"the coefficients of the legendre polynomials are "<<"(constant term first "<<endl;
        cout<<endl;
        for(i=0;i<=n;i++) cout<<beta[i];
        cout<<endl;
        cout<<"and when evaluated at the point"<<x<<"this give"<<evaluate(n,beta,x)<<endl;
}
```

**Sample Data**

```
4
0.25    0.75    -0.25   -0.75   0.50
0.5
```

**Sample Output**

```
the coefficients of the legendre polynomials are ( constant term first )

 2.50000e -1
 7.50000e -1
-2.50000e -1
-7.50000e -1
 5.00000e -1

and when evaluated at the point 5.00000e -1     this gives         8.39844e -1
```

## 5.5 GENERALISED LEAST SQUARES METHODS

### 5.5.1 Weighted Least Squares

Whilst in practice an approximating function is usually a polynomial there is no reason at all why we should be restricted in this way. In certain circumstances it may prove profitable to use a linear combination of functions other than monomials (or, equivalently, orthogonal polynomials). Just what form these other functions should take we return to later and for the moment we consider in an abstract sense a more general approach to least squares approximation.

Let $\left\{\phi_i(x):i = 1,2,\ldots,n\right\}$ be a set of known functions which we refer to as expansion functions (or basis functions). Let

$$L(\alpha,x)=\sum_{j=1}^{n}\alpha_j\phi_j(x) \qquad (5.34)$$

be a linear approximating function. By linear we mean that the parameters $\alpha_j : j = 1,2,\ldots,n$ (the expansion coefficients) appear linearly in (5.34). Polynomial approximation is therefore a special case of the more general form being considered here and corresponds to the choice

$$\phi_j(x) = x^{j-1} \qquad j = 1,2,\ldots,n$$

if the polynomial is expressed in the 'normal' form.

In section 5.4 we saw that if the domain of interest is [-1m1] the choice

$$\phi_j(x) = P_{j-1}x \qquad j = 1,2,\ldots,n$$

(where $P_{j-1}(x)$ is a Legendre polynomial) leads to the polynomial coefficients being found as the solution to a system of equations in which the coefficient matrix is diagonal. The aim here is to achieve a similar situation for the more general case. For the moment the only condition that we impose on the basis functions is that they be *linearly independent*, that is, it must not be possible to express one of the basis functions as a linear combination of the others. (Without this restriction the coefficient matrix will be singular).

The least squares approximation of the form (5.34) to the function *f(x)* on the finite interval [a,b] minimises with respect to the $\alpha_j s$ the quantity

$$\int_a^b\left(f(x)-L(\alpha,x)\right)^2 dx. \qquad (5.35)$$

This basic definition may be modified by introducing a weighting function w(x)>0 so that (5.35) becomes

$$\int_a^b (f(x) - L(\alpha,x))^2 w(x)dx.$$

This weight function allows us to have some control over the approximating function; it can be used to emphasise (or reduce) the square error $(f(x) - L(\alpha,x))^2$ over certain sections of [a,b]. In addition it allows us to use polynomials which are orthogonal on [-1,1] other than the Legendre polynomials (see subsection 5.5.2).

Corresponding to (5.11) we introduce the function

$$I(\alpha) = \int_a^b (f(x) - L(\alpha,x))^2 w(x)dx$$

which we wish to minimise with respect to $\alpha_1, \alpha_2, \ldots, \alpha_n$. As before, at a minimum

$$0 = \frac{\partial I(\alpha)}{\partial \alpha_i}$$

$$= -2\int_a^b \left( f(x) - \sum_{j=1}^{n} \alpha_j \phi_j(x) \right) \phi_i(x) w(x) \, \mathbf{d}x \qquad i = 1,2,\ldots,n$$

or, rearranging,

$$\sum_{j=1}^{n} \alpha_j \int_a^b \phi_i(x)\phi_j(x) w(x) \, \mathbf{d}x = \int_a^b f(x)\phi_i(x) w(x) dx \qquad \mathrm{i} = 1,2,\ldots,\mathrm{n}$$

and these are the normal equations corresponding to (5.13). To keep the notation as simple as possible we introduce the inner product

$$(g(x), h(x)) = \int_a^b g(x)h(x)w(x)dx$$

for any two functions $g(x)$ **and** $h(x)$. Then, the optimal values of the $\alpha_j s$ are given by

$$\sum_{j=1}^{n} \alpha_j \left(\phi_i(x), \phi_j(x)\right) = \left(f(x), \phi_i(x)\right) \qquad i = 1,2,\ldots,n$$

