

Science and Computers II: Project 1

This project is intended for students who do not have prior experience with UNIX and the C programming language, or would like to refresh their knowledge before attempting the later projects. The project will introduce the UNIX operating system, the `vi` editor, the `gnuplot` plotting program and basic C programming. The first few pages of this project are intended to be used as a reference. You will become familiar with these commands as you use them in the practical sessions.

The Unix Operating System

What is an operating system?

It is the resident software on a computer which enables you to log on, edit a file, print files, etc. The Unix operating system is widely used in both servers and workstations. There are many variants of the Unix operating system which come from different vendors (e.g Apple's Mac OS X, Sun Microsystems' Solaris, IBM's AIX, etc.). You will be using the Windows PCs in the computing lab (or your own computer) to log into a server running Linux, an open-source implementation of the Unix operating system.

Unix is case sensitive, i.e. it distinguishes between upper and lower case letters in filenames, thus `file1`, `File1` and `FILE1` are three separate files.

The following is a list of basic Unix commands:

<code>cp file1 file2</code>	makes a second copy of <code>file1</code>
<code>mv file1 file2</code>	renames <code>file1</code> as <code>file2</code>
<code>cat file</code>	lists <code>file</code> on the screen
<code>rm file</code>	permanently remove <code>file</code>
<code>diff file1 file2</code>	list the differences between the two files
<code>cd</code>	change to home directory
<code>cd directory</code>	moves to selected directory
<code>cd ..</code>	move to the next higher directory
<code>pwd</code>	prints the absolute path of the current directory
<code>mkdir directory</code>	create the named directory
<code>rmdir directory</code>	permanently delete the named directory
<code>ls</code>	gives a listing of files in the current directory
<code>ls -l</code>	as above but more information
<code>ls -a</code>	lists all files in current directory, i.e including those beginning with <code>.</code>
<code>file filename</code>	prints the file type of <code>filename</code>
<code>!string</code>	repeats last command starting with <code>string</code>
<code>!!</code>	repeat last command
<code>CTRL-c</code>	hold down the control key and press <code>c</code> to kill a process

```

enscript -o filename.ps filename  converts the text file filename to a PostScript file
                                  called filename.ps
pd2pdf filename.ps                converts the PostScript file filename.ps to the
                                  PDF file filename.pdf
kpdf filename.pdf                 view the PDF file filename.pdf

```

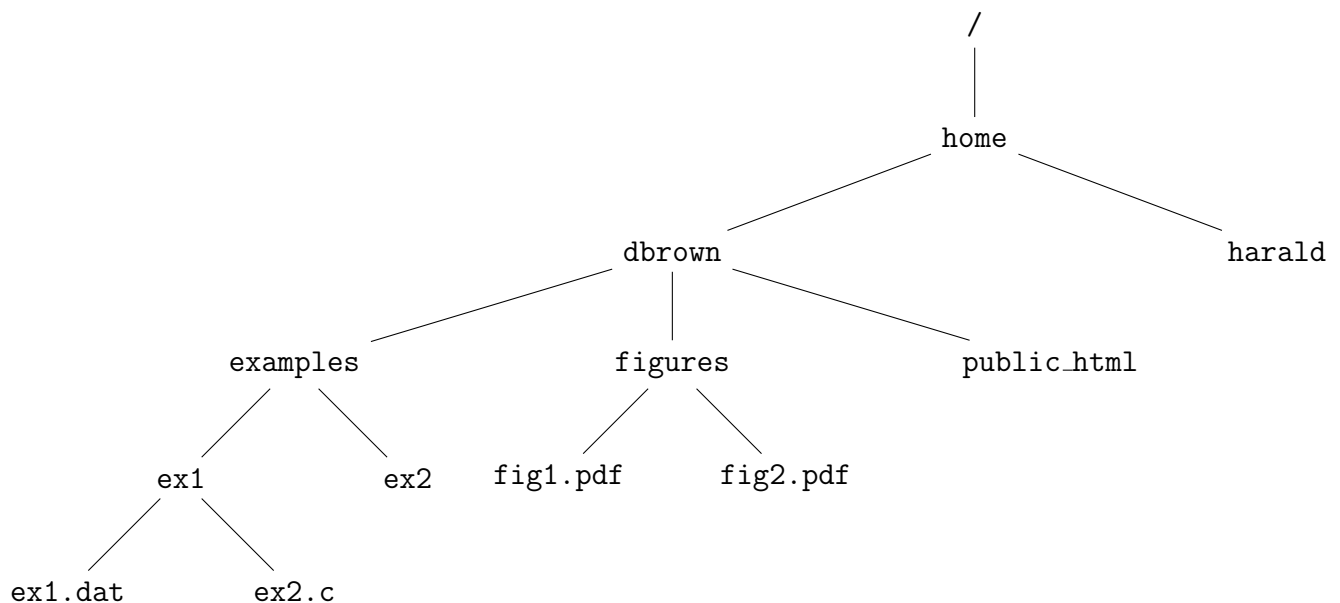
The character `*` is a *wildcard*—it can be used in a command to represent any sequence of characters, e.g.

```
ls ftn*
```

gives a listing of all the files in the current directory beginning with the letters `ftn`.

Warning: be very careful using the command `rm` with wildcards—you may accidentally delete files that you want to keep!

The directory tree—an example



Assuming your home directory is `/home/dbrown`, to go from the directory `ex1` to `figures` you can use any of the following procedures:

- `cd`
 `cd figures`
- `cd ../../figures`
- `cd /home/dbrown/figures`

The Unix command `man` can be used to access the online manual. For example:

```
man enscript
```

will print out the manual and all options for the `enscript` command. It can also be used to get information about C standard library functions, e.g. `man printf` will print documentation for the `printf()` function.

Text editors

There are two editors available on the Linux server: **vi** and **nano**. The **vi** editor has two modes of operation and takes some getting used to at first. It is worth leaning **vi**, as it ubiquitous on Unix systems and very powerful. However, if you find that trying to learn **vi** is slowing you down, you can use **nano** which has an interface similar to the Windows notepad.

To start the **nano** editor, type

```
nano filename
```

at the command line prompt. The available commands are given at the bottom of the window. The character `^` means “hold down the control (CTRL) key.” For example, to exit **nano** hold down control and press the **x** key.

To start **vi**, type

```
vi filename
```

at the command prompt. The two modes of operation in **vi** are:

command mode: each key represents a command, so when you type the keystrokes are interpreted as commands which move the cursor around, etc. The editor always starts up in this mode.

insert mode: this is the normal mode of operation in most editors—any key you press will be entered as a character in the file. Use the keys **i**, **a**, **o**, or **R** to enter insert mode and **ESC** (the escape key) to return to command mode.

Below is a list of **vi** commands:

- To move around the file:
 - h** - move the cursor one space to the left
 - l** - move the cursor one space to the right
 - j** - move the cursor up one line
 - k** - move the cursor up down line
(alternatively the arrow keys can be used)

 - 0** - go to start of current line
 - \$** - go to end of current line

 - ^f** (hold down the control key and press **f**) - move forward full screen
 - ^d** (hold down the control key and press **d**) - move forward half screen
 - ^u** (hold down the control key and press **u**) - move up half screen
 - ^b** (hold down the control key and press **b**) - move up full screen

 - :n** - move to line number **n**
- To enter insert mode:
 - i** - insert before cursor
 - a** - insert after cursor
 - o** - opens new line below current one
 - R** - overwrite existing textExit insert mode by pressing escape.

