Programming in C

Senior Freshman Computational Physics laboratory, Trinity College Dublin

CONTENTS:

1.0 A first program

In order for a computer to obey our instructions we must put them in a simple code that the computer can understand. To do this the instructions must be precise and detailed. The instructions which C programmers give to the computer are called the **source code**, but the computer must translate it into the language which it uses itself for the execution of programs. This is known as the **object code**. This translation process is known as **compiling** and at this stage any errors of syntax which have been made in the program are shown up. The computer cannot execute the program if the instructions are not clear or if the correct syntax has not been used, so it sends back error messages which point out where the problem is. The task for the programmer is then to modify the program in a way that the computer can understand. Computers are easily confused and the error may merely be the omission of a semicolon or a closing bracket.

The best way to understand how C works is to write a simple program. The following program shows you how to command the computer to print to the screen.

(1) Open a new file by typing vi demo.c
(2) Enter the programme given below
(3) Save it and exit using the :wq vi command

```
#include<stdio.h>

main()

{
        printf("Cogito ergo sum.\n");
}
```

- The first line of the program is a message to the compiler that it should link to the standard input-output header file (abbreviated to 'stdio') which includes a library of definitions of commands such as **printf( )** which means 'print to the screen'**.**
- **main( )** is the heading which labels the core part of the program, that is, its main function which is contained in curly brackets { and }. Note the position of these in the layout of the program - placed so that it is easy to see at a glance where the main function begins and ends. The final } terminates the program execution. It's like writing "The End".
- **printf( )** is a pre-defined function in the **stdio.h** file. The specific details of what is to be printed must be contained in brackets, as shown in the program above.
- The set of characters inside the inverted commas which are to be printed are known as a **string**
- **\n** means 'go to a new line'
- Each command is terminated by a semicolon.
- Make sure to close the curly brackets at the end of the program.

To compile the program type

gcc -o demo demo.c

this will create an executable file called demo (the name is given after the –o option in the Gnu C compiler command you just gave).

Type

ls –ls

to see that demo has been created. To run the programme type

./demo

The ./ may be required to tell the operating system to look for demo in the current directory.

1.1 Error and warning messages

If you make a syntax error in entering your programme the compiler will issue an error message and demo will not be created. An example of a syntax error would be to omit the semi colon from the line containing the print statement. A less serious error will result in the compiler issuing a warning message. It is a good idea to investigate why the warning was issued, even though the compiler may produce a working programme.

1.2 Using a control string

In general the format of the print command is **printf("control string", other parameters);** The control string does not need to be written out in detail. For example, an integer may be represented as **%d** and the value of this integer is specified as another parameter after the comma. These symbols are called "conversion specifications" and several may appear in the string, each identified by a different parameter after the comma. The program below (write.c) shows how this works.

```
#include<stdio.h>

 main()
 {
        printf("Printing to the screen\n");
        printf("%s%s%d\n","This is a string ", "and here is an integer: ",
        542);
        printf("%s%c\n", "this is a character: ", 'k');
        printf("\n%s%d%c%d%c%d\n", "The date is ",24,'.',6,'.',97);
        printf("%s%f\n%s%e","this is a floating point number: ",35421.79,
        "This is the same number in scientific notation: ",34521.79);
 }
```

- The **conversion specifications** used here are as follows (see also Table 1 in the appendix):

Character	%c	char
Integer	%d	int
Floating point	%f	float
Scientific notation	%e	float
String	%s	char*

Note that a string is enclosed by double inverted commas, a character by single inverted commas, and a number does not require any such enclosure.

Exercises:

1. Type out the program above. See what happens if you omit the final **}** or **;** and then try to compile the program. Make a note of the error messages that these mistakes generate.
2. Experiment with spacing in the control string and see what happens when a new line (\n) character is omitted.

1.3 Reading from the Keyboard

This program requests an input and then reads it from the keyboard. In order to do this, variables must be used. The variables must be declared at the top of the program (see section II.1.2) and their types defined. This tells the computer to save space in its memory for the variables to be used in the program. The amount of memory put aside depends on how you define the variable. For example, an integer requires 16 bits but a floating point number requires 32. (See Table 1 in the appendix.) It is therefore important not to try to fit a number in a box which is too small for it, e.g. do not put a float into a variable designed for the input of an integer. This would confuse the computer and produce a strange outcome to your operation, if it works at all! Look at **read.c** below:

```
#include <stdio.h>

main( )
{
        int a, b;
        char one, two;

        printf("Enter two integers with a space or <RET> between: ");
        scanf("%d%d", &a, &b);
        printf("The integers are %d and %d.\n", a, b);
        printf("\nEnter two characters\n");
        scanf(" %c %c", &one, &two);
        printf("%s %c %s %c\n", "The characters are", one, "and ", two);
}
```

- **int a, b** instructs the computer to put aside memory space for two integer variables which will be named a and b. Codes for other types of variable include **char** for character, **float** for floating point number, **double** for long floating point variables.
- **scanf( )** instructs the computer to read the values that have been entered on the screen and store their values in the appropriate variable. It uses **white space** to distinguish between inputs to be assigned to different variables. **White space** means a space, a tab space or the use of the <RET> key (which produces a new line.) If you type in '3 4 5' or '3 4 5' it will recognise the three figures as separate inputs and assign them appropriately. However, '345' is identified as a single number.
- It is essential that the variable names to which the information is to be assigned are prefixed with the **&** character which indicates the address where the variable in question is held within the memory.

- Not all variable names are acceptable to C. For instance, C keywords such as **int**, **while**, **float**, etc., are restricted and cannot be used as variable names (see Table 5 at back of manual). C also treats upper and lower case letters as distinct. Hence Count, COUNT and count will be treated as three distinct variables.

Exercises:

1. Enter a figure as a character
2. Enter characters which are not letters
3. Enter a negative integer
4. Examine the effect of changing the spacing in the second part:
   scanf("%c%c", &one, &two); Can you explain the result?
5. Write a similar program to read **floats**.

2.0 Mathematical Operations

```
/*      maths1.c
        This programme performs a variety of simple arithmetical operations. */

#include <stdio.h>

main( )
{
        int a, b, isum, iproduct, idifference;
        float c, d, sum, product, difference, quotient;

        printf("Enter two integers, a and b: ");
        scanf("%d%d", &a, &b);
        isum = a+b;
        idifference = a-b;
        iproduct = a*b;
        quotient = a/b;
        printf("\n%s %d %s %d %s %d %s %f\n", "a+b=", isum, "a-b=",
        idifference, "a*b=", iproduct, "a/b=", quotient);
        printf("\nEnter two floating point numbers, c and d: ");
        scanf("%f%f", &c, &d);
        sum = c+d;
        difference = c-d;
        product = c*d;
        quotient = c/d;
        printf("\n%s %f %s %f %s %f %s %f\n", "c+d=", sum, "c-d=",
        difference, "c*d=", product, "c/d=", quotient);
}
```

- The comment at the top of the program is enclosed in /* and */. This informs the computer to ignore what is inside these marks . Comments are useful in setting out a program clearly so that it can be easily understood by others and indeed yourself if you wish to come back some time in the future to alter your file.