or in matrix form as

$$A\alpha = \mathbf{b} \qquad \mathbf{(4.36)}$$

where

$$A_{ij} = \left(\phi_i(x), \phi_j(x)\right) \qquad i,j = 1,2,\ldots,n$$
$$b_i = \left(f(x), \phi_i(x)\right) \qquad i = 1,2,\ldots,n.$$

The ideas of orthogonality can be extended to this more general case if we say that a set of basis functions $\left\{\phi_i(x) : i = 1,2,\ldots,n\right\}$

is orthogonal on the interval [a,b] with respect to *w(x)* if

$$\int_a^b \phi_i(x)\phi_j(x)w(x)dx = 0 \qquad i \neq j.$$

Further, we say that the set is orthonormal if

$$\int_a^b \phi_i^{\,2}(x)w(x)dx = 1$$

If we use orthogonal basis functions the coefficient matrix in (5.36) will be diagonal and the solution vector can be obtained immediately. In subsection 5.4.2 we saw that it is possible to generate a sequence of polynomials orthogonal on [a,b] with respect to the weight *w(x)*=1 from the monomials. The ideas presented there are now applied to the more general case.

### 5.5.2 Gram-Schmidt Orthonornalisation process

Let $\psi_i(x) : i = 1,2,\ldots,n$ be a set of basis functions defined by

$$\psi_1(x) = \phi_1(x)$$

$$\psi_i(x) = \phi_i(x) - \sum_{j=1}^{i-1} c_{ij}\psi_j(x) \qquad i = 2,3,\ldots,n$$

so that $\psi_i(x)$ is a linear combination of $\{\phi_j(x): j = 1,2,\ldots,i\}$. For the function $\psi_i(x)$ to be orthogonal we need to find values for the $c_{ij}s$ such that

$$(\psi_k(x),\psi_i(x)) = 0 \qquad k = 1,2,\ldots,i-1$$

Now, suppose that we construct the $\psi_i(x)s$ one by one in the order $\psi_1(x),\psi_2(x),\ldots$ so that at the $i-1th$ stage the functions $\{\psi_k(x): k = 1,2,\ldots,i-1\}$ form an orthogonal set. Then

$$\begin{aligned}(\psi_k(x),\psi_i(x)) &= \int_a^b \psi_k(x)\psi_i(x)w(x)\,dx \\ &= \int_a^b \psi_k(x)\left(\phi_i(x) - \sum_{j=1}^{i-1} c_{ij}\psi_j(x)\right) w(x)\,dx \\ &= \int_a^b \psi_k(x)\phi_i(x)\,w(x)\,dx \\ &\quad - c_{ik}\int_a^b \psi_k(x)\psi_k(x)\psi(x)dx \qquad k = 1,2,\ldots,i-1\end{aligned}$$

remembering that $(\psi_k(x),\psi_j(x)) = 0: j = 1,2,\ldots,k-1,k+1,\ldots,i-1.$

Hence $\psi_i(x)$ will be orthogonal to $\psi_1(x),\psi_2(x),\ldots,\psi_{i-1}(x)$ if

$$c_{ik} = \int_a^b \psi_k(x)\phi_i(x)w(x)\,dx \Big/ \int_a^b \psi_k(x)\psi_i(x)w(x)\,dx$$

and so the process

$$\begin{aligned}\psi_1(x) &= \phi_1(x) \\ \psi_1(x) &= \phi_1(x) - \sum_{j=1}^{i-1}\frac{(\psi_j(x),\phi_i(x))}{(\psi_j(x),\psi_j(x))}\psi_j(x) \qquad i = 2,3,\ldots,n \qquad (4.37)\end{aligned}$$

can be used to generate a sequence of orthogonal functions from the basis $\{\phi_i(x):1,2,\ldots,n\}$.

The process (5.37) is known as the Gram-Schmidt orthogonalisation process and it can be used to generate a sequence of orthogonal functions starting with any linearly independent basis. If we normalise at each stage, that is divide $\psi_i(x)$ **by** $(\psi_i(x),\psi_i(x))^{1/2}$, then the new functions will form an orthonormal basis.

**EXAMPLE 5.11**

Let $[a,b]=[-1,1]$ and choose $w(x)=\left(1-x^2\right)^{-1/2}$. Then, starting with the monomials $\phi_i(x)=x^{i-1}:i=1,2,\ldots$ we have

$$\psi_1(x)=1$$
$$\psi_2(x)=x-\frac{(1,x)}{(1,1)}1.$$

now

$$(1,x)=\int_{-1}^{1}\frac{x}{\sqrt{1-x^2}}dx.$$

make the change of variable $x=\cos(u)$ so that $dx=-\sin(u)du$ **and** $\sqrt{1-x^2}=\sqrt{1-\cos^2(u)}=\sin(u).$ Then

$$(1,x)=-\int_{\pi}^{0}\cos(u)\,du=\int_{0}^{\pi}\cos(u)\,du=\left[\sin(u)\right]_0^{\pi}=0.$$

Hence

$$\psi_2(x)=x.$$

The next polynomial in the sequence, $\psi_3(x)$, is given by

$$\psi_3(x)=x^2-\frac{\left(1,x^2\right)}{(1,1)}1-\frac{\left(x,x^2\right)}{(x,x)}x$$

**and we have**

$$\left(x,x\right)=\left(1,x^2\right)=\int_{-1}^{1}\frac{x^2}{\sqrt{1-x^2}}dx=\frac{\pi}{2}$$
$$\left(1,1\right)=\int_{-1}^{1}\frac{1}{\sqrt{1-x^2}}dx=\pi$$
$$\left(x,x^2\right)=\int_{-1}^{1}\frac{x^3}{\sqrt{1-x^2}}dx=0.$$

**Hence**

$$\psi_3\left(x\right)=x^2-\tfrac{1}{2}.$$

### 5.5.3 Chebyshev Polynomials

In Example 5.11 we generated the sequence of polynomials $1, x, x^2-\frac{1}{2}$ which are orthogonal on [-1,1] with respect to the weight function $\left(1-x^2\right)^{-1/2}$. Now consider the polynomials

