Computer Programming & Numerical Analysis

Version 2.01
Last modified on July 18, 2017

Contents

A Note to the Reader

This Manual is intended for use in the Computer Programming & Numerical Analysis class. **The Manual does not assume any familiarity with either the C programming language which is used or the techniques of numerical analysis**. However, if the reader is familiar with these then it would be of help.

The book is organised into 7 Chapters and they are best gone through in sequence. **Chapter 1** is an introduction to programming in general and to the C programming language in particular. This Chapter may be skipped by readers who already are familiar with the language though it may help to go through it once to familiarize themselves with the notation etc. It also has some sample programs to illustrate the use of various C progamming constructs like loops and conditional statements etc.

Chapter 2 is an introduction to **Gnuplot**, an open source graphics package which we use to generate graphics in this course. Once again it is self contained and does not assume any previous familiarity with **Gnuplot**.

Chapter 3 uses the programming and graphics fundamentals from the earlier Chapters to discuss techniques to numerically find the sum of finite and infinite series.

Chapter 4 discusses various numerical methods to find roots of equations. The methods discussed are the Bisection method, Secant method and the Newton-Raphson method.

Chapter 5 is an introduction to numerically solving ordinary differential equations using Euler's formula and 4^{th} order Runge-Kutta method.

Chapter 6 discusses integration by various methods. Methods using intervals of equal and unequal width are discussed here. The Trapezoidal and Simpson's rule are discussed as examples of methods using intervals of equal width and Gauss-Legendre, Gauss-Hermite and Gauss-Lagauerre quadratures as examples of methods which use unequal intervals.

Chapter 7 is a short discussion of matrices and how to manipulate and use them in C. Here we use arrays and pointers to do various matrix operations.

Each Chapter has several sample programs so that the reader can get an idea of how to write his/her own programs. In addition, each Chapter has some questions and problems at the end which the reader should certainly attempt to answer and solve.

For students who own computers at home, the chances are that you are using a Windows OS based machine. This programming language used in this course is C and the compiler used is **gcc** which comes with a Linux OS. In addition, the graphics package Gnuplot also comes with the Linux OS. For Windows users, there is a simple way to emulate a Linux environment.

Go to **https://www.cygwin.com/** and download and install the program Cygwin. This will come in two versions- Cygwin32 and Cygwin64. If you have a 64-bit computer then install Cygwin64, else install Cygwin32. Once installed, run the program as you run any windows program, that is by double clicking on the icon. This will open a small window. In this type **startx**. This will open a terminal on your screen, exactly like the one you see in the laboratory on the Linux machines. It is as if, your Windows machine has turned into a Linux machine. You can run all the Linux OS programs like **Gnuplot, gcc, emacs** etc. in this terminal. Once you are finished, just type exit and you will return to your Windows environment.

We would very much like to get your suggestions regarding how to improve this Manual. In addition, if there are any errors or misprints that are spotted in the Manual, we would like to hear from you. Please send a mail with the suggestions/errors etc. to **shobhit.mahajan@gmail.com** making sure you quote the version number of the Manual as well as the Modification date of the Manual you are using. The version number and date are on the title page of the Manual.

Finally, programming is like cycling or swimming- no amount of reading can ever teach you how to cycle or swim. The only way to learn is to actually do it. Yes, you will lose balance and fall initially but ultimately you will learn how to ride a bicycle. It is the same with programming- unless you actually write your programs, you will never ever learn how to program. Here too, you will initially make mistakes which later on you will think are silly. But everyone who learns programming does this. So there is nothing to be ashamed or scared of. Just start writing and running your own programs and you will see how quickly you will become good at it. The important part will be to develop the ability to debug a program- that is to say, when a program has errors or is not doing what you intended it to do, then to be able to spot the reason for it. This is what we hope you will learn after this course.

Chapter 1

Introduction

This course on Computer Programming is meant to introduce students to both computer programming as well as to numerical analysis. Although it expects students not to have forgotten what they learnt about numerical methods in their undergraduate course, it makes no great demands on them in this respect.

1.1 General Introduction

Most of you would be familiar with using the computer for browsing the Internet. So what is a computer? We can divide a computer into **Hardware** and **Software**. Hardware consists of devices, like the computer itself, the monitor, keyboard, printer, mouse and speakers. These are called **Input/Output (IO)** devices. However, the real bits of hardware lie inside the computer. These are the various chips to perform all the operations, store commands and data etc. The main processing chip is called the **Central Processing Unit (CPU)**. Apart from this there are specialised chips for functions like storing data and instructions (**Random Access Memory**), graphic interface, ethernet connectivity etc. There could also be storage devices for data like a Hard Disk, CD-ROM, SD card etc.

Software is the name given to the programs that you install on the computer to perform certain types of activities. The first thing we need in a computer is an **Operating System (OS)**. You are all familiar with the various Operating Systems- Windows, Linux, Apple iOS and of course Android. Yes, Android is an operating system for mobile platforms like your phone. Software also includes all the other programs which allow us to use the computer- word processing software like MS Word, Spreadsheet software like MS Excel, games, email software etc. All these are called **Application software**.

1.1.1 Operating System

Of course we all know that at the most basic level, the computer works by manipulating electric charge and the language it uses is the language of 0 and 1. The manipulation of electric charge is done by

an enormously complicated set of electrical circuits comprising of millions of microscopic transistors, capacitors and resistors etc. To give you an idea, the Qualcomm Snapdragon chip in your Android phone has more than a billion transistors!
Obviously, to make the computer useful, it is important to be able to communicate with it not in the language of 0s and 1s but in ordinary language. An **operating system** (OS) is a software that, after being initially loaded into the computer by a boot program, manages all other programs in a computer and provides programmers/users with an interface used to access those resources. An OS processes system data and user input, and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system. Thus we can say that an OS is a translator and a manager which allows us to use the computer.

☞***In this course we shall use the Linux OS.***

For students who own computers at home, the chances are that you are using a Windows OS based machine. This programming language used in this course is C and the compiler used is **gcc** which comes with a Linux OS. In addition, the graphics package Gnuplot also comes with the Linux OS. For Windows users, there is a simple way to emulate a Linux environment.

Go to **https://www.cygwin.com/** and download and install the program Cygwin. This will come in two versions- Cygwin32 and Cygwin64. If you have a 64-bit computer then install Cygwin64, else install Cygwin32. Once installed, run the program as you run any windows program, that is by double clicking on the icon. This will open a small window. In this type **startx**. This will open a terminal on your screen, exactly like the one you see in the laboratory on the Linux machines. It is as if, your Windows machine has turned into a Linux machine. You can run all the Linux OS programs like **Gnuplot, gcc, emacs** etc. in this terminal. Once you are finished, just type exit and you will return to your Windows environment.

1.1.1.1 Linux Operating System

The name Linux was coined in 1991 by Linus Torvalds after his own name and is supposed to indicate any open sourced operating system based on a Unix platform. The word open source implies that typically all underlying source codes can be freely modified, used, and redistributed by anyone. The system's utilities and libraries usually come from the GNU operating system, announced in 1983 by Richard Stallman.
There are several distributions (popularly referred to as distros) of the Linux OS eg. Red Hat, SUSE Linux (Novell) and Debian; and other derived distros like Fedora, Ubuntu (Canonical Ltd.), Gentoo, PCLinuxOS, CentOS, etc. Broadly speaking, the mother distros (Red Hat, SuSE, Debian) differ mostly in the areas they specialize or otherwise their focus.

☞***In this course we shall use Scientific Linux.***

Since most of you may not be very familiar with working on the Linux operating system, it is useful to get familiar with the commands so that it is easier to manipulate your files and programs. A few commonly used commands are given below.

A FEW GENERAL LINUX COMMANDS

Some useful linux commands are given here (**you don't have to type $, which shows the command prompt**).

$pwd
print working directory. The command tells you what is your present work directory.

$ls
lists files in a given directory. There are many options available with ls, which you can find using the command **man ls** or **ls –help**. In fact, if you are not sure of all the options for **any** command, you can use this, that is **man command** where 'command' is the command you want more information on. Thus, **man cd, man mv** etc. can be used.

$cd
change directory. To see what is stored in any subdirectory, we must first of all change the current directory by using the cd command. Suppose we wish to change to the include directory. The required command then is
$cd include
This when followed by the ls command will display the files and subdirectories stored in the include directory. To return to the main directory we must execute
$cd ..
Here .. means the parent directory (or one directory up).

$mkdir mydirectory
creates a directory with the name mydirectory
$ls [option(s)]
If you run ls without any additional parameters or options, the program will list the contents of the current working directory in short form.
The options can be

```
$ls -l
```

This will give you a detailed list of the directory contents

```
$ls -a
```

This will also display hidden files

```
$ls -al
```

This will list all files in the current working directory in long listing format showing permissions, ownership, size, and time and date stamp.

```
$cp filename1.c filename2.c
```

copies one file to another file. Original copy filename1.c will also exist after the operation.

```
$mv -i filename1.c filename2.c
```

moves one file to another file. Original copy filename1.c will **NOT** exist after the operation. The flag -i ensures that the computer asks (something like Do you want to proceed?) before executing the command.

```
$rm -i filename1.c
```

deletes/removes a file. Once again the flag -i ensures a check before the command is executed.

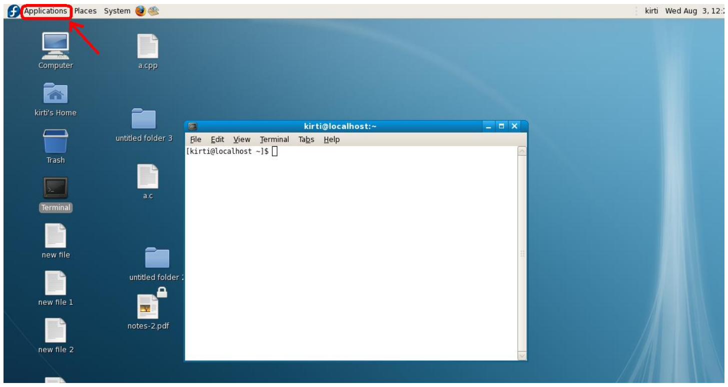
☞ ***You should familiarise yourself with all these commands since you might be using them extensively throughout this course. By default your present working directory will be your home area. Please create all your programs etc. in this directory. Please do not create a directory on the desktop since it would be difficult subsequently to copy files from it in case there is a server problem and you might lose all your work.Subsequently, if you want to organise the directory into various sub-directories with different topics, you can always do that using the commands above.***

GETTING STARTED:

Each of you has been given a userid and a password. Enter userid and the password

EXACTLY as you have been given. This will then allow you to log into the system. When you log in, you are working in your area. The desktop has a directory by the name of HOME. By default, all your work is stored in this directory.

Almost all the work that one does in this lab is done in the terminal mode.To open a terminal, use the mouse and click on (shown as circled on the figure) Applications → System Tools → Terminal. This will open a terminal as shown in the figure. The cursor will be on the command prompt [userid@localhost]$, where userid is your user name.



```
[kirti@localhost ~]$
```

Figure 1.1: **Screen Shot of opening screen after logging in**

1.1.2 Compiler

For our purposes, we would be using the computer to solve problems. For this purpose we need to learn a programming language. There are many programming languages like C, C++, Fortran, APL etc. After one writes a program to do a specific task, say find the sum of the first 10 digits, one inputs the program into the computer. For this purpose we need to use an **Editor**. We shall discuss the various editors below.

Once the program is entered and stored in the computer, one would need to run it. Here there is a problem- the computer, as we are aware only understands a language which is called a lower level language (e.g. Assembly language or Machine language) while our porgram is a set of instructions in English. To translate our instructions into a language which the computer understands, we need a program called a **Compiler**. The name"compiler" is primarily used for programs that translate source codes from a high-level programming language (like C, C++, Fortran, etc.) to a lower level language (e.g., assembly language or machine language). The original sequence is usually called the source code and the output the object code. The most common reason for wanting to translate source codes is to create an executable program. **In this course, we would be using C programming language**

and hence the native compiler with Linux, which is GNU C.

1.1.3 Editors

As we saw above, we need a **text editor** to type and save our program. A text editor is a computer program that allows you to input, modify and assemble text. For instance, in MS Windows, you may be familiar with the NOTEPAD or WORDPAD. A text editor is different from a Word Processing program (like MS WORD) in that it has limited capabilities of formatting and processing text. In Linux there are several editors, like **gedit, emacs, vim** etc. You can use any one of these to make your first program. Emacs and gedit are all menu based and so it is easy to find commands much like you are used to in Notepad and Wordpad. However, if you are interested, there are many tutorials available on the Internet for learning to use these editors efficiently. For instance, (***http://www.jesshamrick.com/2012/09/10/absolute-beginners-guide-to-emacs/***) is a comprehensive guide to using Emacs.
Similarly, for using gedit, (***https://help.gnome.org/users/gedit/stable/***) is a good guide.

1.1.3.1 Edit/Write your own program

Here, we are showing you how to make your first program using **emacs**, but it can be done using any of your favorite editors. On the command prompt of a terminal, type

$emacs &

This will open a new screen. Using keyboard, type the following program:

Program 1.1

```
/* This program prints ‘‘My First C programme’’ */
 #include<stdio.h>
 #include<math.h>
main()
 {
  printf(”My First C program \n”);
 }
```

After writing this program, click on the taskbar item

File → Save

This will open a window with the Save File As option. Enter the desired name of the program, like **first.c** in the New File Name and click OK.

☞***Most students find gedit somewhat simpler to use for editing programs. You should try to use both of them and see which one you prefer and then use it. Just make sure that the programs are properly saved.***

1.1.4 Programming Language

The computer language that we shall use is C, distributed by GNU. The reasons for this choice are threefold:

1. C is a scientific language much like Fortran or C++ or Pascal in structure, but has certain advantages on syntax
2. C has features that allow the programmer to organize programs in a clear, easy, logical way. For example, C allows meaningful names for variables without any loss of efficiency, yet it gives a complete freedom of programming style, including flexible ways of making decisions, and a set of flexible commands for performing tasks repetitively (for, while, do). Also it has a vast library of mathematical functions that can be harnessed to extensive computations.
3. C is useful not only in scientific computations but is also used by computer scientists who focus more on machine level programming or web designing. This is obviously a great advantage both in an academic as well as in a non-academic world.

1.2 Introduction to GNU C

On the command prompt of a terminal, type

```
$gcc -- version
```

This will type the version of the compiler you are using. Note that this compiler will compile programs written in C.

The language C is case sensitive. For example, the variable lower case x is different from the variable upper case X in C.

1.2.1 Create your first C program

This is a three step process:

1. First, you need to write and save your program in a text file using a **TEXT EDITOR**
2. Then you compile this program file using the gcc compiler to create an executable file.
3. Finally, you can **EXECUTE** or **RUN** the executable file created in step 2 to get the output

1.2.2 Compile/Run/Execute your first C program

Now that you have a program written in C programming language which you have named **first.c**, let us see how we can run the program to get the results. You can call the program by ANY name as long as the extension is **.c**. However, it helps to name programs in such a way so that later on you know by looking at the name what it might do. Also avoid very long names and those with too many special characters like _ etc. Compilation is necessary since the computer obviously does not understand the English language in which you have written the program. Further, compilation also takes all the header files (see below) like **stdio.h, math.h** etc and "attaches" them to the program you have written. These header files have predefined functions and routines which make it easier to use the programming language.
Compilation and Executing or running a program is a **two** step process.

1. First one has to **compile** the program to create an executable file
2. Second the compiled executable file has to be **run**.

   Let us see how this is done for our program **first.c**.

   In order to compile and create an executable, type

   $ gcc <program_filename.c> -o <executable_filename>

So, in our case, for example,

$ gcc first.c –o output.x

will compile the program saved in the text file with the name **first.c** and create an executable **output.x** in the same directory. The extension (.c) is meant to indicate the nature of this file. You can give the command ls to check if **output.x** is created. Now we have created an executable file, in our case **output.x** which can be executed or run by the machine.

1.2.2.1 Run your first C program

In order to run the executable file produced above, type at the command prompt,

```
$ ./output.x
```

Note that ./ is necessary before **output.x** to make sure that the file **output.x** in the current directory is executed.

Important: If you dont give a name to the output file (like we have given above (output.x), then the compiler, uses a default name. *The default name is a.out.* So for instance, instead of typing as above, if you compiled your first.c program as

```
$ gcc first.c
```

then the executable file created will be a.out and NOT output.x. This can be run by typing the following at the command prompt:

```
$ ./a.out
```

If everything goes fine, then it will print "This is my first C program" on the screen and will return back to the command prompt.

CONGRATULATIONS! You are successful in writing, compiling and executing your first C program.

> ☞***As pointed out above, there is no need to name the executable files and clutter up your area. An executable file can not be read or edited and so there is no reason to have different executable files in your directory. PLEASE DON'T CREATE ANY EXECUTABLE FILES EXCEPT THE DEFAULT ONE that is a.out. This way your directory will not be filled with useless files. In any case, if you want to run a program, all you need to do is to compile it again and run the freshly created new a.out.***

Although you have written a simple program which prints out some text, we don't really know what the meaning of each of the commands is. The first thing one has to understand about a programming language is that the rules of any programming language are very precise and strict. One has to follow them exactly otherwise the program will not work. In this respect, a computer programming language is unlike natural languages that we speak. In our spoken language for instance, even if we use a grammatically incorrect expression, most of the times the listener can understand what we are trying to say. In the case of programming languages, there is no such margin- the compiler , which is what translates our program into machine language will not understand if the program is written incorrectly even in a small way and return an error. In other words, natural languages have a fair degree of redundancy while a programming language has none.

To start learning how C programming language works, let us write another program, this time something which is a little bit more complex than our first program.

Program 1.2- sinx.c , Version 1

```
/* sinx.c − This program calculates sin(x) */
 # include <stdio.h>
 # include <math.h>

 main ()
 {
   float x,y;
   printf(”Supply the value of x in radians\n”);
   scanf(”%f”,&x);
   y = sin(x);
   printf(”%f,  %f\n”, x,y);
 }
```

After keying in the program above and saving it to a text file **sinx.c**, compile/execute and run your programme. Before you compile delete file **“a.out”** if it already exists. In any case, the compiler will overwrite any existing **a.out** file and replace it with the new one. Then type,

$ gcc sinx.c –lm

The program with the name **sinx.c** will compile and create the default executable **a.out** in the same directory. You can use the command **ls** to check if **a.out** is created. Why did we have to add **–lm** while compiling this program while we did not add this in compiling the first program? The reason for this is that this program has a statement **#include <math.h>**. We need this because we want the program to compute the value of $\sin(x)$ and this function is available in the file **math.h** as you will see below. Whenever we want to include the file **math.h** in any program, during compilation, we need to add the qualifier **–lm**. Since every single program that you write in this course would be using some pre-defined mathematical function, one will always have to include the file **math.h**. **Therefore, please remember to always add this qualifier whenever you compile any program.**

In order to run the executable file,

```
$ ./a.out
```

If every thing goes well the message in the seventh line will be printed on the screen that is

```
$ Supply the value of x in radians
```

and the cursor will begin flashing in the next line prompting you for input. If you now key in the number 3.14159 (thereby wanting to compute sin(π)) and press $<$ **ENTER** $>$, the numbers 3.14159. ,0.000000 will flash on the screen for an instant; after that the screen returns to the **COMMAND PROMPT**.

Hints for understanding the above program:

1. The first line in the above program is a comment and it is not part of the program in the sense that it is not compiled. You can give a **comment** anywhere in the program using the format /* **(Comment)** */. Any line in the program which begins with /* and ends with */ is not read by the compiler. Another way to write comments is to start the line with //. Anything between /* and */ is only for your convenience. It makes no difference to the program. Similarly, any line starting with // is not read by the compiler. The comments are very useful in making programs more readable and self- explanatory. You will find them of great use when you write longer programs and when you try to look at programs that you wrote earlier and whose details you would have forgotten. Writing comments is good programming practice since then anyone who looks at the program can understand it easily.

2. The next two lines beginning with **#include** append two header files to the programme. # is called a **preprocessor directive**. **stdio** stands for standard input output while the file extension **.h** indicates that it is a header file. These header files are there in the compiler directory of the computer. The **stdio.h** file governs the input via **scanf** statement and output via the **printf** statement. There are other means of input/output about which you will learn later. The **math.h** header file contains declarations of in-built mathematical functions like sin, cos, etc. Header files, as we mentioned, are files containing previously defined functions or routines which allow us to simplify the program. Thus, for instance, in the absence of **math.h**, every time that one wanted to find the cosine of an angle, or the square root of a number etc., one would have to write out explicit instructions to do it. The header files make it convenient for us to use various well known functions and operations. Again, please remember that the **syntax** of how to use these predefined routines and functions is fixed and one cannot change it.

3. The 4th line indicates the start of the main program. **Every program must have the function main()**. The 5th line is an open brace ({) **which signals the beginning of the main program. Correspondingly the last line contains a close brace (}) which signals the end of the program EVERY C-program has to have these features.**

4. The sixth line contains the **declaration** that the variables x and y are **floating point numbers**. Each variable in a C program has to be declared in accordance with its nature such as **float**ing point, **int**eger, **char**acter, **l**ong integer, **double** float etc. An integer variable, i, is declared as int i; similarly, a float variable x is declared as float x. **All these declarations must be made at the very beginning of the main program. It is good programming practice to initialize the variables to some nominal values (usually** 0**).** For a detailed discussion of variables, see below.

   Note the semicolon at the end of the statement. **All executable statements in C have to end with a semicolon.** To save space, one can write more than one statement in a line, each separated by a semi-colon but it is not advisable. Proper indentation of programs is crucial as it can often help us avoid common syntax errors.

5. The seventh statement prints the message ”Supply the value of x in radians” on the screen and the **n** in inverted commas causes the cursor to go to the beginning of the next line. **Any matter appearing within the inverted commas in a printf statement appears as it is on the screen** except for the special characters such as \n, %f, %d, %6.2f etc. The meaning of these special characters will be explained as we go along. The messages such as the one in line 7 in the above program are used as prompts, in this case to ask the user to input the value of x (in radians) to the program. The program now waits for you to supply the value of x.

6. Once you supply the value, the eighth statement is the input statement requiring you to supply the value of the variable x. If you have declared x as a **floating point number**, then it is best to supply x as a decimal fraction (**For example x = 2.0 instead of x = 2**). **scanf** is a function which will take the value of the input you have given on the screen and stores it in the memory. The **&x** in this statement corresponds to the **pointer** to the **memory address** of the variable x. (If you do not understand the pointers at this stage, do not worry, it would become clear later. It must be noted, though, that the pointer feature is a major advantage of C over other advanced languages like Fortran). The next statement computes sin(x) by making use of **math.h** and assigns it to the variable y. For details of assignment of variables etc. see below the section on Variables in C.