- Deleting text:
 - `x` - deletes the single character under the cursor
 - `dw` - deletes the current word
 - `dd` - deletes the current line
- Cutting and pasting:
 - `p` - inserts most recently deleted text after current cursor position
 - `yy` - copies current line into buffer - insert using `p`
- Pattern searching:
 - `/pattern` - find first occurrence of `pattern`
 - `//` - find next occurrence of `pattern`
 - `?` - find previous occurrence of `pattern`
- Substitution:
 - `:s/old/new/` - replaces first occurrence of `old` with `new` on current line
 - `:s/old/new/g` - replaces all occurrences of `old` with `new` on current line
 - `:%s/old/new/g` - replaces all occurrences of `old` with `new` on file
- Writing and quitting:
 - `:q` - quits file without writing changes - the editor will not allow you to do this if you have altered the file—to override type `:q!`
 - `:w` - write changes to disk without quitting
 - `:wq` or `:x` - write changes to disk and exit the editor
 - `:w filename` - writes amended version to `filename`
 - `:r filename` - reads `filename` into current document
- Other commands:
 - `r letter` - replaces character under cursor with `letter`
 - `u` - undoes previous command
 - `~R` (control-shift-r) - redo previous command
- Multiple operations can be specified by typing a number immediately before the command, e.g. `3dd` deletes the next 3 lines and `26j` moves down 26 lines.

Introduction to Unix

For many of you this will be your first experience of using a Unix workstation. There are many different exercises in this section. Don't think that you have to complete them all in the first class. The idea is that you become familiar with these at your own pace. If all of this is very new to you, you should plan on spending extra time on these exercises before the next class.

You will be developing and running your programs on a Unix server in the physics cluster room. You will use a program called `ssh` to log into the server. You will also use a program called an "X server" to allow the Unix server to display graphics on your computer. On the physics Windows machines in room 115, both these programs are combined into a program called X-Win32. Documentation for X-Win32 is available online at:

http://www.starnet.com/support/xwin32/9.2/whskin_homepage.htm

Note for Mac OS X users: If you have a Mac OS X laptop, you can log into the server from your laptop. To do this, open the Terminal application (under Applications/Utilities) and type:

```
ssh -X -Y username@sugar-dev1.phy.syr.edu
```

replacing `username` with the username you are assigned in class. You will then be prompted for a password. The `-X` and `-Y` options are important, as they allow the server to display graphics on your laptop. Once your laptop connects to the server, you will be prompted for your password. Enter this and you will be logged into the Unix server.

In this session *everyone* should carry out parts 1–3 on the Windows machines in Room 115. You can log into the Windows machine using your SU NetID.

- 1. Create a session for logging onto the server.** The first time you log use X-Win32 you need to create a session for logging into the server. Start the X-Config program (found under "Start → Programs"). In the X-Win Configuration window, select the Sessions tab. In the Session window, select the folder where the new session will be created (e.g. "My Sessions"). Push the Wizard button and the Session Wizard window will open. In the Session Wizard window, highlight SSH, enter the name for this session (e.g. "phy308 session"), then press the "Next" button. The session wizard then asks you to enter the host to connect to. The server we will be using for this class is `sugar-dev1.phy.syr.edu`. Enter this in the "Host" field and press the "Next" button. In the next Session Wizard window, enter your Login name and Password to access the host, then press the Next button. You should use the Login name and Password that you are assigned in the first class. In the next Session Wizard window select "Linux" from the list of commands in the text window and press the "Finish" button.
- 2. Log onto the server.** Start X-Win32. In the tray, right-click the X-Win32 icon. Highlight and click the session you just created. The session is launched and the status window opens which logs the progress of launching.
- 3. Change your password.** The first time you log onto the server, you should change your password from the one assigned to you. Your new password should contain at least 8 characters and be a mixture of letters, numbers and special characters (such as

!@#%&*()_+=). Choose something that you will remember, but do not select a plain text word. To change your password, type

```
passwd
```

You will then be asked to enter your old password and your new password (twice to ensure that there are no typing errors—remember that your password will not appear on the screen).

4. Practice using Unix and the `vi` editor—these are some suggestions that you might like to try. Type `ls` to determine your file structure. You probably won't have any the first time you log into the server. This is not quite true—if you type `ls -a` you will find that you already have a few hidden files.

Make a directory called `practical1`.

Copy the file `/home/dbrown/gravity_html/teaching/phy308/intro.txt` into this directory. Open the file in `vi` and try moving the cursor around the file. Use pattern searching to find each occurrence of the word “Unix.” Delete the fourth paragraph and then move the second paragraph so that it becomes the first paragraph.

Copy the file `/home/dbrown/gravity_html/teaching/phy308/prime.txt` into the same directory. List the file using `cat`. There are two typing mistakes “hsa” in the first line should be “has” and “prmie” at the end of the paragraph should be “prime.” Use `vi` to correct to make these corrections and add the next few prime numbers to the list.

Create a new file in `vi` and enter a few lines of text, e.g. your name, address and date of birth. Do not be too concerned about typing errors at first. You can go back and alter these once you have entered all the information.

Convert the text file `intro.txt` to PostScript using `enscript` and then to PDF using `ps2pdf`. View the resulting PDF file using `kpdf`.

Use `pwd` to determine your current directory. Change back to your home directory by giving the absolute path name.

5. Display some graphics using `gnuplot`—there is another plotting exercise later in this project, but you should run `gnuplot` to check that you can display graphics from the Unix server on your local computer. To run `gnuplot` simply type

```
gnuplot
```

Mathematical functions such as $\cos x$ can be plotted by giving the command

```
plot cos(x)
```

A list of other functions available for plotting can be found by typing

```
help expressions functions
```

If you encounter any problems displaying the plot of $\cos x$, see me. To exit `gnuplot` type

```
quit
```

6. Finally... you should always log out when you finish a session. To do this, just type `logout` at the command prompt. Do not forget to also log out of the Windows machine, if you are using a lab machine.

Introduction to C

We will begin with an example C program. We will point out the relevant features of this program below.

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

/* program to evaluate the polynomial  $x^3 + 2x^2 - 120x + 11/3$  */
/* for a value of x given by the user on the command line.      */

int main ( int argc, char *argv[] )
{
    /* declare variables */
    float x; /* the independent variable x */
    float poly; /* the value of the polynomial at x */
    FILE *fp; /* a pointer to the output file */

    /* parse the command line argument */
    x = atof( argv[1] );

    /* substitute in polynomial */
    poly = pow(x,3.0) + 2.0*pow(x,2.0) - 120.0*x + 11.0/3.0;

    /* create a new file called output.txt */
    fp = fopen( "output.txt", "w" );

    /* write the output to the file */
    fprintf( fp, "For x = %f the value of the polynomial is %f\n",
        x, poly );

    /* close the output file */
    fclose( fp );

    /* exit the program with return code zero */
    return 0;
}
```

Note the use of the brackets () and {}. The brackets () are used in conjunction with function names, whereas {} are used to delimit the C statements associated with that function. Also note the use of the semicolons. A semicolon ; is used to terminate C statements. C is a free-form language and long statements can be continued from one line to the next (as in the fprintf function call). The semicolon informs the C compiler that the end of the statement has been reached. Since the language is free-form, you can use as much whitespace as you like to make your programs look readable.

Program order

We begin by looking at the overall structure of the program. The general form of a simple C program is