- Even though the quotient of the integers a and b is declared to be a floating point, the value returned is in fact the integer below a/b. This is because **dividing an integer by another always returns an integer value.** For example, 5/2 is 2, 13/7 is 1, 1/2 is 0.
- = is not the same as an 'equals' sign in algebra. It is called the **assignment operator** as it assigns the value resulting from the operation on the right to the variable on the left. **a=b** does not mean that **a** and **b** are equal but instructs the computer to assign the value of **b** to the variable **a.**
- Note that the last statement spreads over two lines. A line of code can be broken in this way at any point where a space is allowed - after a comma in the parameter list in this case - unless the break occurs within a string. To insert a break when typing a string use the character '\' before pressing <RET>, to indicate that the string will continue on the next line.

Exercises:

1. Try writing statements to evaluate more complicated arithmetic expressions - precedence of
   operations and use of brackets is exactly as in ordinary algebra. (See Table 3 at the back of manual.)
2. Find out what the operator **%** does. This operator requires two integers (e.g. 3%5) and has
   the same precedence as * and /. It will not work with floats - see what error message you get
   if you try to use it with floats.
3. Write a program which asks for an angle in degrees to be entered, converts that angle to radians, and prints the answer on the screen. (2π radians = 360 degrees). π is a constant number which must be defined for the computer. It is possible to type in 3.1415 wherever it is needed but it is much more efficient to define π at the beginning of the program. To do this use the command: #define PI 3.1415 which is placed at the very top of the program with the #include commands. Any constant can be defined in this way rather than inserting the actual number all the way through the program. (It is called a **symbolic** constant.) It is also useful if you want to change the constant (for example, to use a value for PI correct to a larger number of digits). If this is the case you only have to change the definition at the top and will not have to go through the program changing each value manually.

2.1 More Mathematical Operations

This section introduces the operators ++, **--**, **+=**, **-=**, ***=** and **/=**. The hierarchy of operators and use of brackets is the same as in ordinary algebra (see Table 3 in the appendix for further details).

The effects of the operators ++ and -- depend on whether they are placed before or after the variable, as demonstrated by the following example:
Consider the statement **x = y++;** Both ++y and y++ increase the value of y by 1, but in the case of y++, the value of y is increased **after** the original value of y has been assigned to x. Therefore, if y = 5 initially, after the operation x = y++ the value of x is 5 and the value of y is 6. (This is one of the operations that the program **maths2.c** below performs.) However, if the expression were phrased

x = ++y; the outcome would be different. This instructs the computer to increase the value of y by one first, and then assign this new value to x. Thus after the operation x=++y both x and y would equal 6.
The **--** operator works according to the same logic.

Other expressions used in the program **maths2.c** are **c+=2** and **d-=4**. These mean respectively, increase the value of c by 2, and, decrease the value of d by 4. Similarly *= and /= respectively multiply and divide the variable on the left by the number on the right.
Examine, compile and run **maths2.c**:

```
/*      maths2.c        More arithmetical operations */

#include <stdio.h>

main( )
{
        int a, b, c, d;

        printf("Enter two integers, a and b: ");
        scanf("%d%d", &a, &b);
        c=a++;
        printf("After the operation c=a++, a=%d, c=%d\n", a, c);
        d=++b;
        printf("After the operation d=++b, b=%d, d=%d\n", b, d);

        c += 2;
        d -= 4;
        printf("\nNow c=%d, d=%d\n", c, d);

}
```

Exercises:

1. Edit the program to find out the effects of the other operators. Also try their effects on floats.
2. Write a program to evaluate the distance s travelled along a straight line in time t and the speed v at time t, where $s=ut+at^2/2$ and $v=u+at$. The program should ask for values of u, a and t to be entered, and print the answer to the screen. Test your program in the cases of zero acceleration and of acceleration under gravity ($a=g=9.8ms^{-2}$).

2.2 Predefined mathematical functions

Just as the functions **printf( )** and **scanf( )** are defined in the header file **stdio.h**, some useful mathematical functions are contained in the header file **math.h**. These include those used in the program **maths3.c** which is printed below. For example,

sqrt(x) = $\sqrt{x}$
fabs(x) = $|x|$
pow(x,y) = x^y

See Table 4 in the appendix for other predefined functions in **math.h**. Note that the functions in Table 4 must be fed a value of x which is a double precision floating point number, and thus x should be declared as a variable of the type **double**, and the appropriate conversion specification is **%lf**. The function then returns the value appropriate to that particular value of x, and this will also be a double precision floating point number. (**fabs()** is so called to distinguish it from **abs()** which takes an integer parameter, and is of integer type. **abs()** is defined in **stdlib.h**).

```
/*      maths3.c
        This program introduces some predefined mathematical functions. */

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

main( )
{
        double x, y;

        printf("Enter two numbers, x and y: ");
        scanf("%lf %lf", &x, &y);
        printf("The square root of x is %lf\n", sqrt(x));
        printf("The absolute value of y is %lf\n", fabs(y));
        printf("x to the power y is %lf\n", pow(x,y));
}
```

Exercise:
Change the program so that it calculates the position and speed of a simple harmonic oscillator at any desired time t:

$$s = a \cos(\omega t+\varphi), \qquad v = -a\,\omega \sin(\omega t+\varphi)$$

The program should ask for the amplitude (a in m), the angular frequency (ω in s^{-1}, related to the time T for 1 oscillation by $\omega=2\pi/T$) and the constant phase angle (φ) in radians.

3.0 Iterations.

Often we require a computer program to repeat the same operation or set of operations a number of times. This is achieved very concisely by using loops which cause the statements within the loop to be repeated until a certain condition is reached. In C there are three main types of loop: the 'for' loop, the 'while' loop, and the 'do-while' loop.

N.B. When learning how to use iterations it is easy to create a continuous loop accidentally. One way to interrupt the program is to restart the computer by holding down the keys <CTRL><ALT><DEL> simultaneously. However you will lose any unsaved work by this method, so make sure you save all your work BEFORE you compile and run any program.

3.1 The 'for' loop.

The general form of the **'for'** loop is:

```
for(initialisation; condition; increment)
        statement;
```

The **initialisation** sets the initial value for the variable to be used.
The **condition** must be true in order for the loop to continue.
If it is found to be true the statement is executed. The **increment** then increases or decreases the value of the variable as instructed, and this new value is now operated on by the loop.