7. The next statement prints the value of two numbers x and y. In this statement **%f** describes the nature of the variable viz., x is a floating point variable denoted by f in %f. For integers we would use **%d**. The **n** once again is to bring the cursor to the next line though it is not required in this case.

8. The final line, as in **ALL** C programs is a closing brace indicating to the compiler the end of the program.

☞ ***Variables and constants in C can be given any name. However, the compiler will only take the first*** 8 ***letters as meaningful and ignore all the others. As mentioned earlier, C treats variables or constants in small case (like x, a, pi etc.) differently than those in upper case (like X, A, PI etc.). So please be careful. Also, by convention, lower case letters are normally used to denote variables while upper case letters are used to denote constants.***

1.2.3 Variables in C

There are different kinds of variables in C. Single precision Floating point (usually just called floating point), double precision Floating point (usually called double), integers, long integers, character variables. These are all different and to see the difference, one must understand how the computer stores these numbers.

Let us start with the simplest kind, that is **int**ger variables. An **int** variable can store values from -32768 to $+32767$, that is from -2^{15} to $2^{15} - 1$. What happens when we declare a variable, say a as **int a** in the program? The compiler on reading this line in the program tells the computer memory to allocate 16 bits (or 2 bytes) of memory space to the variable and call it in its address location as **a**. (*Why is the value of the number stored as **int** only 15 bits while we need 16 bits to store it?*). The format specifier for an **int**eger variable is **%d**.

Now, later on in the program, suppose you give a value of 156 to the variable **a**, you would typically write

$$a = 156;$$

in the program. **The character "=" in C is used for assignment and NOT as an equality** What this statement will do will be to tell the computer that the memory in the address location called **a** , should store the number 156. That is important because it allows one to use the same variables repeatedly. Thus, in ordinary mathematics, a statement like

$$x = x + 10;$$

will make no sense. In C, it simply means: Take the value stored in the memory location called x, add

10 to it and finally, store the result in the same memory location, that is now the value in the memory location of x is the new value, which is the old value plus 10. You can see how this makes a program much easier to write or else everytime we wanted to do an operation on a variable, we would need to define a new variable.

While in most compilers, the **int**eger variable is 16 bits or 2 bytes long, there are some compilers where it is 32 bits or 4 bytes long. In most compilers, there is another variable class defined which can take care of longer integers. This is called the **long int**eger which is 32 bits long and therefore can take values from 2147483647 to +2147483647, that is from -2^{31} to $2^{31} - 1$. The format specifier for **long int**eger is **%ld**.

Similarly, the next kind of variables is the **single precision floating point variable**. These variables have a precision of 6 digits and range from 10^{-36} to 10^{36} and unlike the integer variable, it needs 32 bits (or 4 bytes) of memory space to be stored. Now it is clear that it would be difficult to store this many numbers in 32 bits. So what the computer does is to store it as a number between 1 and 2 scaled with a power of 2. Thus, for instance, 1.0 is stored as $+1.0 \times 2^0$ and -5.0 is stored as -1.25×2^2 etc. So we see that we need appropriate storage for three things:

1. The sign of the number which means 1 bit of storage.
2. The exponent of 2 which uses 8 bits of storage.
3. The mantissa, that is the number which multiplies the power of 2 therefore has 23 bits of storage left. Now since there is little point in storing 1, and the mantissa will **always** be less than 2, the 1 is ignored and the rest of the mantissa after the decimal point is stored as an integer variable of 23 bits.

This way we can store very large and very small numbers. In our case, this will be sufficient. The format specifier for single precision floating point variable is **%f**.

However, in certain cases, more precision is required and so there is another class of floating point variables, called **double precision floating point variables**. These have 14 digits of precision and can range from 10^{-303} to 10^{303}. They are stored just like the single precision floating point variables but with the exponent having 11 bits instead of 8 bits and the mantissa having 52 bits. The sign bit remains the same. Thus to store a double precision floating point number we need 64 bits or 8 bytes. The format specifier for double precision floating point variable is also **%f**.

Then there are **char**acter variables where one can store a **single** character. It is 1 byte long. The format specifier for character variable is **%c**.

Example 1.3- sinxtab.c (Version 1)

Now let us try to run a variant of the previous program which will tabulate the value of sin(x) at twenty equally spaced values of x lying between $x1$ and $x2$ that you would specify. The listing of this program is as follows:

Program 1.3- sinxtab.c , Version 1

```
/* sinxtab.c - tabulates values of sin x */
 # include <stdio.h>
 # include <math.h>
main()
 {
   float x, x1, x2, y, dx, pi;
   int i, n=20;
   pi = 4.0*atan(1.0);
   printf(”Supply x1 and x2 in units of pi\n”);
   scanf(”%f, %f”, &x1, &x2);
   x1 = x1*pi;
   x2 = x2*pi;
   dx = (x2-x1)/(float)(n);
   for(i=0; i<= n; i++)
   { x = x1 + i * dx;
     y = sin(x);
     printf(”%6.2f %6.2f\n”, x, y);
   }
 }
```

Hints for understanding the above program:

1. As in the case of floating point variables, **integer variables must also be explicitly declared in the beginning.** It is possible to assign values to integers in the declaration statement itself, as is done for n in the above program.

2. The **atan** function evaluates arc tangent (arctan) of its argument.

3. The **for** statement sets a sequence i = 0 to n in unit steps. i++ increments i by one and is identical to i = i + 1. All statements after the braces on the next line until the closing braces (in the 14th line) are executed for every value of i in the sequence defined above. This procedure is called **looping**. **NOTE that there is no semicolon after the for statement.**

4. **%6.2f** is a format symbol which stands for a floating point number of total field of six spaces with two places after decimal.

5. The space between **%6.2f** and the second **%6.2f** is to separate the numbers adequately when printed on the screen.

1.2.4 Loops in C

Frequently we would like to repeat some operation for a specified number of steps. In this case, the **for** loop in C is very useful. A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a **specific number of times**. The syntax for a **for** loop is as follows:

```
for ( init; condition; increment ) {
   statement(s);
}
```

The **init** step is executed first, and **only once**. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the **for** loop.

After the body of the **for** loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the **for** loop terminates.

Thus, in our example above, the for loop is

```
for(i=0;i<=n;i++)
{
x = x1 + i * dx;
     y = sin(x);
     printf("%6.2f %6.2f\n", x, y);
}
```

The **init** step is **i=0**, that is the variable **i** which is an integer variable is initialised to 0. The the **condition**, that is $i <= n$ is evaluated. If it is true, which it will be at first since $n = 20$, the body of the loop, that is the statements contained within the curly braces are executed. After this is done once, the flow goes back to the **increment** statement, which in this case is $i++$. Now, recall that the initial value of i was $i = 0$. The statement $i++$ tells the computer to take the value stored in the memory location addressed by i, and whatever the value is there, increment or increase it by 1 and then store the new value in the **same** memory location. After the increment, the condition, that is $i <= n$ is evaluated again and since it would be found to be true, since the value of i after the first run is now $i = 1$, the process repeats itself. This continues till the time the **condition** becomes false when the loop terminates.

Looping or recursion is a very important part of programming and there are many ways in which one can use the various looping structures available in C. Remember that the **for** loop is best used when we want to perform certain statements for a **known** number of steps.

Another kind of looping structure is the **do-while** loop.

Frequently, when we want to repeat certain steps in a program, we do not know how many times we have to repeat the steps, but we do know what the final value of a particular parameter or variable we want. Then it is best to use the **do-while** loop. The structure of a **do-while** loop is as follows:

```
do {
   statement(s);
} while( condition );
```

Here **statements** are the steps you want the computer to repeat and **condition** is the statement which if true means that the steps keep on repeating. Thus in the previous example, we could replace the **for** loop with

```
i=0;n=20;
   do
   { x = x1 + i * dx;
     y = sin(x);
     printf("%6.2f %6.2f\n", x, y);
     i++;
   }while(i<=n);
```

The structure and the meaning of the various lines should be obvious in this loop construction.

Another useful loop structure is the **while** loop. The structure of the **while** loop is as follows:

```
while(condition) {
   statement(s);
}
```

Notice that this is almost exactly like the **do-while** loop except one very important difference. In this loop structure, like the **for** loop, the condition is checked at the **top** of the loop. In the **do-while** loop, the condition is checked after the loop has been executed atleast once, that is the condition is checked at the bottom of the loop. This is a very important difference. In our example above, we could write the same loop with a **while** loop as follows:

```
i=0;n=20;
  while(i<=n)
   { x = x1 + i * dx;
     y = sin(x);
     printf(”%6.2f  %6.2f\n”, x, y);
     i++;
   }
```

We can also have nested loops that is one or more loops inside a loop. The nested loops do not have to be of the same kind, that is one can have **for** loops inside a **do-while** loop etc. **The only thing one has to be careful of is that the loops are completely nested. That means that the inner loops should close before the outer loop closes. This is extremely important and you must remember this.**

Change of Type (typecasts)

Very often one has to change a variable from one type to another; for example **int** to **float**, **float** to **double**, **float** to **int** etc. Thus **float(n) OR (float)(n) converts the value of n into float type. Note that whereas n remains of the type int and its value in the above example remains** 20**, the value of (float)(n) becomes** 20.0**.** Similarly, the statement (int)(x*y+0.1*exp(x)) will, for example, evaluate the expression (x*y+0.1*exp(x)), which is of the type **float** and then convert it into type **int**. Thus we may have a statement like

i = (int)(x*y+0.1*exp(x));

Of course i must already have been declared as an **int** variable. If the expression on the RHS evaluates to 3.7, the integer i will have the value 3; if it evaluates to −3.8, i will have the value −3.

☞***Whenever we divide by a floating point number with an integer, we should always as good programming practice, change the type of the integer in the denominator to float. Otherwise there is a chance that the program will do integer division and give strange results.***

Example 1.3 (version 2)

An alternative program

Program 1.3- sinxtab.c , Version 2

```
/* sinxtab_ver2.c - tabulates values of sin x */
 # include <stdio.h>
 # include <math.h>
main()
 {
   float x,x1,x2,y,dx,pi;
   int i,n=20;
   pi = 4.0*atan(1.0);
   printf(”Supply x1 and x2 in units of pi\n”);
   scanf(”%f,%f”,&x1,&x2);
   x1 = x1*pi;
   x2 = x2*pi;
   dx = (x2-x1)/(float)(n);
   for (x=x1; x <= x2; x+ = dx)
   { y = sin(x);
     printf(”%6.2f %6.2f\n”,x,y);
   }
 }
```

to set up the sequence: x=x1, x1+dx, ...x2 (=x1+n*dx). **Here x+=dx is an abbreviation for x = x + dx. Similarly x*=(expression) implies x = x * expression, etc.** Note that the integer variable i is no longer required in the program; therefore it may be deleted from the declaration statement int i, n = 20. Rerun the program. The results of these two programs should be identical.

☞ ***The ability of a programming language to do this recursion, that is*** $x = x + dx$ ***or*** $x = x * dx$ ***is amongst the most powerful things in programming. Of course, as a mathematical statement it makes no sense. But if we understand how a computer works, then it will be clear. When we define a variable, say*** x ***as a floating point variable, the computer assigns a memory space which typically is*** 4 ***bytes or*** 32 ***bits. This memory space is labelled*** x***. Now whenever we have a value of the variable*** x ***in the program, it stores that value in this memory location. Now imagine what happens when the program encounters a statement like*** $x = x + 1.6$***. It takes the value of the variable*** x ***stored in the memory labelled*** x ***and adds*** 1.6 ***to the number. The new number is again stored in the same memory location and labelled as*** x***. Now after this step, whenever the variable*** x ***is called in the program, it will return the new value of*** x***. This kind of recursion allows programs to be very versatile and***

flexible.

1.3 Functions in C

You have already used library functions like sin, cos, atan, etc. To be able to appreciate the modular structure of C language, it is essential for you to learn how to make your own function routine which performs a specific computation when called by the main program. In this manner, the main program will simply become a calling program and call various functions to perform their respective computations.

A function is a group of statements that together perform a task. Every C program has at least one function as we have seen. This is the function main(). One can divide up one's code into separate functions. How one divides up one's code among different functions is up to you, but logically the division is such that each function performs a specific task. Think of a function as a blackbox which sits inside the main program, which incidentally is also a function! When the function is called by the main program or any other function, then the blackbox is supplied with some input values or parameters. The blackbox performs certain operations on the input values according to how you have defined the function and then returns the value computed to the main program.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. We already know that the C standard library provides numerous built-in functions that your program can call. For example,printf(), scanf(), and many more functions. In C Language, a function is the same as a method or a sub-routine or a procedure, etc. In some languages these are not the same.

The general form of a function definition in C is as follows

```
return_type function_name( parameter list )
 {
   body of the function
}
```

A function definition in C, as we have seen, consists of a function header and a function body. Here **return_type**, **function_name** and **parameter list** are the components of the function header. Here **return_type** tells the compiler what kind of function it is that one is defining. A function may or may not return a value. The **return_type** is the data type of the value the function returns, that is if the function returns a floating point variable, then the **return_type** is **float**, if an integer then **int** and so on. However, it is not always the case that a function has to return a value. Some functions perform the desired operations without returning a value. In this case, the **return_type** is the keyword **void**. Remember that printing something does not count as returning a value. We have already encountered examples of such void functions- **printf, scanf** and even **main() are examples of such functions. function_name** is basically the name by which the function is going to be called or addressed in the program. **parameter list** is essentially what goes into the function or the blackbox. Whenever we call the function in the main program, we need to specify the input parameters. The **parameter list** thus has the type, order, and number of the parameters of a function. However, just like one can have functions which do not return Parameters are optional; that is, a function may contain no parameters. Finally, the **function body** contains a collection of statements that define what the function does.

To see how all this works, let us consider an example.

Example 1.4

For example if you wish to compute the function

$$f(x) = x * x * x + \sin x * \log x$$

in your program, you can do so in two ways.

```
# include <stdio.h>
# include <math.h>

/*
   Function with name f of type float and has one
   argument which is of type float
*/

 float f(float x)
 {
   float z;
   z = x*x*x + sin(x)*log(x);
   return (z);
 }

/*Main function which will call above function*/

main()
{
  float y;
  float f(float x); /* function declaration */
printf(‘‘Supply  the value for which you want to evaluate the function\n”);
  scanf(”%f”,&y);
  printf(”%6.2f  %6.2f\n”, y, f(y));
}
```

Here **return_type** is float since the function is going to return a floating point number. The **functon_name** is simply **f** and the **parameter list** is x. Note that we need to define the kind of variable the parameter x is, in this case float. This method of writing a function returns to the calling program the value of the function for the specified argument x. Alternatively, we could define the expression as follows:

float f(float x)
{
return (x*x*x + sin(x)*log(x));
}

Note that in this case we did not use the **local float variable** z to first calculate the result and then return the value. A local variable within the function is one which has no relevance outside the function. Note that in this case, x, the input variable is **not** local but is passed into the function. Similarly the value of the function is passed out but stored locally as z.

☞ ***The global variables of the function (in our example the variable x) need to be supplied whenever the function is called. Of course though the variable is called x in the function definition, one can call the function in the main program for any variable or even a constant. Thus in the above example, we saw that the function f(x) was called for the variable y which was supplied to the program by the user. Alternatively, we could have asked the program to print f(1.7) instead of f(y). Remember that the function of more than one variable, whenever it is called in the main program, should have exactly the same number of and type of variables specified in exactly the same order.***

Since this is a simple one-line function, we can also use an alternative way to define it. This is called preprocessor directive. The syntax would be

#define f(x) ((x)*(x)*(x) + sin(x)*log(x))

There must be

(a) **A space between #define and f(x) and between f(x) and the beginning of the function.**

(b) **The argument of the function should always be within brackets wherever it appears and the entire function should also be enclosed within brackets**.

The return type of inline functions (int/float etc.) is decided by the argument x. The complete program may look like the following:

```
#include <stdio.h>
#include <math.h>
#define f(x) ((x)*(x)*(x) + sin(x)*log(x)) /* inline function */

/*Main function which will call above function*/
main()
{
    float y;
    printf(”Supply  the value for which you want to    evaluate the function\n”);
    scanf(”%f”, &y);
    printf(”%6.2f %6.2f\n”, y, f(y));
}
```

☞***It is ALWAYS a good idea to declare function prototypes in the main program except for the inline functions. Notice that both these styles of functions have NO semicolon at the end of the line. ALWAYS remember to enclose ALL variables in an inline function in parenthesis.***

We have just seen an example of defining functions in C. However, the above example was fairly simple and did not really need a function to work. We could as well have written a variable z and then printed the values of z for different values of x. The following example will bring out the real utility of functions in C.

Example 1.5

Let us take another example: We wish to evaluate the function

$$\begin{aligned} f(x,n) &= x^n - 1 + e^x \text{ for } x \leq 0.0 \\ &= x^n - \log(1+x) \text{ for } x > 0.0 \end{aligned}$$

where n is an integer. Obviously, such a function can not be written in the inline style easily. However, in the first style it may be written as:

```
float f(float x, int n)
{
float y,z=1.0;
int i;
for(i=1;i<=n;i++)
 {
  z*=x;
}
if (x < 0.0)
{
 y = z - 1 + exp(x);
}
else
{
 y = z - log(1+x);
}

return (y);
}
```

and the full program may look like the following:

```
# include <stdio.h>
# include <math.h>

/*
   Function with name f of type float and has two
   arguments, one of type float and other of type int.
*/

float f(float x, int n)
{
    float y,z=1.0;
    int i;

    for(i=1;i<=n;i++){
       z*=x;
    }
   if (x < 0.0) {
     y = z - 1 +exp(x);
   }
   else {
     y = z - log(1+x);
   }

   return (y);
}

/* global function declaration */
float f(float x, int n);

/* main function will call f */
main()
{
  float q;
  int j;
  printf(”Supply  the value of x(float), n(int)\n”);
  scanf(”%f, %d”, &q, &j);
  printf(”x=%6.2f n=%d f=%6.2f\n”,q, j, f(q,j));
}
```

Note that here we have defined a function of two variables, one of which, x, is a float variable and the other, n, is an integer variable. We can, in fact, define in the same way functions of several variables.

1.3.1 Decision Making in C

You would have realised by now that a C-program is basically the encoding of an **algorithm** in the C-programming language. An algorithm is essentially a step by step method to solve a particular problem. We use algorithms all the time without realising it. Thus, for instance, suppose you want to wake up **exactly at 4am**. Then typically you would need to set an alarm for 4am. The algorithm then can simply be

1. Pick up mobile
2. Open Alarm App
3. Check time of existing alarm
4. **If** Alarm already set for 4am, go to 6
5. **Else** Set alarm for 4am
6. Shut the Alarm App on your mobile

Now this might seem trivial and obvious, but notice that this is exactly what you do. Basically, any particular task can be broken up into discrete steps and the set of these steps to complete a task is called an algorithm.

In the above algorithm, notice that there is a decision to be made. The decision is to check the time of the existing alarm. Then, if the alarm is already set for 4am, then one does nothing and shuts the Alarm App. Else one sets the alarm for 4am. It is these decision structures that we need to implement in C.

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. There are many kinds of decision making structures in C.

The simplest one is the **if** statement. The structure

```
if(boolean_expression) {
 statements to be executed if boolean expression is true
}
```

Thus for instance, we could have the following program

```
#include <stdio.h>
 main ()
 {

      int a = 17;

     if( a < 100)
      {
            printf("a is less than 100\n" );
   }
   printf("value of a is : %d\n", a);
   }
```

The **boolean expression** in this program is " if $a < 100$". The statements inside the curly braces following **if** will be executed **only if** the expression is true. If not, the control will pass to the line immediately after the **if** structure. In the above case, since $a = 10$, and therefore the boolean expression is true, the program will execute the statements inside the **if** and print " a is less than 100". After this it will go as usual to the next line after the **if** block and print " value of a is : 17".

The most commonly used decision structure is the **if....else** structure.

```
if(boolean_expression)
{
   statement(s) will execute if the boolean expression is true
}
else
{
   statement(s) will execute if the boolean expression is false
}
```

In the program above, we could also use this structure as follows.

```
#include <stdio.h>
 main ()
 {

   int a = 179;

   if( a < 100 )
   {

      printf(”a is less than 100\n” );
   }
   else
   {

      printf(”a is not less than 100\n” );
   }

   printf(”value of a is : %d\n”, a);
   }
```

Here in the **if** part of the structure, a boolean expression is checked. In this case since the variable $a = 179$, the boolean expression $a < 100$ is clearly false. Therefore the statements inside the **if** are not executed. Instead, the control goes to the **else** part of the structure. Notice that here there is no condition and therefore if the boolean expression in the **if** is false, then the statements within the **else** structure are always executed. In this case, the statement is a simple printing of “a is not less than 100”. Note again that one can have as many statements within the **if** and **else** structures. Also note that if the boolean expression in **if** is true, then the statements within the **else** structure are **not** executed.

Another decision structure which is used frequently is the **if...else if...else if......else**.Basically, one can use an **if** statement followed by an optional **else if...else** statement. This is very useful to test various conditions using single **if...else if** statement. The syntax is as follows

```
if(boolean_expression 1)
{
   Statements that are executed when the boolean expression 1 is true
}
else if( boolean_expression 2)
```

```
 {
   Statements executed when the boolean expression 2 is true
}
else if( boolean_expression 3)
{
  Statements  executed when the boolean expression 3 is true
}
else
{
   Statements executed when the none of the above condition is true
}
```

There are several things one must keep in mind when using the **if...else if....else** structure.

1. An **if** can have zero or one **else** and it must come **after** any **else if's**.
2. An **if** can have zero to many **else if**'s and they must come before the **else**.
3. Once an **else if** succeeds, that is the boolean statement corresponding to it is true, then none of the remaining **else if**'s or **else** will be tested.

So for instance, we can use this construct in the program above as follows.

```
#include <stdio.h>
main ()
 {

   int a = 179;

   if( a < 100 )
   {

      printf(”a is less than 100\n” );
   }
   else if (a <150)
   {
     printf(”a is less than 150\n” );
     }
      else if (a <175)
   {
     printf(”a is less than 175\n” );
     }
   else
   {

      printf(”a is not less than 175\n” );
   }

   printf(”value of a is : %d\n”, a);
   }
```

In this case, since the value of the variable *a* is 179, the boolean statement corresponding to **if** and the two **else if**s is false and therefore the statement in the **else** will be executed and we will get " a is not less than 175" and then the "value of a = 179".

Once again, we can always use nested **if... else** statements or even nested **if...else if...else** statements. However, one should be careful that the loops are completely nested.

Finally, there is another kind of decision making structure called the **switch** statement. This used when a variable is to be tested for equality against a list of values. Each value is called a **case**, and the variable being switched on is checked for each **switch case**. The syntax is as follow:

```
switch(expression) {

   case constant-expression  :
      statement(s);
      break; /* This is optional */

   case constant-expression  :
      statement(s);
      break; /* This is optional */

   /* you can have any number of case statements */
   default : /* This is Optional */
   statement(s);
}
```

There are several things to keep in mind when one uses this structure.