```
pre-processor include directives
main()
{
    /* comments which are ignored by the compiler */
    declare local variables to function main;
    statements associated with function main;
    return code;
}
```

Let us now look at what these sections involve.

Comments

Comments in C are enclosed between the symbols `/*` and `*/`, for example

```
/* this is a comment */
```

Comment lines are not interpreted by the compiler. Comment lines are essential—although you may know exactly how your program works when you are writing it, it may not be so clear if you try to use it again in six months time. The comment lines should describe what the program does and any special features of the program. Comments may appear at any point in the program. Comments may span several lines in your program, for example

```
/* this is a comment
this is still in the comment
now we close the comment */
```

However, for readability it is suggested that you begin and end comments on the same line.

Pre-processor include directives

C is a small, but powerful programming language. The language uses only 32 keywords:

```
auto break case char const continue default do double else enum
extern float for goto if int long register return short signed sizeof
static struct switch typedef union unsigned void volatile while
```

Compared to a language like Python, C is quite primitive. For example, there is no “print” keyword to print a string to the screen. C provides a set of standard libraries of functions to perform many common tasks that are not part of the core language. When you use functions from these libraries, you must use `#include` pre-processor directives to include *header files* in your program. Header files tell the compiler what the standard library functions look like. In the example program there are three pre-processor directives:

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

Note the use of the angle brackets (< and >) around the name of the header file. These indicate that the header file is to be looked for in a list of standard locations built into the compiler. Sometimes you will see the use of double quotes, e.g.

```
#include "myheader.h"
```

which indicate that the current working directory should be searched for the required header file. This will be true when you write your own header files but the standard header files should always have the angle brackets around them.

In the example program, the `stdio.h` header provides the functions `fopen`, `fprintf` and `fclose`, the header `math.h` provides the function `pow`, and the header `stdlib.h` provides the function `atof`.

Note that pre-processor statements, such as `include` *do not* use semi-colons as delimiters. There are other pre-processor directives, but `#include` is by far the most common. All pre-processor directives begin with a `#` and must start in the first column.

Main program

All C programs must have one and only one `main` function. It is the first user-written function executed when the program starts. The main function organizes the functionality of the rest of the program. The statements associated with the main function are enclosed in curly brackets immediately after the function is declared.

The main function also has access to the command arguments given to the program when it is run. In C this is done through the variables `argc` and `argv` passed to the main function by the operating system. `argc` contains the number of arguments passed and `argv` contains the arguments as strings. We will learn more about these later.

At the end of the main program is a `return` statement. The value returned from the main function becomes the exit status of the process. This must be an integer value and is 0 in the example program.

Declaration statements

The purpose of the declaration statement is to define the variables used by the program. Unlike Python, C requires all variables to have a defined type before they can be used. In this example, only three variables are declared: `x` and `poly` are declared as real variables and `fp` is declared as a file pointer (more on this later). The basic data types in C are

int An integer. Linux uses 32 bits to represent a range from $-2,147,483,648$ to $2,147,483,647$. If the `int` keyword is prefixed with the keyword `unsigned`, then the range is 0 to $4,294,967,295$. If you try to exceed this range the result will be garbage and no warning will be given.

float 32 bits are used to represent a *single precision* floating point value with the range $1.17549435 \times 10^{-38}$ to $3.40282347 \times 10^{38}$ (with a similar range for negative numbers). The stored value is accurate to about 6 significant figures. Large and small numbers can be written in exponential form, e.g. 4.3×10^8 is written as `4.3e8` and 6.67×10^{-4} is written as `6.67e-4`.

double 64 bits are used to represent a *double precision* floating point value with the range $2.2250738585072014 \times 10^{-308}$ to $1.7976931348623157 \times 10^{308}$ (with a similar range for negative numbers). Stored values are accurate to about 16 significant figures. Double precision numbers use the same exponential notation as single precision numbers.

char A single character. This is the most basic unit addressable by the machine; typically one byte.

For the time being, we will concentrate on `int` and `float` variables.

Warning: If you do not declare a variable before using it, the C compiler will print the `undeclared` error and fail to build the program when you try to compile it.

Executable statements

The assignment statement

The assignment statement is of the form

```
x = y;
```

where `x` is a variable name and `y` is a mathematical expression or the return value of a function. Note that the `=` sign does not mean “equals” in the mathematical sense. What it means is: evaluate the expression represented by `y` and place this new value in the variable `x`. Thus we can have an assignment statement such as

```
n = n + 1;
```

which means: take the variable `n`, add 1 to this and write the result back into the variable `n`. Note that like all executable statements, the assignment statement is terminated with a semicolon.

The following arithmetic operators can be used in an assignment statement

<code>a + b</code>	a plus b
<code>a - b</code>	a minus b
<code>a * b</code>	a multiplied by b
<code>a / b</code>	a divided by b

The statement which assigns the value of the polynomial to `poly` uses all of these operators. Note that C does not have an operator which evaluates a^b . To do this we must use the `pow` function

```
pow(a,b)    returns a raised to the power b
```

The order which an arithmetic expression is evaluated is

<code>*</code> , <code>/</code>	first
<code>+</code> , <code>-</code>	second

Statements of equal precedence evaluate from left to right.

In order to force the operations to be performed in different orders, parentheses must be used. For example, the expression

$$\frac{a + b}{cd}$$

must be coded as $(a + b)/(c * d)$. You may think that $(a + b) / c * d$ would also achieve the same result, but in fact this produces

$$\frac{(a + b)}{c} \times d$$

as you can easily verify by applying the above rules. Parentheses can be used even when they are not necessary; they can be used to make expressions clearer. As a general rule—*if in doubt use parentheses*.

Warnings

- In integer division, the result is truncated, e.g. if we have the statement $i = 5/3$ the value stored in i is 1.
- Do not mix data types within an expression, e.g. $3.0/5$ may not give the result you expect.
- All operators must be included explicitly, e.g. $4x$ must be coded as $4 * x$.

C includes several other assignment operators, e.g. $a += b$ which is equivalent to $a = a + b$. You should familiarize yourself with these using an online tutorial or The C Programming book.

Input and output

The example program reads the input value of x from the command line when the program is run. The integer variable `argc` and the array `argv` give the number and value of the program's command-line arguments respectively. By convention, the command-line arguments specified by `argc` and `argv` include the name of the program as the first element. If a user types the command `rm file`, Unix will initialise the `rm` main function with `argc = 2` and `argv = ["rm", "file"]`. In the example program we want to take the value of x from the first argument of the program, so we will use second element in the `argv` array, `argv[1]`. (Note: indexes for C arrays start at zero.) Command line arguments are passed to the program as strings, so we use the function `atof` to convert the string argument for x to a float that can be used in the expression for `poly`.

Output to the screen can be produced using the `printf` statement. Text must be enclosed within double quotes. For example

```
printf( "hello, world\n" );
```

prints the string `Hello, world` to the screen followed by a new line (the new line is specified by the string `\n`). To print variables to the screen, the string in the first argument of the

`printf` statement must contain *format specifications*, followed by the arguments which contain the variables to be printed. For example, if `i` is an integer variable we use the format specifier `%d` to print it, e.g.