The **for** loop

```
for(x=10; x>0; x--)
        printf("\n%d", x);
```

prints the numbers 10-1 down the screen. Examine the following program, **for.c**, and try to work out what will be printed on the screen when it is run. Then compile and run it and see if you were right.

```
/*      for.c           The for statement.      */

#include <stdio.h>
#define MAX 10      /* Names of constants are usually written in capitals */

main()
{
        int i, j, k;
        int add=0, sum=0;

        printf("\n");
        for (i=1; i<MAX; i++)
                printf("i=%d, result=%d\n", i, i*i);

        printf("\n");
        for (j=1; j<MAX; j++)
                sum=sum+j*j;
        printf("j=%d, result=%d\n", j, sum);

        printf("\n");
        for (k=1; k<MAX; k++) {
                add=add+k*k;
                printf("k=%d, result=%d\n", k, add);
        }
}
```

- The operators < and > used in this program are examples of relational operators, which are used in conditional statements. Logical operators can also be used in such statements. A list of relational and logical operators is given below, and also in Table 2 of the appendix.

Relational operators:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	not equal to
==	equal to (remember that = does not signify 'equal to' but 'assign the value of the expression on the right to the variable on the left'.)

Logical operators:

&&	AND
||	OR
!	NOT

For example, `x >= 5 && x != 10` means the condition that 'x is greater than or equal to 5 and is not equal to 10'.

Exercises:

1. Run for.c with a different value of the constant MAX .
2. Try using <= instead of < in the condition and see what difference it makes.
3. Write a program which prints to the screen a table of values of log(x) for x=1.0 to 2.0 in steps of 0.1. The table should consist of 2 columns with the value of x in the first column separated by a tab space from the corresponding value of log(x) in the second.
4. See what happens when you compile the program of problem 3 if you try to start from x=0 instead of x=1.0. (log(0) = -∞).

3.2 The 'while' loop.

The general form is

```
while (condition) statement;
```

or

```
while (condition){
        statement 1;
        statement 2;
}
```

Look at **while.c**. (below) The first 'while' loop performs exactly the same function as the first 'for' loop in **for.c**:

```
for(initialisation; condition ;increment)
        statement;
```

is equivalent to:

```
initialisation;
while (condition){
        statement;
        increment;
}
```

```
/*      while.c         The while statement   */

#include <stdio.h>
#define MAX 10

main()
{
        int i=1, j=1, n;
        int add=0, sum=0;
        float x;
        char c;

        printf("\n");
/* The following while loop is equivalent to the first for loop in for.c */

        while (i< MAX){
                printf("i=%d, result=%d\n", i, i*i);
                i=i+1
        }

/* A while statement can be used to read in a list of numbers when the
number of items is not predetermined, as follows:    */

        printf("\nEnter a list of numbers separated by spaces. \n\
Type 0 followed by <RET> to indicate the end of your list. \n");
        n=1;
        while(scanf("%f", &x)==1 && x!=0){
                printf("Entry number %d is %f\n", n, x);
                n++;
        }
        printf("You entered %d numbers", n-1);
}
```

- Note that it is possible (and sometimes necessary) to assign a value to a variable while declaring it. This is called **initialising** the variable (look back at paragraph I.1.3).

It is important to understand how C evaluates **true** and **false**. In a conditional statement, 1 is returned if the condition is true, and 0 if it is false. This is shown in the last section of **while.c** where a while statement is used to read a list of numbers when the number of items is not predetermined:
In the condition while(scanf("%f", &x) ==1 && x != 0), scanf("%f", &x) is only true (returning the value 1) when a number has been typed in, otherwise it is false (returning 0). The loop continues while two conditions are met, i.e. while

1. a floating point number has been entered (i.e scanf("%f", &x) is true and returns the value 1)
2. the number entered is not zero (which the program has instructed the user to enter when their list of numbers is finished). When 0 is entered, the second condition is false, and the returned value 0 causes the program to exit the loop).

Exercises:

1. Experiment to see what happens if you enter a character instead of a float.
2. Write 'while' loops which have the same effect as the other two 'for ' loops in **for.c**. Any of the logical and relational operators can be used in the condition statement.
3. Write a program which asks for a list of numbers to be entered, and calculates their mean and standard deviation and prints them. (The mean of n numbers, a_1, a_2, ……a_n is $\langle a \rangle = (a_1+a_2+\ldots+a_n)/n$, and their standard (or root mean square) deviation is $\sqrt{(\langle a^2 \rangle - \langle a \rangle^2)}$, where $\langle a^2 \rangle$ is the mean of the squares of the numbers: $\langle a^2 \rangle = (a_1^2 + a_2^2 + \ldots + a_n^2)/n$.)

3.3 The do-while loop

The 'for' and 'while' statements test the conditions before performing the operations defined within the loop. So if the condition is not true, even for the first value of the variable, the operations in the loop will never be performed. However, the do-while loop tests the condition at the end of the loop, so the loop operations are always performed at least once. The general form of the loop is

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

The program **dowhile.c** generates a random number each time the loop is performed, and asks whether another random number is required:

```
*/     dowhile.cThe do-while loop.      */

#include <stdio.h>
#include <stdlib.h>          /* For rand()  */
#include <ctype.h>           /* For isspace()  */

main()
{
        char ans;

        printf("\nThis program will print out random numbers\
between 0 and %d.\n", RAND_MAX);

        do{
                printf("%d \n", rand());
                printf("Find another? (Y/N)\n");
                do{
                        scanf("%c", &ans);
                } while(isspace(ans));        /* Skips white space */
        } while(ans=='y' || ans=='Y');        /* Exits unless Y or y input */

}
```

- Note that two new header files have been included in this program:

  ctype.h defines the function **isspace( )** which returns a value of true if white space is encountered and a value false if there is an item to be read from the keyboard. The loop is thus repeated until non-white space is encountered, when the computer moves to the next statement.

  stdlib.h defines the function **rand( )** which picks a pseudo-random integer between 0 and RAND_MAX. RAND_MAX is a constant which is also defined in the file and represents the upper limit of the range of numbers from which C can select a random integer.
  (This is a very primitive random number generator. For information about how such generators work, and prescriptions for better versions see Numerical Recipes in C listed in the references section of the appendix.)

- Note the do-while loop within the main one. This is called nesting .