1. The expression used in a switch statement must have an integral or enumerated type.
2. One can have any number of case statements within a switch. Each case has to be followed by the value to be compared to and a **colon**.
3. The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant.
4. When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
5. When a **break** statement is reached, the switch terminates, and the program goes to the next line following the switch statement.
6. Not every case needs to contain a break. If no break appears, the program will go through to the subsequent cases until a break is reached.

7. A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

As an example of this statement, consider the following program to convert a set of marks between 0 and 4 into grades. 0 marks is an F grade, 1 is D, 2 is C, 3 is B and 4 is A.

```
#include <stdio.h>

main ()
{


   int marks;
   marks=2;

   switch(marks)
   {
      case 0 :
         printf("Your grade is F\n" );
         break;
      case 1 :
         printf("Your grade is D\n" );
         break;
      case 2:
         printf("Your grade is C\n" );
         break;
      case 3:
         printf("Your grade is B\n" );
         break;
      case 4 :
         printf("Your grade is A\n" );
         break;

      default :
         printf("Invalid grade\n" );
   }

}
```

As you can see, if the value of the switch expression that is marks is 3, the statements corresponding to the case 3 will be executed and we will get " Your grade is C" as the output.

If a statement becomes too long, it can be continued in the next line by using \ . A generalization to functions of many variables is obvious. There are also functions that do not return any value to the calling program but perform many other tasks. In fact, most of the work in C is done through such functions. Examples are **scanf, printf**. Even **main ()** itself is one such function. We will come across many more examples of such functions in future.

N.B. The function subprogram can be defined either before or after the main program. **If it is defined after the main program, it has to be identified in the main program eg.**

```
#include <stdio.h>
#include <math.h>

main ()
{

float f(float x, int n);
    .
}

float f (float x, int n)
{
        y=x;
        return (y);
}
```

1.3.2 Function Prototypes

Every user-defined subroutine/function must be declared in the caller function via a **function prototype.** If a function is to be used in many other functions, its prototype must be declared in each one of them. Alternately, one may make a **global declaration** of the function prototype by declaring it outside all functions. For example

```
      float f1(float x, float y)
      {
        ...
        ...
        ...
 }

      void f2(float x, int i)
      {
float f1(float a, float b);
                ...
                ...
                ...
  }

      main()
      {
```

```
float f1(float x, float y);
            void f2(float a, int j);
            ...
            ...
            ...
    }
```

Here we have two **user-defined functions**, f1 of return-type float depending on two input arguments, both of type float, and f2 of return-type void depending on two input arguments, the first of type float and the second of type void. The prototype of f1 has been declared in f2 as well as in **main**; so the function f1 can be used in both of them. However, the prototype of f2 has been declared only in **main** and not in f1. Hence, f2 can be used only in main. **In general it is best to make a global prototype declaration**:

float f1(float x, float y);
void f2(float x, int n);

immediately after the inclusion of header and other files. Now the two functions can be used in main as well as in all other programs. Note that the type of the function and the type and the number of arguments on which the function depends, must match with the prototype declaration; the names of the arguments can be anything or may not even be included. For example, the function prototype in the above example could have been

float f1(float , float);
void f2(float , int);

Note on function names: **You must always give meaningful function names to user-defined functions to improve the readability of your code. Try to avoid giving single alphabet function names.**

1.4 Operators in C

An operator in C is a symbol that tells the compiler to perform specific mathematical or logical functions. There are several kinds of operators in C like Arithmetic Operators , Assignment Operators, Relational Operators, Logical Operators etc.

1.4.1 Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	remainder after division or modulo division

Table 1.1: **Arithmetic Operators in C**

The meaning of the operators +, - and * computes addition, subtraction and multiplication respectively as we know. However, one has to be careful in division. If both the dividend and divisor are defined as integers, say 6 and 4 that is 6/4 , then the result (which should be 1.5 ordinarily) will be 1 instead since the compiler thinks that this is integer division. If one wants to divide integers, one should first change the typecast as discussed above before dividing. The only unfamiliar operator is %, or modulo division. This is an operator which can only be used by integers and gives us the remainder after the division. Thus, 6%4 will give us 2.

Another useful class of operators is the increment and decrement operators. These are the ++ and the $--$ operators which increase or decrease the value of a variable or constant by 1. Remember that these are so called unary operators and so operate on a single operand. A simple example is

```
int main()
{
    int a = 3, b = 15;
    float c = 1.5, d = 12.5;
    printf(”++a = %d, −−b=%d, ++c =%f, −−d=%f”, ++a,−−b,++c,−−d);
}
```

This program will give us an output of $4, 14, 2.5, 11.5$.

1.4.2 Assignment Operators

As the name suggests, assignment operators are used to assign a value to a variable.

Operator	Usage	Meaning
=	x=y	x=y
+=	x +=y	x=x+y
-=	x -=y	x=x-y
*=	x *=y	x=x*y
/=	x /=y	x = x/y
%=	x %=y	x = x%y

Table 1.2: **Assignment Operators in C**

The usage and the meaning should be clear from the table above.

1.4.3 Relational Operators

A relational operator as the name suggests, checks the relationship between two operands. This operator returns only two values, 1 and 0. If the relation is true, it returns 1 and if the relation is false, it returns value 0.

Operator	Usage	Meaning
==	Equal to	3==10 will return 0
>	Greater than	3>10 will return 0
<	less than	3<10 returns 1
!=	Not equal to	3!=10 returns 1
>=	Greater than or Equal to	3>=10 returns 0
<=	Less than or Equal to	3<=10 returns 1

Table 1.3: **Relational Operators in C**

Relational operators are very useful in decision making and also in loops.

1.4.4 Logical Operators

Logical operators are those which if used in an expression return either 0 or 1 depending on whether the expression is true or false. Once again, these are used frequently in decision making in C.

Suppose $x = 3$ and $y = 10$. Then

Operator	Usage	Meaning
&& (Logical AND)	True only if all operands are true	Then $((x == 3)\&\&(y > 100))$ is equal to 0
|| (Logical OR)	True only if either one is true	Then $((x == 3)\vert\vert(y > 100))$ is equal to 1
! (Logical NOT)	True only if the operand is 0	Then $!(x == 3)$ is equal to 0

Table 1.4: **Logical Operators in C**

1.5 Mathematical Functions in C

These are functions which are in-built in C and are declared in the header file < math.h>. Therefore to use these, we must include this header file.

Wherever the argument in these mathematical functions is a double (double precision variable or constant), it can be replaced by a float variable or constant **but should be avoided.** The single precision counterparts of these functions usually have f appended at the end of their names. For example the single precision counterpart of sin() function declared in $< math.h >$ is sinf() and so on. The choice of using single precision (float) or double precision (double) variables for computation depends on the nature of the problem and accuracy desired. However, if the argument is an integer, it cannot be replaced by a float or double.

The trigonometric functions, the inverse trigonometric and the hyperbolic functions are fairly obvious. **Please remember that the arguments of the trigonometric functions and the results of the inverse functions are always in radians and NOT degrees.** The commonly used trigonometric functions are **sin, cos, tan, asin, acos, atan, sinh, cosh, tanh**. Remember that each of these functions is a type double precision floating point and the arguments are also double precision.

The exponentiation and log functions are **exp,log, log10,log2** where the logarithmic function **log** is log_e while **log10** is log_{10}.

The square root and power functions are **sqrt** and **pow**. The syntax of the **pow** function is $pow(a, b)$ where a and b are both double precision and $a >= 0$. The **sqrt** function has a syntax **sqrt(a)** where $a >= 0$.

The **abs** function returns the absolute value of an **integer**. The **fabs** function returns the absolute value of a floating point number.

The **fmod** function has two double precision variables x, y as its argument. When used, it returns the **floating point remainder of** x/y. Thus suppose $x = 9.6, y = 2.7$ then **fmod(x,y)** is the remainder left by 9.6/2.7 and is equal to 1.5.

Finally, there are two functions which may not be very obvious. These are the **ciel** and **floor** functions which both have single floating point arguments. The **ciel** function returns the smallest integer value not less than the argument while the **floor** function returns the value of the largest integer not greater than the argument. Thus if $x = 9.4$, then **ciel(x)** is 10 while **floor(x)** is 9. Some of the functions that you might use in this course are collected in the table below (Table 1.5).

Function	Description
double cos(double x)	Returns the cosine of a radian angle x.
double acos(double x)	Returns the arc cosine of x in radians.
double cosh(double x)	Returns the hyperbolic cosine of x.
double atan(double x)	Returns the arc tangent of x in radians.
double exp(double x)	Returns the value of e raised to the x^{th} power.
double log(double x)	Returns the natural logarithm (base-e logarithm) of x.
double log10(double x)	Returns the common logarithm (base-10 logarithm) of x.
double pow(double x, double y)	Returns x raised to the power of y.
double sqrt(double x)	Returns the square root of x.
double fabs(double x)	Returns the absolute value of x.
double fmod(double x, double y)	Returns the remainder of x divided by y.
double ceil(double x)	Returns the smallest integer value greater than or equal to x.
double floor(double x)	Returns the largest integer value less than or equal to x.

Table 1.5: **Mathematical Functions in C**

1.6 Storing the results of the program in a data file

Frequently, we would need to store the results of a program in a data file to be used by other programs. Thus, for instance, you might have a program which calculates the orbit of a planet, which will give you the position of a planet at various times. You might want to use this to plot the trajectory of the planet using another graphics program. Or you might have some expression which generates some data and you might want to use another program to do some statistical analysis on the data. C allows you to store the results of the program in a data file as well as to read a data file in a program. We will learn essential features of storing data file in C program. The functions that are used to do this are very similar to the **printf** and **scanf** functions that we have already discussed. The relevant functions are **fprintf** and **fscanf**. Also we need to tell the C program the name of the data file which we want to write to or from which we want the data to be read. The following program will make clear the syntax etc of using these constructs.

```
/* Program for evaluating a function & storing the results  in two
data files , data1.txt  and data2.txt*/

#include <stdio.h>
#include <math.h>

main()
{
        float x,y,z;
        FILE *fp=NULL;
FILE *fp1=NULL;
fp = fopen("data1.txt", "w"); /* open a file handle in write mode */
fp1 = fopen("data2.txt", "w"); /* open a file handle in write mode */
        for(x= 0; x <= 6; x+= 0.1)
{
                y=sin(x);
        z=cos(x);

fprintf(fp, "%f \t %f \n", x, y); /* print line to the file */
fprintf(fp1,"%f\t %f\n",x,z);
        }

        fclose(fp); /* close the file handle */
fclose(fp1);
}
```

COMMENTS

1. Include the header file <stdio.h> in the main program to use FILE pointers in the main program.

2. Open two new data files in “w” (write) mode with a call to **fopen()** function. Other modes of opening a file are “a” for append, “r” for read, etc. One can open as many files as one wants but they should have different names, in this case fp, fp1 etc.

3. In order to store data inside the file, use **fprintf()** function in the manner we used **printf()** earlier for printing the output to the standard output (screen). “\ t” corresponds to a tab space and “\ n” corresponds to new line. Note that **fprintf()** takes the first argument as the file pointer **fp** which points to the file opened earlier in the program.

4. Once data is stored in the file, close the data file using **fclose()** subroutine.

1.7 Examples

We give a few examples of programs to solve simple arithmetic problems. These programs will illustrate the use of most of the concepts discussed above.

Example 1.6

Palindrome Numbers are those numbers which do not change when the order is reversed. Thus $11, 22, 33$ etc. are palindrome numbers as are $121, 313, 212, 12321$ etc. The problem is to write a program to check whether a number is a palindrome number or not.
As with all programs, first one has to think of the underlying algorithm that one would use to solve the problem. Think of this as solving the problem without a computer in a sense. Once we have the algorithm, the next step is to code the algorithm in C to make a program which implements the algorithm.

The algorithm for this is straightforward.

1. Supply the number
2. Reverse the digits
3. Compare with original number
4. If both are the same, then output number is a palindrome number
5. Else output number is not a palindrome number

The program below is one way to implement this algorithm.

```
/* Program to check whether a given number is a Palindrome Number*/
#include <stdio.h>
/*We don't need a math.h file since this program does NOT use any predefined functions*/

main()
{
   int n, rev = 0, tem ;
 /* Note that all operations are on integers and result in integers.*/
 /* So all variables are integer type*/

   printf(”Enter a number to check if it is a palindrome or not\n”);
   scanf(”%d”,&n);

   tem = n;

   while( tem != 0 )
    /* Use of a top-checked while loop along with the logical NOT operator*/
   {
      rev = rev * 10;
      rev = rev + temp%10; /* Use of the remainder operator*/
      tem = tem/10;
       /* Note that since everything is integer, this is an integer division*/
   }
/* Use of an if else structure*/
```

```
    if ( n == rev )
    /* Use of the Relational == operator to check if the two variables are equal*/
       printf("%d is a palindrome number.\n", n);
    else
       printf("%d is not a palindrome number.\n", n);



}
```

Example 1.7

This example shows you how to tell whether a given year is a leap year. Recall that to check for leap year, we must have three criteria

1. The year is evenly divisible by 4;
2. If the year can be evenly divided by 100, it is NOT a leap year, unless;
3. The year is also evenly divisible by 400. Then it is a leap year.

This is the algorithm that we need to implement. One way to do it is as follows:

```
/* Program to check for leap year*/
#include <stdio.h>

main()
{
  int year;

  printf("Enter a year to check if it is a leap year\n");
  scanf("%d", &year);
/* We have to check three conditions. So we will use a if.. else if...else structure*/
  if ( year%400 == 0)
    printf("%d is a leap year.\n", year);
    /* Since there is one statement after the if condition, there is no need for {  }*/
  else if ( year%100 == 0)
   /* Recall that in the if..else if..else construction, if any of the conditions*/
   /* is satisfied, control passes out and no other condition is checked*/
    printf("%d is not a leap year.\n", year);
  else if ( year%4 == 0 )
    printf("%d is a leap year.\n", year);
  else
```

```
    printf(”%d is not a leap year.\n”, year);

}
```

Example 1.8 To find the HCF (Highest Common Factor) or GCD (Greatest Common Divisor) of two numbers. Recall that the HCF of two numbers is the highest or largest common factor of the two. Thus, the HCF of 24 and 16 is 8 since $24 = 8 \times 3$ and $16 = 8 \times 2$. We can implement this in many ways. This is one of the simplest way to do it.

```
/* Simple Program for HCF*/
#include <stdio.h>
main()
{
    int n1, n2, i, hcf;

    printf(”Enter two integers: ”);
    scanf(”%d %d”, &n1, &n2);

    for(i=1; i <= n1 && i <= n2; ++i)
/* note that the condition in the for loop is that */
/*the variable i has to be less than both n1 and n2*/
    {

        if(n1%i==0 && n2%i==0)
        /* we need to check if i is a factor of both n1 and n2*/
            hcf = i;
 /* since i is being incremented in the for loop, */
 /*it will check the divisibility till the smaller of n1 and n2*/
    }

    printf(”HCF of %d and %d is %d”, n1, n2, hcf);

}
```

Example 1.9

A very efficient algorithm to find the HCF of two positive integers is the Euclid algorithm. The algorithm is simple. Let m and n , $m < n$ be the two integers. Then if n is exactly divisible by m, the HCF is m. Else, if p is the remainder of the division of n by m, then repeat the same procedure with the pair (p, m). Keep doing this till either the smaller number in the pairs divides the larger number exactly, in which case the smaller number is the HCF or the remainder is 1 in which case the HCF is1.

The previous program was one way to determine the HCF of two numbers. We now will use Euclid's algorithm in a program with a function.

```
/* Program to find HCF of two numbers using Euclid' algorithm*/
    #include <stdio.h>
    int euclidhcf(int x, int y)
    {
        if (y == 0)
        {
            return x;
        }
        else if (x >= y && y > 0)
        {
            return euclidhcf(y, (x % y));
        }
    }

   main()
    {
      int euclidhcf(int x, int y);
        int num1, num2, hcf;

        printf("\n Enter two numbers to find H.C.F. using Euclidean algorithm: ");
        scanf("%d,%d", &num1, &num2);
        hcf = euclidhcf(num1, num2);
        if (hcf !=0)
            printf("The H.C.F. of %d and %d is %d\n", num1, num2, hcf);
        else
            printf("\n Input Numbers not valid\n");
     }
```

Example 1.10

Suppose one wants to find the number of prime numbers upto a given number n. The following program can be used.

```
/* Program to find the number of Prime Numbers upto a given number n*/
#include <stdio.h>
#include <math.h>

main ()
{
int n,m,i,j,k,flag,np;
float x,xi;
printf("input an integer greater than 3   ");
scanf("%d",&n);
np=0;
for(i=3;i<=n;i+=2)
{
xi=i;
m=sqrt(xi);
j=3;
flag=0;
while(j<=m && flag ==0)
{
x=xi/((float)(j));
k=i/j;
if (x==k)
flag =1;
j+=2;
}
if (flag ==0)
{printf("%d  ",i);
np+=1;
}
}
printf("\n The number of primes upto %d is %d" n,np);
}
```

In the program above a few things are worth noting:

1. The use of integer division and float division. In once case, since i, j, k are all integers, the division of i and j will yield an integer. ON the other hand the division of floating point xi with the integer j (which has been made float), yields a floating point number. Of course, if j divides i exactly without a remainder, then the two operations will yield the same result.
2. The use of logical equal to.

3. The use of a flag which can get incremented in case some condition is true and then checked for its value.
4. Finally, the running of the checking for primes (that is exact divisibility) only till the $\sqrt{xi}$. For any number, it should be obvious that one only needs to check for divisibility till the square root of that number.

☞ ***Some of the common errors that you should avoid.***

1. ***Forgetting to include <math.h>.***
2. ***Forgetting to add -lm while compiling a program with <math.h>.***
3. ***Using a comma instead of a semi colon while writing a for loop***
4. ***Putting a semi colon at the end of a for loop , do, while if statements***
5. ***While dividing a floating point number by an integer, not changing the denominator to float type***
6. ***Using nested loops***
7. ***Forgetting to put " " when writing printf and scanf statements***
8. ***Forgetting to use & while using scanf***
9. ***Not stating the function prototype in the program***
10. ***When using multiple if or for or do loops, making sure that the number of braces {} match, that is the number of opening braces are the same as the closing braces.***

1.8 Advanced Topic: Pointers

Pointers are an essential part of C programming language. They provide a very powerful and compact way to doing programming. In fact, it is because of pointers that C programming language is used extensively. C language uses pointers with arrays, structures and functions. Pointers are use in C to access memory as well as to manipulate the address.

So what exactly is a **pointer**? **A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.** The other variable can be of any type. Thus, we can see that pointers are appropriately named since they 'point' to locations in memory. The best way to think of this is to think of a row of lockers in a room. Now each locker has a number associated with it to identify it. These numbers are like the memory addresses of variables. However, what is inside the locker might vary from locker to locker and this is what corresponds to the value of the variable stored. This much is easy and obvious. But what about a pointer? A pointer in this locker room example would be anything in which the number of another locker is stored. So for instance, you could store the number of your locker in another locker. This other locker, which stores only the number of the actual locker with the valuables in it, would be a pointer.

There is usually some confusion about terms- the term pointer can refer either to a memory address or to the variable that stores the memory address.

So what exactly is a **pointer**? **A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.** The other variable can be of any type. There are two types of operations with pointers:

The **reference operator** & gives the "address of a variable".

The **dereference operator** $*$ gives the "contents of an object pointed to by a pointer".

To declare a pointer to a variable do we write

type *name

Thus we can have pointers like **int *ip** (pointer to an integer), **int *fp** (pointer to a floating point variable) etc. Note that pointer variables always point to the **same** type of data. Thus we cannot declare a variable to be float and then have an integer pointer pointing to it.

```
float x;
int* px;
px=&x;
```

is **wrong** because there is a type mismatch.

You are already familiar with the use of pointers. Suppose one wants to input the value of a floating point variable x. Then we know, we need to write

scanf("%f",&x)

We have used the reference operator & to get the address in the memory. To see the use of this, consider the following program

```
#include <stdio.h>
```

```
int main()
{
    int x;              /* A normal integer*/
    int *p;             /* A pointer to an integer ("*p" is an integer, so p
                           must be a pointer to an integer) */

    p = &x;             /* Assign the address of x to p */
   x=107;
    printf( ”value of x= %d\n”, *p ); /* Note the use of the * to get the value */
  }
```

If one compiles and runs this program, it will give you an output

value of $x = 107$

Note that though we are asking the printf statement to print the value of $*p$, it prints the value stored in x. To understand this, let us go through the code. We start with a regular integer variable x. We then defined a pointer to an integer as p. The next line stores the memory location of x in the pointer since we are using the referencing operator &. In the locker room example, this is like our looking at the number of a locker rather then inside it. We then give a value of x as 107. This is stored obviously in the memory location of x. The printf command asks the machine to print $*p$. Remember that the dereferencing operator $*$ looks at the address stored in the pointer p and goes to that address and fetches the value stored in that address which as we have said before was the address of the variable x. This is like opening one locker (the pointer) finding the number of another locker in it (the variable x) and going to that locker to see what is inside that locker and returning it to the screen.

To see how all of this works, consider the following simple code:

```
#include <stdio.h>
int main()
{
   int* pi;
   int i;
   i=220;
   printf(”Address of i:%d\n”,&i);
   printf(”Value of i:%d\n”,i);
   pi=&i;
   printf(”Address of pointer pi:%d\n”,pi);
   printf(”Content of pointer pi:%d\n”,*pi);
   i=1110;
   printf(”Address of pointer pi:%d\n”,pi);
   printf(”Content of pointer pi:%d\n\n”,*pi);
   *pi=5;
```

```
    printf(”Address of i:%d\n”,&i);
    printf(”Value of i:%d\n\n”,i);

}
```

The output that one gets from running this program is as follows:

```
Address of i: 2686784
Value of i: 220
Address of pointer pi: 2686784
Content of pointer pi: 220
Address of pointer pi: 2686784
Content of pointer pi: 1110
Address of i: 2686784
Value of i: 5
```

Let us try and understand how we get this output.:

1. Declaration of pointer and variable, int *pi creates a pointer pi and int i creates an integer variable i. Note that till now we have not devlared the values of pi and i and therefore pointer pi points to either no address or a random address and the variable i is assigned an address but contains a random value.
2. i=220; assigns 220 to the variable i, i.e.,220 is stored in the memory location of variable i.
3. Now when we print &i, we get a value of the address of i and when we print just the variable i, we get its value.
4. pi=&i; assigns the address of variable to i to the pointer pi.
5. Therefore, when we print the address of the pointer, pi we get the same address as i and when we print the value of pi that is *pi, we get the same value as that of i, that is 220.
6. i=1110; assigns 1110 to variable i.
7. Now when we print the address and value of the pointer pi, we see that the address is the same as that of i and the value is the new value of i that is 1110.
8. *pi=5; changes the contents of the memory location pointed by pointer pi to 5. Since the memory location pointed to by the pointer pi is the same as that of i, when we print the value of i, we get the new value that is 5.

