University of Kent

School of Physical Sciences

Module PH302
Computing Skills

Module Convenor: Dr. Jingqi Miao
Room 115; Email: j.miao@kent.ac.uk

January 17, 2014

Acknowledgements

Document Written by Timothy M. Kinnear
Room 104; Email: tk218@kent.ac.uk

Document Contributed to and Edited by:
Dr. Jingqi Miao (Room 115; Email: j.miao@kent.ac.uk),
Philip D. G. Cox (Email: pdgc3@kent.ac.uk),
Piero Canepa,
Paul Cornwall (Email: pc247@kent.ac.uk)

Additional thanks goes to all those who provided proof-reading and feedback during the production of this document. See changelog accompanying document for some details.

Document Originally Written in 2010/2011, typeset using LaTeX 2_ε [29]

Contents

0 Introduction

This document is designed to give an overview of good programming practice as well as general programming concepts. It includes a reference for working on Unix based systems such as Linux, and the Fortran 90 language around which the course is based.

If you have already begun week one, it is recommended to skip straight to 'Week One', Section 1, and read this section later.

0.1 Why Learn to Program?

One question which may come up during the course is 'why are we doing this?' In the case of programming, there are many reasons, from both scientific and general perspectives.

0.1.1 Greater Understanding of How Computers Operate

The general idea behind the majority of programming, as well as general computer use, is the ability to perform tasks and procedures which would otherwise be too complex or time consuming to do manually. For example, keeping the records of several thousand students in a university could be done using files and paper records. However, the ability to access the data more efficiently by orders of magnitude makes it more than worthwhile to keep the majority of the information in a computer system. It is important to remember that in all computer systems, whatever the task and however you interact with it, at some point, it had to be programmed to perform that task. While it may seem that since these things have already been programmed, it may not be worthwhile to understand how they work internally. However, knowing how the computer performs these tasks permits a greater degree of control over the computer, even when not programming. No doubt there have been times when using a computer, where it simply doesn't seem to want to do what you're trying to tell it; by knowing the general way in which computers go about the tasks behind the scene, it becomes far easier to avoid potential problems, and interact with the computer in a way that won't hit upon hidden snags.

This is a general, but important, point. If nothing else is taken away from this course, hopefully the general concept of how computers operate behind the scenes will grant you a more intuitive way of interacting with computing problems that may arise in the future.

0.1.2 Scientific programming

A key aspect of this course is the utility of programming for scientific tasks, be it simulation, data processing, controlling apparatus or any similar use. In this role, even for research which isn't based on programming, it can simplify otherwise tedious or complicated tasks. For example, processing large volumes of any sort of data, for statistical or plotting purposes, is frequently done with programming or scripting. Whilst 'off-the-shelf' applications such as Microsoft Excel can be used for some purposes, it typically is more advantageous to set up automated systems. Writing programs or scripts can permit the complete process of obtaining, processing, manipulating and creating a spread of relevant graphs in one go. This means that whenever data is updated or needs to be reprocessed, other than running the program again with the new rules/data, no additional work is required.

This may not seem especially beneficial, but when you have written 90% of a thesis, paper or report, and you're required to adjust the process which you used for the data analysis, or a particular assumption made in a model, redoing all of the relevant graphs manually, even if the original files still exist, becomes a tiresome and arduous task. How much easier to simply change a line of code, and recreate all of the graphs in one go!

0.1.3 Computational Simulation

Computational simulation permits discrete modeling of highly complex or interdependent systems, with conditions which would prohibit finding analytical solutions through maths alone. For example, modeling of fluid dynamics is one purpose for which this advantage is particularly clear, as attempting to obtain an analytical solution to fluid flow around complex surfaces/bodies would be either extremely difficult, or potentially impossible. The 'Navier-Stokes' equation, around which a large portion of fluid dynamics is based, has no currently known general solution. However, by using discretised computer modeling, it can nevertheless be incorporated into models providing accurate simulation of fluid flows. For those interested, one of the one million dollar 'Millennium Prize Problems' in Mathematics still exists for a solution to the Navier-Stokes equation!

The main idea of computational simulation is to take a continuous problem, and split it up into discrete 'chunks' that can be solved computationally; you create an approximation of the analytical solution. Since as many of these discrete problems can be solved as is desired, scenarios which are highly complex and interdependent can be built up, which would be nigh on impossible to attempt a specific analytical solution for.

More and more, computational simulation has become a key tool for science and technology. These fields extend to chemistry and biology, as well as physics, where problems frequently are not 'simple' in the sense that they can be approximated as corresponding to some known analytical form. Any problem involving complex, non-symmetrical shapes, or are highly dynamic and vary strongly with time and other parameters, are ideal to be targeted with computational methods.

0.2 Languages

Programming languages can broadly be separated into two main overarching categories. Those which would be classed classically as programming languages, and those which fall into the concept of 'scripting languages'. We'll take a brief look at each here.

0.2.1 Compiled Languages

To write a program in the classic programming languages requires several steps. Firstly, the intended code is written in the language in question, the file containing the written code for a given program is called that program's '**source**' code. After this is done, a program called a '**compiler**' is run. In basic terms, the compiler takes the source code, with the words, terms and arrangements which we can understand (the 'syntax' of the language), and converts it into a piece of binary machine code, which can then be executed (run) by the user at any subsequent time, without the use of any other program. These compiled binary codes are invariably the type of program which you will be familiar with on Windows or Apple machines. Code compiled for Linux machines will not run on Windows machines, nor vice-versa. However, the actual language used, regardless of the intended operating system, will be the same. For simple codes, it should be possible to compile the same source code for several systems. However, complex codes will tend to reference capabilities, libraries and user interaction which are system dependant. This means that in these cases, the code would have to be rewritten to some extent in order to be compiled for a different system. This concept, the extent to which a code can or can't be compiled for different and varied systems, is referred to as its '**portability**'.

0.2.2 Examples of Compiled Languages

Some examples of compiled programming languages include:
'**C**' and its more modern successor '**C++**';
The various versions of '**Fortran**', such as F77 and F90;
'**Visual Basic**', used on Windows machines;
There are also various forms of '**Assembly Code**', which are substantially closer to actual machine code than the above examples. They could count as a category of their own, as the differences between a language like 'C' and an Assembler language are extreme.

0.2.3 Scripting Languages

Whilst these languages are not the focus of the course, they are worth explaining in order to form a more complete picture of programming. The exact line between scripting languages and other programming languages is not well defined. However, scripting languages tend to be for smaller tasks, typically those where the action is not time critical or demanding. Scripting languages will usually be substantially '**higher level**' than other programming languages.

The higher level a language is described as being, the further away the commands of that language are from the fundamental machine code they are meant to represent. So, while a more complex and lower level programming language like Fortran may require a dozen commands in order to produce a plot of a graph for defining things such as the space required for the plot or specifying the manner in which to read in the dataset; a scripting language may have a simple single 'plot' command, followed by the location of the data file in order to do the whole thing.

While this may make it seem like scripting languages are much easier to use, this ease of use comes at a cost. Most scripting languages are '**interpreted languages**', meaning that instead of passing the code through a compiler, as with the other programming languages, they are converted and executed at once, line-by-line, by an interpreting program. This means that in addition to the time that would be taken to run the line of code when compiled, you also have to take the time to 'compile' each line as it goes through them. In addition, high-level commands tend to be quite general in their application, meaning that they can be used for a while variety of situations. The result of this is that for a given situation, running the command in the code may require processing a great deal of unnecessary code for the particular scenario in which it has been applied.

This means that interpreted scripting languages will run more slowly for a given task than an optimised lower level programming language. This defines their field of use: for creating a single plot, or renaming a spread of files, the time taken by a scripting language per task may be something like a tenth of a second. A programming language may be

able to do the task ten or even one hundred times faster, but if it's only one task, the time saved writing a simple script compared to a full code in a compiled language becomes worthwhile.