```
printf( "The value of i = %d\n", i );
```

To print several variables, add additional format specifiers and arguments to the function call. This example prints an integer variable `i` and a single precision floating point variable `x`:

```
printf( "The value of i = %d and the value of x = %f\n", i, x );
```

Notice that the second format specifier is `%f` rather than `%d` which formats a floating point number. Some common format specifiers that can be used in a `printf` statement are:

`%d` Print the integer argument is printed as a signed decimal.

`%f` Print the floating-point argument in the style `ddd.ddd` where number of `d`'s after the decimal point is 6 by default.

`%e` Print the floating-point argument in the style `d.ddde+dd` where there is one digit before the decimal point and the number after is 6 by default.

In the example program, the result is written to a file called `output.txt`, rather than to the screen. To do this, the program first opens the file for writing with the `fopen` function, storing a pointer to the resulting file in the variable `fp`. The program then uses the function `fprintf` to write to the file. `fprintf` is similar to the `printf` function, except that the first argument is the file pointer returned by `fopen`. The program then closes the file with the `fclose` function. You can learn more about the functions `printf`, `fprintf`, `fopen`, and `fclose` by using the Unix `man` command. For example typing

```
man fprintf
```

will print the documentation for the `fprintf` function.

Compiling and linking

To run your program, it is first necessary to convert it from C source code into machine code (sometimes called object code). This must then be linked in with other library codes; machine codes which execute the functions used by the program. We will use the GNU C compiler `gcc` to compile and link C programs. To use it, simply type

```
gcc filename.c
```

Note that the filename must contain a `.c` extension, so all your C program filenames should end in `.c`, e.g. `polynomial.c`

The compiler automatically links in some functions, such as `fprintf`, but you need to explicitly tell it where to find some other functions. One example of this type of function is `pow` which is found in the math library. To link in a program that needs the math library, type

```
gcc filename.c -lm
```

The `gcc` command will produce an executable file `a.out`. To run the program simply type `./a.out`

The characters `./` tell the operating system that `a.out` is found in the current directory.

Exercises

1. Use the command

```
cp /home/dbrown/gravity_html/teaching/phy308/polynomial.c .
```

to copy the example program into your current directory. (The dot at the end of the this command tells the computer to create a file with the same name in your current directory.) Use the `ls` command to check that the file is there. Now compile this program using the `gcc` command and repeat the `ls` command. You should now see a new file names `a.out`. If you see an error message saying `undefined reference to 'pow'` then you have forgotten to tell the compiler to link in the math library with `-lm`.

To run the program and evaluate the polynomial for a value of $x = 2.5$, type

```
./a.out 2.5
```

When the run is complete (i.e. your command prompt returns) use `cat` to examine the file `output.txt` and check that the output is correct.

2. Alter the program so that the output is written to the screen, rather than a file. Note: before altering a working program it is safest to make a copy of the file just in case before something goes wrong—e.g. copy the file to `polynomial2.c`.
3. Change the assignment statement so that the program computes the value of

$$\frac{3x^2}{5} + 4x - 2$$

Check that your answer is correct. If not, check your code carefully and consider the above warnings again.

4. In order to demonstrate the effects imposed by the restrictions on the range of integer and real variables copy the files `biginteger.c` and `rlimits.c` from the directory
`/home/dbrown/gravity_html/teaching/phy308`
Compile and run these programs. What do you notice about the results?

5. The relationship between the frequency ν and the wavelength λ of a light wave is

$$\nu = c/\lambda$$

where c is the speed of light. Write a program which reads in a value for the wavelength from the command line and calculates the corresponding frequency. Print out both the wavelength and the frequency. (Assume that $c = 3 \times 10^8 \text{ ms}^{-1}$.)

Math library functions

The C math library contains many useful mathematical functions. A list of the common ones is given below. Do not forget to `#include <math.h>` in your program and link the math library at compilation with `-lm`.

<code>acos(x)</code>	returns the value of $\arccos(x)$ in radians
<code>asin(x)</code>	returns the value of $\arcsin(x)$ in radians
<code>atan(x)</code>	returns the value of $\arctan(x)$ in radians
<code>atan2(y,x)</code>	returns the value of $\arctan(y/x)$ in radians, using the signs of both arguments to determine the quadrant of the return value
<code>ceil(x)</code>	Get smallest integral value that exceeds x
<code>cos(x)</code>	returns the value of $\cos(x)$ in radians
<code>cosh(x)</code>	returns the value of $\cosh(x)$
<code>exp(x)</code>	returns the value of e^x
<code>fabs(x)</code>	returns the value of $ x $
<code>floor(x)</code>	Get largest integral value less than x
<code>fmod(x,y)</code>	Divide x by y with integral quotient and return remainder
<code>log(x)</code>	returns the natural logarithm $\ln(x)$
<code>log10(x)</code>	returns the base-10 logarithm $\log_{10}(x)$
<code>pow(x,y)</code>	returns the value of x^y
<code>sin(x)</code>	returns the value of $\sin(x)$ in radians
<code>sinh(x)</code>	returns the value of $\sinh(x)$
<code>sqrt(x)</code>	returns the value of \sqrt{x}
<code>tan(x)</code>	returns the value of $\tan(x)$ in radians
<code>tanh(x)</code>	returns the value of $\tanh(x)$

An example of a program which uses the math library function `sqrt` is `quadratic.c` in the directory

`/home/dbrown/gravity_html/teaching/phy308`

Graph Plotting

`gnuplot` is an interactive plotting program which is very simple to use. To start `gnuplot` simply type

```
gnuplot
```

at the command line. You will get a number of messages followed by the `gnuplot` prompt

```
gnuplot>
```

You can now enter `gnuplot` commands.

- **To plot mathematical functions** such as $\sin x$, simply type

```
plot sin(x)
```

Most other common functions are available. For a complete list, type

```
help expressions functions
```

- **Interactive help** is available for any command, e.g.

```
help plot
```

produces information about the command `plot`, or simply

```
help
```

lists the available commands.

You can also plot more complex expressions by combining two or more functions, e.g.

```
plot sin(x)*cos(x)
```

- **To plot your own data** you will need a file containing two columns of numbers. For example, if the data is in a file called `output.txt` and columns 1 and 2 correspond to the x and y axes, respectively, then you can type

```
plot 'output.txt' using 1:2 with lines
```

for a line graph, where 1:2 indicates the columns used for the x and y axes. (Note the single quotes around the file name.) An alternative to a line graph is simply to represent each coordinate by a point

```
plot 'output.txt' using 1:2 with points
```

Multiple graphs can be plotted on a single set of axes by separating each plot command by commas, e.g.

```
plot 'output.txt' using 1:2 with lines, 'output.txt' using 1:3 with lines, sin(x)
```

will use the data in column 1 of `output.txt` for the x -ordinate and obtain the y -ordinates from columns 2 and 3 and from $\sin x$.

- **Modifying the basic graph**—many of the parameters will be set automatically, but you may wish to vary them. For example, to change the range on the x -axis so that the graph is plotted in the range $x = 0$ to $x = 5$ type

```
set xrange[0:5]
```

You will not see the result of this immediately. To view the modified graph you must then type

```
replot
```

The axes can be labelled, e.g.