Although in our course, we dont really need to use pointers explicitly, they are extremely useful in advanced programming, especially when one is dealing with linked lists and large amounts of data.

They provide a neat and efficient way for allocation of memory in the machine and are used extensively. We will encounter them again when we study matrices and their manipulation.

1.9 QUESTIONS

1. Can we run a program without having function main()?
2. What header file must be included (#include) if you use printf and scanf statements in your program? What is # in the #include statement?
3. Do we need to put a semicolon ; after #include statements?
4. What do you understand by typecasting? How will you convert an int variable into float in an arithmetic expression?
5. What are the different ways in which a function (other than main()) can be written and called in main() program?
6. Where do we use pointer in a simple code?
7. What will happen if a semicolon is put after the **for** statement? Will the program compile? If yes, then what will be the difference in the output?
8. What will happen if a semicolon is put after the **do** statement? Will the program compile? If yes, what will be output?

1.10 PROBLEMS

1. Make a table of the trigonometric functions $\sin x, \cos x$, and $\tan x$ for $0 \leq x \leq \frac{\pi}{4}$.
2. The function $f(x, y)$ of two variables x, y is defined by

$$\begin{aligned} f(x,y) &= x^2 + y^4 && \text{for } |x| > |y| \\ &= x^2(x^2+1) && \text{for } |x| = |y| \\ &= y^2 + x^4 && \text{for } |x| < |y| \end{aligned}$$

Make a table of the function $f(x,y)$ for $-1.0 \leq x, y \leq 1.0$ at intervals of 0.25 for both x and y.

3. Pythagorean number are those integers which satisfy the relation

$$a^2 + b^2 = c^2$$

Write a program to find all the distinct sets of Pythagorean numbers less than 100.

4. Some integers have a property that they are divisible by the sum of their digits. Thus, 84 is one such number since it is divisible by $8 + 4 = 12$. Let us call them Harshad numbers. Write a program to find all the Harshad numbers between 50 and 70 both inclusive.

5. Fibonacci numbers are those which have the property that each member (apart from the first two) is the sum of the previous two numbers. Thus the Fibonacci numbers are $1, 2, 3, 5, 8, \cdots$. Write a program to generate Fibonacci numbers till 200. Now modify the program to use a function which can be called to generate the Fibonacci numbers till any integer entered by the user.

6. Write a program to calculate the factorial of a given integer. Your program should use a function or subroutine which calculates the factorial.

7. Use the factorial subroutine that you have written for the previous problem to write a program which calculates ${}^nC_r$ and ${}^nP_r$ for given values of n and r. (${}^nC_r = \frac{n!}{(n-r)!r!}$ and ${}^nP_r = \frac{n!}{(n-r)!}$)

8. We know that a quadratic equation of the form $ax^2 + bx + c = 0,\ a \neq 0,\ a, b, c$ real can be solved easily. We also know that the determinant D is given by

$$D = b^2 - 4ac$$

If $D \geq 0$ then the roots are real. Write a program to determine the roots of a quadratic equation. Your program should first determine if $D \geq 0$ and then give the roots. It should report an error if $D < 0$.

Chapter 2

Graphics Using GNUPLOT

2.1 Introduction

GNUPLOT is a command-line program that can generate two- and three-dimensional plots of functions and data. The program runs on all major computers and operating systems (Linux, UNIX, Windows, Mac OSX...).

The software is copyrighted but freely distributed (i.e., you don't have to pay for it). It was originally intended to function as a software for plotting mathematical functions and data but has outgrown its credentials. It now supports many non-interactive uses, including web scripting and integration as a plotting engine for third-party applications like Octave. Gnuplot has been supported and under development since 1986.

Gnuplot supports many types of plots in either 2D and 3D. It can draw using lines, points, boxes, contours, vector fields, surfaces, and various associated text. It also supports various specialized plot types.

Gnuplot supports many different types of output: interactive screen terminals (with mouse and hotkey functionality), direct output to pen plotters or modern printers (including postscript and many color devices), and output to many types of graphic file formats (eps, fig, jpeg, LaTeX, metafont, pbm, pdf, png, postscript, svg, ...). Gnuplot is easily configurable to include new devices.

2.2 Plotting with inbuilt functions of GNUPLOT

We will first demonstrate gnuplot using built-in functions.

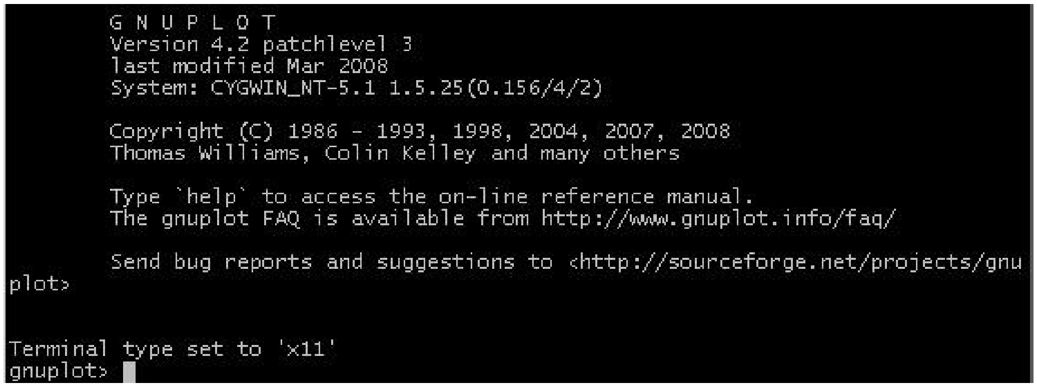
2.2.1 Interactive plotting

In the following example, we plot sin(x) between $-2\pi < x < 2\pi$ with labels on x and y axis.

Open a terminal and type at the prompt

$gnuplot

You will see a screen like this



```
        G N U P L O T
        Version 4.2 patchlevel 3
        last modified Mar 2008
        System: CYGWIN_NT-5.1 1.5.25(0.156/4/2)

        Copyright (C) 1986 - 1993, 1998, 2004, 2007, 2008
        Thomas Williams, Colin Kelley and many others

        Type `help` to access the on-line reference manual.
        The gnuplot FAQ is available from http://www.gnuplot.info/faq/

        Send bug reports and suggestions to <http://sourceforge.net/projects/gnu
plot>


Terminal type set to 'x11'
gnuplot> 
```

Figure 2.1: **Screen Shot of opening GNUPLOT**

On the gnuplot prompt type the following lines one by one (self explanatory)

```
gnuplot> set xlabel ‘x’
gnuplot> set ylabel ‘sin(x)’
gnuplot> set time
gnuplot> set grid
gnuplot> plot [-2*pi : 2*pi] sin(x) title “Sine Wave” with linespoints
```

You will get a plot like this

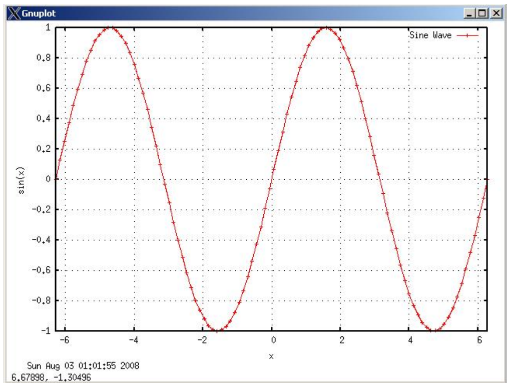


Figure 2.2: **Screen Shot of Sin(x)**

To turn off the grid, you can “unset grid”, to turn off the xlabel, you can type “set xlabel ’ ’ ”. Type **set** at the gnuplot prompt to see all of the options you can turn on and off. To turn on auto-scaling (without any ‘x’ or ‘y’ labels: default), type “set auto” at gnuplot> prompt.

Note that many of the gnuplot keywords including: **using, title**, and **with** can be abbreviated with a single alphabet as: u, t and w but should be avoided by beginners. Also, each line and point style has an associated number.

In order to draw two plots on top of each other, you can replace the last line by

```
gnuplot>plot [-2*pi : 2*pi] sin(x) t “Sine Wave” with linespoints, cos(x) t “Cosine Wave” with linespoints
```

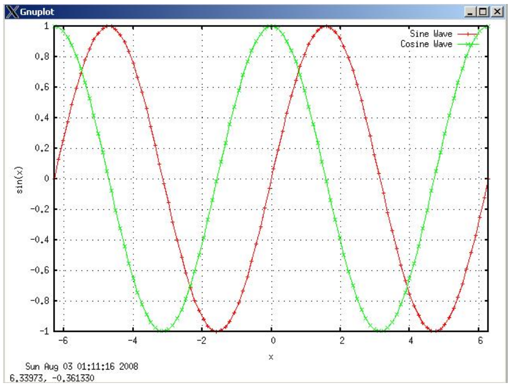


Figure 2.3: **Screen Shot of Sin(x) & Cosine(x)**

You can exit from gnuplot by typing “exit” (or “quit”) on the gnuplot prompt.

```
gnuplot>exit
```

2.3 Saving Plots

To save the above image as an enhanced postscript file with “.eps” extension, instead of displaying it to the screen, enter the following commands:

```
gnuplot> set terminal postscript enhanced color solid 22
gnuplot>set output ‘plot.eps’
gnuplot>set xlabel ‘x’
gnuplot>set ylabel ‘y’
gnuplot> plot [-2*pi : 2*pi] sin(x) title “Sine Wave” with linespoints
gnuplot>set term x11
```

This will create a postscript image file called “plot.eps” of the previous plot. It will be placed in the same folder in which you are working. You can then use any postscript viewer program like “gv” (or “evince”) to open your saved graphics file. Remember, the plot will not appear on the screen when you redirect the terminal type to postscript (first line of the example above), so it may appear as if nothing has happened. Exit from gnuplot prompt, and then type on the terminal

```
$gv plot.eps &
```

Note: once you have finished plotting to a file, you need to set the terminal type back to x11 (last line of the previous example) so as to view subsequent plots on the screen.

2.4 Plotting using script

You can also plot and save above-mentioned example by writing all the commands in a script. When this script is executed in gnuplot, each line is executed in sequence beginning from the top. To do that, open your favourite text editor (gedit, emacs, vi, nedit etc) and then type following lines (exactly the same which you typed interactively at the gnuplot> prompt):

```
set terminal postscript enhanced color solid 22
set output ‘plot1.eps’
set xlabel ‘x’
set ylabel ‘y’
plot [-2*pi : 2*pi] sin(x) title “SineWave” with linespoints
```

set term x11

and save it to the gnuplot script file **plotscript.p** Finally, to execute this script using gnuplot, on the terminal prompt, type

$**gnuplot ./plotscript.p**

This will create .eps file **plot1.eps**. You can directly view this file using gv as in the above example.

2.4.1 Customization

Customization of the axis ranges, axis labels, and plot title, as well as many other features, are specified using the set command. Specific examples of the set command follow. (The numerical values used in these examples are arbitrary.) To view your changes type: **replot** at the gnuplot> prompt at any time.

Action	**Command**
Create a title:	> set title "Force-Deflection Data"
Put a label on the x-axis:	> set xlabel "Deflection (meters)"
Put a label on the y-axis:	> set ylabel "Force (kN)"
Change the x-axis range:	>set xrange [0.001:0.005]
Change the y-axis range:	> set yrange [20:500]
Have Gnuplot determine ranges:	> set autoscale
Put a label on the plot:	> set label "yield point" at 0.003, 260
Remove all labels:	> unset label
Plot using log-axes:	> set logscale
Plot using log-axes on y-axis:	> unset logscale; set logscale y
Change the tic-marks:	> set xtics (0.002,0.004,0.006,0.008)
Return to the default tics:	>unset xtics; set xtics auto

Other features which may be customized using the set command are: arrow, border, clip, contour, grid, mapping, polar, surface, time, view, and many more. The best way to learn is by reading the on-line help information, trying the command, and reading the Gnuplot manual.

FUNCTION	**RETURNS**
abs(x)	absolute value of x, $\|x\|$
acos(x)	arc-cosine of x
asin(x)	arc-sine of x
atan(x)	arc-tangent of x
cos(x)	cosine of x, x is in radians.
cosh(x)	hyperbolic cosine of x, x is in radians
erf(x)	error function of x
exp(x)	exponential function of x, base e
inverf(x)	inverse error function of x
invnorm(x)	inverse normal distribution of x
log(x)	log of x, base e
log10(x)	log of x, base 10
norm(x)	normal Gaussian distribution function
rand(x)	pseudo-random number generator
sgn(x)	1 if $x > 0, -1$ if $x < 0, 0$ if $x = 0$
sin(x)	sine of x, x is in radians
sinh(x)	hyperbolic sine of x, x is in radians
sqrt(x)	the square root of x
tan(x)	tangent of x, x is in radians
tanh(x)	hyperbolic tangent of x, x is in radians

Table 2.1: **Some Inbuilt Functions in Gnuplot**

Bessel, gamma, beta, gamma, and gamma functions are also supported. Many functions can take complex arguments. Binary and unary operators are also supported. The supported operators in Gnuplot are the same as the corresponding operators in the C programming language, except that most operators accept integer, real, and complex arguments. The ** operator (exponentiation) is supported as in FORTRAN. Parentheses may be used to change the order of evaluation. The variable names x, y, and z are used as the default independent variables.

2.5 Plotting using data from a file

Create a data file, data1.txt, using your favourite text editor (say, gedit or emacs) and enter the following data (there are three columns)

$\#n$	n^2	n^3
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Again start gnuplot in a terminal by writing “gnuplot” on the terminal prompt and type the following line

```
gnuplot> plot “data1.txt” u 1:2 with linespoints,“data1.txt” using 1:3 with linespoints
```

gnuplot ignores lines starting with # (comment lines.) Also, you can combine any number of plots in one figure. Thus, in this example, we have plotted n vs n^2 and n vs n^3 by the command **u 1:2** and then again **u 1:3**. It should be clear that the 1 refers to the first column of the data file and the 2 and 3 refer to the second and third column. Similarly, one can plot data from different data files in this manner. The above command will create a plot like that in Fig 2.4.

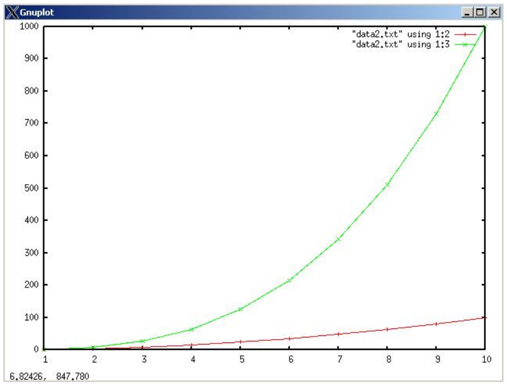


Figure 2.4: **Screen Shot of Plot using data file**

You can modify these plots again using the x and y-labeling.

As we discussed in the previous chapter, one can store the results of any program in a data file by opening a file and writing to it using the command **fprintf**. Once we have the data file, then one can use it as above to plot the results in gnuplot.

You can also save the above plot in a .eps file using the following commands:

```
gnuplot>set terminal postscript enhanced color solid 22
gnuplot>set output ‘plots2.eps’
gnuplot>plot “data1.txt” using 1:2 with linespoints, “data1.txt” using 1:3 with linespoints
gnuplot> set terminal x11
```

Exit the gnuplot and then type **gv plots2.eps** on the terminal command prompt to see the output.

2.6 Periodic Function

Very often we need to plot a function which is periodic: $y = f(x) = f(x+T)$, where T is the period of the function, over a range which covers many periods. The function may be given in an analytic form over one period, say, from $x = 0$ to $x = T$. For example the periodic function

$$\begin{aligned} f(x) &= 1 \quad \text{for } 0 \leq x \leq \pi \\ f(x) &= -1 \quad \text{for } \pi \leq x \leq 2\pi \\ f(x+2\pi) &= f(x) \end{aligned}$$

when plotted in the interval $(-6\pi, 6\pi)$ will generate a plot like the one shown in Fig 2.5.

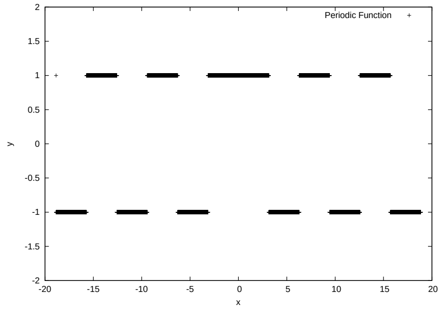


Figure 2.5: **Screen Shot of Plot of periodic function**

How does one plot such a function? We only know the values in the interval $-2\pi \leq x \leq 2\pi$ as given above. Suppose we wish to find the value at $x = 4.7\pi$. We know from the definition of the function that it is periodic with a period 2π. This means it repeats itself exactly after that interval. So we need to find the number of complete intervals of length 2π in this. This is easy to do. Recall from the function **int** which when operating on a floating point number returns the integer value of the number. So if we use $n = \textbf{int}\left(\frac{4.7\pi}{2\pi}\right)$ we will get $n = \textbf{int}\ (2.35) = 2$. This means that there are 2 complete periods from 0 to 4.7π. Now if we take the quantity $x1 = 4.7\pi - 2\pi n$, its value will be $x1 = 0.7\pi$ which is within the range where the function is defined. In this case, since $x1 < \pi$, we have the value $f(x) = 1$. This way we can find the value at any arbitrary point. The program to implement this algorithm can be as follows:

```
#include<stdio.h>
#include<math.h>
#define pi 3.14159

main()
{
  float x,y,z,x1; int n;
   FILE *fp=NULL;
  fp=fopen("res1.txt","w");
  for(x=-(6*pi);x<=(6.0*pi);x=x+0.001*pi)
    {n = (int)(x/(2.0*pi));
    x1 = x - 2.0*pi*n;
    z=fabs(x1);
      if(z >=0 && z <pi)
       {y=1;}
      if(z>= pi && z < 2*pi)
      {y=-1;}
```

```
        fprintf(fp,”%f  \t  %f\n”,x,y);
        }
    }
```

Note the use of **fabs** in the above program since the range of x is from -6π to 6π. Also note the use of the operation **(int)** which just takes the integer part of the argument, which in this case is $\frac{x}{2\pi}$.

After this program is run, you will have a file called **”res1.txt”** in your directory. Now if you can plot the data in the file using **gnuplot**.

gnuplot >**set terminal postscript**
gnuplot >**set output ‘periodic1.eps’**
gnuplot >**set xlabel ‘x’**
gnuplot >**set ylabel ‘y’**
gnuplot >**set yrange** [-2:2]
gnuplot >**plot “res1.txt” u 1:2 title “Periodic Function”**
gnuplot >**set term x11**

This set of commands in **gnuplot** will generate a file **periodic1.eps** with the title **Periodic Function** in your directory. If you now open the file, you will see a plot like that in Figure 2.5.

☞ ***Some of the common errors & good practices while using gnuplot***

1. ***In the plot statement, the name of the datafile to be plotted should always be in “ ”.***
2. ***Before you plot any data from a file, make sure you check the file and confirm that it has the data that you expect. Thus, for instance, in case there is some error in your program, it might give very large or very small numbers or even NAN (Not A Number) which is typically the output when there is a division by zero.***

3. ***Sometimes, because of an incorrect use of a loop structure, either your program goes into an infinite loop, or more frequently will generate a huge datafile. Please remember that a typical page of text (which is what a datafile is) has a size of 1 kilobyte. Thus, if your datafile is of size much larger than a few kilobytes, there is something wrong. In that case, please check and correct your program.***
4. ***Always first see the plot without any extra labels, titles etc. on your screen. If it is of the form that you expect, then only go and add all the extra things like labels, titles, legends etc and save as .eps file.***
5. ***Do not use very long and complicated names for your datafiles or the graphic files (.eps) since then there is a chance you will make a mistake when trying to enter the names. Short, descriptive names are best. Thus, for example, if you are plotting the graph for Problem No. 3 in the graphics chapter, then the program name can be graphprob3.c, the eps file can be graphprob3.eps and the datafile can be a standard data.dat. Remember, that just as there is no need to store and save the output or executable (.out) files for each program, similarly there is no need to store the data files with different names. One can always generate them when needed.***

2.7 PROBLEMS

1. To learn more about how to use gnuplot and some of the concepts from the previous chapter, try to make the following figures in gnuplot. You will have to write the programs to generate and save the data files for these figures and then plot those data files in gnuplot.

   (a) A right angled triangle with vertices at $(0,0),(4,0)$ and $(4,3)$.

   (b) A circle of radius 3 centered at $(5,5)$.

   (c) A box with vertices at $(1,1),(5,1),(5,5)$ and $(1,5)$.

2. You are familiar with Lissajous figures. These are plots that one obtains when one superimposes two perpendicular harmonic motions of different phases and amplitudes. Thus, for instance,

$$x = \sin\theta, \quad y = A\sin(n\theta + \delta)$$

Plot the following functions for θ in the range $0 \le \theta \le 4\pi$ and the parameters A, n and δ given by

(a) $\delta = \frac{\pi}{4}, A = 1$ and $n = 2, 2.5, 3$ ($n = 1$ gives an ellipse. Check.)

(b) $\delta = \frac{\pi}{4}$, $n = 2$ and $A = 0.5, 1, 2$

(c) $n = 2, A = 1$ and $\delta = \frac{\pi}{4}, \frac{\pi}{2}, \pi$.

3. The periodic function $y(x)$ with period 2π is defined as:

$$\begin{aligned} y &= x \qquad 0 \le x < \pi \\ &= 2\pi - x \quad \pi \le x < 2\pi \end{aligned}$$

Plot this function from $x = -6\pi$ to $x = 6\pi$.

4. Plot $|\Theta_{lm}(\theta)|^2$, the square modulus of the orbital wave function for $l = 3, m = 0, \pm1, \pm2, \pm3$. The values of $|\Theta_{lm}(\theta)|$ are given by

$$\begin{aligned} \Theta_{3,0}(\theta) &= \frac{3\sqrt{14}}{4}\left(\frac{5}{3}\cos^3\theta - \cos\theta\right) \\ \Theta_{3,\pm1}(\theta) &= \frac{\sqrt{42}}{8}\sin\theta\left(5\cos^2\theta - 1\right) \\ \Theta_{3,\pm2}(\theta) &= \frac{\sqrt{105}}{4}\sin^2\theta\cos\theta \\ \Theta_{3,\pm3}(\theta) &= \frac{\sqrt{70}}{8}\sin^3\theta \end{aligned}$$

This is a polar plot ($r = |\Theta_{lm}(\theta)|^2, \theta$). You should plot x against y for $0 < \theta < 2\pi$ where

$$x = |\Theta_{lm}(\theta)|^2\cos\theta$$

$$y = |\Theta_{lm}(\theta)|^2\sin\theta$$

5. Spherical Bessel functions $j_n(z)$

where

$$\begin{aligned} j_0(z) &= \frac{\sin(z)}{z} \\ j_1(z) &= \frac{\sin(z)}{z^2} - \frac{\cos(z)}{z} \end{aligned}$$

and the rest can be obtained by the recurrence relation

$$j_{n-1}(z) + j_{n+1}(z) = (2n+1)\frac{j_n(z)}{z}$$

. Using this recurrence relation, plot on the same graph, the spherical Bessel functions in the range $0 \le z \le 5$ at intervals of 0.01 for $n = 0, 1, 2, 3, 4, 5$.