However, for a simulation, or large data processing jobs, where there are lots of tasks, or a task repeated a vast number of times, the difference will be in favour of a lower level language. If a script takes one tenth of a second to process one data point, and a lower level language takes one thousandth of a second, but there are 100,000 data points, the time difference becomes 2 $\frac{3}{4}$ hours for a script, to just under two minutes for a lower level language!

While these values are just illustrative, they show the basic difference in purpose between scripts and compiled languages.

0.2.4 Examples of Scripting Languages

Some examples of interpreted scripting languages include:
'**Python**', an increasingly popular high-level scripting language, with interpreters easily available on Windows, Apple and Linux systems;
'**Perl**, an interpreted language which is partially influenced by 'C';
'**JavaScript**', a language used heavily for internet applications, designed to be similar to the slightly lower level, compiled, '**Java**' code;
Another interesting scripting language class is '**Shell Scripts**'. These languages are strongly integrated with the way Linux terminal '**shells**' process commands. They provide very powerful functions to interact with Linux systems, efficiently and quickly performing a large number of file handling and manipulation tasks, as well as triggering other, non-shell script programs.

0.2.5 Fortran 90

The language focused on in this course is '**Fortran 90**'. Fortran itself is a language dating back to the early days of electronic computing, when programs were still read into a computer by '**punch cards**', literal pieces of card with holes punched in specific places representing instructions to give the computer. The language has moved on from then, but it still shares fundamental roots with the efficient design which was then necessary. The oldest version of Fortran which is still commonly used is '**Fortran 77**', a version written in 1977 (Fortran 77 will henceforth be abbreviated to F77). This was followed in 1990 by a reasonable number of changes forming Fortran 90 (F90). F90 still maintains a large degree of backwards compatibility with F77, with most F77 codes running as F90 codes with few, if any, alterations required.

The next version of Fortran was F95, this was fundamentally similar to F90, but removed a number of inefficient and archaic aspects that permitted compatibility with F77. As such, fewer F77 codes will run as F95 codes. However, its reduced backwards compatibility means that it is not as popular as F90. F95 was followed by Fortran 2003 which in turn was succeeded by Fortran 2008. However, both are still proprietary software, meaning an expensive license fee for obtaining the compiler.

The compiler to be used for this course is the free compiler '*gfortran*', This stand for '*G*NU *Fortran*' where the 'GNU' is a recursive abbreviation meaning '*G*NU's *N*ot *U*nix'. The use of this compiler is explained in the material for the first week.

1 Week One - The PCSPS07 Server, Unix and Your First Programs

1.1 Overview of Course and Assignment Format

This course is taught through weekly topics, each constituting a chapter/section. For each weekly set of topics, you will have three, one hour long workshop sessions. Each week also has a set of associated assignments. These are two program writing tasks, and a set of questions.

By the end of each week you will have been expected to complete the assignments, and to hand in the necessary pieces of work **before the end of the final workshop of the week**. As with all other modules, late submissions *will not* be accepted. If you cannot hand in your work on time for whatever reason, you should talk to your **personal tutor** (neither the course convenor, lecturers, nor any of our demonstrators) who will decide what to do next for you. The only exceptions to this deadline format are **weeks 17 and 23** (Sections 5 and 11), the assignments for which are due in **on Thursday week 18 and Friday week 24 for Sections 5 and 11 respectively** (see the table in Section 1.1.1 for full schedule).

Keep all of this in mind when you go about the work in the course. Ensure that you make best use of the workshop time as it is the only opportunity to get direct help, provided by the module demonstrators. It is **strongly** recommended that you do at least some reading of the week's chapter *before* each workshop. Time spent reading in the workshop reduces the time to program *and* obtain help while doing so.

Each programming assignment conforms to its own slight variation of the mark scheme, dependent on the specific format and aim of the intended program. However, marks are always given for programs which are well written. When writing your programs, make sure to refer to Section 14 on '**good programming practice**'. Regular implementation of these practices will get marks in **all** of the program writing assignments.

All sections/subsections which directly describe assignment material have the suffix 'Assignment Material' in the section/subsection title. Additionally, the specific requirements and questions for each week's work can be found at the end of each weekly section, under the 'Assignments Summary' subsection. **Make sure to refer to these subsections**, they describe what work must be handed in for that week and give the specific criteria of what you are expected to have done.

Also remember that the demonstrators will have copies of your code to mark. If the task was to write a program to produce a result/graph, and you produce the output artificially without making your program work, this *will* be noticed. The aim of the tasks is not the production of the output itself, it is the process of generating it using computer programs written by you, to the specifications stated by the assignment. This course is to develop and assess your abilities to write programs; while it may be entirely possible to produce the required results by means other than this, it benefits no one and *does not* count as valid work.

Also important are the Sections at the end of the document. Firstly the one on working in a Unix environment (Section 13), it is recommended that you read this section alongside this week's work, and/or refer to it throughout the course as a reference. The remaining sections are Section 14 on good programming practice, Section 15 on common error messages and likely solutions; and finally Section 16, a command overview, which should act as a reference for the basics as you go through the course.

There are ten weekly topics with regular workshops and assignments, as well as two weeks (six and twelve) which have in-class tests. Further information on these tests is available in the brief sections describing those weeks (Sections 6 and 12) and will be on Moodle the week prior to the test.

1.1.1 Rooms, Schedule and Printing

The vast majority of this course is made up of workshop sessions, in which you teach yourself the material contained in this document, whilst being able to ask for help from the demonstrators. The first session of the course (Monday 17:00 20/01/13), however, is an introductory lecture, given in RLT1. Be sure to attend this session, as it will cover in full the layout of the course, as well as key concepts in using a unix based system and Fortran programming.

For most weeks, you will be required to hand in two programming assignments, and answers to a set of short questions. Any week which requires such work to be handed in will state in a subsection at the end of that week's section exactly what printouts and answers are needed. In general, this will consist of the source code files for each assignment, any output produced by the programs written, and either hand written or typed up answers to the questions.

For week one, no such work needs to be handed in. However, this does **not** mean that it is not necessary to complete the assignments set. In particular, one of the assignments is directly expanded in week two, not doing it in week one will set you back for that week.

The following is the list of assignments for the course, and at what point they are due in:

Week	Work	Date due in
Week 13/Section 1	No assignments to hand in.	
Week 14/Section 2	Two programming, one set of questions.	Friday (31/01/13).
Week 15/Section 3	Two programming, one set of questions.	Friday (07/02/13).
Week 16/Section 4	Two programming, one set of questions.	Friday (14/02/13).
Week 17/Section 5	Two programming, one set of questions.	Thursday (27/02/13, week 18).
Week 18/Section 6	Sessions continue Section 5 and start Section 7.	
Week 19/Section 7	Two programming, one set of questions + class test.	Friday (07/03/13).
Week 20/Section 8	One programming, one set of questions.	Friday (14/03/13).
Week 21/Section 9	Two programming, one set of questions.	Friday (21/03/13).
Week 22/Section 10	Two programming, one set of questions.	Friday (28/03/13).
Week 23/Section 11	One programming.	Friday (11/04/13, week 24).
Week 24/Section 12	Sessions continue Section 11 + class test.	

For each listed day on which assignments are due in, the deadline is the **end of the workshop session on that day**. This does **not** mean that you can start printing at the end of the session, your work **must** be handed in **before** the end of the session. Plan your work and printing around that deadline.

Whenever needing to print out assignments, you should first transfer your files from PCSPS07 back to your local machine. This can be done with WinSCP on university machines or your own Windows machines, alternatives and equivilents for other types of computer exist. The process for doing this is described fully in Section 1.2.5.

1.2 The PCSPS07 Server

To provide you with the tools necessary to compile and run programs written in F90, you will be given access to log in to a unix-based server called 'PCSPS07'. Setting up your account on PCSPS07 and getting used to working in the provided environment will be an important part of the tasks for the first week.