$$T_i\left(x\right)=\cos\left(i\ \cos^{-1}\left(x\right)\right)\qquad i=0,1,\ldots\qquad(5.38)$$

**so that we have**

$$T_0\left(x\right)=\cos\left(0\right)=1$$
$$T_1\left(x\right)=\cos\left(\cos^{-1}\left(x\right)\right)=x$$
$$T_2\left(x\right)=\cos\left(2\cos^{-1}\left(x\right)\right)=2\cos^2\left(\cos^{-1}\left(x\right)\right)-1=2x^2-1$$

and so on. Then these polynomials are also orthogonal on [-1,1] with respect to the weight function $\left(1-x^2\right)^{-1/2}$ since

$$\left(T_i(x), T_j(x)\right) = \int_{-1}^{1} \cos\left(i\cos^{-1}(x)\right)\cos\left(j\cos^{-1}(x)\right)\frac{1}{\sqrt{1-x^2}}\,dx$$

$$= \int_0^{\pi} \cos(iu)\ \cos(ju)\ du = 0 \qquad i \neq j\ \ i,j = 0,1,\ldots$$

Now, using a standard result, we have

$$\cos\left(i\ \cos^{-1}(x)\right) + \cos\left((i-2)\cos^{-1}(x)\right)$$
$$= 2\cos\left((i-1)\cos^{-1}(x)\right)\ \cos\left(\cos^{-1}(x)\right) \qquad i = 2,3,\ldots$$

so that

$$T_i(x) + T_{i-2}(x) = 2T_{i-1}(x)x$$

or

$$T_i(x) + 2xT_{i-1}(x) = T_{i-2}(x) \qquad (5.39)$$

and so these polynomials satisfy a three term recurrence relation. From (5.39) it is clear that $T_i(x)$ will be of degree $i$ and the next two polynomials in the sequence are

$$T_3(x) = 4x^3 - 3x$$
$$T_4(x) = 8x^4 - 8x^2 + 1.$$

The polynomials $T_i(x)$ are known as Chebyshev polynomials and are from a multiplicative constant they are the same as those generated by the Gram-Schmidt orthogonalisation process. The next example shows how Chebyshev polynomials may be used in least squares approximation.

**EXAMPLE 5.12**

Suppose that we wish to find a least squares approximation to $e^x$ **on** $[-1,1]$ of the form $p_3(x) = \alpha_1 T_0(x) + \alpha_2 T_1(x) + \alpha_3 T_2(x) + \alpha_4 T_3(x)$ where the $T_i(x)s$ are Chebyshev polynomials. Now

$$\int_{-1}^{1} T_i(x)T_i(x)\frac{1}{\sqrt{1-x^2}}dx = \int_0^{\pi} \cos^2(iu)du = \begin{cases} \pi & i=0 \\ \pi/2 & i=1,2,\dots \end{cases} \qquad (5.40)$$

so that if we use a weighted inner product with $w(x) = \left(1-x^2\right)^{-1/2}$ the coefficient matrix in (5.36) is diagonal. We have immediately that

$$\alpha_1 = \tfrac{1}{\pi}\int_{-1}^{1} \tfrac{e^x T_0(x)}{\sqrt{1-x^2}} dx = 1.26607$$
$$\alpha_2 = \tfrac{2}{\pi}\int_{-1}^{1} \tfrac{e^x T_1(x)}{\sqrt{1-x^2}} dx = 1.13032$$
$$\alpha_3 = \tfrac{3}{\pi}\int_{-1}^{1} \tfrac{e^x T_2(x)}{\sqrt{1-x^2}} dx = 0.271495$$
$$\alpha_4 = \tfrac{4}{\pi}\int_{-1}^{1} \tfrac{e^x T_3(x)}{\sqrt{1-x^2}} dx = 0.0443368$$

so that

$$p_3(x) = 1.26607 + 1.13032x + 0.271495\left(2x^2-1\right) + 0.0443368\left(4x^3-3x\right).$$

The integrals which define $\alpha_1, \alpha_2, \alpha_3,$ **and** $\alpha_4$ have been evaluated not analytically but numerically to six significant figures. The subject of numerical integration is not discussed here but the reader may be interested to know that an eight-point Gauss Chebyshev rule was employed.

### 5.5.4 Choosing a basis

Before leaving least squares approximation it is important that something be said about the choice of basis functions. It is advisable to obtain as much information as possible about the overall form of the function to be approximated and then choose a basis which reflects any know behaviour. For example it may be known that the function is even (odd) and hence there is no point in including odd (even) functions in the basis set. Thus a suitable choice of basis functions for the approximation of cos($x$) over $[-\pi/2, \pi/2]$ **would be** $1, x^2, x^4, \dots$ **or,** better still, the polynomials produced by the Gram-Schmidt orthogonalisation process from this set. Another situation that might arise is that in which the function is periodic on $[-\pi, \pi]$. In such a case it is natural to use as an approximating function a linear combination of periodic functions.

## 5.6 APPROXIMATION ON NON-FINITE INTERVALS

We now look at the problem of approximating a function *f(x)* over an interval for which one, or possibly both, of the end points is not finite. To find a least squares approximation the approach is exactly the same as that outlined elsewhere in this chapter but we must employ an appropriate weight function. We simply derive the normal equations and then solve for the polynomial coefficients. Once again orthogonal polynomials can be used to keep the amount of computational effort involved to a minimum.