```
set xlabel 'temperature'
```

There are similar commands such as `set ylabel` and `set title`.

Parametric equations can also be plotted—see the exercise below for details.

- **To save your plot to a file** requires the following sequence of commands

```
set terminal postscript
```

which tells gnuplot to output to a PostScript file, rather than the screen. Then

```
set output 'filename.ps'
```

for example the file name could be `figure1.ps`. Then

```
replot
```

The `replot` command is essential, otherwise on exiting gnuplot you will discover that the file is empty.

- **To exit gnuplot** type

```
quit
```

If you have written the output to a particular file, e.g. `figure1.ps`, then on exiting you should find that such a file has been created. You can then use `ps2pdf` to convert this file to PDF and view it with `kpdf`.

Exercises

1. If a sample of radioactive substance initially contains N_0 radioactive atoms, then at a later time t the number of remaining radioactive atoms is given by

$$N = N_0 e^{-\lambda t}$$

where λ is related to the half life by $\lambda = \ln 2/\tau$.

Write a program which accepts values of τ and λ as input on the command line and computes the proportion of remaining radioactive atoms N/N_0 . Check by hand to verify that the output is correct.

2. This exercise involves the use of `gnuplot`. First we will use `gnuplot` to illustrate beats arising from two cosine waves of slightly different frequency. To produce the resultant of two such waveforms type

```
plot cos(5*x)+cos(4*x)
```

Now try plotting the individual cosine waves as well as the resultant on a single set of axes.

Second, we can use `gnuplot` to generate Lissajous figures. To do this it is first necessary to give the command

```
set parametric
```

The `plot` command now expects two inputs, e.g.

```
plot sin(t), sin(t+pi/4)
```

Try this, and then experiment with other similar examples.

Decision making

In C most decision making is handled by the `if` statement. We will first give a formal definition of the `if` statement and then look at a specific example.

Syntax of the `if` statement

- `if (condition) statement;`
- `if (condition)`
 `{`
 statement block;
 `}`
- `if (condition)`
 `{`
 statement block 1;
 `}`
 `else if (expression)`
 `{`
 statement block 2;
 `}`
 `else`
 `{`
 statement block 3;
 `}`

If condition is non-zero, then the `if` statement interprets it as “true” and executes the appropriate code.

Logical Expressions

Comparators

<code>a < b</code>	meaning	$a < b$
<code>a <= b</code>		$a \leq b$
<code>a > b</code>		$a > b$
<code>a >= b</code>		$a \geq b$
<code>a == b</code>		$a = b$
<code>a != b</code>		$a \neq b$

Operators

<code>!</code>	meaning not	e.g. <code>if (! (a < b))</code>
<code>&&</code>	logical and	e.g. <code>if ((a < b) && (c < d))</code>
<code> </code>	logical or	e.g. <code>if ((a < b) (c < d))</code>

Warning: be careful not use the assignment operator = when you should use the equal to comparator ==. The code

```
int x = 10;
int y = 20;
if ( x = y ) printf( "x equals y\n" );
```

will set the value of the variable x to 20 and print x equals y, since 20 is a non-zero value. The correct code to test if the mathematical expression $x = y$ is true is

```
if ( x == y ) printf( "x equals y\n" );
```

Exercises

Consider the program

```
/home/dbrown/gravity_html/teaching/phy308/quadratic.c
```

which, given the coefficients a , b and c on the command line, finds the roots of the quadratic equation

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

What happens if the value of the discriminant (given by $b^2 - 4ac$) is negative? If you run the program for a quadratic equation with complex roots, you will find that it prints `nan` for the roots. `nan` stands for “not a number,” and is a value that is produced as the result of an operation on invalid input operands, especially in floating-point calculations. In this case, the program was unable to calculate the square root of a negative number. It indicates that the operation was invalid by returning a `nan` result. However, we can avoid the program printing `nan` by using the `if` statement. We will consider examples using the each of the three forms of the `if` statement given above.

1. The simplest way to avoid a crash is to stop the program if the discriminant is negative, i.e. we insert the line

```
if ( discriminant < 0 ) return 0;
```

which causes the main program to return the value of zero and exit if the discriminant is negative.

2. A more informative approach is to use the second form of the `if` statement by inserting the lines

```
if ( discriminant < 0 )
{
    fprintf( stderr, "The roots are complex\n" );
    return 1;
}
```

In this case, if $b^2 - 4ac < 0$, the program uses the `fprintf` function to write a message to a special file pointer called `stderr` (short for *standard error*) and returns with status code 1 (rather than zero). Both of these are standard conventions for error conditions in Unix programs: we write all error messages to `stderr` (which normally just gets printed to the screen) and return 0 if the main program succeeds, otherwise we return non-zero number.

3. The program `quadratic2.c` shows an example of using the third form of the `if` statement to compute complex roots, as well as real roots.
4. `if` statements can also be nested inside other `if` statements.

Note that in the above examples, indenting the statement block improves the readability of the programs.

What happens if you run the program `quadratic.c` with only two arguments instead of three, i.e.

```
./a.out 2.0 3.0
```

You will likely see the error message

```
Segmentation fault
```

printed by the operating system (if you are using Mac OS X, you may see `Bus error` instead). This means that your program has attempted to access some memory that does not exist. In this case, there is no value for `argv[3]`, so your program crashed when it tries to set the value of `c`. To avoid this error, try modifying your program to use the `if` statement to check that the program has exactly three arguments. It should print an error message to `stderr` and exit with a non-zero number if this is not true. *Hint:* remember that the array `argv` also contains the program name as the first element and that C starts counting array indexes at zero.

Warning: It is not advisable to test that two `float` or `double` variables are equal (or not equal), e.g. suppose variables `a` and `b` both have values of 2. We expect the condition `a == b` to be true. However `a` may be represented as 2.000000 and `b` as 1.999999. Consequently, if we test to see if `a == b` the answer will be false.

Control Loops

C has several constructs for constructing control loops, which we will look at now.

The while and do while loops

You can repeat any statement using either the `while` loop:

```
while( condition )
{
    statement1;
    statement2;
}
```

or the `do while` loop:

```
do
{
    statement1;
    statement2;
} while( condition );
```

The `condition` is just a test to control how long you want the statements between the curly braces to carry on repeating. The braces `{` and `}` which have appeared at the beginning and end of the main program can also be used to group together related declarations and statements into a *statement block*. In the case of the `while` loop, before the statement block is executed, the `condition` is checked. If it is non-zero (i.e. `true`) then the statement block is executed, control returns to the `while` statement and if `condition` is still true, the statement block is again executed. If `condition` is zero (i.e. `false`) the statement block is not executed and control moves to the next statement after the `while` loop. In the case of the `do while` loop, the statement block is always executed at least once. If `condition` is true after the block has executed, the block is executed again. The same logical expressions that we introduced for `if` statements can be used in the test condition for the `while` statement.

Consider the following program:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    do
    {
        printf( "Hello, world!\n" );
    } while( 1 );
    return 0;
}
```

The program will print `Hello, world!` to the screen and then check the `while` condition. Since the number 1 in this condition is always non-zero, this program will loop forever, printing `Hello, world!` over and over again. If you want to print `Hello, world!` just ten times, then you need a counter and a test for when that counter reaches 10. A counter in C is a simple variable and you add one to it each time you go through the loop:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int i = 0;
    do
    {
        printf( "Hello, world!\n" );
        ++i;
    } while( i < 10 );
    return 0;
}
```

We have added three things to this program: (i) an integer variable called `i` which is initialized to zero, (ii) a test in the `while` loop that $i < 10$, and (iii) the statement `++i`. The double plus means *increment the variable by one*, so the statement `++i` is equivalent to `i = i + 1`. A similar decrement operator `--i` exists which decreases the variable by one.

Warning: Most C compilers do not initialize variables to zero. It is essential therefore to initialize all variables before incrementing them (e.g. `i` is initialized to 0 in the above example).

Note: if you want to execute a loop a set number of times you should use an integer variable as the counter. C will allow you to use floating point variables, but this may cause problems. The increment and decrement operators only work on integer variables. However, sometimes it is appropriate to use a floating point number in a `while` test, e.g. suppose we did not know that the first positive root of $\cos x$ is $x = \pi/2$ and we want to print the values of $\cos x$ from $x = 0$ to the first root, at increments of 0.01. We could use the program:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    float x, y;
    x = 0;
    do
    {
        y = cos( x );
        printf( "%f %f\n", x, y );
        x = x + 0.01;
    } while( y > 0 );
    return 0;
}
```