1.2.1 Your Account

Firstly, you need to set up and activate an account to use on PCSPS07. This can be accomplished by completing the following steps:

1. In the Windows environment, click 'Start', then 'Search'. This will present you with a dialog box.
2. Type '`putty`' into the search text, and press return.
3. This should locate the *putty* 'ssh client' for you. Double click its icon to run it.
4. For the 'host name' field, enter '`PCSPS07.kent.ac.uk`', this is the address of the PCSPS07 server. After entering this, click 'Open'.
5. For username, type your ITS user code.
6. For password, type your ITS user code, an underscore, and then your university number.
7. This will log you in, and instantly prompt you to change your password.
8. You will need to enter your pre-generated password again, and then a new password, followed by confirmation of the new password.

During the course of the first few weeks, you may encounter a number of terms which are new to you. This guide will try to explain most of these as you go along. In this section, we used the term '*ssh*', this means '*s*ecure *sh*ell'. A 'shell' in the context of Unix is the mechanism by which you interact in a terminal environment. It is this 'shell' that permits the processing of the various commands which you type into the terminal. The ssh system is a means of creating a shell and performing tasks on another computer remotely. You run an ssh client on the machine you are at, which can connect to an ssh server operating on another computer. In this case the other computer is the server PCSPS07. However, pretty much any Unix based machine can be set up as an ssh server and be accessed remotely, even a regular desktop computer.

1.2.2 Logging In

Having created your account, you will now want to log in. If you have not already, read the '*Unix Based Systems*' section of this document (section 13), or refer back to it, in order to help you gain familiarity with working on PCSPS07.

In order to provide a useful graphical interface to the PCSPS07 server, you will be shown how to use a piece of software called 'Xming'. This is a free X window system server for Windows machines. The X graphics system is the display environment used by linux and other unix-based operating systems. To provide a graphical interface to the PCSPS07 server, it will be forwarding (sending) the graphics information from itself to wherever you are. A university Windows machine wouldn't understand that information on its own. Xming acts as an interpreter of the incoming X graphics information, providing a translation for the Windows display system.

To log in and use Xming you must complete the following steps:

1. In the Windows environment, click 'Start', then 'Search'. This will present you with a dialog box.
2. Type '`xlaunch`' into the search text, and press return.
3. This should locate the Xming X Launch Wizard tool. Double click the icon to run it.
4. It will prompt you with four different window configurations, select the 'One Window' mode, and leave the 'display number' entry as zero. Click 'next'.
5. It will now ask for the requested connection mode, select 'Open session via XDMCP' and again click 'next'.
6. It will prompt you for the name of the host, enter '`PCSPS07.kent.ac.uk`' and click 'next'.
7. The next screen is for additional options, no changes need to be made here, leave the 'clipboard' box ticked and click 'next' again.
8. At this point, you are presented with the opportunity to save the current configuration to use again. It is recommended (but not vital) to do so. Ensure that the name you give to the shortcut ends with the '`.xlaunch`' extension, as it is not added automatically. In future, double clicking the new shortcut created using this option will automatically start up your desktop session with all of the relevant settings.
9. Clicking 'Finish' will create an XDMCP session connected to the PCSPS07 server.
10. Before logging out or closing the Xming session, make sure to follow the instructions described in Section 1.2.3.

After this you will find an empty, grey desktop window open (if something else appears, ask a demonstrator). From here you can open terminals and run programs by right clicking anywhere in the window and using the pop-up menu which appears.

The most powerful tool within a Unix system is the terminal. To open a terminal, left click on the empty background to the desktop and select 'terminals' and then 'terminal' (there can be multiple terminal types available, but in this instance only one is configured).

The program used earlier in order to initially create your account, putty, provides direct access to a single terminal within PCSPS07. This is quicker for small tasks than using the full graphical interface, but does not permit you to have multiple terminals simultaneously or let you use any of the graphical text editors. If you would like to use putty regardless, a convenient non-graphical text editor that runs in the terminal window is 'nano', described briefly in Section 13.3.

The advantage of using putty is that it forces you to learn quickly how to do the simpler tasks through the command line, understanding these tasks is important to making best use of a Unix environment. Additionally, there is a version of putty for virtually all machine types and is extremely robust, so this can be used from your home computer (with the same server choice and username/password as you use to connect to PCSPS07 from the workroom computers with Xming).

1.2.3 Closing Xming Session - IMPORTANT

Please ensure that to end an Xming session, you **do not** log out or exit within the desktop session (ie by right clicking and selected 'exit' or 'log out'). Due to an issue with the server, you **must** close Xming itself in order to exit safely. Use the key-combination `alt+F4` or click the '`x`' 'close window' button in the window bar in the top right.

1.2.4 X-Window Forwarding - Linux and Mac (Optional)

If you use a Linux or Mac system, ssh interaction and x-forwarding is generally built-in. Should you want to use graphical forwarding from the PCSPS07 server, in a terminal, type the following:

```
:> ssh -X PCSPS07.kent.ac.uk -l username
```

or alternatively:

```
:> ssh -X username@PCSPS07.kent.ac.uk
```

Both perform the same action. In the first case your username has been specified following an additional command line option (`-l` for *l*ogin-name'), and in the second case it has been specified as an extension of the address you are attempting to ssh to. In both cases the option '`-X`' is specifying that 'X' information is to be forwarded from the ssh server. Obviously, where you see '`username`' in the above lines, you put your own PCSPS07 username.

Note that if you do use Linux or Mac, or have a Fortran compiler on your Windows machine, it is not sufficient to only compile on your home machine. While you are permitted to do this in the course of designing, writing and testing your code, ultimately your code **must** be capable of being run on the PCSPS07 server, and you should ensure that this is the case.

1.2.5 Transferring files using '*WinSCP*'

It is necessary to move files back and forth between your PCSPS07 account (the 'remote' location) and the machine you are physically at (the 'local' location). The simplest way to do this from a Windows machine is by using a program called '*WinSCP*', which is a free sftp (*S*ecure *F*ile *T*ransfer *P*rotocol) client. The university machine have this program available by default, it can also be downloaded from http://winscp.net/.

To transfer files simply open WinSCP, then enter details to connect as you would using putty (server: *PCSPS07.kent.ac.uk*, your PCSPS07 username and password), and the connection type should be 'sftp'. After this, it should display two panels, side by side. Each is a file browser located in a directory, the left-hand panel should display a 'local' directory (a folder on the machine you are working on); the right-hand panel should contain a 'remote' directory (a directory, usually defaulting to your home directory, on your PCSPS07 account). Each of these panels can be browsed separately to access files from either system from anywhere you have access to on each system. To transfer a file between the systems, simply highlight and then drag-and-drop to relevant files from one panel to the other. A description of the transfer will briefly appear, and the file should then be accessible on the system it was deposited in. Note that, by default, WinSCP should only *copy* files, not *move* them, i.e. a transferred file will remain on both systems after the transfer. Keep this in mind if you are intending to free space on your PCSPS07 account without permanently losing files. After transfer, they may have to be separately deleted.

Once files are transferred back from PCSPS07 to your Windows machine, you can open them with a text editor (for source code and output text files) and print in the usual manner for university machines. Note that usual university printing charges **do** apply. Do not print unnecessarily.

1.3 References

Not all information obtained or checked from other sources is referenced directly, as references were used in a piecemeal fashion for various small items throughout the document. However, wherever possible, all sources are acknowledged in the 'references' section at the end of this document (Section 17). In particular, the reference material for the GCC compiler collection and specifically gfortran, the compiler used for this module, have been referred to for many specifics: the GNU GCC compiler collection manual [2]; the GNU Fortran manual [3]; and the GNU Fortran web site [4], all of which are freely available on the Internet via the GNU web page, http://www.gnu.org/[5].