Suppose that we wish to approximate *f(x)* over the interval $]-\infty, \infty[$. Then with respect to the inner product

$$(g(x), h(x)) = \int_{-\infty}^{\infty} g(x)h(x)e^{-x^2}\,dx \qquad (5.41)$$

the polynomials defined by the three term recurrence relation

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x) \qquad n = 2,3,\ldots$$

with

$$H_0(x) = 1,\ H_1(x) = 2x$$

are orthogonal and are known as the *Hermite polynomials.* If we seek an approximation of the form  and use the inner product (5.41) the coefficient matrix in the normal equations is diagonal and so the $\alpha_i s$ can be found as

$$\alpha_i = \int_{-\infty}^{\infty} f(x)H_{i-1}(x)e^{-x^2}\,dx \Big/ \int_{-\infty}^{\infty} H_{i-1}^2(x)e^{-x^2}\,dx.$$

It can be shown that

$$\alpha_i = \int_{-\infty}^{\infty} f(x)H_{i-1}(x)e^{-x^2}\,dx \Big/ \sqrt{\pi 2^{i-1}(i-1)!} \qquad i=1,2,\ldots,n.$$

For semi-infinite intervals we consider without loss of generality $[0, \infty[$ since any other semi-infinite range can easily be mapped onto this interval. With respect to the inner product

$$(g(x), h(x)) = \int_{0}^{\infty} g(x)h(x)e^{-x^2}\,dx \qquad (5.42)$$

the polynomials defined by the three term recurrence relation

$$L_n(x) = (2n-1-x)L_{n-1}(x) - (n-1)^2 L_{n-2}(x) \qquad n = 2,3,\ldots$$

with

$$L_0(x) = 1, L_1(x) = 1 - x$$

are orthogonal and are known as the *Laguerre* polynomials. Using (5.42) the coefficients in the least squares approximation of the form $P_{n-1}(x) = \alpha_1 L_0(x) + \alpha_2 L_1(x) + \ldots + \alpha_n L_{n-1}(x)$ **to** $f(x)$ **on** $[0,\infty[$ are given by

$$\alpha_i = \int_{-\infty}^{\infty} f(x) L_{i-1}(x) e^{-x^2} dx \Big/ L_{i-1}^2(x) e^{-x} dx.$$

It can be shown that

$$\alpha_i = \int_{-\infty}^{\infty} f(x) L_{i-1}(x) e^{-x^2} dx \Big/ \left((i-1)!\right)^2 \qquad i = 1,2,\ldots,n.$$

## 5.7 ALTERNATIVE CRITERIA FOR BEST APPROXIMATION

### 5.7.1 $L_p$ approximation

In the last four sections we have looked at least squares approximations in which the measures (5.10) (or a generalisation of it) has been used to determine the optimal values of the coefficients in the approximating function. As we remarked in subsection 5.3.1 other measures (such as (5.9)) are possible and we now investigate the class of measures which define the $L_p$ approximations.

Recall that the (unweighted) least squares approximation to a given function on the finite interval $[a,b]$ gives

$$\underset{\alpha_1,\alpha_2,\ldots,\alpha_n}{\textbf{minimum}} \int_a^b \left(f(x) - L(\alpha,x)\right)^2 \mathbf{dx}$$

that is, the integral of the square of the residual function $r(x) = f(x) - L(\alpha,x)$ is minimised with respect to the coefficients $\alpha_1,\alpha_2,\ldots,\alpha_n$. The least first *power* solution minimises not the integral of the square but the integral of the absolute value of the residual, that is, it gives

$$\underset{\alpha_1,\alpha_2,\ldots,\alpha_n}{\textbf{minimum}} \int_a^b \left|f(x) - L(\alpha,x)\right| \mathbf{dx}$$

whilst the *Chebyshev (or minimax)* solution gives

$$\underset{\alpha_1,\alpha_2,\cdots,\alpha_n}{\textbf{minimum}}\ \underset{a\le x\le b}{\textbf{maximum}}\left|f(x)-L(\alpha,x)\right| \tag{5.43}$$

and here the maximum absolute value of the residual is minimised. (The quantity (5.43) is a generalisation of the measure (5.9).)

Figure 5.1 illustrates the difference between least squares, least first power and minimax approximation. The function $f(x)$ is to be approximated by the straight line $p_1(x)=\alpha_1+\alpha_2 x$. Fig 5.1(a) shows an arbitrary straight line approximation and 5.1(b) the corresponding residual function. To obtain the least squares solution we need to find those values of $\alpha_1$ **and** $\alpha_2$ which minimise the area under the curve $r^2(x)$ of Fig.5.1(c) between *a* and *b*. The least first power solution gives the minimum area under the curve $|r(x)|$ of Fig.5.1(d) whilst for the minimax solution the maximum deviation of the curve $r(x)$ in Fig. 5.1(b) from the *x*-axis within $[a,b]$ needs to be minimised. Note that in the last three diagrams of Fig.5.1 the given curve crosses, or touches, the *x*-axis at the points at which the curves $y=f(x)$ **and** $y=\alpha_1$ intersect in Fig.5.1(a).