The for loop

Initializing a variable, incrementing it and testing its value to terminate a loop is a very common operation and so many programming languages provide a keyword to do exactly this. In C, the keyword is `for`. The syntax of a `for` loop is

```
for( initialize; condition; increment )
{
    statement1;
    statement2;
}
```

On entering the `for` loop, the statement `initialize` is executed. If the test in `condition` is non-zero, then the statement block is executed. The `increment` statement is then executed. This repeats until `condition` evaluates to zero (i.e. false).

Note: If you want to use the value of an counter in a mathematical expression, you need to convert it to a float (or double) with a “cast.” For example

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int i;
    float x, y;
    float dx = 0.01;
    for ( i = 0; i < 1000; ++i )
    {
        x = (float) i * dt;
        y = cos( x );
        printf( "%f %f\n", x, y );
    }
    return 0;
}
```

The counter variable `i` is cast to a single-precision floating point number using the keyword `(float)` (the parentheses are important) before it is multiplied by `dt`. In this case, the cast is not strictly necessary, as C has various rules which would promote `i` to a float since it is being multiplied by a float, but *if in doubt, use an explicit cast*.

Notice that we still have to declare the variable `i`, but now the initialization, test and increment statements are all contained in one place (i.e. in the parentheses following the `for` statement).

The above program prints out the value of $\cos x$ from $x = 0$ to $x = 10.0$ in steps of $dx = 0.1$. If you want to obtain an evenly sampled array, it is much more reliable to use an integer counter and multiply by a fixed step, rather than use a float (as in the previous example) and add another small float every step.

Exercises

It is strongly recommended that you attempt the first exercise, which follows on from the last set of exercises. This makes use of the `for` loop construct and you are encouraged to use `gnuplot` to plot the results. There are two further examples which make use of the `if else` construct (exercise 2) and a `do while` loop (exercise 3).

1. Starting from the radioactive decay program that you wrote in the previous exercises, include a `for` loop so that it computes the proportion of remaining atoms at a series of times t , e.g. take $\tau = 5 \times 10^4$ s and determine N/N_0 for $t = 0$ to 2×10^5 s in steps of 1000 seconds.

Be careful about the variable type you assign to t . Remember the index of the `for` loop should be an integer (see above) but t is also needed as a floating-point value in the assignment statement, so you should use the appropriate cast.

Remember to check that the program is working properly, e.g. check one of the values by hand. (Note that if you check the value in the first time step this does not verify that the `for` loop is working correctly. Instead check the value for the, say, the third time through the loop, i.e. $t = 2000$ s.)

Write the output to a file and plot it using `gnuplot`.

2. The amount of energy required to raise n moles of an ideal gas through a temperature Δt at a constant volume is given by

$$\begin{aligned} E &= (3/2)nR\Delta t && \text{monatomic gas} \\ E &= (5/2)nR\Delta t && \text{diatomic gas} \end{aligned}$$

where R is the molar gas constant ($R = 8.31 \text{ J mol}^{-1} \text{ K}^{-1}$). Write a program which reads n , ΔT and whether the gas is monatomic or diatomic (i.e. enter the number of atoms in the molecule). The output should give the appropriate energy change. Note that the number of atoms in the molecule is an integer so if you parse this from the command line you should use the function `atoi` to convert the value of `argv` to an integer, rather than `atof`. Check your answers by hand.

3. Modify the program in exercise 1 above to use a `do while` construct so that the values of N/N_0 are calculated for times t until $N/N_0 < 0.1$.
4. Copy the program

```
/home/dbrown/gravity_html/teaching/phy308/hello.c
```

and compile it. If you run the program it will loop forever. The `while (1)` test puts the code in an infinite loop. To stop a program from executing press `CTRL-c` (hold down the control key and press `c`).

Double precision variables

We need to be aware of the fact that `float` variables are only represented to an accuracy of approximately 6 decimal places. This should be sufficient for most back-of-the-envelope calculations, however small errors in floating-point arithmetic can grow when mathematical algorithms perform operations an enormous number of times. A few examples are matrix inversion, eigenvector computation, and differential equation solving. These algorithms must be very carefully designed if they are to work well. As you have seen in previous exercises, `float` variables are also limited in their range from $\sim 10^{-38}$ to $\sim 10^{38}$. For computations that require higher accuracy, or larger ranges, we can use `double` variables. When you encounter the GNU Scientific Library (GSL) later in the class, you will find that GSL uses `double` variables for all computations.

Exercises

1. Copy the program

```
/home/dbrown/gravity_html/teaching/phy308/round.c
```

and examine the code. `x` and `y` appear to be two equivalent mathematical expressions, however if you run the program and plot the results in `gnuplot`, you will find that the results differ. This shows you that you should take great care in the way that you code a mathematical expression.

2. Try running the program

```
/home/dbrown/gravity_html/teaching/phy308/series.c
```

giving the input values $x = 2$ and an accuracy of 1×10^{-8} . The calculated value is given as `nan`. This is because with single precision `float` variables the program cannot achieve the desired degree of accuracy. Alter the program so that it uses double precision variables and show that the desired answer can be achieved.

Arrays

An array allows several values to be associated with a single variable name. Arrays can be of any data type, i.e. `int`, `char`, `float` or `double`, e.g.

```
int a[5];
float x[10];
float d[7][15];
```

C starts counting array elements at zero, so the array declared as `a[5]` contains the five elements `a[0]`, `a[1]`, ..., `a[4]`. Any array element can be specified by either referring to the element directly, e.g.

```
x[2] = 1.0;
```

or by using any integer expression, e.g.

```
for ( i = 0; i < 3; ++i )
{
    x[4+(2*i)] = 3.0;
}
```

The above expression sets values `x[4]`, `x[6]` and `x[8]` to 3.0.

Note: the index which specifies an array element must be an integer variable. The reason is obvious—a real variable cannot be used even if it has an integral value, e.g. 6.0, since the actual stored value may be 5.999999. Obviously it makes no sense to refer to the 5.999999th element of an array. An example of the use of arrays is given in the example program `dotprod.c`.

Arrays must have constant dimensions so that the compiler knows how much memory space to allocate to each one. The easiest way to set the dimension of an array is to use the `const` (constant) keyword to declare an integer constant, e.g.

```
const int n = 10;
float a[n], b[n], c[n];
```

If we now change the dimensions of the arrays it is only necessary to alter the one value in the `const` statement. Note that a variable that is declared constant may not be altered within the program. Obviously variable defining the length of the array must be declared before any array declaration which makes use of that parameter. An example of this appears in the program `dotprod.c`.

Pointers

Pointers are a very powerful, but primitive function of the C language. Pointers are a throwback to the days of low-level assembly language programming and as a result they are sometimes difficult to understand and subject to subtle and difficult-to-find errors. However, before we can look at function calls in C we need to understand the basics of pointers.

You should now be used to accessing areas of the computers memory using variables. If we declare a variable, e.g.