Of additional interest is the Fortran related literature in the Templeman Library. As of writing, several of the key books available include: '*Fortran 90 Programming*' [6] (Notable as five copies of this text should be available in the library); '*Fortran 90/95 Explained*' [7]; '*Advanced Scientific FORTRAN*' [8]; and '*Fortran 90*' [9]; as well as two volumes of '*Numerical Recipies in Fortran 90*' [10] [11]. Care should be taken when finding a suitable book for an overview or guide to Fortran, as many books on Fortran 77 are still in use. This course is based specifically on Fortran 90, which features a number of changes from F77. The concepts in F77 programming are broadly the same, but specifics will differ, particularly in relation to more modern programming uses and standards. All references to the available library texts are valid to the date of the last update to this part of the document (05/12/13).

1.4 The Basics of Programming

All of the programming done as part of this course can be described as 'linear'. Based on the code which you provide the program with, the computer will attempt to execute instructions one by one, in order, until it successfully completes or it encounters a problem. This idea of the computer working step by step through the instructions given to it is vital to understanding the programming covered by this course. Always keep in mind that any code that you write for a program, the program does not know what it was you intended. It will simply do exactly as instructed. As such it must be instructed in the right way if it is to operate in the way you want it to. It is this translation between what the outcome of a specific series of the limited instructions available to a program will be, and what your intent is, which is the basis for attempting to write code of any sort.

1.4.1 File Editing

Firstly, in order to create and edit text files, you will need to open an editor. A number of editors are described in Section 13.3, we recommend either 'nano' or 'gedit'. If you are in the desktop environment, right click the desktop, select 'accessories' and then 'gedit' to open an instance of gedit. Alternatively, type the following command into a terminal:

```
:> gedit &
```

This will open up a gedit window. Select 'File', 'New' in the menu bar, or 'New' button in the toolbar to begin a new file. Alternatively, you can use the following command:

```
:> gedit filename &
```

Where filename is the name of any file. If the filename already exists, it will try to open that file to edit it. If a file of that name does *not* yet exist, it will create a new file of that name for you, as well as opening it to edit as though it were already there.

In both of the above commands, the line was ended with an ampersand, '`&`', this is a special instruction to get the terminal to background the process which you are running. Many commands run in the terminal itself, only permitting you to type more commands once they have been finished or closed, by adding the ampersand, it runs the command in the background instead, letting you continue to use the terminal.

Instead of gedit, you can also use nano, it is a minimal text editor and only runs in the terminal. If you attempt to background it *will* appear to begin running, but will not give you an opportunity to interact with it. This is because it does not have a graphical interface of its own and is designed to be run within the terminal window. Whenever you use the ampersand to background a process, you may notice that before the program opens, the terminal displays a number in square brackets, this is the background ID for that terminal of the command you chose to run. If you accidentally run a program designed to run in the terminal, such as pico, and it runs but you can't access it, you can type '`fg number`' where the value for '`number`' is the background ID for that terminal. `fg` is the command for '*f*ore*g*rounding' a backgrounded process. The background numbers given when using the ampersand do *not* transfer between different terminals, however. This means you can't use this method to background a process in one terminal, and then foreground it in another.

1.5 Using Unix Systems

It is strongly recommended that you either read, or refer to, Section 13 in order to gain a better understanding of using the Unix environment as a whole. At the very least it is helpful to learn the `man` command (Section 13.1.1) which you can then use to find out information about other commands using the terminal itself, as and when needed.

1.6 Simple Program

To begin looking at coding in Fortran, simple programs tend to be of the 'hello world' variety. This involves setting up possibly the most basic program that can be made. When it runs, it prints the message 'hello world' to the screen. Once you've seen this, we can look at how one compiles Fortran code, then go through the various lines to understand what they mean (the numbers to the left are just the line numbers, to make it easier in this document to refer back to specific lines, not related to anything you need to type):

```
1  PROGRAM helloworld
2
3    IMPLICIT NONE
4
5    !print out the phrase "hello world" to the terminal
6    WRITE(*,*) 'Hello World'
7
8  END PROGRAM helloworld
```

1.6.1 Compiling

This is the entire content of the program, so if you type this into a file, save it as '`helloworld.f90`' and compile it, it should run and print 'Hello World' into the terminal that it was run in. There are several different compilers which turn Fortran code into programs, the one used in this course is a free compiler called '`gfortran`'. This stand for '*G*NU *Fortran*' where the 'GNU' is a recursive abbreviation meaning '*G*NU's *N*ot *U*nix'. Many useful pieces of software created by GNU are available (all free), more can be found out at http://www.gnu.org/[5]. In order to run `gfortran` on this 'hello world' code to compile it, you need to type the following (assuming the code has been saved as '`helloworld.f90`', if you have not already done this, do so now):

```
:> gfortran helloworld.f90
```

This will create a compiled program of your code, and name it '`a.out`' (this name may differ on some systems, depending on what is set up as the default name). To run this program, simply type:

```
:> ./a.out
```
You should see a line saying 'Hello World', followed by a new empty prompt.

The code for a given program is know as the '*source code*' or simply '*source*' for that program (it being the code from which the program was sourced). You may see files with code written in them referred to as such within this course.

Obviously the name '`a.out`' is not terribly helpful, certainly not if you plan to write many programs for different things. While you could rename each of these programs manually after compiling them, this would be tedious. For this reason `gfortran` features an 'optional argument' (see the explanation of the '`ls`' command in section 13.1.1 for more on 'options'). This optional argument, '`-o`' for '*o*utput' also expects an argument of its own. Wherever you put the `-o`, whatever word follows it will be taken as the name you wish to give to the compiled program. Therefore, running:
```
:> gfortran -o helloworld helloworld.f90
```
Will instead compile a program by the name of '`helloworld`', which can similarly be run by typing the command:
```
:> ./helloworld
```
Important Note: Whatever arguments and options you use for `gfortran`, you should **always** place the name of your source code as the **final** argument. Do not follow it with options, as this can lead to unexpected behaviour. Additionally, be very careful not to mix up the source code name, and the intended program name. If you accidentally type the following:
```
:> gfortran -o helloworld.f90 helloworld
```
With the source code name and desired program name transposed, it will think that you want to compile lines *from* the program file, and *create* a new program called '`helloworld.f90`', if it attempts to do this **it will overwrite your source code file with an empty file which it prepares to fill with the compiled code**. As such it is very important to check for this before compiling programs.

1.6.2 What the code does

We will return again now to code itself, and see what each line does. All F90 programs must start with a line with the format:

```
1 PROGRAM helloworld
```

Where `helloworld` is whatever you have chosen to call your program. This tells the compiler that the code for program '`helloworld`' starts here. Note that this name is not the same as the name of your code file, in that it should not end in '`.f90`'.

```
3   IMPLICIT NONE
```

This is a special instruction called '`IMPLICIT NONE`'. This line should always be included at the start of your programs. The purpose of `IMPLICIT NONE` will be explained later in the course. For the moment, assume that it is required for every program that you write.

```
5   !print out the phrase "hello world" to the terminal
```

This line, in blue and beginning with an exclamation mark, is called a '**comment**'. A comment is text which is added to the code file of a program in order to aid in explaining what the code is supposed to do, but which will not actually perform any action when the program runs. This is done so that when you return to sections of code later, you can easily remember what they were intended to do and how they work. Also so that anyone else reading your code can be shown how it works and what it's supposed to do.

In F90, all comments begin with an exclamation mark. This tells the compiler that any other text on the rest of the line is **not** code. This means that you can write anything in a comment and it won't prevent the program from compiling or running. Frequently, comments are used to add reminders for what you intend to include later, such as '`!put write statement here`'. Were you to write this without the exclamation mark, the compiler would attempt to read the words as a set of invalid code commands and would fail.