The three approximations considered so far in this subsection are members of the $L_p$ family of approximations. Formally we have the $L_p$ approximation to *f(x)* on $[a,b]$ gives

$$\underset{\alpha_1,\alpha_2,\cdots,\alpha_n}{\textbf{minimum}}\ \underset{a\le x\le b}{\textbf{maximum}}\left|f(x)-L(\alpha,x)\right|^p dx \tag{5.44}$$

and this definition may be generalised further by the insertion of a weight function, *w(x)*, inside the integral. It can be shown that as $p\longrightarrow\infty$ the approximation which gives (5.44) tends to the minimax $(\textbf{or } L_\infty)$ approximation which we now consider in detail.

## 5.7.2 Minimax approximation

In Example 5.11 and subsection 5.5.3 we introduced a sequence of polynomials, the Chebyshev polynomials, which are orthogonal on [-1,1] with respect to the weight $w(x)=\left(1-x^2\right)^{-1/2}$. These polynomials play an important role in the computation of minimax polynomial approximations and begin this subsection by stating a very important property that they possess.

**THEOREM 5.1 (Minimum deviation)**

Of all polynomials of degree $n$ with leading coefficient one, $T_n(x)/2^{n-1}$ has minimum maximum amplitude (that is, smallest deviation from zero) on [-1,1].

A proof of this theorem may be found in Ralston and Rabinowitz (1978). It is fairly easy to see from the three term recurrence relation (4.39) that $T_n(x)/2^{n-1}$ is a polynomial of degree $n$ with leading coefficient 1.

Now

$$\max \quad |T_n(x)| = 1$$

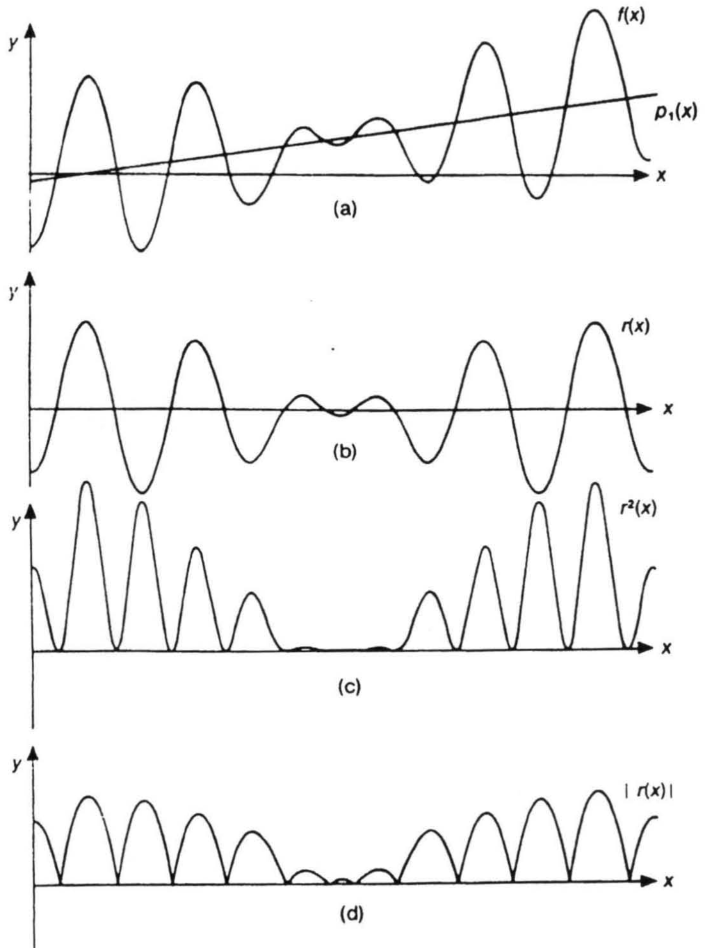


**Fig. 5.1**

polynomial $q_n(x)$ of degree $n$ and let $p_{n-1}(x)$ be that polynomial which gives

$$\underset{\alpha_1,\alpha_{21},\ldots\alpha_{n1}}{\textbf{minimum}}\ \underset{-1\le x\le 1}{\textbf{maximum}}\left|q_n(x)-\sum_{i}^{n}\alpha_i x^{i-1}\right|$$ .Then if the leading coefficient of $q_n(x)$ **is** 1

$$r_n(x)=q_n(x)-p_{n-1}(x)$$

Which is also of degree $n$ must, by the minimum deviation theorem, be equal to $T_n(x)/2^{n-1}$ and so the coefficients which define $p_n(x)$ may be obtained by equating terms.

**EXAMPLE 5.13**

To find the minimax polynomial approximation of the form

$$P_3(x)=\alpha_1+\alpha_2x+\alpha_3x^2+\alpha_4x^3$$

to $x^4$ **on** $[-1,1]$ we write

$$x^4-\alpha_4x^3-\alpha_3x^2-\alpha_2x-\alpha_1=x^4-x^2+1/8$$

so that $\alpha_4=0,\alpha_3=1,\alpha_2=0,\alpha_1=-\frac{1}{8}$ **and so** $p_3(x)=x^2-\frac{1}{8}$. Note that we have 'lost' the leading term here; this is to be expected since from (5.39), it is clear that the polynomials $T_0(x),T_2(x),T_4(x),\ldots$ involve only even powers of x whilst the polynomials $T_1(x),T_3(x),T_5(x),\ldots$ involve only odd powers.

The approximation of a polynomial of degree $n$ by a polynomial of degree one less provides an insight into the computation of Chebyshev approximations, a more general result being provided by the following theorem.

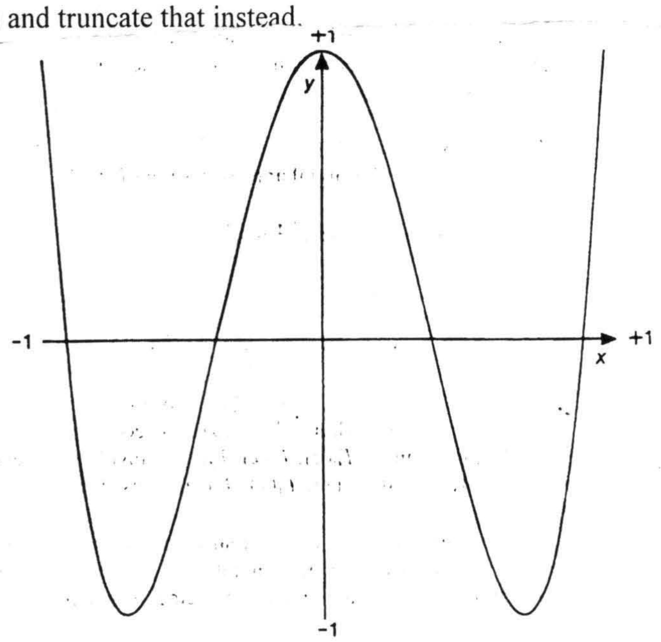
**THEOREM 5.2 (Characterisation theorem for $L_\infty$ approximation)**

Let $f(x)$ be a continuous function on the finite interval $[a,b]$ **and let** $p_{n-1}(x)$ be the corresponding minimax polynomial approximation of degree at most $n-1$. Then the residual function $r(x) = f(x) - p_{n-1}(x)$ equioscillates on $n+1$ distinct points in $[a,b]$. (See Fig. 5.3.)

This theorem (Phillips and Taylor (1973)) can be extended to a much wider class of basis functions subject only to a few minor constraints - see Powell (1981) for further details. Moreover it can be shown (Powell, 1981) that if a polynomial can be found such that the residual function equioscillates on n+1 points of [-1,1] then that polynomial must be the minimax approximation. Algorithms which use the equioscillation property to find a minimax approximation are rather complex in nature. However we now show that it is possible to find a polynomial approximation which almost satisfies the Chebyshev criterion without difficulty.