Exercises:

1. Modify the program to generate a random number between 0 and 1.
2. Write a program to print to the screen a table of values of x and sin(x). The value of x should
   appear in the left hand column separated by a tab space from the corresponding sin(x) in the right hand column. Each different value of x should appear on a new line. Start from x = 0, and evaluate sin(x) for 20 further values, ending at x = 2π.
   NB. To 9 decimal places π = 3.141592654. It is clearly meaningless to quote values of x and
   sin(x) to a greater accuracy than this. Choose the number of places of decimals to which your table will be accurate, and use the 'precision specifier' (see bottom of Table 1) to restrict the number of decimal places printed. (e.g. **%.4lf** will print a double precision floating point
   number correct to 4 places of decimals.)

3.4 Selection statements

The **if** and **if-else** statements are used to instruct the computer to execute one statement if a particular condition is true, but to execute an alternative statement if the condition is false. The general form is:

```
if (expression) statement;
else statement;
```

The **else** clause is not essential. If it is absent, execution proceeds from the statement following the **if** clause when the condition is false.

Look at **ifelse.c** , compile and run it.

```
/*      ifelse.c                The use of selection statements.  */

#include <stdio.h>

main( )
{
        int  i, j, k, a, b, c;
        int min, middle, max;

        printf("\nEnter three integers: \n");
        scanf("%d %d %d", &i, &j, &k);

        if (i<j)
                a=i;
        else
                a=j;
        if (a<k)
                min=a;
        else
                min=k;
        printf("The smallest is %d\n", min);

        printf("This part of the program will order three integers by size.\n");
        printf("Enter three integers:\n");
        scanf("%d %d %d", &a, &b, &c);
        if (a<b && a<c) {
                min=a;
                if (b<c) {
                        middle=b;
                        max=c;
                }
                else    {
                        middle=c;
                        max=b;
                }
        }
        else  {
                if (a>b && a>c) {
                        max=a;
                        if(b<c) {
                              min=b;
                              middle=c;
                        }
                        else   {
                                 middle=b;
                                 min=c;
                        }
                }
                else    {
```

```
                        middle=a;
                        if (b<c) {
                                min=b;
                                max=c;
                        }
                        else    {
                                min=c;
                                max=b;
                        }
                }
        }
        printf("%d , %d, %d", min, middle, max);
}
```

- The first section determines which of three integers is the smallest. It consists of a sequence of two **if-else** statements. (There is information about sorting routines in Numerical Recipes in C - see references section in the appendix.)
- The second section prints three integers in order of size. Note the nesting of the **if-else** statements and the use of **{** and **}** when more than one statement is to be processed after the **if** condition.
- If no **else** clause is included, the program executes the statement(s) associated with **if** provided the expression is true. Otherwise it proceeds directly to the statement which follows **if**(condition){statement};.
- Note that statements nested within an **if** clause are indented, as are nested clauses. This is common practice for readability and helps prevent trivial mistakes such as the omission of a bracket.
  N.B. Although statements can be nested they may **never overlap**, i.e. each inner loop must be completely contained in the outer loop.

4.0 Functions

Functions are an essential element of **structured programming**. This involves dividing a large, complex program into specific tasks, to be performed in sequence. Each specific task is defined in a separate function, which performs its task independently of the rest of the program. The sequence in which the separate tasks are performed is determined by the **main** function (which you have already used.) The main function **calls** other functions in the appropriate sequence, and the command is simply the name of the function to be called, with the values of any parameters that are needed in brackets after the name.

Often the called function will return a value to the main function. However it may also change the value of a **global** variable. A global variable is one that is recognised by the complete program, i.e. by main() and all other functions. This means that the variable must be declared in exactly the same way in all other relevant functions.

Like variables, all functions must be declared. This must be done before the function is called. So if main() is the first function in the program, and a second function, name(), is called from within it, the function name() must be declared at the top of a program, above the beginning of main(). The declaration is called a **function prototype**, and takes the general form

```
type  name(parameter list);
```

type specifies the type of object that will be returned by the function. For example, if the function returns a floating point value the type will be **float** . If the function does not return anything the type will be **void**.
name is the name of the function.
parameter list is the set of variables (separated by commas) on which the function depends.

main( ) is a function which does not usually return a value, and in this case may be assigned the type **void**. However the type specifier for main() is usually omitted, which results in the assignment of the default type, which is **int**.
Examine the program **function.c** which demonstrates the use of a second function called by the main function.

```
/*    function1.c
      This program contains the main function and a second function called by main.   */

#include<stdio.h>

 int square(int y);                        /*function prototype  */
 main()
 {
         int x, answer;                    /*local variable declaration  */

         printf("/nEnter an integer value between 1 and 100: ");
         scanf("%d", &x);
         answer = square(x);               /*calls the square() function  */
         printf("\nThe square of %d is %d.", x, answer);
 }

 int square(int y)                 /* function header, and declaration of parameter y  */
 {
          int  y_squared;                          /* local variable declaration   */

          y_squared=y*y;
          return y_squared;
 }
```

- The **main** part of the program will often be quite short, and even in complex programs may consist almost entirely of a sequence of functions which are called in sequence, and defined below main().
- Note that the function header is identical to the function prototype but without a semicolon.
- After the command return value; has returned the appropriate value to the main function, the operation of the program continues (using that returned value) from the point in **main** at which the function was called.
- Local variables, declared within a function, cannot be used outside that function.
- The **parameter(s)** on which a function depends are variables (e.g. y in the prototype and header for the function square(y) in this program). When the function is called by main, this variable is assigned a particular value, called the **argument** (the current value of variable x in this case.)

5.0 Using files

It is often more useful to place the results of a calculation in a file rather than print them to the screen. In particular, a data file, `filename.dat,` containing one data point per line, can be used with **Gnuplot** to plot graphs of the data. (See section 5). A data point is a pair (in 2D) or triple (in 3D) of values, separated by a space or tab. By default **Gnuplot** takes the values in the first column to be the x coordinates of the points, and those in the second to be the y coordinates. If there is a third column, the values are taken to be the z coordinates.
The program below writes values of x and cos(x) to a file which is called 'cosine.dat'

```
/*   datafile1.c  This program writes data for cos(x) to a file.   */

#include <stdio.h>
#include <math.h>
#define  PI 3.14159265

main()
{
        double x, d=0;
        int i=0, n=0;
        FILE *cosp;                             /* file type declaration  */

        printf("\nEnter the number of values of x between x=0 and x=2PI\n");
        scanf("%d", &n);

        if((cosp=fopen("cosine.dat", "w"))==0){
                printf("cannot open file\n");            /*  returns error message if file
                exit(1);                                     cannot be opened            */

        }
        d=2*PI/(n-1);                   /* calculates the interval in x to be used. */
        x=0                             /* initialise  x  */

        for(i=0; i<n; ++i){
                fprintf(cosp, "\t%lf\t%lf\n", x, cos(x));
                x+=d;
        }
        fclose(cosp);
        printf("data for cosx has been written to the file cosine.dat");
}
```

- **FILE *cosp** is the file declaration, and **cosp** is the address of an area of memory where all the information associated with the file cosine.dat is to be stored. cosp is called a **pointer** to a memory area, and ***cosp** declares the pointer cosp. (To declare a pointer, its name is prefixed by an asterisk.) The 'p' in the name cosp has been included to remind us that it is a pointer name. A FILE type is defined in the header file stdio.h, and the definition includes information about the current state of the file (e.g. open or closed) as well as its contents. You can find out more about pointers in section 7.