The most important point of comments is to explain the *purpose* of pieces of the code. This is not the same as explaining what a line of code actually *does*, however. Any line of code should be able to be understood in terms of the instruction it causes the program to perform. However, it does not tell a reader *why* it does that, or what the *intention* of a sequence of several lines of code is. It is vital to include this extra information to make the code readable to someone other than yourself, and marks in the assignments are given specifically for making the code sufficiently well commented. Examine Section 14.1 for more details on the correct method of commenting your code. Note also that the marking criteria for your assignments includes suitable use of comments in your code, so is something you should understand.

```
6   WRITE(*,*) 'Hello World'
```

The next line along is known as a 'write statement', it tells the program to output something. If you want to output to the terminal screen, it is written '`WRITE(*,*)`'. Note that the symbol between the two asterisks is a *comma*, not a period.

There are other forms of this for outputting to other locations such as files, and this will also be explained later in the course (See Section 2). Whatever follows the `WRITE` statement on the line is the content which the program is being told to output. In this case, the output is a '**string**'. A 'string' is the term for any sequence of normal characters, letters, symbols or numbers in a non-mathematical context. In other words, if you wanted to add two numbers together, they could not be strings, but if you wanted to store or display a phrase, it would be a string.

Defining a string is done by enclosing the text you wish to make into a string with inverted commas, `'`. You must have both an 'opening' inverted comma, to tell the program that your string has begun, and a 'closing' inverted comma, to tell the program that the string has ended and that anything else on the line is code again. The lack of one of these will prevent your code from compiling. If this happens, you will likely be given an error message of the form of:

```
helloworld.f90:6.11:

WRITE(*,*) 'Hello World
           1
Error: Unterminated character constant beginning at (1)
```

Understanding messages like this is very important for 'debugging' your codes. This is the process of finding and removing problems that the program has. In the case of this error message, 'Unterminated character constant' refers to a string not correctly having the closing inverted comma. The '`1`' points at the *beginning* of the string which is causing the problem. The compiler doesn't know where you intended the string to end, so it cannot point to where the other inverted comma *should* be. This is the case with many error messages, they will only be able to indicate the beginning of a problem, as they cannot know what your intentions for the code actually were.

Also important is the first line of the error '`helloworld.f90:6.11:`', this states the file which the problem was found in, then a colon, then what appears to be a decimal number. This number is actually two numbers, first the line on which the problem occurred ('`6`' in this case), then a full stop, then the 'column' (how many characters along the line) at which it thinks the problem begins (the eleventh letter along in this case, including all symbols and spaces in the line).
While the line and the location on the line is then shown (the middle `WRITE(*,*) 'Hello World` line with the `1` on the line below) this number can still be useful, especially if you have many lines which appear similar in your code and you don't want to search through to find the one which exactly matches the line shown in the message.

```
8 END PROGRAM helloworld
```

The final line of the code has the form `END PROGRAM programname`. This tells the compiler that there is no more code to read for program `helloworld` (or any other name). This is required by all programs that you will write. The program name used here must match the one used for the first line's '`PROGRAM`' command, if it does not, the program will not compile. The error message for this problem would likely look like the following:

```
helloworld.f90:8.26:

END PROGRAM helloaaaaworld
                          1
Error: Expected label 'helloworld' for END PROGRAM statement at (1)
Error: Unexpected end of file in 'helloworld.f90'
```

In this example, rather than the expected '`helloworld`', the code has been changed to have the word '`helloaaaaworld`'. When the compiler reads an `END PROGRAM` line, it checks the subsequent word against all of the program labels that have already been defined. In this case there is only one, '`helloworld`' here. However, this is not always the situation, and it can be the case that there is more than one such program label.

It may appear that the importance of the error messages has been overstated. However, it is almost guaranteed that, at some point, one of your own codes will not work as expected. Messages such as those shown here are vital in determining what is causing the problem. For more information on errors, see section 15.

If you have followed the preceding steps, you have now written, compiled, run and (hopefully) understood, your first program as part of this course, Congratulations!

In subsequent chapters, we shall now explore the other central aspects and components of programming, all branching from this one point.

1.7 Generalised Programs

Having been shown a specific program example, we can now look at the more general form of program organisation in Fortran:

```
PROGRAM programname
```

```
  IMPLICIT NONE

  !Declarations

  !Code

END PROGRAM programname
```

1.7.1 Introduction to Variables

The main form of the code should already be familiar to you. A program is started and ended with the `PROGRAM` statements; the top of the code always has the `IMPLICIT NONE` statement, then the main body of the code is written. The one phrase which may not be familiar to you is 'declarations'. This section, after `IMPLICIT NONE` but before the actual instructions of the code, is where you set up any 'variables'. Variables are names used to store things like numbers or text which you may not want to write into the code; it may receive values after you've compiled the code, for example. Look at this following example involving variables:

```
1  PROGRAM variables_example
2
3    IMPLICIT NONE
4
5    INTEGER :: A
6
7    REAL :: B
8
9    A = 4
10
11   B = 2.6
12
13   WRITE(*,*) A, B
14
15 END PROGRAM variables_example
```

In this example, we have defined two variables: an '`INTEGER`', which we have named '`A`' and a '`REAL`' variable, which we have named '`B`'.

`INTEGER` and `REAL` variable types are both for storing numbers. `INTEGER`s, obviously, store whole, integer values, and `REAL` stores '**floating point**' numbers, essentially decimal values. So the value of '`4`' is suitable for both `INTEGER` and `REAL` (by default, a `REAL` will read this as '`4.0`'). However, an `INTEGER` variable cannot be assigned a value of `1.9`, say. If you try and do this, it is not certain how the program will react as it will depend on how the compiler and system are set up. This is referred to as 'undefined behaviour', where there is nothing in the intrinsic Fortran rules to state what the 'correct' thing for the compiler to do is.

In this example we have also listed two separate variables, separated by a comma, after the `WRITE` statement on line 13. You can list multiple variables for a single `WRITE` statement and they will be printed out sequentially on the same line. Having multiple `WRITE` statements will print multiple lines.

Each variable to be used must be defined at the beginning and have a name unique from all other variables. The enigmatic `IMPLICIT NONE` line of code relates to this. In Fortran it is possible to create variables with implicit types without needing to explicitly define them in the way described so far. However, this can cause substantial problems in that a misspelled variable, rather than causing an error, will simply create a new variable with that misspelt name. It is considered good practice to *always* explicitly define variables at the beginning of your code, and therefore the `IMPLICIT NONE` command is used to inform the program that *no* implicitly typed variables will be used.

1.7.2 Use of Variables - Assignment Material

Important Note: The following will be the first part in which you will need to write a program in the style of the weekly assignments, however, due to the first week being short, you will **not** be required to hand this in.

First, you must create a file with the name '`assign_1_1.f90`'. Into this file you must enter the following code:

```
PROGRAM assign_1_1

  IMPLICIT NONE

  INTEGER :: I
```

```
  REAL :: N

  I = 10
  N = 3.6875

  WRITE (* ,*) I

  WRITE (* ,*) N

END PROGRAM assign_1_1
```

As you should be able to determine, this creates two variables, a `REAL` and an `INTEGER`, assigns a value to each and then prints both numbers out on separate lines. Having written this program, compile it with the following command:
`:> gfortran -o assign_1_1 assign_1_1.f90`
Remember that the '`-o assign_1_1`' tells the compiler to set the *o*utput filename to `assign_1_1`, and the final `assign_1_1.f90` tells the compiler that the source code you want to use to create the program is in that file.
Note: Do not mix up the `assign_1_1` and `assign_1_1.f90`! If you follow the `-o` option with the name of the source code file, it will overwrite it when it attempts to create a program of that name and you will lose your code!

Having compiled your code, you now need to run it. To do this, type the following:
`:> ./assign_1_1`
After entering this, you should hopefully see `10` and then `3.6875000`, followed by a new empty command prompt.

One capability given to you by the terminal is '**output redirection**'. In the case of this program, you have output to the terminal itself. This output is referred to as 'stdout' for '*st*an*d*ard *out*put'. It is possible to redirect items sent to stdout to a file instead. This is done with the output redirect function, called in the following manner:
`:> ./assign_1_1 > assign_1_1.out`
This instructs the computer to run your program, `assign_1_1`, but to instead send any output (the '`>`') to a file (the '`assign_1_1.out`'). Note that if the file you instruct the program to redirect its output to already exists, *it will be overwritten*. Take care to ensure not to output over an existing file. Also note that the '`.out`' is simply the layout that has been chosen for this module, a redirect does not need to be to any file type in particular.