Chapter 3

Finite & Infinite Series

3.1 Introduction

A commonly encountered problem in Physics is the numerical evaluation of mathematical functions like the exponential, circular, hyperbolic or hypergeometric function for given values of their arguments and up to some specified accuracy. Most of these functions can be represented by infinite series, infinite products or continued fractions. Some examples of series representation of a function are given below:

$$
\begin{aligned}
e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots && (3.1)\\
J_n(x) &= \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{-\left(\frac{x^2}{4}\right)^k}{k!(n+k)!} && (3.2)\\
\log(1+x) &= \sum_{j=1}^{\infty} (-1)^{j-1} \frac{x^j}{j} && (3.3)
\end{aligned}
$$

We describe below some numerical methods for evaluating both finite and infinite series. [Though representation of a function in terms of continued fractions and infinite products is also quite useful, we shall not discuss these here.]

3.2 Finite Series

Before looking at the evaluation of infinite series, let us see how to evaluate finite series since the case of infinite series will be an extension as we shall see. We take the example of the following series with a total of $(n+1)$ terms:

$$S_n(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \tag{3.4}$$

Here each term is of the form $\frac{x^i}{i!}$; with $i = 0, 1, 2, \cdots, n$. As long as n is a small number, there is no problem and we can actually evaluate each term and then sum them up. However, if we wish to find the sum of this series for large n, say $n = 20$, there is a serious problem - the computer cannot handle "large" numbers and 20! is a "very large" number ($\sim 2.4 \times 10^{18}$). So clearly we need to find another way to summing of series with very large or very small terms.

We overcome this problem by not evaluating individual terms of the series. Instead we find the ratio of two consecutive terms, t_i and t_{i-1}. Suppose this ratio is R. Then $t_i = Rt_{i-1}$. Since R is usually a small number, it is possible to find all the terms, given the first term t_0, by assigning to i the values $1, 2, 3 \cdots$. By adding these terms we get the required sum.

In the specific example of the series Eq.(3.4) above, we can easily see that

$$t_i = \frac{x^i}{i!}$$

$$t_{i-1} = \frac{x^{i-1}}{(i-1)!}$$

Therefore

$$R = \frac{t_i}{t_{i-1}} = \frac{x}{i}$$

Thus, starting with $t_0 = 1$, we get

$$i = 1 \quad t_1 = Rt_0 = x$$

$$i = 2 \quad t_2 = Rt_1 = \frac{x}{2}x = \frac{x^2}{2}$$

$$i = 3 \quad t_3 = Rt_2 = \frac{x}{3}\frac{x^2}{2} = \frac{x^3}{3 \times 2} = \frac{x^3}{6}$$

and so on. We then define a quantity called the **$\mathbf{j^{th}}$ partial sum** S_j as

$$S_j = \sum_{i=0}^{j} t_i$$

Note an interesting property of this quantity. Any partial sum is by definition the sum of the previous partial sum and the term itself. Thus

$$S_5 = \sum_{i=0}^{5} t_i = \left(\sum_{i=0}^{4} t_i \right) + t_5 = S_4 + t_5$$

This is a property we can use to sum the series iteratively. Thus, the algorithm for summing a finite series to a given number of terms is simple.

1. Find t_0 or t_1, the first term of the series.
2. Find R, the ratio of the i $^{\text{th}}$ term.
3. Find S_0 or S_1, the first partial Sum.
4. From t_0 or t_1 and R, find the next term.
5. Add the next term to the first partial Sum to get the second partial Sum.
6. Repeat this process till we get the required partial Sum which is the Sum of the finite series.

The following program can carry out this process:

Program ser1

```
/* program for evaluating a finite series */
#include <stdio.h>
#include <math.h>

main()
{ float x,t,s;
  int n,i;
  printf("supply x and the number of terms n \n");
  /* if n=20, the last term is $x^{20}  /19!$ */
  scanf("%f,%d",&x,&n);
  s=1.0; t=1.0;  /* Initial values of  sum s and the first term t */
  /* The following loop evaluates the terms and sums them */
  for (i = 1; i < n; i++) /* i starts at 1 ,t_0 term is the initial value*/
  { t* = x/i;   s+ = t; } /* x/i is simply the ratio R */
    printf("\n");
    printf("x=%6.2f,n=%d,sum= %12.5e",x,n,s);   }
```

In this program, the statement $s+ = t$ generates the partial sums $S_2(x), S_3(x), \cdots$ while $t* = \frac{x}{i}$ generates the successive terms for $i = 1, 2, \cdots, n$.

☞**Note that the order of the statements $t* = \frac{x}{i}$ and $s+ = t$ is important. What happens if they are interchanged? Note also that the initialization $s = 1.0$ and $t = 1.0$ must be done outside the loop over i. What happens if these are done within the loop?**

Of course, instead of taking the ratio of t_i and t_{i-1}, we could also take the ratio of t_{i+1} and t_i. In this case,

$$t_{i+1} = \frac{x^{i+1}}{(i+1)!}$$

$$t_i = \frac{x^i}{i!}$$

Thus,

$$R = t* = \frac{t_{i+1}}{t_i} = \frac{x}{i+1}$$

Note here that i will now start from 0 and the first term is $t_0 = 1$. These two methods are equivalent provided we take care of the initializations of i and t.

In the above program the statement:

printf(”\n”);

shifts the cursor to a new line. This device is used to make the output more readable. If necessary more than one such statements can be included. In the statement

printf(”x=%6.2f,n=%d,sum= %12.5e”,x,n,s);

the sum s is printed in **e-format** with a total field of 12 characters with five places after the decimal. The following examples of numbers expressed in the e-format illustrate its use:

Number	e-format
-.00234	-2.3400e-03
5643.1	5.64310e+03
1.5648	1.56480e+00

All these numbers have a total field of 12 characters, including the signs before the number and after e.

Run the program given above for various values of n (for a fixed x) and for various values of x (for a fixed n) and study the behaviour of $S_n(x)$ as a function of x and n. Print your results in the form of a table and also display them in the form of graphs $S_n(x)$ vs. x for fixed n, or $S_n(x)$ vs. n for fixed$x$.

You would have noticed by now that to obtain the results for a new set of input parameters (x and n in this case), you have to run the program again. This is an unnecessary irritant and can be avoided by introducing one more loop outside the loop over i. The outer loop can prompt you to feed new values of n or x or both. Look at the following program:

Program ser2

```
/* program for evaluating a finite series */
#include <stdio.h>
#include <math.h>

main()
{float x,t,s;
 int n,i;
 printf("\n supply x and the number of terms n \n");
 scanf("%f,%d",&x,&n);
 do
 {s=0.0;t=1;
  for (i=2;i<=n;i++)
  {s+=t; t*=x/(i-1);  }
   printf("\nx=%6.2f n=%d sum= %12.5e",x,n,s);
   printf("\n\n");
   printf("enter x and n (n -ve to break the loop)\n");
   scanf("%f,%d",&x,&n);
 }
 while (n>0);   }
```

Notice that after this program has run for one set of values of x and n and printed the result, it prompts you to input new values of x and n. It also tells you that by feeding zero or a negative value for n you can break the loop and come out of it, since the loop will run only for positive values of n. This is indicated by the while statement at the end of the **do** loop. **Remember that in C or C++, a do loop must end with a while condition.** Also note that while the initialization $s = 0.0$; $t = 1.0$; is outside the **for** loop, it is inside the **do** loop. WHY?

If you wish to plot $S_n(x)$ against n (for a fixed x) you can do so by modifying the above program. Store variables of your program in a .txt file and then use gnuplot to plot that data. But now you will have to change n regularly. A better way to plot would be to replace the **do-while** loop by a **for** loop in which n changes by a fixed step.

3.3 Infinite Series

Whereas a finite series can always be summed in principle, the sum of an infinite series has a meaning only if the series is convergent. So it must be ensured that the series under consideration is indeed convergent before one embarks on its evaluation. For finite series, the number of terms to be summed is given in advance. This is the reason that we can use a **for loop** to evaluate the sum as you would remember from Section 1.2.2.1. However, in the case of an infinite series, obviously an infinite number of terms cannot be summed. So how do we sum an infinite series? The answer lies in the fact that if the series is convergent, then by definition it means that adding more and more terms to the partial sum, changes the partial sums by smaller and smaller amounts. Thus, if we decide that we want the sum of an infinite series to a **given accuracy**, then we can stop adding the sums. In effect, what we are doing is actually summing again a finite series though here we do not before hand how many terms we need to some to achieve the desired accuracy.

To illustrate, consider the simple case of $\sin(x)$. We know that this function can be written as an infinite series,

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!} \tag{3.5}$$

Following the exact same method as above for a finite series, we can see that the k^{th} term is

$$t_k = (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

while the $(k-1)^{th}$ term is

$$t_{k-1} = (-1)^{k-1} \frac{x^{2k-1}}{(2k-1)!}$$

Thus the ratio

$$R = \frac{t_k}{t_{k-1}} = -\frac{x^2}{(2k+1)(2k)}$$

Clearly the first term, t_0 is x. What about s? The initial partial sum is obviously the initial term. Thus the initial values are $t = x = s, k = 1$.

We can write a program to sum this series to any number of terms for a given value of x, say $x = \frac{\pi}{4}$. We know that the result of $\sin(\frac{\pi}{4}) = .7071$. The program below will evaluate the series upto increasing number of terms till 10. For each term, we will print the value of that term and the partial sum.

```
#include<stdio.h>
#include<math.h>
#define pi 3.14159

main()
{
  float x,t,sum,z,acc=0.00001; int i;
   FILE *fp=NULL;
  fp=fopen(”res.txt”,”w”);
x=pi/4.0;
       sum = x; t = x ;i=1;
        for(i=1;i<=10;i++)
         {            t*=-x*x/((2*i+1)*2*i);
          sum+=t;
           fprintf(fp,”%d \t %f \t %f \t sin(x)=%f\n”,i,t,sum,sin(x));
         }
       }
```

The results of this are in the Table 3.1.

n	t_n	S_n	$\sin(\frac{\pi}{4})$
2	-0.080745	0.704652	sin(x)=0.707106
3	0.002490	0.707143	sin(x)=0.707106
4	-0.000037	0.707106	sin(x)=0.707106
5	0.000000	0.707106	sin(x)=0.707106
6	-0.000000	0.707106	sin(x)=0.707106
7	0.000000	0.707106	sin(x)=0.707106
8	-0.000000	0.707106	sin(x)=0.707106
9	0.000000	0.707106	sin(x)=0.707106
10	-0.000000	0.707106	sin(x)=0.707106

Table 3.1: **Finite Series Sum for** $\sin(\frac{\pi}{4})$

As you see, this being a very rapidly converging series, after the first four terms, the partial sum really does not change and so adding more and more terms will not help. So instead of adding up a large number of terms, we can add a few terms and get the desired result. Of course, the successive terms after the $n = 5$ are not really zero but very small numbers which are being evaluated to zero because the variable defined is a single precision floating point variable.

So the question is how does one know when to stop adding more and more terms? Or what is the same thing, how do we check for the desired level of accuracy? Clearly, what we see from the example above is that if the relative value of the term to be added to a partial sum is very small compared to the partial sum itself, then it will not change the partial sum significantly. Thus the quantity that one would want to evaluate and see if it is small enough is

$$\textbf{accuracy} = |\frac{t_n}{S_{n-1}}|$$

If this quantity is smaller than some pre-determined amount, then we can safely terminate the summation.

One measure of accuracy can be in terms of the number of decimal places up to which the result is required to be correct. A better method of defining accuracy, and the one we normally use, is in terms of the number of **significant figures** up to which the result is required to be accurate. (Why is this method better?) In either case, the number of terms needed to obtain the desired accuracy is determined during the process of evaluation itself and is not known in advance. We have already seen in our discussion of **Loops in C** that if we don't know the number of iterations, we need to use a **do-while** loop instead of a **for** loop.

The process of addition of terms continues as long as the magnitude of relative contribution of the term to be added, i.e., —term/sum— remains larger than the desired accuracy. The strategy for finding successive terms remains the same.

☞ **Significant Figures: RULES**

1. **The leftmost NONZERO digit is ALWAYS the MOST significant digit.**
2. **When there is NO decimal point, the rightmost NONZERO digit is the least significant digit.**
3. **In case of a decimal point, the rightmost digit is the least significant digit EVEN IF IT IS A ZERO.**
4. **The number of digits between the most and least significant digits are the number of significant digits.**

Thus for instance, $22.00, 2234, 22340000, 2200.$ all have four significant digits. When one is adding, subtracting, multiplying or dividing numbers, then the result should be quoted with the least number of significant figures in any one of the quantities being used in the operation of adding, multiplying etc. In your intermediate calculations, always keep **ONE MORE** significant digit than is needed in the final answer. Also when quoting an experimental result, the number of significant figures should be one more than is suggested by the experimental precision.

Two things that need to be always avoided are
Writing more digits in an answer (intermediate or final) than justified by the number of digits in the data.
Rounding-off, say, to two digits in an intermediate answer, and then writing three digits in the final answer.

While dropping off some figures from a number, the last digit that one keeps should be rounded off for better accuracy. This is usually done by truncating the number as desired and then treating the extra digits (which are to be dropped) as decimal fractions. Then, if the fraction is greater than $\frac{1}{2}$, increment the truncated least significant figure by one. If the fraction is smaller than $\frac{1}{2}$, then do nothing. If the fraction is exactly $\frac{1}{2}$, then increment the least significant digit only if it is odd.

Now we are ready to write the program to sum the infinite series for $\sin(x)$ in the range $0 \leq x \leq 2$ upto a desired accuracy. The program can ask you what level of accuracy you would want. The program will be like the one given below.

Program inf ser1

```
/* program for evaluating an infinite series for Sin(x) */
     #include <stdio.h>
#include <math.h>
main()
{
   int i;
   float t,s,x, acc;
   float PI=4.0*atan(1.0);
printf( ”Enter the value of the accuracy desired\n”);
scanf(”%f”,&acc);
for(x=0;x<=2*PI;x+=0.1)
{
          i=1;
           s=t=x;
          do
          {
             t*=-x*x/((2*i+1)*2*i);
             s+=t;
             i+=1;
          } while (fabs(t/s) > acc);
          printf(”x=%f   sin(x)=%f\n”,x,s);}

}
```

This program will print a table of x and $\sin(x)$ as evaluated from the infinite series for $\sin(x)$. You can of course print the results into a file and then plot it using **gnuplot**. The above program is the prototype of all such programs to evaluate convergent infinite series.

One can of course repeat the whole exercise with the ratio of $\frac{t_{k+1}}{t_k}$. Then the initialisation of k will be from 0 as should be obvious. Also the ratio R will be different. One can easily check that the ratio is then

$$R = \frac{t_{k+1}}{t_k} = -\frac{x^2}{(2k+3)(2k+2)}$$

Of course results we will get from using either of these methods will be same.

☞ ***Some of the common errors & good practices***

1. ***Before attempting to write the program to evaluate the sum of an infinite series, always start with calculating the k th and the (k+1) th from which you can determine the common ratio.***
2. ***When you have the expressions for these terms, check by putting in the values of k to see if they indeed give the series that you want to sum.***
3. ***Make sure that the initialisation, that is the initial values of k, s, t are correct.***
4. ***Make sure that the order of incrementing t and s is correct.***
5. ***Some series might involve a factorial function which needs to be evaluated in the program. For this purpose, you can use the facotrial function that you have written in the problems of Chapter 1.***

3.4 Questions

1. Why do we evaluate ratio of consecutive terms, and not individual terms while evaluating sum of a series?
2. What is the necessary condition to sum up an infinite series?
3. What are the possible ways of defining accuracy while summing up an infinite series? Which one is better and why?

4. Which loop do you generally use for summing up a finite series? What about infinite series?
5. Compare the behaviour of some simple functions at large x (like Prob.2 and Prob.3) and explain the discrepancy.

3.5 Problems

1. Write a program to evaluate the sum up to 20 terms of the series

$$1 + \frac{1}{x^2} + \frac{1}{x^3} + \frac{1}{x^4} + \cdots$$

   for a given $x(0 \leq x \leq 2)$, and compare your result with the analytic sum of the series.

2. Evaluate $\cos(x)$ using the series

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

   accurate to four significant places. Plot $\cos(x)$ vs x in the range $0 \leq x \leq \pi$.

3. Write a program to evaluate $J_n(x)$ to an accuracy of four significant figures using the following series expansion:

$$J_n(x) = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{x^2}{4}\right)^k}{k!(n+k)!}$$

   Plot $J_n(x)$ against x for $0 \leq x \leq 10$ and $n = 0, 1, 2$. Compare with the known behaviour of these functions and explain the discrepancy at large x.

4. . Evaluate $F(z)$ given by

$$F(z) = \cos\left(\frac{\pi z^2}{2}\right) \sum_{n=0}^{\infty} \frac{(-1)^n \pi^{2n} z^{4n+1}}{1.5.9.\cdots.(4n+1)}$$

   correct to four significant figures, for $0 \leq z \leq 1$, at intervals of 0.1.

5. Write a program to plot the sum of the following series:

$$f(z,n) = \sum_{k=0,2,4}^{\infty} \frac{z^k}{2^{n-k} k! \Gamma\left(\frac{1}{2} + \frac{n-k}{2}\right)}$$

for $n = 2$ and z in the range $0 \leq z \leq 5$. You would require the following relations:

$$\Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$$

$$\Gamma(z+1) = z\Gamma(z)$$

6. Write a program to plot the following function:

$$f(z) = C\left(1 + \frac{z^3}{3!} + \frac{1.4z^6}{6!} + \frac{1.4.7z^9}{9!} + \cdots\right)$$

where $C = 0.35503$, for z in the range $-10 \leq z \leq 0$, at intervals of 0.05.

Chapter 4

Root Finding

4.1 Introduction

Assume that a function $f(x)$ is continuous within the interval (x_1, x_2) on the real line. A point x_0 within this interval is said to be a root of the equation $f(x) = 0$, if $f(x_0) = 0$. In the Figure 4.1, we see that the points $x = a$ and $x = b$ are the roots of the function $f(x) = -6x^2 + 45x - 3$. We shall discuss methods of finding all the real roots of a given equation in a specified interval of the variable x.

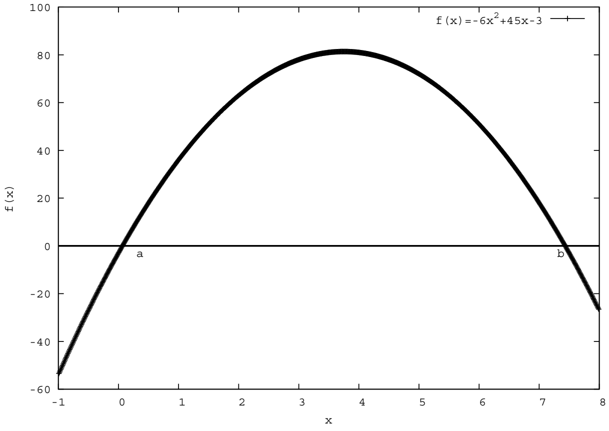


Figure 4.1: **Root of a function**

Given this definition of the root of a function, it seems clear that the easiest way to find the roots would be to evaluate the function at various points and see where it is zero. However, in practice that is not very convenient since one cannot decide beforehand how small an interval to evaluate the function. However, we can still find the rough location and the number of roots in a given interval. To do this, we need to simply tabulate the function at sufficiently large number of points within the given interval. Notice that if the function $f(x)$ is zero, which it is by definition at the location of the root, the value of the function will change sign at that point. That means that if the function $f(x)$ is positive (negative) at values of the argument x less than the root, it will be negative (positive) at values of x larger than the root. This is clear from the Figure 4.1. So all one has to do is to find two points, say x_1 and x_2 where the function has different signs. Then we are assured that there exists atleast one root in the interval (x_1, x_2).

Once the rough location of a root has been found, our next task is to find the root to a prescribed accuracy by narrowing down the interval (x_1, x_2) within which it lies.

Most methods of root finding depend upon what is called the process of **iterations**. We shall describe three such methods here. These methods are general and apply to any kind of function. There are other methods that apply specifically to polynomials and can find even complex roots, but we shall not discuss them here.

☞***In any program to find the roots of a function, the first step should ALWAYS be tabulation of the function to determine the rough value of the roots. This tabulation does not have to be very exact but should be coarse. The idea is to have some idea of the rough location of the roots.***

4.2 Bisection Method

Suppose the function $f(x)$ varies as shown in the Figures(4.2 & 4.3) below. Let two points x_L and x_R be on opposite sides of the root x_0. In the bisection method, one determines the sign of the function at the mid-point, x_M, of x_L and x_R:

$$x_M = \frac{(x_L + x_R)}{2}$$

If $f(x_M) = 0$, then x_M is the required root. If $f(x_M)$ is not zero and has the same sign as $f(x_L)$ (Fig.(4.3) then the root must lie between x_M and x_R. In this case we replace x_L by x_M and repeat the process of **halving the interval and comparing the signs**. In case $f(x_M)$ has a sign opposite to that of $f(x_L)$ (Fig.(4.2)), then the root must lie between x_L and x_M. In this case we look for the root between x_L and x_M by replacing x_R by x_M and repeating the entire process. Since the root is known to lie between x_L and x_R and this interval $(x_R - x_L)$ is halved in every successive approximation, the root can be determined to any desired accuracy.

When do we stop this process? We can see that as we continue this process, the interval $|x_R - x_L|$ becomes smaller and smaller. However, bear in mind that it is not the absolute value of this quantity that is of relevance but the relative value. When this interval relative to the value of $|x_R + x_L|$ becomes smaller than a given accuracy, we can say that we have got the root to the desired accuracy. In other words, the process of bisecting the interval is repeated while $\frac{|(x_R - x_L)|}{|(x_R + x_L)|}$ remains greater than the desired accuracy. Since, by definition, at a root the function must be zero, alternatively the condition for the termination of the loop can depend on the value of the function itself, i.e., the loop is repeated while $|f(x_M)|$ is greater than or equal to some very small number, say 10^{-6}.