- Before information can be written to a file, the file must be opened. The simplest command to use is

```
pointername = fopen("filename", "w")
```

  fopen creates a file called **filename** if it does not already exist, and **"w"** tells the computer to write to the file. This command will overwrite whatever is already contained in the file if it already exists. (**"r"** would tell the computer to read the file, and **"a"** to append information to the end of the file). fopen() is a pointer name, as this command implies. In this program it associates the pointer cosp with the file cosine.dat. In order to check for possible problems, a more complicated set of file-opening statements is used in **datafile.c**. If the file cannot be opened, then the expression

```
cosp=fopen("cosine.dat", "w")
```

  is false (`==0`), and the program generates the error message "cannot open file" before quitting the program. Exit(1) tells execution to stop due to an error.

- A file must always be closed after use, and this is done by the command **fclose(pointername)**. (The functions **exit( )**, **fopen( )**, and **fclose( )** are all defined in **stdio.h**.)

- **fprintf** is the command used to write to a file. The form is the same as that for **printf** except that there is an additional parameter which tells the program the address to print to:
  fprintf(filepointername, "command string", other parameters)

- **fscanf** is not used in this program but it is used to read from the file whose pointername is specified before the command string.

6.0 Using GNUPLOT

GNUPLOT is a windows based, command-driven interactive plotting program. It is case sensitive (i.e. it distinguishes lowercase from capital letters).
The GNUPLOT window has a set of pull-down menus across the top, including:

- **File**, which contains **Open** and **Save** options
- **Plot**, which contains the commands **Plot**, **3D Plot** and **Data filename**. This last enables you to open a data file (with extension .dat) containing data to be plotted.
- **Functions**, which contains a list of predefined functions.
- **Axes**, which includes options for defining the range of values to be shown along the x, y or z axes.
- **Styles**, which allows you to choose to plot using points, lines, etc.

Commands may be entered either using these menus, or by typing the appropriate command directly after the prompt **gnuplot>**.

Plotting predefined functions

Select **Plot** from the pull-down menu (or type **plot** - with lowercase p - at the gnuplot prompt). Select a function from the **Functions** menu, and (to show it is a function of x) type '(x)' immediately after the function name. For example, if you have chosen the sine function, at this

stage you should see the following line on your screen:

gnuplot> plot sin(x)

Now press <RET>, and the graph of sin x will be plotted on a new window.

Changing the range

GNUPLOT uses a default range of x and y unless you define these ranges. To change the x range, select **X Range** from the **Axes** menu, enter the lower limit in the dialogue box which appears, and press <RET>. Then enter the upper limit in the new dialogue box and press <RET>. Now click on the REPLOT button at the top of the window, and your most recent graph will be replotted with your chosen range of x values. Alternatively, select the range and then enter the plot command for your chosen function. The y and z ranges may be chosen similarly.

Symbols used by GNUPLOT

- GNUPLOT uses * and / to represent multiplication and division respectively. The multiplication sign must always be inserted explicitly. For example, GNUPLOT will understand 2*x, but not 2x.
- To enter the square of x, enter 'x*x'. Similarly x to the power 3 should be entered as 'x*x*x' etc.
- GNUPLOT recognises the term **pi**, so, for example, to plot sin x from x=0 to x=2π you can type '2*pi' for the upper limit of the x range. (Notice that the sign '*' must be used to indicate multiplication).

Plotting user-defined functions:

To plot a function which does not appear in GNUPLOT's list of functions, first define the function, f(x) for a 2-dimensional graph, or f(x,y) for a 3-dimensional one. For example:

gnuplot> f(x)=3*x*x-4

Now use the plot command from the pull-down menu (or type 'plot') and type 'f(x)' after the plot command. The graph will be plotted when you press <RET>. (If you do not like the default choice of range on the x axis, change the range as described in paragraph 4.2 and repeat the plot command (or press the **REPLOT** button.)

Variable names

By default, GNUPLOT assumes that the independent variable for the **plot** command is x (2D), and the independent variables for the **splot** command are x and y (3D). They are called the dummy variables. The **set dummy** command changes these default dummy variable names. For example, it may be more convenient to call the dummy variable t when plotting functions of time. The following commands plot sine and cosine as functions of t, on the same graph :

gnuplot> set dummy t

gnuplot> plot sin(t), cos(t)

To set two dummy variables in the case of a 3-dimensional plot, use the **set dummy** command with the two variables separated by a comma:

```
gnuplot> set dummy u, v
gnuplot> f(u,v)=u*u+v*v
gnuplot> splot f(u,v)
```

Plotting data from a data-file

Data contained in a file can be displayed by specifying the name of the data file (enclosed in single quotes) after **plot** or **splot**. After choosing the appropriate plot command, select **Data filename** on the **Plot** menu to obtain a window in which you may browse to find the file you want. Data files have a '.dat' extension (filename.dat), and should contain the information for plotting one data point per line. (Lines beginning with # will be treated as comments and ignored). For **plot**, each data point represents an (x,y) pair, and each line of the data file should contain an x value followed by a blank space and then the corresponding y value. For **splot**, each point is an (x,y,z) triple. Again the numbers on each line of the data file must be separated by blank space, which divides each line into columns.

Plotting data from a data-file containing multiple columns

It is possible to select two columns to plot from a data-file containing more than two columns. For example, the command

```
plot 'datafile'  using 1:3
```

will use the first column of the file datafile.dat as the x column, and the third column as the y column. To show two curves on the same plot, using column 1 of the data-file as the x column and columns 2 and 3 as separate functions of x, type

```
plot 'datafile'  using 1:2 , 'datafile'  using 1:3
```

Alternatively the following two lines will have the same effect:

```
plot 'datafile'  using 1:2
replot 'datafile'  using 2:3
```

7.0 Arrays

Programmers use arrays to store many elements of the same type (e.g. integer, float, etc.). Only one variable name is used, and the elements are distinguished from each other by a subscript. An array must be declared (just like any other variable), at the beginning of a program, and the number of elements must be specified at this point so that the computer can set aside a block of memory of sufficient in size. For example, an array with 5 integer elements would be declared as int array[5]. The individual elements are named array[0], array[1], array[2], array[3], array[4].