Make sure to keep both of these files, the source, `assign_1_1.f90`, and your output `assign_1_1.out`, since you will need to submit both as part of this weeks assignment. Additionally, add suitable comments to your code (as described in Section 14.1).

If you now open this file, you should find the same two numbers, one on each line, as the output originally displayed to the screen.

At the top of your code file (and **all** of your subsequent code files) you should put comments containing your name and student number, the date at which you last edited the code file and the purpose of the code file. This is considered mandatory for *all* code you write. If you are editing a piece of code which is someone else's, it is in this section that you would put your own name as well as when and why you are editing the code, while leaving their information intact. For the purposes of this course, unless explicitly told otherwise, you should **not** be editing or making use of code which is not written by you, or provided in this document. Doing so without acknowledging the original author is considered plagiarism, and doing so when not instructed even if references to the original author are included will **not** constitute valid work. The aim of this module is to assess *your* ability to code, not that of anyone else.

1.8 More on Maths

1.8.1 Initialising Variables

When a variable is first given a value, that variable is said to have been '**initialised**'. When you declare a variable, it tells the computer to reserve a short section in memory to be able to contain that variables value. It does not yet tell the computer what that section of memory should contain as simply declaring a variable does not give it a value. So what happens if you don't give a variable a value before trying to use it?

Make a copy of your first program and call it `example_1_1.f90`, and remove the lines giving `N` and `I` values. Compile this as `example_1_1`, then run it first without a redirect. What has the program outputted? If you run the program multiple times, does it always output the same thing?

After a variable has been defined, but before it has been initialised, the 'box' in memory for that variable is effectively full of random bits of whatever happened to be in that spot of memory previously. Failing to initialise variables before using them can cause a great deal of problems which can be quite difficult to track down. Therefore, always make sure that a variable has some form of value before it is used!

1.8.2 The Meaning Behind Variable Types

In the previous example the types of variable used were `INTEGER` and `REAL`. The difference between an `INTEGER` and `REAL` is both the way it is stored in memory, and the way it is processed. All computer data is stored in binary. A 'bit' is a single zero or one, a 'byte' is eight 'bits', a kilobyte one thousand bytes etc. In most cases, a Fortran `INTEGER` is 4 bytes long, or 32 bits. This can store any whole number in the range between -2147483648 and 2147483647. A `REAL` variable is different, however. It is effectively stored as two separate numbers, one which represents the floating point part of the number (also referred to as the 'mantissa'), and a separate number for the exponent. For a number of the form $a.aaaa \times 10^b$, the parts represented by the 'a's are stored separately to the parts represented by the 'b's. This gives you more limited precision than storing just the number, but a vastly greater range of values can be represented. In most cases the default `REAL` is also 4 bytes long, has a precision of about 7 decimal places, and can store numbers with minimum and maximum exponents of 10^{-126} and 10^{+127}.

The reason that we can only state that in 'most cases' `REAL`s and `INTEGER`s are 4 bytes long, is that the definition of what constitutes a `REAL` and `INTEGER` is dependent on the machine, *not* the language. The sizes of the default variables are defined by the computer, not the Fortran language. This should not be an issue for the assignments set throughout the course, but is worth bearing in mind.

1.8.3 More Variable Types in Fortran

In addition to `REAL` and `INTEGER`, there exist a variety of other variable types. Firstly there is a variation on `REAL`, the '`DOUBLE PRECISION`' variable type. This is effectively the same kind of floating point style number that a `REAL` is, but rather than stored in four bytes, it is stored across eight, providing much greater precision and range but at the expense of twice the amount of memory or storage.

Another variable type is that of '`CHARACTER`'. This is the type used to store single, or by extension, multiple ascii characters to form text. As mentioned as part of the 'hello world' program is the concept of a 'string'. A string is what stored text is called. In Fortran strings are made of a multiple length `CHARACTER` variable. The specifics of using string variables is covered later in the course (primarily in Section 8).

Another variable type is that of '`LOGICAL`'. These are simple variables with only two states: true (notated within Fortran as '`.TRUE.`') and false (notated as '`.FALSE.`'). These are used for logical/conditional tests, their use will be covered extensively in the section on '`IF`' and '`CASE`' statements (Section 3).

The final type we will consider here is the type called '`COMPLEX`'. This type is designed to store complex numbers. In terms of structure, it is effectively two `REAL` variables back-to-back, one representing the real portion of the complex number, and the other the imaginary part. The mathematical routines in Fortran intrinsically support this type, meaning that true complex operations can be performed without having to write additional routines.

1.8.4 Maths in Fortran

Now we will look at a slightly more complicated program, and show how variables can be used to perform useful mathematical tasks:

```
1  PROGRAM discriminant
2
3    !************************
4    !code written by U. N. Known, last edited 31/03/10
5    !Program to find discriminant for a quadratic equation of form ax^2 + bx + c
6    !************************
7
8    IMPLICIT NONE
9
10   !set up variables, a, b and c are the coefficients, ans is to store the answer
11   REAL :: a, b, c, ans
12
13   a = 2.0 !!
14   b = 3.0 !Initial Conditions
15   c = -2.0 !!
16
17   ans = b**2 - 4 * a * c !calculate discriminant
18
19   WRITE(*,*) ans
20
21 END PROGRAM discriminant
```

In this program we have created four REAL variables, a, b, c and ans in order to calculate the 'discriminant' part of the quadratic formula, where the quadratic formula is:

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

and the discriminant is the term within the square root:

$$b^2 - 4ac$$

Now, depending on the values chosen for the coefficients, the program will output the discriminant of the quadratic formula. Looking at line 20, defining ans, we can see how this is done. The regular maths-style symbols (+ for plus, - for minus, * for multiply and / for divide) can all be used to combine variables. To the left of the equals symbol is always the variable whose value we are setting; there can only be one variable on this side of the assignment. On the right side of the equals, any combination of variables with mathematical operators. Note that there is no implicit multiplication, if you put 'C = A B' this will not implicitly multiply A and B together, you must use 'C = A * B'.

To raise a value or variable to a power, you use double asterisks, as shown with the 'b**2' part of line 20, representing b^2. This covers the key mathematical operations available, although more will be covered later, including more complicated functions like sin and cos.

Also important to note is that this example of code has been furnished with suitable comments. Firstly, the author and date of the last edit, as well as the purpose of the program. In the case of the programs you write as part of this module, you should also include the assignment number.

1.8.5 Assumed and Returned Variable Types

Every value within a program has to be interpreted as a specific variable type. This means that for an expression like a = b + 3, the 3 has to be understood by the computer as one of the variable types described, does it choose INTEGER or REAL? It has to be one of these (or one of the other numerical variable types) in order to understand how mathematical operations on it will work. A single number, with no decimal place ('3' for example), will be automatically interpreted as an INTEGER; one with a decimal place will be assumed to be a REAL (3.0 for example).

It is necessary to know that the result of any of the intrinsic mathematical operators depends on what its operands are (the values that it is operating on). 'Mathematical operators' including such as +, -, *, / and others like SQRT introduced in the next subsection.

If you have an expression 3 + 5; both operands, 3 and 5 are INTEGERs, therefore the result of the addition (the operation) will also be an INTEGER. This knowledge is vital in some circumstances. Consider the operation 3 / 4, we would presume this would give us 0.75, however it would not. As both the 3 and the 4 are expressed as INTEGERs, integer division will be performed. In other words, how many whole times 4 will 'go into' 3. For this example, the result of 3 / 4 will be 0, 4 cannot be made to 'go into' 3 an integer number of times (the 'remainder', 3 in this case, would not be given in this operation). This could be fixed by phrasing the operation as 3.0/4.0 instead; with both operands interpreted as REALs, the result will also be REAL and give 0.75.