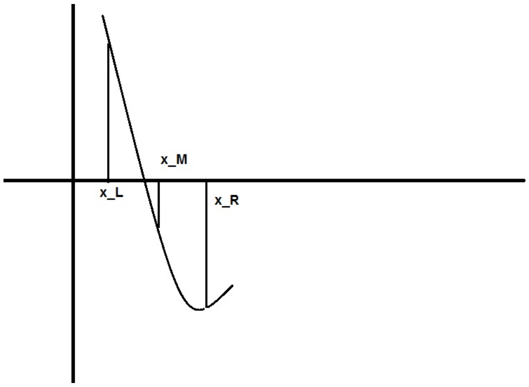


Figure 4.2: **Bisection Method**

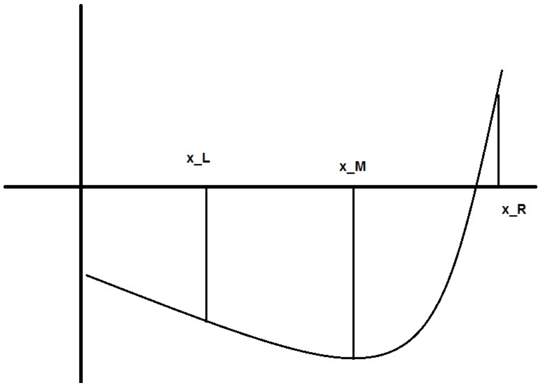


Figure 4.3: **Bisection Method**

The bisection method has the advantage that the procedure is guaranteed to converge. The disadvantage is that the method is slow. Another disadvantage is that roots lying close to each other may be difficult to separate.

Now we can write down the algorithm for using the bisection method to determine the roots of a given function.

1. Tabulate the function $f(x)$ in the given interval in which the roots are to be found.
2. Determine the rough location of the root or roots. This means determining the two values x_R and x_L of x between which $f(x)$ changes sign.
3. Determine the midpoint x_M of the interval $x_R - x_L$.

4. Check the sign of $f(x_M)$. If it is the same as x_R replace x_R by x_M or if it is the same as x_L replace x_L by x_M.
5. Repeat this process till the quantity $\frac{|x_R - x_L|}{|x_R + x_L|}$ becomes smaller than the required accuracy.
6. Then x_M is the root. As a check, evaluate $f(x_M)$ and see that it is smaller than the required accuracy.

As an example of the bisection method, suppose we wish to find the roots of

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

between 0 and 10. The program below will first tabulate the function and then ask you to input the range in which the root lies. This range is easy to see when one has tabulated the function since the value of the function will change sign. It will also tell you the number of roots within the desired interval that is 0 and 10 since the function will change sign at those values. However, this is just a rough value of the root. The actual bisection routine then takes the inputs that you give from the table (that is the number of roots, the range in which the roots lie) and gives the exact value of the roots to the desired accuracy.

```
/* Bisection Method*/
#include <stdio.h>
#include <math.h>
float f(float x)
{ return (sin(x)-x*cos(x));
}
main()
{ float x,xm,xl,xr,acc=0.00001,xinc=0.5,z,a,b,x1,x2;
  int n,i;
  float f(float x);
printf(”Enter the minimum value of x\n”);
scanf(”%f”,&a);
printf(”Enter the maximum value of x\n”);
scanf(”%f”,&b);
for (x=a;x<=b;x=x+0.1)
{printf(”%f \t %f\n”,x,f(x));
}

printf(”input no of roots”);
  scanf(”%d”,&n);

  for (i=1;i<=n;i+=1)
```

```
   {   printf("\ninput x1,x2");
           scanf("%f,%f",&x1,&x2);
   for(x=x1;x<=x2;x+=xinc)
   {
    if(f(x)*f(x+xinc)<0)
     {xl=x;xr=x+xinc;
      do
       { xm=(xl+xr)/2.0;

         if(f(xm)*f(xl)>0)
          {xl=xm;}
         if(f(xm)*f(xl)<0)
          {xr=xm;}
          z=fabs((xl-xr)/(xl+xr));
          printf("xm=%f  f(xm)=%f   z=%f    acc=%f\n ",xm,f(xm),z,acc);
        }
      while(z>acc);
      printf("\nroot=%f   f(xm)=%f  z=%f  acc=%f\n\n",xm,f(xm),z,acc);

    }
   }
 }
 }
```

4.3 Secant Method

The Secant method uses a secant line to approximate $f(x)$ in the neighbourhood of a root. In this method also we need two points to begin with. It differs from the bisection method in that the two points need not be on the opposite sides of the expected root as long as they are sufficiently close to it. However it is always better to choose the points to lie on the opposite sides of the root. Let x_1 and x_2 be the two points close to the root x_0. Consider the two points on the curve $y = f(x)$ having coordinates (x_1, y_1) and (x_2, y_2), where $y_1 = f(x_1)$ and $y_2 = f(x_2)$. The equation of the secant line through these points is given by

$$y - y_2 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_2) \tag{4.1}$$

which intersects the x axis (that is at $y = 0$) at the point x_3 where

$$x_3 = \frac{x_1 y_2 - x_2 y_1}{y_2 - y_1}$$

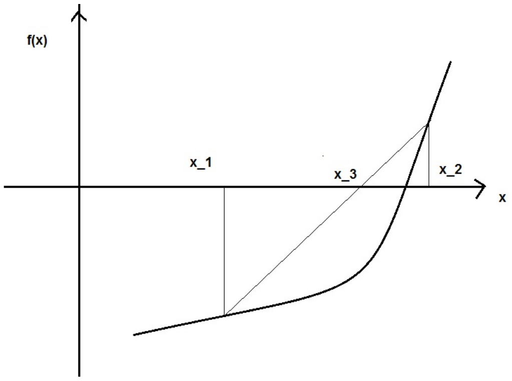


Figure 4.4: **Secant Method**

As can be seen from the Fig(4.4), x_3 provides a better approximation to the actual root x_0. The process of finding the secant line is repeated with the points x_2 and x_3: we use a do while structure and replace x_1 by x_2 and x_2 by x_3 within the loop. The loop is terminated when the magnitude of the difference between the two successive approximations to the root becomes less than the desired accuracy. One may, as in the case of the bisection method, put the condition for the termination of the loop on the value of the function instead of the value of the root.

The secant method usually converges much faster than the bisection method. It has the disadvantage however that the procedure is not guaranteed to converge.

The algorithm for this method should be clear from the description above. Let us use it to find the roots of the function that we used in the discussion on Bisection method.

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

between $x = 0$ and $x = 10$.

```
/* Secant Method*/
#include <stdio.h>
#include <math.h>
float f(float x)
{ return (sin(x)-x*cos(x));
}
main()
{ float x,acc=0.00001,z,a,b,x1,x2,t,x3;
  int n,i;
  float f(float x);
printf(”Enter the minimum value of x\n”);
scanf(”%f”,&a);
printf("Enter the maximum value of x\n");
scanf("%f",&b);
for (x=a;x<=b;x=x+0.1)
{printf("%f \t %f\n",x,f(x));
}

printf("input no of roots");
  scanf("%d",&n);

  for (i=1;i<=n;i+=1)
  {   printf("\ninput x1,x2");
          scanf("%f,%f",&x1,&x2);
  do
  {
  f1=f(x1);
  f2=f(x2);
  x3= (x1*f2-x2*f1)/(f2-f1);
   x1=x2;
  x2=x3;
  t=fabs(f2);
  }while(t>acc);
      printf("\nroot=%f   f(x2)=%f   acc=%f\n\n",x2,f2,acc);

   }
  }
```

4.4 Newton-Raphson Method

This method is based on the Taylor expansion of a function. In geometrical terms it is equivalent to using the tangent to a curve at a given point rather than the secant between two points to find the root.

Let $x = a_0$ be a root of the equation $f(x) = 0$, let a be a close approximation to it, and let $h = a_0 - a$. Since a is close to a_0, h is expected to be small. If we can find h, then we would know the root exactly, since $a_0 = a + h$. Now, using the Taylor expansion, we know that

$$f(a_0) = f(a+h) = f(a) + hf'(a) + \frac{h^2}{2!}f''(a) + \cdots \tag{4.2}$$

If h is sufficiently small, then

$$\frac{h^2}{2!}f''(a) \ll hf'(a)$$

We can therefore neglect terms of order higher than h, and since $f(a_0) = 0$ (since a_0 is the actual root and therefore by definition, $f(a_0) = 0$), we get

$$h \approx -\frac{f(a)}{f'(a)}$$

This provides an estimate of how far the guess value, a, is from the actual root, a_0. In the next iteration $a + h$ is taken as the new approximation to the root, a_0. This iterative process is continued as long as $|\frac{h}{a}|$ remains greater than the desired accuracy.

Another way to understand the algorithm is as follows: Choose any starting point x_0 as the initial guess for the root of $f(x) = 0$. Draw a tangent to the curve $f(x) = 0$ at $(x_0, f(x_0))$. Then the equation of this tangent is simply

$$y = f'(x_0)(x - x_0) + f(x_0)$$

This tangent will intersect the x-axis ($y = 0$) at

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

We check if $f(x_1)$ is smaller than the accuracy required. If yes, x_1 is the root. Else, we repeat the process of drawing a tangent to the curve at $(x_1, f(x_1))$ and do this process iteratively till we get the required accuracy. Figure 6.1 will give you an idea of how this works. As you can see, the guessed value of the root, that is the point at which the tangent crosses the x-axis gets closer and closer to the actual root.

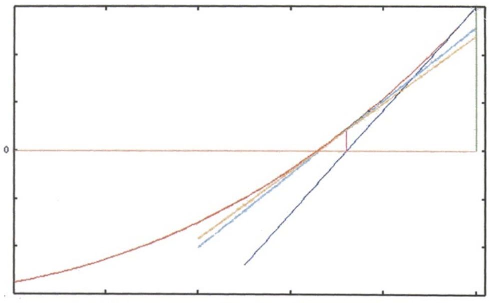


Figure 4.5: **Newton Raphson Method**

It is clear that larger the numerical value of the derivative $f'(x)$ in the neighbourhood of the given root, smaller is the correction h that has to be added to obtain the next approximation to the root. Newton-Raphson method is therefore particularly convenient when the graph of the function is steep in the neighbourhood of the given root. On the other hand, if the derivative $f'(x)$ is small near the root, h will be large and computing the root by this method will be difficult (sometimes even impossible).

The main advantage of this method is that only one point is required to start the procedure. Also the method usually converges very rapidly. However, the disadvantage is that the derivative of the function has to be calculated, and this may not be easy. It is also not guaranteed that the method will converge to a root from any arbitrary starting point. In fact some times the method is known to fail even if one starts reasonably close to the expected root.

We can use the Newton-Raphson method to determine the roots for the function $\sin(x) - x\cos(x)$. Note there we do not need to tabulate since we can start anywhere. However, it is always a good practice to tabulate the function to get to know the rough value of the root. This way, when you get the actual root, you can see if it is correct since if it is very different from the rough value that you have obtained from the tabulation, then there is some error in the program.

```
/* Newton-Raphson*/
#include<stdio.h>
#include<math.h>
float f(float x)
{ return (sin(x)-x*cos(x));
}
float g(float x)
{return (x*sin(x));
}
main()
{
  float a,x,f,g,ainc=0.1,xinc=0.1,f1,f2,h,acc=0.00001;
  float f(float(x);
  float g(float x);

 FILE *fp=NULL;
 fp=fopen("res.txt","w");
        do
           {f1 = f(x);
            f2 = g(x);
            h  = -f1/f2;
            x  = x+h;
            }
            while(fabs(h/x)>acc);
               fprintf(fp,"%f   \t %f\n",x,f1);
          }
```

☞ ***Some of the common errors & good practices***

1. ***Always tabulate the function to get some idea of the roots of the function before you find the exact roots.***
2. ***The Bisection method will always give you a root of the function. However, if the roots are very close to each other, it may not find all the roots.***
3. ***Secant method though faster than the Bisection method is not guaranteed to discover the root of the function.***
4. ***Newton-Raphson method is fast and convenient but again, it is not guaranteed to converge.***
5. ***Newton-Raphson method should be avoided if the slope of the function is steep near the root.***

4.5 Questions

1. Compare the behaviour of some simple functions at large x (like Prob.2 and Prob.3) and explain the discrepancy.What do you understand by iterative methods? How do you find out (roughly) if a given equation has more than one root?
2. How many initial points are needed for the three different methods? How do you select them?
3. What are the advantages and disadvantages of the three methods studied for root-finding?

4.6 Problems

1. Find the roots, accurate to four significant figures, of the equation

$$e^{ax} - bx^2 = 0$$

in the range $-1.0 \leq x \leq 1.0$ for
i) a= 1.0, b = 5.0

ii)a=-1.5, b=10.0

by three iteration methods, that is Bisection, Secant and Newton-Raphson methods. In each case, determine the number of iterations necessary to obtain the desired accuracy.

2. Using the series expansion for $J_n(x)$,

$$J_n(x) = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{x^2}{4}\right)^k}{k!(n+k)!}$$

find its two lowest positive roots for $J_0(x)$ to an accuracy of four decimal places. **Note that in this problem you have been asked to find the roots to accuracy of 4 decimal places and not 4 digits.**

3. The equation

$$f(x, y) = 0$$

defines y as an implicit function of x. . As an example, consider

$$f(x, y) = x^3 + y^3 + xy + 1 = 0$$

For any given x, this is a cubic equation in y; so y can be found by obtaining the roots (one or three real roots) of this equation, say by secant method. Plot y as a function of x, for $-1.5 \leq x \leq 1.5$. If for some value of x there are three real roots, (y_1, y_2, y_3), plot all the three points $(x, y_1), (x, y_2), (x, y_3)$. You may assume that $-2 \leq y \leq 2$.

4. Choosing equally spaced values of t in $(0, \frac{2\pi}{\omega})$, solve the Kepler equation for ψ

$$\psi - \epsilon \sin \psi - \omega t = 0$$

Use the solution to plot the orbit whose radial coordinates are given by

$$r = a(1 - \epsilon \cos \psi)$$
$$\cos \theta = \frac{\cos \psi - \epsilon}{1 - \epsilon \cos \psi}$$

Take $\omega = 1.0$, $\epsilon = 0.8$ and $a = 2.0$. Remember that time t, is only a parameter. The equation has to be solved for each t in the given interval. For each t, the initial value of ψ can be chosen to be t.

5. The Chebyshev polynomials of the second kind can be generated from the recurrence relation,

$$U_{n+1} = 2xU_n - U_{n-1}$$

The first two polynomials are given by

$$U_0(x) = 1$$

$$U_1(x) = 2x$$

Plot the function $U_4(x)$ and find all its roots in the interval $[-1, 1]$.

Chapter 5

Ordinary Differential Equations

5.1 Introduction

An ordinary differential equation (ODE) is an equation involving derivatives of an unknown function of one independent variable. ODEs are very important tools in modeling a wide variety of physical phenomena - oscillating systems, electrical networks, chemical reactions, satellite orbits, etc. Therefore the solution of ODEs is of great importance in physical sciences, ecology, economics and social sciences. We examine some of the ways of solving such equations.

Consider the general first-order differential equation:

$$y' = \frac{dy}{dx} = f(x, y); \qquad y(x_0) = y_0$$

The function $f(x, y)$ and the initial condition (x_0, y_0) are given. The exact solution of this equation is a curve in the $x - y$ plane passing through the point (x_0, y_0) . Solving this differential equation means finding $y(x)$, the equation representing this curve. In general, we are required to find the solution, $y(x)$, from $x = x_0$ to some final point, $x = x_f$. For this purpose the region from x_0 to x_f is divided into n equal intervals, each of length $h = \frac{(x_f - x_0)}{n}$. Some approximation procedure is then employed to obtain y_{i+1} from (x_i, y_i) , where $y_i = y(x_i)$,and $x_i = x_0 + ih$. Thus, beginning with (x_0, y_0) one obtains (x_1, y_1) . The whole process is repeated n times (using **for or do while** loop) to finally obtain y at $x = x_n = x_f$.

Various approximation techniques are available for the numerical solution of differential equations; of these we shall discuss the Euler's formula and the Runge-Kutta methods.

5.2 Euler's Formula

Consider the equation $y' = f(x, y)$. If y_i and y_{i+1} are the solutions at points x_i and x_{i+1}, then we may obtain y_{i+1} from the approximation

$$\frac{y_{i+1} - y_i}{x_{i+1} - x_i} \approx y' \approx f(x_i, y_i)$$

The interval between two consecutive values of x is h, the step size: $(h = x_{i+1} - x_i)$. Therefore

$$y_{i+1} = y_i + hf(x_i, y_i) + O(h^2)$$

or

$$\Delta y_i = y_{i+1} - y_i \approx hf(x_i, y_i) \tag{5.1}$$

neglecting $O(h^2)$, the error term of order h^2. Beginning with (x_0, y_0) , this gives $y_1 = (y_0 + \Delta y_0)$ at the point $x_1 = (x_0 + h)$. Similarly y_2 can be calculated starting with (x_1, y_1) , and so on. Notice that this last equation is essentially a Taylor's expansion of $y(x)$ with only the first two terms retained.

The Euler method is not recommended for practical use because it is not very accurate when compared to other, fancier, methods with an equivalent step size. This is because the error term is of order h^2. Moreover, the method is not very stable. Since it is a very simple method, it is useful to obtain a rough guess of the expected solution.

5.3 Runge-Kutta methods

A straightforward extension of this method for better accuracy would be to retain higher order terms in Taylor's expansion. However, the calculation of higher order derivatives that this straightforward extension requires is often cumbersome; as a result, this straightforward extension is rarely employed in practice.

Consider however the use of a step like Equation(5.1) to take a trial step to the mid-point of the interval. Then use the value of both x and y at that mid-point to compute the real step across the whole interval:

$$k_1 = hf(x_i, y_i)$$

$$k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2})$$

$$y_{i+1} = y_i + k_2 + O(h^3)$$

This symmetrization cancels the first order error term, h^2, and makes the method second order, i.e., the error term is proportional to h^3. This is called the **mid-point or second order Runge-Kutta method.**

The basic philosophy of the Runge-Kutta methods is to find y_{i+1} by evaluating the derivative of y not only at the initial point (x_i, y_i) , as is done in the Euler method, but also at the end point and some intermediate points. The increment, Δy, instead of being just $hf(x_i, y_i)$, is some kind of an average of all these derivatives. The intermediate points are so chosen that the terms of order $h^2, h^3, \cdots$ in the expression for the error cancel out. So, the error is considerably reduced, being proportional to a much higher power of h.

The most widely used method in this class of methods is the 4th-order Runge-Kutta method, RK4. The error in this case is $O(h^5)$. For the solution of the differential equation

$$y' = f(x, y); \qquad y(x_0) = y_0$$

the RK4 method leads to the following formula:

$$y_{i+1} \equiv y(x_{i+1}) = y(x_i + h) = y_i + \Delta y_i$$

where the increment Δy_i is given by

$$\Delta y_i = \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

where

$$\begin{aligned} k_1 &= hf(x, y) \\ k_2 &= hf(x + \frac{h}{2}, y + \frac{k_1}{2}) \\ k_3 &= hf(x + \frac{h}{2}, y + \frac{k_2}{2}) \\ k_4 &= hf(x + h, y + k_3) \end{aligned}$$

To understand this procedure better, it is instructive to look at what each of these steps is doing.

1. We start with k_1 which is just the slope multiplied by h at the beginning of the procedure, that is at the initial point. (remember that the differential equation is $y' = f(x, y)$.)

2. Now we use this slope k_1 to move halfway to the interval, then k_2 is the estimate of the slope at that midpoint multiplied by h. This is better than the estimate k_1.

3. Again, we use k_2 to move halfway to the interval and get k_3 as the estimate of the slope which is better than k_2.

4. Finally, we use k_3 to go all the way to the end of the interval to get k_4 as the estimate of the slope at the endpoint of the interval.

5. To get the value of the quantity y at the endpoint, we use a weighted sum of all these slopes to get an accurate estimate of the slope and multiply it by the interval itself.

Just as in the Euler's method, here also beginning with (x_0, y_0), we obtain $y_1 = y_0 + \Delta y$, i.e., $y(x_1), x_1 = x_0 + h$; then from y_1 we obtain y_2 and so on. The method is very simple to implement as a function which calculates y_{i+1} given (x_i, y_i) and h. The accuracy of the result can be checked by repeating the calculation with a larger number of steps, say $2n$, and observing the resulting convergence.

To compare the two methods, that is the Euler and Runge-Kutta (RK-4) method, let us consider a simple first order differential equation. Of course, Euler's method as well as the RK-4 method can be used for higher order equations as well. Consider the equation

$$\frac{dx}{dt} = xt^2 + t^2 \sin(t^3) \qquad x(0) = 1 \tag{5.2}$$

We will solve this equation exactly and compare the exact analytical solution with the numerical solution obtained with Euler's method and RK-4 method. The exact solution is

$$x(t) = -\frac{3}{10}\cos(t^3) - \frac{1}{10}\sin(t^3) + \frac{13}{10}e^{\frac{t^3}{3}} \tag{5.3}$$

The program below will solve the differential equation and plot the results from the analytical calculation, Euler and RK-4 methods.

```
#include<stdio.h>
#include<math.h>

float f(float t, float x)
{ return (x* t*t + t*t* sin(pow(t,3))); }

main()
{ float f(float t, float x);
  float t=0,x=1.0,h=0.05,s,ti,tf;int i,n; float k1,k2,k3,k4;
  FILE *res1;
   res1=fopen("res1.dat","w");
   FILE *res2;
   res2=fopen("res2.dat","w");
FILE *res3;
   res3=fopen("res3.dat","w");
     ti=0;
   tf=1.0;
    do
        {
            x = x + h*(x* t*t + t*t* sin(pow(t,3)));
            t = t + h;
    fprintf(res1,"%f  %f\n",t,x);
     }
       while(t<=tf);

       for(t=h;t<tf;t+=h)
       {s=-0.3*cos(pow(t,3)) - 0.1 * sin(pow(t,3)) + 1.3 *  exp(0.33*pow(t,3));
       fprintf(res2,"%f    %f\n",t,s);
     }

  n=(tf-ti)/h;
 t=ti;x=1;
  for  (i=1;i<=n;++i)
     {   k1=h*f(t,x);
             k2=h*f(t+h/2.0,x+k1/2.0);
             k3=h*f(t+h/2.0,x+k2/2.0);
             k4=h*f(t+h,x+k3);
 x+=(k1+2*k2+2*k3+k4)/6.0;  t+=h;
   fprintf(res3,"%f    %f\n",t,x);
  printf("%f,%f\n",t,x);
           }
           }
```

The results when plotted are shown in Figure (5.1).