### 5.7.3 Chebyshev Series Approximation

In section 5.2 a polynomial approximation to $f(x)$ was obtained by truncating its Taylor series expansion. Clearly we can also express the Taylor expansion as a linear combination of Legendre or Chebyshev polynomials and truncate that instead.



Minimax approximation

$(n = 3)$

Fig. 5.3

Let

$$f(x) = \sum_{i=0}^{\infty}{}' a_i T_i(x)$$

(The prime on the summation sign indicates that the first term in the series is halved. This is a conventional nicety only and merely serves to make life easy for us later on.) Now since the Chebyshev polynomials are orthogonal on [-1,1] with respect to the weight $\left(1 - x^2\right)^{-1/2}$ we have

$$\begin{aligned}\int_{-1}^{1} \frac{f(x)T_j(x)}{\sqrt{1-x^2}} dx &= \int_{-1}^{1} \sum_{i=0}^{\infty}{}' a_i T_i(x) T_j(x) \frac{1}{\sqrt{1-x^2}} dx \\ &= \sum_{i=0}^{\infty}{}' a_i \int_{-1}^{1} T_i(x) T_j(x) \frac{1}{\sqrt{1-x^2}} dx \\ &= \frac{\pi}{2} a_j \quad j = 0,1,\ldots.\end{aligned}$$

(note the effect of the prime in relation to the result (5.40). Making the usual change of variable we have

$$a_j = \frac{2}{\pi} \int_0^{\pi} f\left(\cos(\theta)\right) \quad \cos(j\theta) \mathbf{d}\theta. \qquad \mathbf{(5.45)}$$

To obtain the Chebyshev series coefficients $a_j : j = 0,1,\ldots.$ we do not need to know any of the derivatives of $f(x)$. However, we do need to be able to evaluate the integrals in (5.45) and the easiest way to do this is to use one of the numerical techniques described in Chapter 6. In particular, the *N*-panel trapezium rule gives

$$a_j \approx \frac{2}{N} \sum_{k=0}^{N}{}'' f(x_k) T_j(x_k) \qquad j = 0,1,\ldots$$

with

$$x_k = \cos\left(\frac{\pi k}{N}\right) \qquad k = 0,1,\ldots,N$$

where the primes on the summation sign indicates that both the first and the last terms are to be halved.

Suppose now that we truncate the Chebyshev series expansion so that

$$p_{n-1}(x) = \sum_{i=0}^{n-1}{}' a_i T_i(x)$$

may be regarded as a polynomial approximation of degree at most $n$-1 to $f(x)$. Then the residual function is given as

$$r(x) = f(x) - p_{n-1}(x) = \sum_{i=0}^{\infty}{}' a_i T_i(x) - \sum_{i=0}^{n-1}{}' a_i T_i(x) = \sum_{i=n}^{\infty} a_i T_i(x).$$

If the series converges rapidly then the form of $r(x)$ will be dominated by the term $a_n T_n(x)$ provided that a suitably large value of $n$ has been taken. This means that the residual function behaves like a Chebyshev polynomial so that, effectively, $p_{n-1}(x)$ is the minimax approximation to $f(x)$.

Chebyshev series approximation on intervals other than [-1,1] is quite straightforward as the next example shows.

EXAMPLE 5.14

Suppose that we wish to find a Chebyshev series approximation to $e^x$ valid on [0,1]. Then we need to make the change of variable

$$z = 2x - 1. \qquad (5.46)$$

Now $x = (z+1)/2$ so we need to compute the Chebyshev coefficients for $e^{(z+1)/2}$ and, using the program of the next subsection, the first five terms are

$$a_0/2 = 3.50677$$
$$a_1 = 0.850391$$
$$a_2 = 0.105208$$
$$a_3 = 0.00872238$$
$$a_4 = 0.000542785$$

The series is converging at a reasonably fast rate and if we truncate after the term involving $a_3$, to give

$$p_3(x) = \frac{a_0}{2} T_0(2x-1) + a_1 T_1(2x-1) + a_2 T_2(2x-1) + a_3 T_3(2x-1)$$

the residual will be dominated by the term $a_4 T_4(2x-1)$. Note that the change of variable (5.46) must be used when evaluating the Chebyshev polynomials. Now, from the definition (5.38) $|T_4(2x-1)| < 1$, so that

maximum

$$|e^x - p_3(x)| \approx |a_4| = 0.000542785$$

$x \in [0,1]$

Looking at the output from the preceeding we see that $|e^x - p_3(x)|$ takes a value which is close to $|a_4|$ at five points and that the sign of the residual at these points is alternately positive than negative.

## 5.7.4 A program for Chebyschev Series Approximation