1.8.6 More Maths in Fortran - Assignment Material

Important Note: As with the first 'assignment' in this chapter, this subsection is to get you used to writing assignment style material, but you will **not** need to hand this in. It is vital that you complete the work, however, as it is expanded upon in subsequent weeks.

For this subsection, you will need to create a file assign_1_2.f90, into this, you should type the contents of the 'discriminant' program in the previous subsection. Modify the header comment to something suitable for your own work. **IMPORTANT NOTE:** You can leave out the 'U. N. Known' name in this case, but be aware that *whenever* using code written by someone else, you *must* reference that you have done so, and what you have. Failing to do this will count as plagiarism. For the purpose of this course, unless stated otherwise, you should *not* use anyone else's code regardless of whether you reference or not, the purpose of this module is to teach and assess *your own* ability at coding, not someone else's.

Having written the code into your assign_1_2.f90, to ensure that it works, try compiling and running it using:

```
:> gfortran -o assign_1_2 assign_1_2.f90
:> ./assign_1_2
```

If it outputs '25.000000', it has functioned correctly. We will now proceed to make alterations to this program to get it to output the two values of 'x' for the quadratic formula. Firstly we will find the solution equivalent to the instance of the quadratic formula where we use 'plus the square root':

$$x_{\text{plus}} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

Later we will look at the other instance, for 'minus the square root':

$$x_{\text{minus}} = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

To begin with, add a new variable on the line defining the other `REAL` variables, call this '`x_plus`'. After the line calculating the discriminant, add the following line of code:

```
x_plus = (-b + SQRT(ans))/(2 * a)
```

Here we have shown two useful techniques, the first of these is using brackets to ensure that the mathematical operations are carried out in the correct order. Bracketed statements are always evaluated ('calculated') first, the 'deeper' the bracket is nested within other brackets, the earlier it will be evaluated. This follows the regular 'BODMAS' (*B*rackets *O*ther *D*ivision *M*ultiplication *S*ubtraction) or 'BIDMAS' (*B*rackets *I*ndices *D*ivision *M*ultiplication *S*ubtraction) algebra rule you should be familiar with.

The second technique is illustrated through the reuse of the `ans` variable. Rather than calculate the entire formula on one line for one variable, it is possible to use several extra variables to calculate parts of a formula, and then combine those instead, making the maths clearer in your code.

The final thing to note is the use of an 'intrinsic' maths function; `SQRT`. An intrinsic function is effectively a special command to perform a more complicated task in one go, or a literal mathematical function such a *sin* or square root. In this case we include the square root function, which is used like this:

```
SQRT(var)
```

Where '`var`' is some variable of one of the floating point types (`REAL`, `DOUBLE PRECISION` or `COMPLEX`).

Variables should also have names which are sensible for their purpose. In this case the name '`ans`' is a bit too generic, it is not named in line with it's purpose, replace it with '`discrim`' or similar, to represent the fact that it is the discriminant. You should need to change it in three places, the line where it was defined, the line where you assign it a value, and the line where you use it to calculate '`x_plus`'. Now also change the `WRITE` statement such that it outputs `x_plus` instead of `ans`.

Your final task to complete the coding for this assignment is to add a suitable variable and calculation to find the 'minus the square root' instance of the quadratic formula, as shown in Equation (2); then add a suitable `WRITE` statement to print this value *as well as* the 'plus the square root' value already calculated.

Remember to create sensible comment lines, as well as choosing a sensible and descriptive variable name. Having done all of this, compile and run your program to test if it works. If it appears to work, run it again, redirecting the output as follows:

```
:> ./assign_1_2 > assign_1_2a.out
```

What happens if you choose other values for the coefficients, a, b and c? Try running it with several sets of values of your own, though you needn't output these to a file. Now try running it with the values: `a = 2.0`, `b = 3.0` and `c = 2.0`. What happens, and why might it be the case? It is worth knowing that '`NaN`' means '*N*ot *a N*umber'. After running with no redirect to see what the result is, run it again, redirecting as follows:

```
:> ./assign_1_2 > assign_1_2b.out
```

The following is not a task to complete, but begin thinking about what sort of things would be necessary to make a program that more completely covers the different possibilities for the solution(s) of x from the quadratic formula.

1.9 Assignments Summary

Important Note: This section lists the expected work to be done for the first week in the format used for subsequent assignments. However, you do **not** need to hand this work in. There is **no** work to hand in for week one.

1.9.1 Programming Assignment 1

1. Write the program `assign_1_1` as described in Section 1.7.2.
2. Produce an output file by redirecting the output from the program using the command:
   ```
   :> ./assign_1_1 > assign_1_1.out
   ```

1.9.2 Programming Assignment 2

1. Write the program `assign_1_2` as described in Section 1.8.6.
2. Follow the instructions in Section 1.8.6 to add the calculation for `x_plus`, as well as the changes to the `ans` variable.
3. Through the creation of a new variable and writing of a new calculation, improve the code to calculate `x_minus`.

4. Using the original values for the coefficients, a, b and c, produce an output file by redirecting the output from the program using the command:
   ```
   :> ./assign_1_2 > assign_1_2a.out
   ```
5. Change the values of the coefficients to: $a = 2.0$, $b = 3.0$, $c = 2.0$.
6. Produce an output file of this by redirecting the output from the program using the command:
   ```
   :> ./assign_1_2 > assign_1_2b.out
   ```

Important Note: Make sure to keep the source code to this program after completing the assignment, it is used as the basis for another assignment later in the course.

1.9.3 Questions

In addition to the programming tasks, there will generally also be a short set of questions designed to test your knowledge regarding the programs themselves, as well as the general concepts covered by the Section (you may also need to refer to the section on using Unix, Section 13). For this week consider the following questions, as with the two programming assignments, you do **not** need to hand in these questions for this week (there will be a file uploaded to moodle with the answers next week to check your own answers):

1. What would you type to create a directory called 'mydirectory'?
2. What would you type to move a file 'myfile' from your current directory into your new 'mydirectory' directory?
3. How would you now move into and then out of 'mydirectory'?
4. In the second program assignment, `assign_1_2`, what do you think has happened which produced `NaN` to appear as the values for x?
5. What variable type would be most suitable for each of the following parameters, give a reason in each case:
   (a) Age
   (b) Year, Month, and Day
   (c) The value of pi
   (d) Name
   (e) Whether or not someone owns a car
   (f) Impedance in an A/C circuit.

2 Week Two - Using Input and Output in Programs

2.1 User Input

2.1.1 Basic I/O Concepts and the 'READ' Command

A vital part of programming is the ability to provide a means of inputting data and options to a program. Whilst in the previous examples values for variables have been defined in the code, this is not usually desirable as changing the values requires recompiling the code. It is preferable to be able to compile the program once, and provide any various different inputs to it when it is run. This avoids the need to compile every time you wish to change the input.

The most simple form of input is interactive user input, this is where the program runs, prompts the user to input some parameter or variable value and then reads in what the user types. This can be done using the `READ` command.

The basic format of the `READ` statement is similar to the `WRITE` statement, to read a user input from the terminal and put it in a variable '`var`' you would use the statement:
`READ(*,*) var`
This will cause the program to wait at this point until the user has entered a value and pressed return. Note though that this is subject to the same issues of incompatible variables that regular variable assignments are. If `var` is an `INTEGER` variable and the user enters '`0.4`', this will cause the same rounding problem suffered by assigning a floating point number to an `INTEGER` variable as in the previous chapter.

Worse still is if the users enters a 'value' which is not a number at all, typing letters and/or symbols rather than numerical data. There are ways to make checks when reading input, to ensure the user has entered a sensible value, this is generally referred to as 'input validation'; this concept will be covered in more detail later.