N.B. In C **the first variable of an array always has the subscript [0]**. Therefore the elements in array[n] are array[0], array[1], array[2], …, array[n-1].

Elements of an array can be used in a program in the same way as variables of the same type. For example, a vector could be seen as an array. Suppose we have a velocity in 3-dimensional space: we specify it by giving the x, y, and z components, v_x, v_y and v_z, and these are treated as elements in an array velocity[3]. (In vector notation we denote the velocity by **v**, which really means the set of three numbers which are its components). Elements can be used individually in a calculation. For example, if the speed is required , $v = |\mathbf{v}| = \sqrt{v_x^{\,2} + v_y^{\,2} + v_z^{\,2}}$, and the C statement generating this would be:

```
v = sqrt(velocity[0]*velocity[0] + velocity[1]*velocity[1] + velocity[2]*velocity[2]);
```

or perhaps more neatly:

```
vsquared = 0;
for(i = 0; i < 3; i++)
        vsquared = vsquared + velocity[i]*velocity[i];
v=sqrt(vsquared);
```

Look at the program **array.c** which asks for 10 numbers to be entered and evaluates their sum and the sum of their squares, treating the numbers as the elements of a 1 dimensional array.

```
/*      array.c
        This program sums the elements of an array and also their squares.      */

#include <stdio.h>
#define N 10

main()
{       float x[N], sum, sum2;
        int i;
        sum=0;                          /* Initialises variables to zero */
        sum2=0;

        for(i=0; i<N; ++i) x[i]=0;      /*  Initialises all array elements to zero - not strictly
                                            necessary in this case.     */
        i=0;            /* Note you need to set i back to 0 before you use it again -*/
                        /* what happens if you don't ?  */
        do{
                printf("Enter the value for x[%d]: \n",i);
                scanf("%f", &x[i]);
                i++;
        }
        while(i<N);
        for(i=0; i<N; ++i){
                sum += x[i];
                sum2 += x[i]*x[i];
        }

        printf("The sum of the elements is %f.\n", sum);
        printf("The sum of the squares of the elements is %f.\n", sum2);
}
```

- Note that in general all the elements in an array should be initialised (i.e. assigned a value) so that you know what value is in each element before you start. This is most easily done with a 'for' loop. If this is not done the computer will use the value that had previously been stored in that section of memory, with unpredictable results. In array.c this initialisation is not strictly necessary because the value for each element is read in from the keyboard.
- In small arrays such as this the elements could have been initialised when the array was declared:

```
int array [ 5] = {0, 0, 0, 0, 0};
```

The for loop which initialises the array elements in the program above would then be unnecessary.

7.1 Multidimensional arrays

Arrays can be multidimensional. For example, a 2 dimensional array (a matrix) for representing a chess board could be **chess [8] [8]**. This array has 64 elements: **chess[0] [0], chess[0] [1], chess[0] [2],…, chess[3] [4], chess[3] [5],…, chess [7] [6], chess[7] [7]**. Note the order in which they are presented, with the second dimension of the array (the 'column' number, in matrix terms) being incremented first.

8.0 Pointers

In the work that you have done so far, the particular section of computer memory allocated to a particular variable or array by the compiler has been unimportant to the programmer. The value of a variable or a series of values stored in an array were accessible by making reference to the variable name or array name and array index. However, the C Language also permits the programmer to have access to, and use of, the actual addresses at which variables or array values are stored, using a language feature called **pointers**.

A **pointer** is a variable which stores a value which is the memory address of another variable or element in an array. This address is always an integer. The number of bytes required for an address depends on the type of variable to be stored there: a character requires 1 byte (=8 bits), an integer requires 2 bytes, a floating point number requires 4 bytes, and a double precision floating point number requires 8 bytes. (See Table 1 in the appendix.) Pointers have already been used in writing to a file in section 4.

8.1 Pointer declaration

When a pointer or pointer array is declared, it is distinguished from an ordinary variable or array by being preceded by an asterisk, *. The value of a pointer is always an integer, but the total size of the memory area it points to must also be included in the declaration. This is determined by the type of the variable to which the pointer refers, as explained above, and this is called the pointer **base type**. It is this type which must be included in the pointer declaration. For example, look at the statements

```
int a, *xp;
double b, *yp, *zp[10];
```

The first declares an ordinary integer variable a, and a pointer xp to an integer variable. The second declares a variable b, of type double, a pointer yp to a variable of type double, and a pointer zp to an array which contains 10 elements of type double. Although the pointers yp and zp have integer values, their base types are double, corresponding to the type of the variables they point to. The names xp, yp and zp here include a 'p' as a reminder that they are names of pointers. It is often convenient to choose pointer names which contain some reminder that they refer to pointers rather than to ordinary variables.

When declaring a pointer to a file, as in section 4, the declaration is of the form

```
FILE *pointername;
```

The FILE type is defined in stdio.h , and includes, as well as the file contents, information about the state of the file, and whether it is being written or appended to or read from.

Although it is often unnecessary to initialise ordinary variables, this is never the case with pointers**.**

Pointers must always be initialised before they are used. An unitialised pointer can cause your program or operating system to crash, as it may inadvertently alter the contents of a section of memory which is vital to the program or operating system. A pointer is initialised using a statement such as,

```
xp = &a;
```

In this context, the & operator is the '**address of**' operator, and in the statement above, the address of the ordinary variable named a is assigned to the pointer xp. After this initialisation, pointer operations involving xp will only affect the section of memory which has been assigned by the compiler to the variable a.

The second operator which may be applied to a pointer is the '**value of**' operator, *. The statement,

```
b = *xp;
```

assigns the value of the variable stored at the memory address indicated by the pointer xp, to the variable b. Look at the sequence of statements below:

```
int a, b, *xp;          /* declare integers a, b, pointer xp        */
a = 35;                 /* initialise a     */
xp = &a;                /* assign the memory address of a to xp  */
b = *xp;          /* assign the value of the variable whose memory address is stored at xp
to b */
```

This is equivalent to,

```
int a, b;
a = 35;
b = a;
```

8.2 Pointer arithmetic

Since the value of a pointer is an integer, an **integer may be added to a pointer**, and pointers may be **incremented**, **decremented**, or **subtracted from each other,** but they cannot be added, multiplied or divided.