```
float x;
```

the compiler allocates enough memory in the program to store an integer and gives it that memory location the name `x`. You can the use this name to store data in this memory location. For example:

```
x = 10.0;
```

is an instruction to store the data value 10 in the area of memory named `x`. The computer access its own memory not by using variable names but by using a memory map with each location of memory uniquely defined by a number, called the address of that memory location. A pointer is a variable that stores this location of memory. In more fundamental terms, a pointer stores the address of a variable, i.e. *it points to the variable*.

A pointer has to be declared just like any other variable since a pointer is just a variable that stores an address. A pointer to a `float` variable can be created with

```
float *p;
```

Adding a asterisk in front of the variable name declares it to be a pointer to the declared type. Once you have declared a pointer variable you can begin using it like any other variable. To interact with pointers, C defines two operators `&` and `*`:

- `&` The ampersand operator returns the memory address of a variable. That is, when you apply it to a variable such as the variable `x` above, it will return the memory address where the value of `x` is stored.
- `*` The asterisk operator is used in conjunction with pointer variables. If you place `*` in front of a pointer variable then the result is the value stored in the variable pointed at. That is, if `p` stores the address, or pointer, to another variable then `*p` is the value stored in the variable that `p` points at. The `*` operator is called the *de-referencing operator*. You should try not to confuse it with multiplication or with its use in declaring a pointer.

An example of the use of pointers is given in the following code:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    float x; /* declare memory for a float variable called x */
    float y; /* declare memory for a float variable called y */
    float *p; /* declare memory for a pointer to a float */

    x = 10.0; /* assign the value of 10.0 to the variable x */
    p = &x; /* assign the memory address of the variable x to the pointer p */

    /* use the de-reference operator to store the value of x in y */
    y = *p;

    /* print the value to the screen */
    printf( "x = %f\n", y );

    return 0;
}
```

More about input and output

So far we have considered how to write output to the screen or to a file using the `printf` and `fprintf` functions:

```
printf( "Hello, world!\n" );  
fprintf( stdout, "Hello, world!\n" );
```

Note that if the file pointer used in the first argument of `fprintf` is the `stdout`, the `printf` and `fprintf` statements are equivalent. We have also used `argc` and `argv` to read input from the command line. There is also a function which reads data from a file: `fscanf`. In this section, we are going to look at file input and output in more detail.

The `fopen` function

We have seen the use of the `fopen` function in the program `polynomial.c` to open a file for writing: `fp = fopen("output.txt", "w");` The variable `fp` must be declared as a file pointer: `FILE *fp;` The `fopen` function, like all the file-system functions, requires the header `stdio.h`. The first argument to `fopen` is a string which gives the name of the file to open and the second argument is the *mode* in which the file is opened.

Common modes are `"r"` to open the text file for reading, `"w"` to create a new text file for writing (overwriting any existing file of the same name), and `"a"` to append to an existing text file. See the Unix man page for `fopen` for more details.

The `fscanf` function

The `fscanf` function allows us to read input for the program. Consider the following program:

```
#include <stdio.h>  
  
int main( int argc, char *argv[] )  
{  
    int i;  
    float x;  
  
    printf( "enter an integer: " );  
    fscanf( stdin, "%d", &i );  
    printf( "you entered %d\n", i );  
  
    printf( "enter a float: " );  
    fscanf( stdin, "%f", &x );  
    printf( "you entered %f\n", x );  
  
    return 0;  
}
```

The first argument to the `fscanf` function is a file pointer—in this case `stdin` or *standard input*, reading from the user. The next argument is a format statement identical to that used by `fprintf`. For each format specifier in the format statement, an addition argument is given which tells `fscanf` where to store the input data. These arguments must be *pointers* to the variables where you want to store the input data. In the first `fscanf` statement, the argument is `&i` which passes a pointer to the variable `i` to the function.

In many cases we may want to read data from a text file. To do this first `fopen` the file with the read ("r") mode and then use the `fscanf` function with the resulting file pointer. Suppose that we have a file called `input1.txt` which contains three columns; a column of integers, and two columns of floats:

```
1 3.202 182.010
2 3.212 190.232
3 3.473 208.212
```

The following example program `read1.c` reads this file into the arrays `a`, `x` and `y`:

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
    const int n = 3;
    int i, code, a[n];
    float x[n], y[n];
    FILE *fp;

    fp = fopen( "input1.txt", "r" );
    if ( !fp )
    {
        fprintf( stderr, "unable to open input file\n" );
        return 1;
    }

    i = code = 0;
    while( code != EOF && i < n )
    {
        code = fscanf( fp, "%d %f %f\n", a + i, x + i, y + i );
        fprintf( stderr, "read %d elements\n", code );
        ++i;
    }

    for ( i = 0; i < n; ++i )
    {
        fprintf( stdout, "%d %f %f\n", a[i], x[i], y[i] );
    }

    return 0;
}
```

This example contains several of the constructs that we have seen so far, so let us walk through it now. First the variables are declared and then `fopen` is called to open the input file. We then use an `if` statement to check that the value of the file pointer `fp` is not zero. If `fopen` fails to open the file, it will set `fp` to zero, so this checks that the file has been successfully opened. It is always a good idea to check the value returned by `fopen` in this way before using it in an `fscanf` or `fprintf` function.

Next the program sets the values of the variables `i` and `code` to zero. It then uses a `while` loop to call `fscanf` to read the contents of the file. `fscanf` reads in the values from a single line of the file and returns the number of values read, which is stored in `code`. When the end of the file is reached, `fscanf` sets `code` to the special value `EOF` (which means end of file).

The test in the `while` loop repeats the loop as long as `code` is not equal to `EOF` and the counter `i` is less than 3. This breaks the loop at the end of the file and ensures that if there are more than three lines in the file, only three array values will be filled.

When we call `fscanf` we need to pass pointers to the memory locations where the input data should be stored. In C there is a very close connection between pointers and arrays. When you declare an array as

```
int a[10];
```

you are in fact declaring a pointer `a` to the first element in the array. That is, `a` is exactly the same as `&a[0]`. The only difference between `a` and a pointer variable is that the array name is a constant pointer you cannot change the location it points at. Another way of writing `&a[0]` is `a`. Similarly, `&a[3]` can be written as `a + 3`. We use this construct to pass pointers to the individual array elements to the `fscanf` function.

Finally the program prints the values it read in from the file.

Exercises

1. This exercise is concerned with file i/o.

Copy the files `read1.c` and `input1.txt` from the directory
`/home/dbrown/gravity_html/teaching/phy308`

and compile and run the program. Examine the program and add comment statements describing what the code does. Make a second copy of the file `input1.txt` and called `input2.txt`. Delete the last line of the file, so it only has three lines in it. Modify the program to read its input from the `input2.txt` and re-run the program. What happens? Try adding a fourth and fifth line to the input data file and re-run the program. What happens?

Revert to the original version of `read1.c` and re-write the loop using a `for` loop, rather than a `while` loop.

Add a fourth column containing a floating-point number and modify your program read it into a new array called `z`.

2. This exercise is to practice the use of arrays.

A particle executes simple harmonic motion such that the position, velocity and acceleration of its motion at time t is given by

$$\begin{aligned}x &= B \cos \omega t \\v &= -\omega B \sin \omega t \\a &= -\omega^2 B \cos \omega t\end{aligned}$$

where B is the amplitude and ω is the angular frequency.

Write a program which reads in values for ω and B (suggested values are $\omega = 2 \text{ rad s}^{-1}$ and $B = 0.4 \text{ m}$) and then use a `for` loop to calculate the values of x , v and a at, say, 10 intervals between $t = 0 \text{ s}$ and $t = 5 \text{ s}$. Once you have calculated all the values of x , v and a use another `for` loop to print them out to a file. Use the appropriate format statement so that your results appears in columns which can be plotted in `gnuplot`.

Modularization

What is modularization? In many cases a program does not contain all the instructions required by the computer. Instead it requires part of the calculation to be performed by other functions, e.g. `pow`, `fprintf`, etc. These are C standard library functions that we have already met. In C, you can also write your own functions.

Why modularize? There are several reasons:

- in a complex program each function can be tested separately,
- using functions avoids duplicating frequently used code,
- it is easy to use functions to other programs.

Functions

A function is simply a block of C code that are have grouped together and given a name. For example we can define a function that prints a string