```
//a program to evaluate Chebyshev series coefficients and to check the approximate
//equioscillation property of a truncated series
#include<iostream.h>
#include<iomanip.h>
#include<math.h>

#define MAXHIGHESTCOEFFICIENT 50;
#define TOL 0.000005;

class indextype
{public:
        int indextype;
 };

 class coeffsvector
```

```
{public:
                float coeffsvector[50];
};

float f(float x)
{ float f;
        f = exp(x);
        return f;

}

float g(float theta )
{
         float g;
        g=f((cos(theta)+1.0)*0.5)*cos(i*theta);
        return g;
}

float evaluate(indextype n, coeffsvector coeff, float x)

        float pn,pnminus1,pnminus2,sum;
        indextype i;

        pn=x;
        pnminus1=1.0;
        sum = coeffs[0] + coeffs[1]*x;
        for (i=2;i<=n;i++)
        {
                pnminus2 = pnminus1;
                pnminus1= pn;
                pn = ((2.0*i-1.0)*x*pnminus1-(i-1.0)*pnminus2)/i;
                sum = sum + coeffs[i]*pn
        }
        evaluate = sum
        return evaluate;
}



main()
{
        float halfpi,pi,step,xvalue,approximatevalue;
        indextype i,hihghestcoefficient;
        int j,numberofevaluationpoints;
        int converged;
        coeffsvector  coefficients;
        void trapeziumrule(float integrate(float x),float lower,upper,tolerance,int maximundepth,int
monitorprogress);
        pi=4.0 * arctan(1.0);
        halfp = pi *0.5;
        cin>>highestcoefficient,numberofevaluationpoint;
        cout<<endl;
        for (i=0;i<=highestcoefficient;i++)
        {
                tapeziumrule(g,0.0,pi,tol,10,false,converged,coefficient[i]);
                coefficients[i]=coefficients[i]/halfpi;
                if (converged==0) cout<<"approximate to the "<<i<<"coefficient "<<tol<<endl;
        }

 cout<<"the chebyshev polynomial coefficients are "<<"(constant term first)"<<endl;
 cout<<endl;
 for (i=0;i<=highestcoefficient;i++)
 cout<<coefficient[0]*0.5;
 cout<<endl<<endl;
 cout<<"table of approximate function values"<<endl;
 cout<<endl;
 cout<<"     x          approximate value          error"<<endl;
 step = 1.0/(numberofevaluations-1);
 xvalue=xvalue +step;
 approximatevalue=evaluate(highestcoefficient-1,coefficient,2.0*xvalue-1.0);
 cout<<xvalue<<endl<<approximatevalue<<f(xvalue)-approximatevalue
 }
}
```

```
// this procedure is missing
void trapeziumrule(float integrate(float x),float lower,upper,tolerance,int maximundepth,int
monitorprogress);
{


}
```

Sample Data

```
4       9
```

Sample Output

```
the chebyshev polynomial coefficients are ( constant term first )

3.50677e 0
8.50391e -1
1.05208e -1
8.72246e -3
5.42886e -4

table of approximate function values

      x                       approximate value          error
0.00000e  0                   9.99481e -1                5.18557e -4
1.25000e -1                   1.13365e  1               -5.01618e -4
2.50000e -1                   1.28431e   0              -2.83301e -4
3.75000e -1                   1.45473e   0               2.63886e -4
5.00000e -1                   1.64818e   0               5.43621e -4
6.25000e -1                   1.86793e   0               3.16132e -4
```

|  |  |  |
|---|---|---|
| 7.50000e -1 | 2.11725e 0 | -2.53918e -4 |
| 8.75000e -1 | 2.39942e 0 | -5.47871e -4 |
| 1.00000e 0 | 2.71771e 0 | 5.74272e -4 |

### 5.7.5 Least First Power Approximation

Finally in this section we look at the least power approximation problem. The solution to this problem is, in certain circumstances, surprisingly easy to calculate as shown by the following theorem.

THEOREM 5.3

Let $f(x)$ be a continuous function on the finite interval [-1,1] and let $p_{n-1}(x)$ be the least first power polynomial approximation to $f(x)$ of degree at most $n-1$. Suppose that the residual function $r(x) = f(x) - p_{n-1}(x)$ has precisely n zeros. Then the position of each zero is independent of $f(x)$ and is given by

$$\theta_i = \cos\left(\frac{i\pi}{n+1}\right) \qquad i = 1,2,\ldots,n.$$

This theorem (proof in Powell, 1981) says that, provided certain conditions are satisfied, the computation of an $L_1$ approximation reduces to finding values for the polynomial coefficients such that

$$r(\theta_i) = f(\theta_i) - p_{n-1}(\theta_i) = 0 \qquad i = 1,2,\ldots,n.$$

that is, we need to find values for the coefficients which ensure that the approximation passes through the coordinates $\left\{(\theta_i, f(\theta_i)) : i = 1,\ 2,\ldots,n\right\}$. This type of problem is known as interpolation problem and is discussed in the next chapter. The following example indicates the way in which the process works.

EXAMPLE 5.15 Consider the approximation of $e^x$ on $[0,1]$. For the general $[a,b]$ the zeros of the residual function are given by

$$\theta_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{i\pi}{n+1}\right) \quad i = 1,2,\ldots,n$$

so that for [0,1]

Choosing $n = 3$ it is fairly easy to see that the polynomial

$$p_2(x) = \frac{(x-\theta_2)(x-\theta_3)}{(\theta_1-\theta_2)(\theta_1-\theta_3)}e^{\theta_1} + \frac{(x-\theta_1)(x-\theta_3)}{(\theta_2-\theta_1)(\theta_2-\theta_3)}e^{\theta_2} + \frac{(x-\theta_1)(x-\theta_2)}{(\theta_3-\theta_1)(\theta_3-\theta_2)}e^{\theta_3}$$

passes through the coordinates $\{(\theta_1, e^{\theta_i}), (\theta_2, e^{\theta_2}), (\theta_3, e^{\theta_3})\}$

Rearranging we get

$$p_n(x) = e^{\theta_1}\frac{\theta_2\theta_3}{(\theta_1-\theta_2)(\theta_1-\theta_3)} + e^{\theta_2}\frac{\theta_1\theta_3}{(\theta_2-\theta_1)(\theta_2-\theta_3)} + e^{\theta_3}\frac{\theta_1\theta_2}{(\theta_3-\theta_1)(\theta_3-\theta_2)} - x\left\{e^{\theta_1}\frac{\theta_2+\theta_3}{(\theta_1-\theta_2)(\theta_1-\theta_3)} + e^{\theta_{23}}\frac{\theta_1+\theta_3}{(\theta_2-\theta_1)(\theta_2-\theta_3)} + e^{\theta_3}\frac{\theta_1+\theta_2}{(\theta_3-\theta_1)(\theta_3-\theta_2)}\right\} + x^2\left\{\frac{e^{\theta_1}}{(\theta_1-\theta_2)(\theta_1-\theta_3)} + \frac{e^{\theta_2}}{(\theta_2-\theta_1)(\theta_2-\theta_3)} + \frac{e^{\theta_3}}{(\theta_3-\theta_1)(\theta_3-\theta_2)}\right\} \approx 1.0153 + 0.85030x + 0.83298x^2.$$

Note that Theorem 5.3 says that $p_2(x)$ is the $L_1$ quadratic approximation on [0,1] to any function which satisfies the conditions of that theorem and which passes through the given coordinates.

The result of Theorem 5.3 holds provided that the residual function has precisely n zeros. This is not as stringent a condition as it might seem at first sight since the condition holds if the first n derivatives of

$f(x)$ are continuous on [-1,1] and $f^{(n)}(x)$ is everywhere positive in [-1,1] (and this means that the solution obtained in Example 4.15 is valid). The points $\{\theta_i : i = 1,2,\ldots,n\}$ (the interpolation points) are the interior points of [-1,1] at which $T_{n+1}(x)$ takes on its maximum value. They are also the zeros of $U_n(x)$, the Chebyshev polynomial of the second kind of degree *n*. The polynomials $\{U_n(x) : n = 0,1,\ldots\}$ are orthogonal on [-1,1] with respect to the weight function $w(x) = \sqrt{(1-x^2)}$ and may be defined by

$$\begin{aligned} &U_0(x) = 1; \quad U_1(x) = 2x; \\ &U_n(x) = 2x\ U_{n-1}(x) - U_{n-2}(x) \quad n \geq 2. \end{aligned}$$

## C++ RUN TIME PERFORMANCE COMPARED TO VISUAL BASIC AND FORTRAN

Comparative analysis of C++ runtime performance is herewith conducted with Visual Basic and Fortran. A problem to calculate the sum of solution of a polynomial of degree 20 evaluated at point 1 to 100000 step 1 is used to obtain average processing speed of the three compilers as a means of justifying comparative advantage of C++. Below are the C++ codes and the results obtained with similar execution in Visual Basic and Fortran.