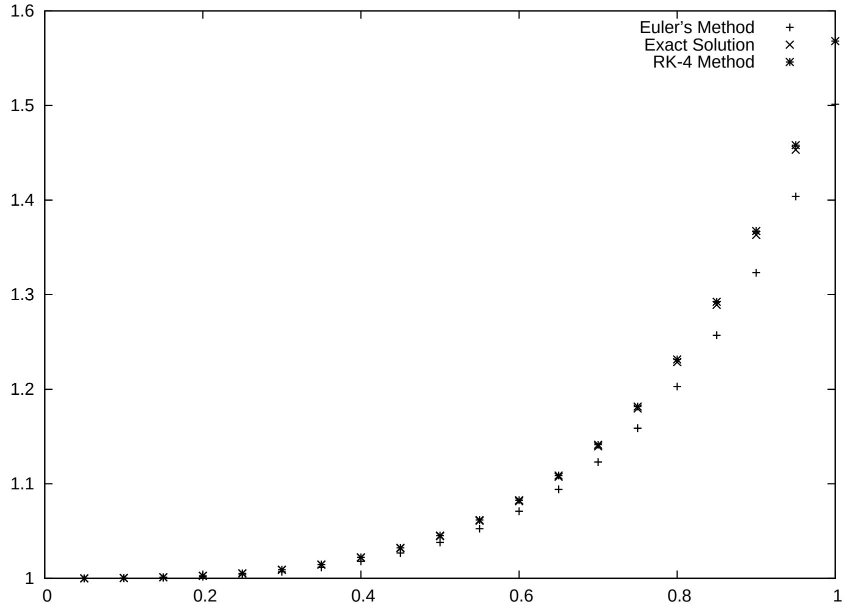


Figure 5.1: **Exact Solution, Euler's method and RK-4 method**

Note that the RK-4 solution is almost identical with the exact solution while the Euler's method gives us a solution which is a good approximation but not very exact. Explore the above program with a smaller step size for the Euler's method and check how the accuracy improves.

5.4 Simultaneous Equations of First-Order

The Runge-Kutta method (or the Eulers method for that matter) can be extended to a system of any number of coupled first-order or higher order differential equations. To solve a pair of simultaneous differential equations:

$$y' = \frac{dy}{dx} = f_1(x, y, z); \qquad z' = \frac{dz}{dx} = f_2(x, y, z)$$
$$y(x_0) = y_0 \qquad z(x_0) = z_0$$

we evaluate the quantities

$$
\begin{aligned}
k_1 &= hf_1(x_i, y_i, z_i) & m_1 = hf_2(x_1, y_1, z_i) \\
k_2 &= hf_1\left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}, z_i + \frac{m_1}{2}\right) & m_2 = hf_2\left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}, z_i + \frac{m_1}{2}\right) \\
k_3 &= hf_1\left(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}, z_i + \frac{m_2}{2}\right) & m_3 = hf_2\left(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}, z_i + \frac{m_2}{2}\right) \\
k_4 &= hf_1\left(x_i + h, y_i + k_3, z_i + m_3\right) & m_4 = hf_2\left(x_i + h, y_i + k_3, z_i + m_3\right)
\end{aligned}
$$

This leads to (at $x_{i+1} = x_i + \Delta h$)

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

$$z_{i+1} = z_i + \frac{1}{6}\left(m_1 + 2m_2 + 2m_3 + m_4\right)$$

Notice that any second-order differential system

$$y'' = f(x, y, y') \qquad y(x_0) = \alpha,\ y'(x_0) = \beta$$

can be written as two coupled first order equations by putting $y' = z$

$$y' = z,\ z' = f(x, y, z),\ y(x_0) = \alpha,\ z(x_0) = \beta$$

This is a special case of the system considered above. In general, an nth order ODE can be converted into a system of n first order ODEs in a like manner.

In case of three simultaneous first order ODE's:

$$y' = f_1(x, y, z, u),\ z' = f_2(x, y, z, u),\ u' = f_3(x, y, z, u)$$

we introduce one more set of quantities in addition to k_is and m_is, and calculate these in a similar manner. Extension of the method to any number of coupled ODE's is obvious.

As an example, consider the simple mass m on a spring with spring constant k. The equation of motion is given by

$$m\frac{d^2y}{dt^2} + ky = 0$$

We impose the initial conditions of $y(t = 0) = A$ and $y'(t = 0) = 0$. The solution to this is trivial and for the given initial conditions it is simply

$$y(t) = A\cos(wt) \quad w = \sqrt{\frac{k}{m}}$$

We choose $k = m = A = 1$ for simplicity. Then we can easily solve the second order differential equation by RK4 method outlined above. The program can be as follows:

```
/* Solving y''+ y' =0
   with initial condtions y(x=0) = 1 and y'(x=0) =0 using RK4  */
#include<stdio.h>
#include<math.h>
#define f1(t,y,z) ((z))
#define f2(t,y,z) (-y)

main()
{    float h=0.01,t,y,z,k1,k2,k3,k4,m1,m2,m3,m4,r;
 FILE *fp=NULL;
 fp=fopen(”res.txt”,”w”);
t = 0.0; y = 1.0; z = 0.0;  /* Initial conditions*/
      do
          { k1 = h*f1(t,y,z);
            m1 = h*f2(t,y,z);
            k2 = h*f1(t+h/2.0,y+k1/2.0,z+m1/2.0);
            m2 = h*f2(t+h/2.0,y+k1/2.0,z+m1/2.0);
            k3 = h*f1(t+h/2.0,y+k2/2.0,z+m2/2.0);
            m3 = h*f2(t+h/2.0,y+k2/2.0,z+m2/2.0);
            k4 = h*f1(t+h,y+k3,z+m3);
            m4 = h*f2(t+h,y+k3,z+m3);
            y  = y+(k1+2.0*(k2+k3)+k4)/6.0;
            z  = z+(m1+2.0*(m2+m3)+m4)/6.0;
            t  = t+h;
r=cos(t);
  fprintf(fp,”%f \t %f \t %f\n”,t,y,r);
          }
      while(t<=10.0);
 }
```

The graph can be plotted for the RK4 solution and the Exact solution and you can see from Figure 5.2 that the numerical solution is very good.

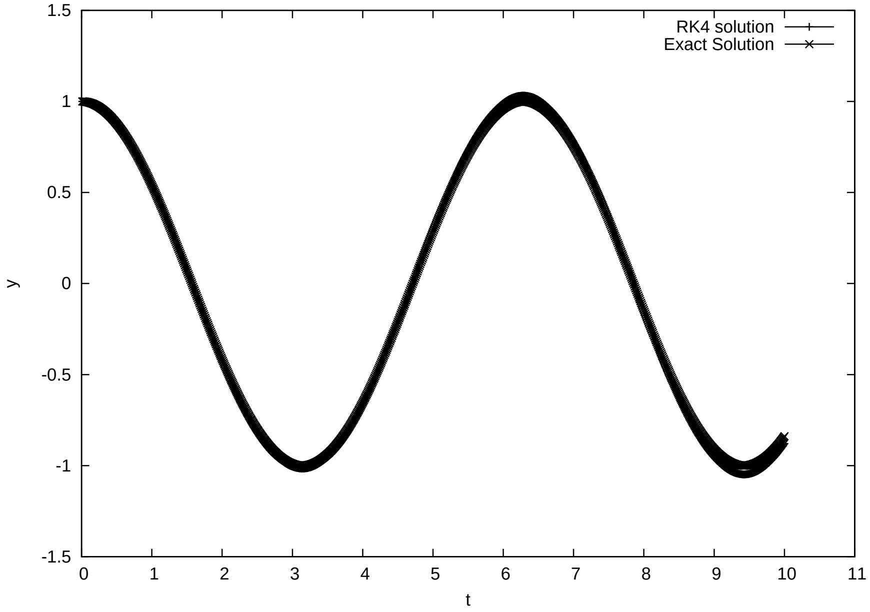


Figure 5.2: **Exact Solution and RK-4 method for Mass on a Spring**

☞ ***Some of the common errors & good practices***

1. ***While using the RK4 method, please make sure that the order of*** k_1, k_2, k_3, k_4 ***is maintained.***

2. ***In RK4 method, care should be taken to see that the order of incrementing the dependent and independent variables is correct.***

3. ***In solving simultaneous equations using the RK4 method, please ensure that the incrementing of the*** k_i ***and*** m_i ***etc is alternate.***

4. ***Always make sure that the initial values of the dependent and independent variables, that is the initial conditions are correctly implemented outside the iteration loop.***

5. ***Note that the division by*** 2 ***in the values of the quantities*** k_i ***is written as a division by*** 2.0***. This is to ensure that the division is not integer division as pointed out in Chapter 1.***

5.5 Problems

1. For the differential equation

$$\frac{dy}{dx} = x + y \quad ; y(0) = 1,$$

tabulate $y(x)$ for $1 \leq x \leq 5$ at intervals of 0.1 for different choices of the step size h ($h = 0.01, 0.005, 0.002, 0.0001$), along with the analytic solution. Use all the three methods for their comparative study. **Note that though the tabulation is required between$x = 1$ and $x = 5$ only, the process of solving the equation has to begin from $x = 0$, since the initial condition is prescribed at that point. Also notice that tabulation has to be done at intervals of 0.1 only though the step size, h, is much smaller than that.**

2. The ODE describing the motion of a pendulum is

$$\theta'' = -\sin\theta$$

The pendulum is released from rest at an angular displacement α i.e. $\theta(0) = \alpha, \theta'(0) = 0$. Use the RK4 method to solve the equation for $\alpha = 0.1, 0.5$ and 1.0 and plot θ as a function of time in the range $0 \leq t \leq 8\pi$. Also plot the analytic solution valid in the small θ approximation ($\sin\theta \approx \theta$).

3. A simple "prey-predator" system is modeled by the set of equations

$$\dot{x} = \gamma_1 x - \gamma_2 xy, \qquad \dot{y} = -\gamma_3 y + \gamma_4 xy$$

where $x(t)$ and $y(t)$ represent respectively the prey and predator populations as functions of time. The term $\gamma_1 x$ tells us that the prey population grows in proportion to its own population while $-\gamma_2 xy$ says that it decreases as a result of encounters with the predators. The second equation says that the predator population decreases in proportion to its own population (to model competition for food between its members) and increases as a result of encounters with the prey (by providing food for the predators). Solve these equations for $\gamma_1 = 0.25, \gamma_2 = \gamma_4 = 0.01$ and $\gamma_3 = 1$ with the initial values $x(0) = 100$ and successively $y(0) = 5, 10, 15, 20,$ and 25. Plot $x(t)$ versus $y(t)$ for $0 \leq t \leq 20$.

4. Solve the following differential equation:

$$f_1(x)\frac{d^2y}{dx^2} + f_2(x)\frac{dy}{dx} + f_3(x)y = f_4(x)$$

with

$$y(0) = 0 \text{ at } x = 0$$
$$\frac{dy}{dx} = 1 \text{ at } x = 0$$

where

$$f_1(x) = f_2(x) = 1, \quad f_3(x) = 4x$$

and

$$f_4(x) = \sum_{i=0}^{10} \frac{(-1)^i x^{2i}}{(2i+1)!}$$

and plot the result from $x = 0$ to $x = 1$.

5. Do numerical integration on the following differential equations (Lorenz equations) with integration step size, $\Delta t = 0.01$:

$$\frac{dx}{dt} = -10(x - y) \qquad \frac{dy}{dt} = \alpha x - xz - y \qquad \frac{dz}{dt} = xy - \frac{8}{3}z$$

Plot the trajectories (after removing transients)

a) in $x - y, x - z, y - z$ planes, and
b) in $x - t, y - t, z - t$ planes,

for the following values of the parameter α:

i)$\alpha = 5.0$ (fixed point solution)
ii) $\alpha = 50.0, 125.0, 200.0$ (chaotic motion)
iii) $\alpha = 100.0, 150.0, 250.0$ (periodic motion)

Choose any reasonable initial conditions.

6. To plot the bifurcation diagram for the logistic map:
A difference equation is a particular form of recurrence relation which is derived from a differential equation. Consider a difference equation

$$x_{n+1} = \alpha x_n(1 - x_n) \quad , x_0 \in [0, 1]$$

Here α is a parameter $\alpha \in [0, 4]$. Choose a single initial value x_0 of x in the range given for x. This can be any value EXCLUDING $0, 1$ and 0.5. For this value of x_0, solve the difference equation for various values of α in the range given, taking $\Delta\alpha = 0.05$. Thus you will have 100 values of α. For the solution to the equation for each α, remove the first 10^4 iterations since these are transients. Keep the next 100 iterations for each α and plot a graph between x and α.

Chapter 6

Integration

6.1 Introduction

Evaluation of the definite integral $\int\limits_a^b f(x)dx$ is equivalent to finding the area enclosed by the curves represented by the integrand $y(x)$, the two ordinates at a and b, and the x-axis (Figure (6.1)). Numerical integration or quadrature, as it is usually called, aims at calculating this area. To do this the required area is divided into several strips by erecting ordinates in the interval between a and b . The area of each strip is found by some suitable approximation and these areas are summed to get the total area. In some procedures the width of the strips is kept equal, while in others it is unequal. It is of course assumed that the given integral has a finite value.

6.2 Methods Based on Intervals of Equal Width

6.2.1 Trapezoidal Rule

Let the range of integration, a to b , be divided into n equal intervals, each of width $h = \frac{(b-a)}{n}$. Let the ordinates at the $(n+1)$ points, $x_0 = a, x_1 = a+h, \cdots, x_n = an+h = b$, be denoted by $y_0 = y(x_0), y_1 = y(x_1), \cdots, y_n = y(x_n)$. If we now approximate each segment of the curve between two successive ordinates by a straight line, each strip becomes a trapezium.

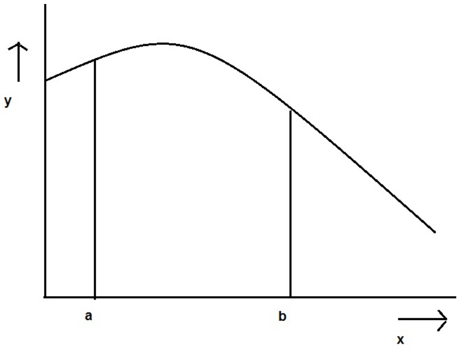


Figure 6.1: **Trapezoidal Rule**

We can find the area of each trapezium and add. The result is

$$\begin{aligned}\int_a^b f(x)dx &\cong \frac{(y_0+y_1)}{2}h + \frac{(y_1+y_2)}{2}h + \cdots + \frac{(y_{n-1}+y_n)}{2}h \\ &= \frac{h}{2}\left[y_0 + y_n + 2\sum_1^{n-1} y_i\right] \\ &= \frac{h}{2}\left[y(a) + y(b) + 2\sum_1^{n-1} y(a+ih)\right] \qquad (6.1)\end{aligned}$$

This formula is called the **Trapezoidal rule**. It can be shown that for trapezoidal rule the leading term in the error is

$$\text{Er} = -\frac{(b-a)h^2 y''(\xi)}{12} \qquad (6.2)$$

where ξ is some point in the interval (a, b) and y'' denotes the second derivative of y. **Clearly, if the integrand is a linear function of its argument, all derivatives of order two and higher vanish, and the Trapezoidal rule produces an exact result**.

As an example, consider the integral

$$I = \int_a^b \frac{1}{(1+x)^3}$$

We will evaluate it using the Trapezoidal Rule. The algorithm is straightforward. We divide the interval into n equal segments and then evaluate the function at each segment and multiply it with the length of the segment to get the area of the segment as explained above. Finally, we add the area of all the segments to get the area under the curve which is the value of the definite integral. The program could be as follows:

```
 /* Trapezoidal Rule*/
 #include<stdio.h>
#include<math.h>
float f(float x)
{
return (1/((1+x)*(1+x)*(1+x)));
}
main()
{
float f(float x);
int i,n;
float a,b,s=0,y=0,h;
printf(”Enter the no of intervals =”);
scanf(”%d”,&n);
printf(”Enter the lower limit=”);
scanf(”%f”,&a);
printf(”Enter the upper limit=”);
scanf(”%f”,&b);
h=(b-a)/n;
for(i=1;i<=n-1;i++)
{
s=s+f(a+i*h);
}
y=(f(a)+f(b)+2*s)*h/2.0;
printf(”the value of the integral is=%f”,y);

}
```

6.2.2 Simpson's Rule

A method that gives reasonably accurate results and is quite often used for numerical integration is **Simpson's rule**. If in the previous example, we approximate the function over two adjoining segments by a quadratic rather than a linear function, we obtain what is known as Simpson's rule:

$$\begin{aligned}
\int\limits_a^b f(x)dx &= \left(\frac{h}{3}\right)[y_0 + 4(y_1 + y_3 + \cdots + y_{n-1}) + 2(y_2 + y_4 + \cdots + y_{2n-1}) + y_n] \\
&= \left(\frac{h}{3}\right)\left[y_0 + y_n + 4\sum_{i=1}^{\frac{n}{2}} y_{2i-1} + 2\sum_{i=1}^{\frac{n}{2}-1} y_{2i}\right] \\
&= \left(\frac{h}{3}\right)\left[y(a) + y(b) + 4\sum_{i=1}^{\frac{n}{2}} y(a + (2i-1)h) + 2\sum_{i=1}^{\frac{n}{2}-1} y(a + 2ih)\right] \qquad (6.3)
\end{aligned}$$

It is obvious that in this case n must be even. Simpson's rule generally yields better results than the Trapezoidal rule because we use a quadratic rather than a linear approximation to each segment of the curve. The leading term in the error in Simpson's rule is proportional to the fourth derivative of y ; therefore it produces exact results in the case of integrands that are general polynomial of order three.

For both methods, the first step is to choose an appropriate interval width h . Equivalently, one can choose the number of intervals, n . But for certain exceptions, Simpsons rule yields reasonably accurate result for $n \sim 50 - 100$. In the trapezoidal rule, for same kind of accuracy, a much larger number of points may be needed.

☞ **Remember that for Simpson's rule n has to be even and that the various terms have to be multiplied by proper weights.**

The following example will make the implementation of Simpson's rule clear.

Consider the integral

$$\int\limits_a^b \frac{1}{(1+x)^3}$$

A program for solving this might look like this

```
/* this is a program to test the Simpson subroutine */
#include <stdio.h>
#include <math.h>
float f(float x)
{   return  (1/((1+x)*(1+x)*(1+x)));
```

```
 }
main()
{   float simpson(float a,float b,int n);
float f(float x);
    int n;
    float a,b,integral;
    printf(”\n input a,b,n  ”);
    scanf(”%f,%f,%d”,&a,&b,&n);
    /* REMEMBER n should be even*/
    integral=simpson(a,b,n);
    printf(”\n\nIntegral=  %f\n”,integral);
}
float simpson(float a,float b,int n)
{   int i;
    float f(float(x));
    float x,y;
    float h=(b-a)/n;
    float s=f(a)+f(b);
    for (i=1;i<n;i+=2)
    {   x=a+i*h;
             y=f(x);
             s+=4*f(x);
    }
    for (i=2;i<n;i+=2)
    {   x=a+i*h;
             y=f(x);
             s+=2*f(x);
    }
    s*=h/3.0;
    return s;
}
```

6.3 Methods Based on Intervals of Unequal Width

6.3.1 Gauss Quadrature

We have noted above that the trapezoidal rule gives exact results for polynomials of order one, whereas Simpson's rule gives exact results for polynomials of order three. We can do better than this if we do not insist on the integrand being evaluated at equal intervals. Suppose we approximate any finite integral by:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{n} w_k f(x_k) \tag{6.4}$$

and choose the $2n$ unknowns - the points x_k and the weights w_k - such that we get an exact result as long as $f(x)$ is an arbitrary polynomial of order $(2n-1)$, i.e.,

$$\begin{aligned}\int_{-1}^{+1}\left[A_0 + A_1x + A_2x^2 + \cdots + A_{2n-1}x^{2n-1}\right]dx &= \sum_{k=1}^{n} w_k\left[A_0 + A_1x_k + A_2x_k^2 + \cdots + A_{2n-1}x_k^{2n-1}\right] \\ &= 2\left[A_0x + \frac{A_2}{3} + \frac{A_4}{5} + \cdots + \frac{A_{2n-2}}{(2n-1)}\right]\end{aligned}$$

where the last step is got by explicit integration. Since this relation must be satisfied for arbitrary $A_1, A_1, A_2, \cdots, A_{2n-1}$, we obtain the conditions

$$\begin{aligned}\sum_{k=1}^{n} w_k x_k^{2i-2} &= \frac{2}{2i-1} \qquad & i = 1, 2, \cdots, n \\ \sum_{k=1}^{n} w_k x_k^{2i-1} &= 0 \qquad & i = 1, 2, \cdots, n\end{aligned}$$

These $2n$ equations can, in principle, be solved for the $2n$ unknowns, w_k and x_k . One can show that the x_ks are the zeros of the Legendre polynomial $P_n(x)$. The weights w_k are also related to the Legendre polynomials.

As a simple illustration, for $n = 2$ one gets the equations

$$\begin{aligned}w_1 + w_2 &= 2 \\ w_1x_1^2 + w_2x_2^2 &= \frac{2}{3} \\ w_1x_1 + w_2x_2 &= w_1x_1^3 + w_2x_2^3 = 0\end{aligned}$$

with the solution

$$\begin{aligned}w_1 = w_2 &= 1 \\ x_1 = -x_2 = \frac{1}{\sqrt{3}}\end{aligned}$$

We see that x_1, x_2 are indeed zeroes of $P_2(x)$.

The approximation

$$\int_{1}^{b} f(x)dx \approx \sum_{k=1}^{n} w_k f(x_k)$$

with the w_k's and x_k's determined as above, is called the **Gauss-Legendre quadrature, or simply Gauss quadrature**. As the above analysis shows, this procedure should, in general, prove to be superior to the Trapezoidal rule or Simpson's rule. The accuracy of the quadrature improves with increasing the order of the polynomial, n . For most functions and for reasonable accuracy, $n = 8$ is usually quite adequate. The weights w_k and the points x_k are given in standard tables for different values of n . Since x_k are symmetric about $x = 0$, and the weight factors for the two symmetric zeroes are the same, only the positive values of x_k and the corresponding w_k need be read in. The negative values of x_k are generated from the positive values. Thus, all one has to do in practice is to decide upon the value of n, read w_k and x_k from the tables and evaluate the series on the RHS of Equation(6.4) which gives the value of the integral on the LHS. Actually, one should perform the calculation for two values of n, say $n = 6$, and $n = 8$ and verify that the two results agree with each other to the desired accuracy. If the agreement is not satisfactory, a still higher value of n should be used.