```
hello()
{
    printf( "Hello, world\n" );
}
```

This is not a very useful function; functions are much more useful when they manipulate input values and return output values. We can define a function that takes two values x and y and prints their sum:

```
add_numbers(float x, float y)
{
    printf( "x + y = %f\n", x + y );
}
```

The function is declared to take two floating-point variables. We would call this function in the main program with

```
int main( int argc, char *argv[] )
{
    float a = 5.0;
    float y = 1.0;

    add_numbers( a, y );

    return 0;
}
```

In C, functions are completely separate from the main program and other functions—therefore functions will not assign values to a variable just because that variable has the same name as a variable in the main program. This means that all the variables which are required by the function must be passed via the argument list. It also means that names of the variables passed to a function do not need to be the same as those in the function statement. However there must be a one-to-one correspondence between the variables in the argument list in the main program and that in the definition of the function, and the variables must be of the same type (or cast to the appropriate type).

Note that the variables within the argument list of a function are not altered within the function. This means that functions are isolated, and nothing survives after they have finished executing. To be useful there has to be a way of getting data out of a function. We can do this with the `return` statement. To use this we also have to declare the return type of the function

```
float add_numbers(float x, float y)
{
    float result;
    result = x + y;
    return result;
}
```

In this example, we have also declared a *local variable* called `result` and we return the value of `result` to the calling function. You can define as many local variables as you like in a function. `return` statements can occur anywhere within the function, not just as the last instruction—however, a `return` always terminates the function and returns control back to the calling function.

Suppose you want to return more than one value from a function. For example, we might want to write a function that returns the positive and negative roots and the discriminant of a quadratic equation. We might try the following:

```
float quad_solve( float root_p, float root_n, float a, float b, float c )
{
    float d = b*b - 4.0*a*c;
    root_p = (-b + sqrt(d)) / (2.0*a);
    root_n = (-b - sqrt(d)) / (2.0*a);
    return d;
}
```

and call the function with:

```
quad_solve( p, n, 7.0, 15.0, -2.0 );
```

to store the roots in the variables `p` and `n`. In fact this does not work. This is because the function arguments `root_p`, `root_n`, `a`, etc. are also local variables to the function. Changing their values in the function does change the values of the variables passed in the calling program.

To do what we want, we must modify the function to take *pointers* to variables. Then we can use these pointers to access the memory locations of variables in the calling function. To do this, the example would become

```
float quad_solve( float *root_p, float *root_n, float a, float b, float c )
{
    float d = b*b - 4.0*a*c;
    *root_p = (-b + sqrt(d)) / (2.0*a);
    *root_n = (-b - sqrt(d)) / (2.0*a);
    return d;
}
```

and call the function with:

```
quad_solve( &p, &n, 7.0, 15.0, -2.0 );
```

Notice the use of the * and & pointer operands.

We have not yet addressed how to make functions accessible to a main program. If we keep our functions in the same .c file as the main program, the only requirement is that the function's type has to be known before it is actually used. One way is to place the function definition earlier in the program than it is used—for example, before `main()`. The only problem is that it is often clearer if the main program comes at the top of the program listing. The solution is to declare the function separately at the start of the program. For example

```
#include <stdio.h>

/* declare functions used by main */
float quad_solve( float *root_p, float *root_n, float a, float b, float c );

/* main program */
int main( int argc, char *argv[] )
{
    /* main program goes here */
}

/* function to solve quadratic equation */
float quad_solve( float *root_p, float *root_n, float a, float b, float c )
{
    float d = b*b - 4.0*a*c;
    *root_p = (-b + sqrt(d)) / (2.0*a);
    *root_n = (-b - sqrt(d)) / (2.0*a);
    return d;
}
```

Warning: If the function type is not known to the main program before the function is used, you can obtain strange results as the compiler will assume that all the arguments to the function are integers. To catch for the use of functions that are not properly typed before being used add the `-Wall` argument to the C compiler which will print a warning if you do this (it will also print other warnings, so using `-Wall` is a good idea generally). For example to compile the polynomial program, use

```
gcc -Wall polynomial.c -lm
```

Exercises

1. Modify the `polynomial.c` function from the previous exercise to use a function to compute the roots of the the polynomial equation.

2. Numerical differentiation

The simplest approach to performing numerical differentiation is to determine the gradient of a function over a small range. We will apply this to the curve $\sin x$, for which

$$\frac{d}{dx}(\sin x) \approx \frac{\sin(x+h) - \sin(x)}{h}$$

where h is small.

Write a function to evaluate the right hand side of this expression for a given x and h . Then write a program which reads in values of x and h , calls the function to determine the approximate value of $d \sin x / dx$ and then prints this value out together with the true value (given by $\cos x$).

Project 1 Report

These two exercises form the assessed work for project 1. They build on material from the previous exercises, however your report need only contain sufficient material to answer the questions in this section.

Numerical Differentiation

You should have a program containing a function which evaluates $d \sin x / dx$ by calculating the gradient of the tangent over a range h . Change the program so that it calls the function for a range of values of x between 0 and π radians. Print the output of the function, the true value and the error (i.e. the difference between these two values), tabulating your output.

At which points are the errors greatest? Explain why.

By using different values of h determine an approximate relation between the magnitude of the error and the size of h .

Does using `double` variable, rather than `float` variables change your conclusions about the errors? Explain why.

Simple Harmonic Oscillator

Take your existing program which computes the values of $x(t)$, $v(t)$ and $a(t)$ for the simple harmonic oscillator and place the main part of the calculation in a function with the prototype

```
int shm( double *x, double *v, double *a, int n,  
         double B, double omega, double t_start, double t_end );
```

where `x`, `v` and `a` are pointers to arrays of dimension `n`. The function should use a `for` loop which calculates the values of x , v and a for n equally spaced values between t_{start} and t_{end} , stores them in the arrays and returns them to the calling function.

The main program should read in values of ω and B (suggested values are $\omega = 2 \text{ rad s}^{-1}$ and $B = 0.4 \text{ m}$), determine the time period and call the function. On exiting the function, the main program should write x , v and a to a file. Finally, using `gnuplot`, plot both v and a as functions of x on the same set of axes. Decrease the sampling interval, i.e. increase the size of the array, until the curves are smooth—if you have written the program well this should only involve one small alteration to the program.