Incrementing a pointer xp using xp++ (equivalent to xp+1) changes the indicated memory address to the beginning of the next address of that type. Suppose xp has base type int, then it points to the address of an integer variable which is 2 bytes long. If the address (i.e. the value of xp) is 1000, then xp++ has the value 1002, since the next address for an integer variable is 2 bytes further on in the memory. Similarly if yp is a pointer of base type double, and yp has the value 2000, yp++ will have the value 2008.
Thus adding an integer n to a pointer changes the value of the pointer by nb, where the value of b depends on the base type: b=2 for int, 4 for float and 8 for double.
For example,

```
int a, b, numele, *xp, *yp;        /* declare variables and pointers */
xp = &a; yp = &b;                  /* initialise pointers            */
xp = xp + 10;          /* xp points to an address 10 adresses beyond the original one */
xp++;                  /* increment xp to xp+2 (since xp has base type int) */
xp--;                  /* decrement xp to xp - 2                 */
numele = xp - yp;      /* numele is the number of memory locations of base type int
                              between xp and yp */
```

8.3 Pointers and arrays

Arrays consist of contiguous memory locations, for example, an integer array data[5] could be stored in the memory locations thus:

Memory location	1000	1002	1004	1006	1008
Array element stored	data[0]	data[1]	data[2]	data[3]	data[4]

Since an array consists of a number of different elements, a pointer cannot point to the whole array but only to its first element. The name of the pointer is simply that of the array, without brackets. **Arrays are the only case in which an asterisk is not needed to signify a pointer**. In the case above, **data** is identical to **&data[0]** (which indicates the address where the first element of the array is stored). Since the name of an array is a pointer constant, it cannot be changed for the duration of program execution.

Pointers are useful for referencing particular array elements, and this can be faster than the usual means of referencing array elements. For example, the fifth array element of array data can be referenced using,

```
int a, data[5], *xp;          /* declare variable, array and pointer      */
xp = data;                    /* initialise pointer       */
a =*(xp+4);       /* xp points to the first element of data, xp+4 points to the fifth element */
```

An equivalent pair of statements is,

```
int a, data[5];
a = data[4];
```

(Remember that the first element is data[0], so the fifth is data[4].)

8.4 Use of pointers in functions

A function can be called **by value**, i.e. by passing the **value of the argument** to the function or it may be called **by reference**, i.e. by passing a **pointer to an argument**, instead of the argument itself. Both methods of calling a function are illustrated below.

```
/* pointer.c   This program calls a function by value and a function by reference.*/

#include <stdio.h>
int square(int y);                                  /* Function  prototype */
int cube(int *xp);
main()
{
int y = 5, *xp;
xp = &y;
printf("%d\n%d",square(y),cube(xp));                /* or cube(&y) */
}

int square(int y)                                    /* Function header  */
{
int y_squared;
y_squared = y*y;
return y_squared;
}

int cube(int *xp)                                   /* Function header  */
{
int y_cubed;
y_cubed = *x p* *xp * *xp;
return y_cubed;
}
```

So far it has only been possible to pass single values as arguments to a function, and to return single values. If a function is to return more than one value, pointers must be used. For example look at the following main function which calls a function addten():

```
        main(){
        int x=2, y=3;
        addten(&x, &y);
        }

void    addten( int *px, int *py)
        {
```

```
        *px =*px + 10;
        *py = *py + 10;
        }
```

The function addten() adds ten to the numbers stored at the addresses &x and &y, pointed to by *px and *py. In this case the function does not return any values and so is of type void.

To pass several variables to a function, or to return several variables, it is convenient to use arrays. Array names are pointers, as explained in section 4. If the pointer to an array is passed to a function, then the function knows the address and can process the whole array by moving from one element to the next by incrementing the address. The function must also be told the size of the array which it is being passed so that it knows when to stop incrementing the pointer! To let a function 'know' the size of an array, the size can be passed to it as an argument. Thus the function is passed two arguments - a pointer to the first element of the array and an integer specifying the number of elements in the array. A function header might be:

```
        int squared(int *p_array, int x);
or      int squared(int array[], int x);
```

where x is the number of elements in the array. The following program passes an array to a function in order to find the largest element.

```
/*largest.c       Finds the largest element of an array  */

#include<stdio.h>

int array[10], num;
int largest(int *p, int y);

main()
{
        for(num=0; num<10; num++)
        {
                printf("\nEnter an integer value:");
                scanf("%d", &array[num]);
        }

        printf("\n\nThe largest value is %d", largest(array, 10));
}

int largest(int *p, int y)
{
        int num=0, biggest= -12000;

        for(num=0; num<y; num++)
        {
                if(p[num] > biggest)
                biggest = p[num];
        }
        return biggest;
}
```

9.0 The xxgdb debugger

10.0 Numerical methods of integration

See Kreyszig, Advanced Engineering Mathematics, 7th edn. sections 18.5 and 20.3

10.1 Definite integrals

The problem is to evaluate an integral of the form $\int_a^b dx\, f(x)$ for a given function *f(x)* between fixed limits $x=a$ and $x=b$. Approximate methods are based on the interpretation of this integral as the area under the curve $y=f(x)$ between the given end points.

In the simplest possible case, the area under the curve is approximated by a large number of rectangles of equal width $h=\Delta x$, where the height of the $(i+1)^{th}$ rectangle is $f(x_i)$, with $x_i=a+ih$. If there are are n such rectangles, then the width of each is $h=(b-a)/n$. The value of the integral is given approximately by the sum of the areas of the n rectangles:

$$\int_a^b dx\, f(x) \approx \sum_{i=0}^{n-1} f(x_i)\, h \qquad (1)$$

The trapezoidal rule

An improvement on the simplest approximation (1) is to estimate the area of each section of width h as if it were a trapezoid. The area of the section between x_i and x_{i+1} is then approximately

$$\frac{h}{2}\Big(f(x_i) + f(x_i + h)\Big) \qquad (2)$$

and we sum these areas from i=0 to $n-1$.

The error in this approximation is given by ε, where

$$K f^{(2)}_{max} \le \varepsilon \le K f^{(2)}_{min,} \qquad K = -(b-a)^3/(12\, n^2) \qquad (3)$$

and $f^{(2)}_{min}$ and $f^{(2)}_{max}$ are the minimum and maximum values of the second derivative of f(x) in the range $a \le x \le b$.

Simpson's rule

Simpson's rule is based on approximating the curve f(x) bounding the top of each section of area by a quadratic function. In this case the interval $b-a$ must be divided into an even number of steps, $2n$, each step being of the same length $h=(b-a)/(2n)$. The endpoints of the steps are:

$$x_0=a,\ x_1=a+h,\ x_2=a+2h,\ \ldots\ldots\ldots\ x_n=a+2nh=b$$

The function is approximated in the first double interval by $f(x) \approx Ax^2 + Bx + C$, where A, B and C are calculated by fitting this quadratic function to the known values of f at x_0, x_1 and x_2 . This approximate function is integrated from x_0 to x_2 and the result expressed in terms of h, $f_0=f(x_0)$, $f_1=f(x_1)$ and $f_2=f(x_2)$.