```
//--------------------------------------------------------------------------------------------------------------
// program to compute the sum of solution of a polynomial of degree 20 evaluated at point 1 to 100000
#include <stdio.h>
#include <math.h>
#include <iostream.h>
#include <time.h>
//--------------------------------------------------------------------------------------------------------------
int main(void)
{
int x;
double result, total;
time_t t1, t2, t3;
t1 = time(NULL);
for (x = 1; x < 10001; x++)
 {result = x ^ 20 + x ^ 15 + x ^ 13 - 2 * x ^ 16 + x ^ 8 - 20 * x ^ 5 + 10 ;
   total = total + result;
 return x;
 }
 t2 = time(NULL);
 t3 = t2 - t1;
 prinf ("Result of evaluation, starttime, endtime, and time difference are %s", total, t1, t2, t3;
  }
//--------------------------------------------------------------------------------------------------------------
```

**PROCESSING SPEED**

| VISUAL C++ | VISUAL BASIC | MSFORTRAN |
|---|---|---|
| 1.58 secs | 3.47secs | 5.22secs |

**Deduction**

From this simple analysis, Visual C++ is indeed a better and faster compiler.

## GENERAL REMARK

Obtaining reliable numerical results of any given mathematical problem is clearly a factor of the iterative technique or mathematical model utilized. From the work done on this project, it is a big relief that very tedious repetitive and cumbersome manual calculations are simplified with C++. For example, the process of finding the solution in n unknowns of a linear simultaneous equation which is proportional to $n^2$ irrespective of whether Jacobi or Gauss - Seidel (method is adopted) is without any doubt a cumbersome manual calculation with n say, 100. In this case, it not only suffices to use C++ (which object orientation engender high processing speed) but that the iterative technique with faster convergence rate (Gauss - Seidel) be utilized to save processing time.

The same argument can be put forward concerning the process of finding a root of a function using functional iteration and Newton - Raphson iteration. Pairwise comparison of the two processes indicate a very quick convergence after just 3 iterations in the later approach while convergence was achieved with the former after 15 iterations. The implication of this is very clear in that for every given non-linear algebraic equation in which a root is desired, the best iteration program to use is the Newton - Raphson iteration program.

Also, the simplicity of the *class* and *object* coding feature of C++ makes it possible to develop structured programs for the approximation of numerically defined functions and finding solutions to ordinary differential equations.

## RECOMMENDATION

It has been demonstrated that solution to every mathematical problem can be obtained through a numerical step-by-step algorithmic approach. For such approaches it has also been demonstrated that there exists structural coding know how in C++ which can easily be used to reduce run time and to obtain reliable results to specified degree of confidence. It is therefore recommended that

1) *students of mathematics and statistics be exposed adequately to C++ OOP programming as a veritable tool for mathematical computing.*
2) *C++ be applied to complex mathematical calculations in engineering, hydrology, astrology etc.*
3) *further works be encouraged for the application of C++ to advanced mathematical models.*

# REFERENCES

4. Broyden, C. G. (1975) Basic Matrices. An Introduction to Matrix Theory and Practice, Macmillan, London

5. Conte, S. D. and De Boor, C. (1980), Elementary Numerical Analysis, Algorithmic Approach (3rd Edition), McGraw - Hill, Kogakasi, Tokyo

6. Johnson, L. W. and Riess, R. D. (1982), Numerical Analysis (2nd Edition) Addison - Wesley, Reading, Massachussets

7. Ladd, S. R. (1992), Applying C++, Colorado Springs, CO

8. Phillip C. and Cornelius B. 1986, Computational Numerical Methods, Ellis Howood Ltd

9. Young M. J. (1998), Mastering Visual C++6, Sybex Inc., Alameda,CA