The Gauss quadrature as described above applies only when the limits of integration are $(-1, 1)$. However, an integral with any finite limits (a, b) can be transformed into one with limits $(-1, 1)$ by a linear transformation of the independent variable:

$$I = \int_{a}^{b} f(z)dz = \frac{(b-a)}{2}\int_{-1}^{+1} f(\frac{(a+b)}{2} + \frac{(b-a)}{2}x)dx$$

where

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

Now applying Gauss quadrature we have the result

$$I = \frac{(b-a)}{2}\sum_{i=1}^{n} w_i f(z_i)$$

where

$$z_i = \frac{(a+b)}{2} + \frac{(b-a)}{2}x_i \tag{6.5}$$

The x_i's and w_i's are obtained from the tables, the z_i's are then obtained from x_i's with the help of Equation(6.5), the given function $f(z)$ is evaluated at these points and the series on the RHS is evaluated. Finally the result is multiplied by the factor $\frac{(b-a)}{2}$ to obtain the value of the integral.

6.3.2 Gauss-Laguerre Quadrature

Approximating integrals over the semi-infinite interval $(0, \infty)$, by a sum:

$$\int_0^\infty e^{-x} f(x)dx \approx \sum_{k=1}^{n} w_k f(x_k)$$

is known as **Gauss-Laguerre quadrature**. [**note that only $f(x_k)$ appears on the RHS in the sum and not the full integrand $e^{-x_k} f(x_k)$** .] The weights w_k and the points x_k are obtained from considerations similar to the case of Gauss Quadrature, viz., that the series on the RHS give exact results whenever is an arbitrary polynomial of order $2n - 1$. In the present case it turns out that the x_k's are zeros of the Laguerre polynomial $L_n(x)$ and w_K's are also related to the Laguerre polynomials. Again, w_k and x_k are available in standard tables.

6.3.3 Gauss-Hermite Quadrature

This quadrature is used to evaluate integrals over the interval $(-\infty, \infty)$ using the approximation:

$$\int_{-\infty}^{\infty} e^{-x^2} y(x)dx \approx \sum_{k=1}^{n} w_k y(x_k)$$

where x_k's are the zeros of the Hermite polynomial $H_n(x)$ and the weights w_k's are also related to the Hermite polynomials. As in the case of the Legendre polynomials, zeros of the Hermite polynomials are also placed symmetrically about the origin and therefore the number of data points to be fed to the computer is halved. The x_ks and w_ks can be read from the tables.

6.3.4 General Considerations & Programming Tips

Keying in the data every time we wish to evaluate an integral is not a good programming practice. One should key the data in a file and include this file in the program using the **#INCLUDE metastatement**. Since Gauss-Legendre, Gauss-Laguerre and Gauss-Hermite quadratures use different sets of zeros and weights, it is best to make three include files and use them according to the integral to be evaluated.

To make such a file for various values of n one uses the **switch statement**. This statement tests a single expression and provides different actions depending on its value. The general form of switch is as follows:

```
switch(n)
```

```
{
case 1:
     [statements]
     break;
case 2:
     [statements]
     break;
...
}
```

In this example, if n has value 1, the statements following case 1 : are executed (**notice the colon after 1**). If $n = 2$, then statements following case 2 : are executed and so on. So for different values of n, we can store zeros and weights in the form of arrays and can call them for the desired value of n.

The three files, **GAUSS.C, LAGUERRE.C and HERMITE.C** have been provided to you in the include directory. These contain the data for the Gauss-Legendre, Gauss-Laguerre and Gauss-Hermite quadratures respectively, for some specific values of n: n = 4,6,8,10,12 for Gauss-Legendre and Gauss-Hermite; n = 4,6,8,10 for Gauss-Laguerre quadrature. **The required file must be included only after n has been defined**. For example, if you wish to find an integral using the Gauss quadrature for $n = 8$ then your program may be

```
main()
{...
 n = 8;
 #include <GAUSS.C>
```

☞ **For all three quadratures, the points x_i and the weights w_i are represented by arrays $x[i]$ and $w[i]$ respectively. Hence, in your program also you are obliged to use the same variable names, $x[i]$ and $w[i]$. Also note that the index, i, begins with zero and not 1. Further, due to the symmetry of the zeros and weights for the Gauss and Gauss-Hermite quadratures, the index runs from 0 to $(\frac{n}{2} - 1)$: only half the values are supplied in the table, the other half have to be generated by symmetry. For Laguerre quadrature the index runs from 0 to $(n - 1)$.**

As an example, consider the integral

$$\int_a^b \frac{\arctan x}{x^2} \qquad a = 5, b = 10$$

We can of course do it using the Trapezoidal method or Simpson method. However, let us see how it can be done using the Gauss-Legendre Quadrature. Recall that the standard limits for the integral in Gauss Quadrature is -1 to 1 while here we have 5 to 10. So we need to use Eq 6.5 to find the corresponding z_i to the x_i. The program can then be as follows:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define f(x) (atan(x)/((x)*(x)))

main()
{float c,d,z,z1,x[20],w[20],a=5.0,b=10.0,s;
 int i,n=6;
 #include<GAUSS.C>
 c=0.5*(b+a);d=0.5*(b-a);
 s=0.0;
 for (i=0;i<=n/2-1;i++)
 {z=(c+d*x[i]);z1=c-d*x[i];
 s+=w[i]*(f(z)+f(z1));}
 s*=d;
 printf(”s=%f”,s);
}
```

Note that the for loop runs from 0 to $\frac{n}{2}-1$ because remember that the array $x[i]$ and $w[i]$ run from $i = 0$ to $n-1$ and the weights and the zeroes, that is $w[i]$ and $x[i]$ are symmetric. Thus for $n = 6$ the zeroes are $x[0], x[1], x[2]$. So the for loop reads the zeroes alternately for z and $z1$ above where $z = \frac{b+a}{2} + \left(\frac{b-a}{2}\right) x[i]$ and $z1 = \frac{b+a}{2} - \left(\frac{b-a}{2}\right) x[i]$.

Similar programs can be written for evaluating semi-infinite integrals (with limits $0, \infty$) using Laguerre quadrature and infinite integrals (with limits $-\infty, \infty$ using Hermite quadrature.

☞ ***Some of the common errors & good practices while using gnuplot***

1. ***While using Trapezoidal Rule, the larger the number of segments, the better is the accuracy***
2. ***In Simpson's Rule algorithm, make sure that the value of*** n ***is even.***
3. ***The for loop for the odd and even terms starts at*** 1 ***and*** 2 ***and has an increment of*** 2.
4. ***The weights with which the odd and even summations are multiplied are different. It is*** 4 ***in the odd case where the loop starts from*** $i = 1$ ***and*** 2 ***in the even case where the loop starts from*** 2.

5. ***Remember to divide the result of the sum of the areas obtained in Simpson's rule by*** 3***.***
6. ***While using quadratures, care should be taken to include the file GAUSS.C or HERMITE.C or LAGUERRE.C only AFTER defining the integer variable*** n ***and specifying its value.***
7. ***While using the quadratures, remember that the arrays run from*** 0 ***onwards and NOT*** 1***. The for loops therefore which reads the arrays should always start from*** 0***.***
8. ***The zeroes of the Gauss and Hermite quadratures are symmetrical and therefore the for loops should run from*** 0 ***to*** $\frac{n}{2}-1$***. For Laguerre quadrature, the for loop should run from*** 0 ***to*** $(n-1)$***.***

6.4 Problems

1. Integrate to an accuracy of 1 in 10^5 for given limits a and b:

$$\int\limits_a^b \frac{\arctan x}{x^2}, \qquad a = 5, b = 10 \qquad (Answer : 0.142208)$$

   Use Trapezoidal & Simpson rule.

2. The time period of a pendulum is given by the integral

$$T = 4\int\limits_0^{\frac{\pi}{2}} \frac{1}{1-\sin^2\left(\frac{A}{2}\right)\sin^2 x}dz$$

   where $A$is the amplitude of oscillations. For small amplitudes it is possible to approximate the time period to

$$T_1 = 2\pi\left[1+\left(\frac{A}{4}\right)^2\right]$$

Plot T, T_1 and the percentage difference between T and T_1 as functions of A for $0 < A < \pi$.

3. Let $R(\theta)$ be the polar coordinates of a particle moving under a central force. Then θ is given as a function of R by the expression:

$$\theta(R) = \int\limits_{r_0}^{R} \frac{dr}{r^2 \left[\left(\frac{2mE}{l^2}\right) - \left(\frac{2mV(r)}{l^2}\right) - \frac{1}{r^2}\right]^{\frac{1}{2}}}$$

Plot the orbit of the particle for $V(r) = -\frac{k}{r}$ (inverse square law force). Use Gauss quadrature for the evaluation of the integral. The upper limit,R is to be varied from r_0 to r_m, where r_0 and r_m are the two zeroes of the factor in the square brackets. Take $m = l = k = 1$ and

i) $E = -0.25$ (This gives $r_0 = 0.6, r_m = 3.4$ approximately)

ii)$E = 0 (r_0 \approx 0.5, r_m \approx 5)$

4. Locate the smallest positive root of the function $F(x)$, given by:

$$F(x) = \int\limits_{0}^{\pi} \cos\left[x^a \cos(t)\right] \sin^{2n+1} t dt$$

to an accuracy of 4 significant figures, for $n = 1$ and $a = 1.5$.

5. Use the integral representation of the Bessel function:

$$J_n(z) = \frac{1}{2\pi} \int\limits_{0}^{2\pi} \cos(z \cos(x)) dx$$

to find its zeroes in the range $0 \leq z \leq 12$ by secant method.

6. The spherical Bessel function of order n is given by

$$j_n(z) = \frac{z^n}{2^{n+1} n!} \int\limits_{0}^{\pi} \cos(z \cos\theta) \sin^{2n+1} \theta d\theta$$

Find all the roots of $j_2(z)$ between 0 and 10.

Chapter 7

Matrices

7.1 Introduction

In Physics, we are all familiar with vectors and matrices. For instance, the position vector, the electric field vector etc. are examples of vectors. Though in general vectors are defined as objects in any space which have a specific property under rotations about an axis, we usually choose a basis and then the vector is specified in terms of the projections on the basis. Thus, in 3-dimensions, we usually choose the mutually perpendicular x, y, z basis and then any vector is simply a set of three numbers giving us the components or projections along these three directions. For instance, we can represent a vector $\vec{X} = 2\hat{x} + 5\hat{y} + 7\hat{z}$ as

$$\vec{X} = \begin{bmatrix} 2 \\ 5 \\ 7 \end{bmatrix}$$

Similarly, there are many instances where we use matrices in Physics. The rotation matrix is one example. In general, whenever we have tensors of rank 2, we can write them as matrices. Thus the rotation matrix in two dimensions, **R** can be written as

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

We are already familiar with the use of numbers (both integers and floating point numbers) in C programming. In this Chapter, we shall see how to manipulate vectors and matrices.

7.2 Arrays in C

An **array** is a collection of data items, all of the same type, accessed using a common name. A one-dimensional array is like a list,a two dimensional array is like a table. C language places no limits on the number of dimensions in an array. Some texts refer to one-dimensional arrays as vectors, two-dimensional arrays as matrices, and use the general term arrays when the number of dimensions is unspecified or unimportant. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

You already know that a simple variable like x (which may be of type **float** or **int**), can be assigned only one value at a time. Once a new value is assigned to x, the earlier value is lost. While such variables suffice for most purposes, some applications, in particular those involving vectors and matrices, need variables with indices, such as x_i or a_{ij}. **These variables constitute arrays**. For example, x_0, x_1, x_2, x_3 form the elements of a **one-dimensional array**. Similarly, $a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{12}, a_{20}, a_{21}, a_{22}$, are the elements of a **two-dimensional array** of 3 rows and 3 columns. A two-dimensional array of m rows and n columns is said to be of order $m \times n$.

7.2.1 Declaration of Arrays

Like 'ordinary' variables, arrays also have to be 'declared'. If the elements of an array are numbers of type float, the array is also of type float, and is declared as such along with other variables of type **float**. If the elements are intended to be all integers, the array is of type **int** and declared accordingly.

Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [ ] brackets for each dimension of the array. That is , a variable is declared as an array by using square brackets along with its name. Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets. **Dimensions used when declaring arrays in C must be positive integral constants or constant expressions.** For example,

```
int i, j, intArray[ 10 ], number;
        float floatArray[ 1000 ];
        int tableArray[ 3 ][ 5 ];
```

are all valid examples of declaring arrays. In the above, the array **intArray** is a 1-dimension has 10 elements which are integer variables while the array **floatArray** is a 1-dimensional array of size 1000 floating point numbers. Similarly, the array **tableArray** is a 2-dimensional array of 3 rows and 5 columns filled with floating point numbers.

7.2.2 Initializing & Using Arrays

Arrays may be initialized when they are declared, just as any other variables. The initialization data is placed in curly braces following the equals sign. **Each entry is separated by a comma.** An array may be partially initialized, by providing fewer data items than the size of the array. The remaining array elements will be automatically initialized to zero.

Like simple variables, elements of arrays can also be assigned values in the declaration statement. For example, we may have statements of the following kind in a program:

```
float x[3] = {1.0,2.0,3.0}, y,z,h = 0.1;
int i,j=1,a[5] = {5,3,11,2,3};
float m[2][3] = {{0.1,0.2,0.3},{0.4,0.5,0.6}};
```

Notice the use of curly brackets in assigning values to the elements of arrays. The declaration and initialization of the array $m[2][3]$ above generates the matrix

$$m = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \end{bmatrix}$$

since in C arrays are read row by row. Once an array has been initialized in the manner explained above, its elements can be called in the program by its indices. For example, within an expression, $a[2][3]$ means the element a_{23}.

☞**Thus, in the declaration statement, $a[2][3]$ denotes a two-dimensional array of two rows and three columns, while in an expression $a[2][3]$ refers to a particular element of the array, viz., the element a_{23}. So be careful!**

Now that we have declared the arrays, how do we use them?
Elements of an array are accessed by specifying the index of the desired element within square [ ] brackets after the array name. Array subscripts must be of integer type.

☞ **Array indices start at zero in C, and go to one less than the size of the array.** For example, a five element array will have indices zero through four. The elements of the array $x[3]$ are $x[0], x[1]$ and $x[2]$. This is because the index in C is actually an offset from the beginning of the array. The most common mistake when working with arrays in C is forgetting that indices start at zero and stop one less than the array size. Arrays are commonly used in conjunction with loops, in order to perform the same calculations on all (or some part) of the data items in the array. **Moreover, the indices can have only non-negative integer values.**

To illustrate these points, consider this simple program to generate the first 10 Fibonacci numbers. Recall that each Fibonacci number is the sum of two previous Fibonacci numbers.

```
#include <stdlib.h>
```

```
 #include <stdio.h>
  main()
  {
int i; /* declaring an integer variable i */
int fibonacci[ 10 ];/* declaring  a 1-dimensional integer array fibonacci with 10 elements*/

           fibonacci[ 0 ] = 0; /* Initialising the first element of the array fibonacci*/
           fibonacci[ 1 ] = 1;  /* Initialising the second element of the array fibonacci*/

           for( i = 2; i < 10; i++ )
               fibonacci[ i ] = fibonacci[ i - 2 ] + fibonacci[ i - 1 ];

           for( i = 0; i < 10; i++ )
               printf( ”Fibonacci[ %d ] = %f\n”, i, fibonacci[ i ] );
           }
```

This will generate the following output:

```
Fibonacci[0]=0
Fibonacci[1]=1
Fibonacci[2]=1
Fibonacci[3]=2
Fibonacci[4]=3
Fibonacci[5]=5
Fibonacci[6]=8
Fibonacci[7]=13
Fibonacci[8]=21
Fibonacci[9]=34
```

Compare this with the program you would have written in Problem 5 in Chapter 1 to see how the use of arrays makes things simpler.

As an example of the use of 2-dimensional arrays, consider the following sample program which adds 2 2 × 3 matrices and prints the result.

```
  #include <stdio.h>
  main( )
  { float a[2][3] = {{0.1,0.2,0.3},{-0.2,-0.4,-0.6}};
    float b[2][3] = {{0.3,0.4,0.5},{0.6,0.7,0.8}};
    float c[2][3];
```

```
int i,j;
for(i=0; i<=1; i++)
{   for(j=0; j<=2; j++)
    {   c[i][j] = a[i][j] + b[i][j];
        printf(”%f\t”,c[i][j]);    }
        printf(”\n”);      }    }
```

Run this program and see the output. The output will be in the form of a matrix.

$$\begin{bmatrix} .4 & .6 & .8 \\ .4 & .3 & .2 \end{bmatrix}$$

This has been accomplished by using two **printf** statements, one in the inner loop and one in the outer loop. If the second **printf** statement is omitted, the output will place all the elements in a single row.

We must be careful while declaring the order of an array. **C does not check array bounds.** The statement:

```
float a[3][3];
```

means that a has been declared as an array with three rows and three columns. If somewhere in the program we use $a[4][5]$ in some expression, no error message will be sent but the output that is produced may be total garbage. It is safer to declare an array of order higher than we are likely to use in the program.

7.3 The Function matalloc

The above method of declaring arrays has some drawbacks. We are most used to defining the indices of the rows and columns of matrices (as well as rows for vectors) starting from 1. Recall that the elements of an $m \times n$ matrix $\mathbf{A}$, are usually written as

$$a_{ij} \quad i = 1, 2, \cdots, m \qquad j = 1, 2, \cdots, n$$

In C, as we have seen above, the arrays, whether one or two dimensional, have their indices starting from 0. This is obviously inconvenient. However, there is a more serious matter. The way in which we have described arrays above, allows us to **define an array of a fixed size only**. Thus if we have written a program for inverting a 5×5 matrix, it can be used to invert only a 5×5 matrix. For inverting, say a 10×10 matrix a totally new program will have to be written. This is of course very cumbersome. What we would like to do is to develop general programs for various matrix operations as function programs and then use these functions for matrices of a specific order in a particular application.

In the absence of knowledge of the order of the matrices, how do we make declarations to reserve memory for storing them in the memory space of the computer, or call and use programs that have been developed for matrices of any order? In C, this problem is solved by using the functions called **malloc and calloc**. **These functions are contained in the header file stdlib.h and this file must, therefore, be included at the top of the program.** These functions allocate contiguous blocks of memory via pointers. The functions **calloc and malloc** have a similar purpose; **calloc** has the additional feature that it sets initial values of all elements to zero. These functions are used for dynamic memory allocation in C which is very useful especially in dealing with arrays.

We have written a function **float **matalloc(int m, int n)** that allocates memory space to an $m \times n$ matrix. This function will generate a column vector if we put the number of columns equal to one (i.e. $n = 1$.) and a row vector if $m = 1$. So, no separate function is required to generate vectors. This program has been written in such a way that the indices take values starting with 1 in accordance with the standard notation familiar to us. It is contained in the file <**MAKEMAT.C**> **in the include directory, and you can access it via #include** <**MAKEMAT.C**>.

To illustrate use of the above function, we give below a program that calls this function to allocate memory space to two matrices, **a** and **b**, assigns elements to the matrix **a**, makes a copy of this matrix (the matrix **b**) and then prints the matrix **b**. Here the elements of the matrix a are given by the formula:

$$a_{ij} = \frac{i^2 + j^2}{i + 2j + 2} \quad i, j = 1, 2, \cdots, n$$

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include < MAKEMAT.C>
main ()
{ int i,j,m=3, n=2;
  float **a, **b;
  a = matalloc (m,n);    b = matalloc (m,n);
  /* now we can give elements of the matrices */
  for (i=1; i<=m; i++)
  { for (j=1; j<=n; j++)
    { a[i][j]=(float)(i*i+j*j)/(float)(i+2*j+2); b[i][j]=a[i][j];
      printf(”%f\t”, b[i][j]);  }     printf(”\n”);    }   }
```

In the above program, note the use of the pointer for the 2-dimensional arrays (matrices) a and b. Essentially think of the 2-dimensional array as a pointer to a pointer to a floating point number (in the case where the entries of the array are of the type **float** as above. If the entries are of the type **int**, then the array can be thought of as a pointer to a pointer to an integer. (For details of pointers, see the Advanced Topic on pointers, Section 1.8 in Chapter **??**.

If we have to allocate memory space for two matrices **a** and **b** of size 2 × 3 and 3 × 5 respectively, then the first, second and third lines after **main()** in the above program would be replaced by:

```
int m=2,n=3,l=5;
      float **a,**b;
      a = matalloc(m,n);     b = matalloc(n,l);
```

However, note that the facility of assigning values to various elements of a matrix while defining it, viz.:

```
float a[2][3] = {{0.1,0.2,0.3},{-0.2,-0.4,-0.6}};
float b[2][3] = {{0.3,0.4,0.5},{0.6,0.7,0.8}};
```

is lost when a matrix is defined via the function **matalloc**. So, each element may have to be assigned its numerical value individually. Thus, for example

```
...
              ...
              for(i=1;i<=n;i++)
                { for(j=1;j<=m;j++)
                  { scanf("%f",&x);   a[i][j]=x;}}
```

allows you to enter the values of the $n \times m$ elements, $a[i][j]$, one at a time. Also note that there is a problem with the statement:

```
{
scanf("%f",&a[i][j])
}
```

Instead one has to use

```
{
scanf("%f",&x);   a[i][j]=x;
}
```

☞ ***Some of the common errors & good practices***

1. ***Dimensions of arrays must always be declared a integers.***
2. ***Arrays in C are read row by row.***
3. ***The array indices in C start at* 0 *and run to one less than the size of the array.***

4. ***Make sure the header file stdlib.h is included in your program.***
5. ***By including the file makemat.c one can start the array indices at*** 1.
6. ***C does not check array bounds. So be careful of declaring the arrays of sufficient size.***

7.4 Problems

1. Create separate functions for (a) adding, (b) subtracting, (c) multiplying, (d) printing and (e) finding trace of matrices and (f) transferring elements of one matrix to another. For example:

```
void matprint(float **a,int m,int n)
        { int i,j;
           for (i=1; i<=m; i++)
          { for (j=1; j<=n; j++)
             {printf (   %f\t     ,a[i][j]); }   printf (   \n   ); } }
```

   is the function to print an $m \times n$ matrix **a**. The statement (in main) to print a $l \times m$ matrix **x** will simply be **matprint(x,l,m)**;. **Note that $**$ is present along with the name of the matrix in the function but not in the calling program. Also note that most of these programs are of the type void. The value of the calculated matrix is not returned to the main program in the usual way.**
   Test each individual function by using it in a short program to check if it is working. Once you have checked all the individual functions, make a file called **matops.c** of these functions by assembling them together. This file obviously cannot be used by itself in a standalone fashion, since it only has functions listed in it. But now, whenever you have any program to write with matrices, you just include this file that is write **#include < matops.c>** in the program and call the relevant functions to carry out print, copy, add or multiply etc. operations on the matrices in your program.

2. Write a program which will generate two 4×4 matrices **a** and **b**, where $a_{ij} = \frac{i}{(i+j)}$ and $b_{ij} = \frac{j}{(i+j)}$, obtain the commutator $c = [a, b]$ and print its elements in a matrix format. Your program should use the program **matops.c** that you have written in the previous problem.