The contribution of the area from each double interval is estimated in a similar way, and the results summed to give the following approximation:

$$\int_a^b dx\, f(x) \approx \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots\ldots + 4f_{2n-1} + f_{2n} \right) \qquad (4)$$

Frequently this is written in terms of the double step length $H=2h$, and can be written as either of the two summations below:

$$\int_a^b dx\, f(x) \approx \frac{H}{6}\left(f(a) + 2\sum_{i=1}^{n-1} f(a+iH) + 4\sum_{i=1}^{n} f(a+(i-\frac{1}{2})H) + f(b) \right) \qquad (5)$$

$$\int_a^b dx\, f(x) \approx \frac{H}{6}\sum_{i=1}^{n}\left(f(a+(i-1)H) + 4f(a+(i-\frac{1}{2})H) + f(a+iH) \right) \qquad (6)$$

The error in this approximation is of the order ε, where

$$K' f^{(4)}{}_{max} \le \varepsilon \le K' f^{(4)}{}_{min}, \qquad K' = -(b-a)^5/(180\,(2n)^4) \qquad (7)$$

and $f^{(4)}$ is the fourth derivative of $f(x)$.

10.2 Differential equations

Simple Euler method

Suppose that we wish to solve the differential equation

$$\frac{dx}{dt} = f(x) \qquad (1)$$

given some initial value of x, x_o. For some functions, f, it is possible to obtain an analytic solution, in which case we would find a function $x(t)$ which satisfies equation (1) and $x_o = x(0)$. In some cases an analytic solution of this form cannot be found and the solution must be found numerically.

The simplest way to integrate an equation of the form (1) numerically is the ***simple Euler method***: Given an initial value $x = x_o$ we can estimate the value of x at time $t_1 = t_o + \Delta t$ using the first two terms of a Taylor series

$$x_1 \approx x_0 + (t_1 - t_0)\left.\frac{dx}{dt}\right|_{t_0} \qquad \text{or} \qquad x_1 \approx x_0 + \Delta t\ f(x_0) \qquad (2)$$

where $\Delta t = (t_1 - t_o)$.

The numerical solution to the problem is a set of pairs of data points, (x_i,t_i), where each time t_i is

separated from the next by the interval Δt . The graph of the function *x(t)* may then be plotted from this set of data points.

Improved Euler method

The Euler Method is rather crude and it is possible to improve on it:

(i) make an *estimate* of the slope at $t_i + \Delta t$ using the Euler Method, giving an approximate value $f(x_i + \Delta t)$

(ii) use the average of the slope in (i) and the slope $f(x_i)$ at t_i to calculate x_{i+1}.

This is equivalent to using the trapezoidal rule at each step of the integration. The result is

$$x_{i+1} \approx x_i + \frac{\Delta t}{2}\left(f(x_i) + f(x_i + \Delta t)\right) \tag{3}$$

The second term on the rhs is the *average* of the slopes mentioned above. Note that the slope at the end of the interval has been estimated *since we do not actually know where the new value of x lies*.

The improved Euler method is correct to second order, i.e. to terms of order $(\Delta t)^2$.

Runge-Kutta method

It is possible to derive a series of higher order approximations for x_{i+1} which use values of the slope computed at several intermediate points along the step interval. These are Runge-Kutta Methods (RKM). The title ***Runge-Kutta Method*** with no qualification usually refers to the 4th order RKM which is explained in this section. (The Improved Euler Method is the 2nd order Runge-Kutta Method). Just as the improved Euler method corresponds to using the trapezoidal rule for each step, the Runge-Kutta method corresponds to using Simpson's rule.

At the end of the ith step of the integration we calculate

$$k_1 = f(x_i) \tag{4a}$$

$$k_2 = f(x_i + \frac{\Delta t}{2}) \tag{4b}$$

$$k_2 = f(x_i + \frac{\Delta t}{2} k_1) \tag{4c}$$

$$k_2 = f(x_i + \Delta t\, k_2) \tag{4d}$$

and the 4th order RKM approximation for the value x_{i+1} at the end of the next step is

$$x_{i+1} = x_i + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{5}$$

APPENDIX

TABLE 1 : C NUMERIC DATA TYPES

INPUT TYPE	PRECISION SPECIFIER AND DECLARATION CODE	MEMORY REQUIRED IN BYTES	MINIMAL RANGE
character	%c char	1	-128 → 127
integer	%d int	2	-32 768 → 32 767
floating point	%f float	4	6 digits of precision
long integer	%ld long	4	-2,147,483,648→2,147,483,648
double precision floating point (long float)	%lf double	8	10 digits of precision
scientific notation	%e		
string	%s char *		
minimum field width specifier (displays integer in space at least e.g.8 characters wide)	e.g. %8d		
precision specifier (displays float with e.g. .4 decimal places)	e.g. %.4f		

TABLE 2 : THE C OPERATORS

ARITHMETICAL OPERATORS	
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus : gives the remainder when the first operand is divided by the second
--	Decremental : subtract 1
++	Incremental :add 1
RELATIONAL OPERATORS	
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

LOGICAL OPERATORS	
&&	AND
||	OR
!	NOT

[There are other C operators which you have not yet encountered which can be used when you progress to more complex programming.]

TABLE 3 : PRECEDENCE OF C OPERATORS

PRECEDENCE	OPERATOR
Highest	( ) [ ]
	! ++ --
	* / %
	+ -
	< <= > >=
	!= ==
	&
	&&
	||
Lowest	= += –= *= /=

TABLE 4 : FUNCTIONS INCLUDED IN math.h

sin(x)		exp(x)	e^x
cos(x)		log10(x)	$\log_{10}(x)$ (log to the base 10)
tan(x)		sqrt(x	$\sqrt{x}$
sinh(x)		log(x)	ln x (natural log)
cosh(x)		asin(x)	$\sin^{-1}(x)$
tanh(x)		acos(x)	$\cos^{-1}(x)$
fabs(x)	|x|	atan(x)	$\tan^{-1}(x)$
pow(x, y)	x^y		

TABLE 5 : RESERVED KEYWORDS

char	simplest C data type
do	used with while statement
double	C data type
else	signals alternative statement to be executed when a condition evaluates as false.
float	C data type
for	C loop command
if	Changes program flow based on a true/false decision
int	C data type
long	C data type
return	returns a value to the main function
void	indicates that a function does not return anything

References

- Aitken and Jones, Teach Yourself C programming in 21 Days, Sams (1995).
- Kreyszig, Advanced Engineering Mathematics, Section E, Wiley.
- Koonin, Computational Physics, Benjamin/Cummings (1986).
- Press et al, Numerical Recipes in C, Cambridge University Press.
- Schildt, C: The Complete Reference (2^{nd} edition), Osborne (1990).