2.1.2 Example

For a functional example of a program using a `READ` statement, the following is provided:

```
1  PROGRAM variables_example
2
3    IMPLICIT NONE
4
5    INTEGER :: A
6
7    REAL :: B
8
9    WRITE(*,*) 'Please input an integer value:'
10
11   READ(*,*) A
12
13   WRITE(*,*) 'Please input a floating point value:'
14
15   READ(*,*) B
16
17   WRITE(*,*) A, B
18
19 END PROGRAM variables_example
```

In this example, the `WRITE` statements on lines 9 and 13 provide prompts to the screen to let the user know that input is expected and what it should be. The following `READ`s on lines 11 and 15 take the typed data in, and puts each into the stated variable. For each `READ` a sequence of variables can be taken at once, by listing the variables expected after the `READ` statement separated by commas. Then, the code will expect several numbers from the user, which can be separated by a variety of delimiting characters (the most common being a space). If the user enters several space separated variables when only one variable is expected, an error will not be raised; in this case it will simply take the first entry, and disregard the remainder of the line. However, if the input expects more variables than the user enters, it will continue to wait until it has been given the expected number of values.

2.1.3 Other I/O Concepts

Using this style of interactive input is sometimes useful, but is discouraged for situations where you are likely to run the program many times with similar options, where you want to more easily repeat the input without typing each value separately every time. Interactive input is also discouraged for instances where there are a great many input parameters.

In these cases, it is more common to either use an input file, with a list/lists of the values, so you are easily able to change one out of many and run the program again straight away. The other is to use command line arguments for your program, similar to those you have already observed for things like the `gfortran` compiler. In this case it is clear why command line arguments can have an advantage. By using the up key in the terminal you can go back to the entire compile command to recompile after changes. This would be much more time consuming if you had to write every argument each time, especially if the gfortran program has to ask you questions such as whether you'd like to use the default output name or not.

The next subsections will illustrate both of the concepts just described: inputting values from a file, and inputting values from the command line.

2.2 Input/Output From/To a File

2.2.1 Opening Files

Frequently it is advantageous to read information from a file, either for input of options/settings or data. Both reading and writing from a file is done with the same commands as reading from or writing to the command line, with one slight alteration. You will remember that all of the `READ` and `WRITE` commands have required an additional pair of arguments: `(*,*)` These are for defining *where* the statements are reading/writing from/to and *how* they are reading/writing.

To read from a file one of these two asterisk filled fields needs to be used. The first argument defines the 'file unit' to be used. The file unit is basically the reference number for the file or other output being used. The second argument is the formatting rule for how to read or write the data.

To open a link to a file and assign a file unit number to it, we must use the '`OPEN`' command. This is used in the following manner:
`OPEN(unitnumber,FILE=filename)`
In this case, '`unitnumber`' is the file unit number which we would like to assign this file to. This number can be between 1 and 99, with the exception of 5 and 6. Earlier, we covered the concept of 'standard in', `stdin`, and 'standard out', `stdout`, these input and output locations are handled exactly as files are, and as such have unique file unit numbers of 5 for `stdin` and 6 for `stdout`. In regular usage, the asterisk automatically uses either 5 or 6, for `stdin` and `stdout`, depending on whether you are using `READ` or `WRITE`.

2.2.2 Performing I/O with a File

So let's say we have a file containing settings to be read in when our program starts. Calling this file '`setup.dat`', the process for reading in data from this file would be the following:

```
1  PROGRAM file_io
2
3    IMPLICIT NONE
4
5    INTEGER :: A
6
7    REAL :: B
8
9    OPEN(10,FILE='setup.dat')
10
11   READ(10,*) A, B
12
13   CLOSE(10)
14
15   OPEN(11,FILE='output.txt')
16
17   WRITE(11,*) A, B
18
19   CLOSE(11)
20
21   WRITE(*,*) A, B
22
23 END PROGRAM file_io
```

After defining our variables, we `OPEN` the `setup.dat` file as unit '`10`' on line 9; there is no particular reason for 10 being a particularly suitable unit number, different numbers may be more or less suitable for different purposes. The only vital factor is keeping in mind the reserved unit numbers and upper limit.

After this, we read in two variables through the READ command on line 11, A and B, in this instance these are just generic values. Note the use of the unit number of 10 in the READ command.

Another important command is now used on line 13. The 'CLOSE' command tells the program to stop its link to the opened file. Practically, this process is done automatically when the end of the program is reached. However, it should be considered a necessity to explicitly close files as soon as it is possible to do so.

Now we will perform output to another file. This new file is opened on line 15 in the same manner as line 9, however with a new filename and the unit number 11. A WRITE statement is then used on line 17 to put our two variables into the file; then the file unit is closed on line 19.

Finally, for the sake of this demonstration program, the values of A and B are also printed to the screen.

2.2.3 The Program in Operation

Were you to try to run this program straight away, you would get an error. Without a 'setup.dat' file already existing, the program presumes it must need to create it. However, having done so, it attempts to read data out of the new empty file, and instantly reaches the end of the files contents without finding anything. In order to run the program successfully we must create the input file for it to use. By now you should hopefully be fairly familiar with how to create plain text files with particular names. Create one now, in the same directory as your code/executable, called 'setup.dat'. It should contain two numbers, separated by a space, on the first line. These should be an INTEGER, then a REAL, the same as will be read in on line 11.

Now when the program is run (if you have already compiled, there should be no need to do so again), you should see the two numbers you entered printed to screen. Now, in a terminal within the same directory as your code and input file, you should also find a new file created, 'output.txt', which also now has the two numbers in.

In order to understand the manner in which the READ statement is extracting the text from the file, try various different contents in your setup.dat file. For example: separating with a comma rather than a space; having more or fewer values than required; having non-numerical characters in the file; or similar. It is through experimentation like this, finding the limits of what works and why, that you gather an intuition about what constitutes valid code, and what might be the problem if code doesn't work.

It is worth noticing that, whatever changes you make to the input, the file containing the output is overwritten from scratch each time. This is presuming that the code reaches line 15, if you tried the example where too few, or incorrect, inputs are present in the input file, the code will stop at line 11, where we attempt to read the input variables. This means that the code never gets to the point where it opens, and hence tries to rewrite, the output file.

2.2.4 Writing Files From Scratch or Appending Data

Overwriting the content of a file when producing output is not always desirable. To change this, when opening the output file, you can add an additional argument, 'POSITION' as follows:

```
OPEN(11,FILE='output.txt',POSITION='APPEND')
```

This will mean that each WRITE command, data will only be written to the end of the file, added on to whatever was previously there.

In some compilers, there exists the extension that the APPEND attribute can be applied to the ACCESS parameter, as in: `OPEN(11,FILE='output.txt',ACCESS='APPEND')`, however, the availability of this feature depends on the version of the system/compiler being used.

Try replacing the existing 'OPEN' command on line 15 of the previous program, then recompile. Running the program several times and checking the output.txt file will show that the output is gradually being expanded upon, with each instance of output added to a new line.

2.3 Command Line Arguments (Optional)

In the section on Unix (Section 13) and on how to compile code (Section 1.6.1) you will have seen use of command line arguments for the programs and commands covered. This is such as adding the name of your code after the command for the Fortran compiler, in order to tell it what you want to compile.

It is possible to implement this same system into your own Fortran codes too.

2.3.1 Using 'GETARG'

The key command in obtaining arguments from the command line is '`GETARG`', it is used as follows:
CALL GETARG(argnumber,targetvar)
Where '`argnumber`' is the index of the command line arguments, counting from the first word provided after the name of your program itself when you run it. Hence, were you to run a program called '`action`' with two arguments '`1.5`' and '`5`' as in this example:

```
:> ./action 1.5 5
```

running `GETARG(1,targetvar)` would extract the '`1.5`' and `GETARG(2,targetvar)` would extract the '`5`'.

The second part of the `GETARG` call, the part called '`targetvar`' in the previous examples, is the name of the variable which you want to read the command line argument into. Unfortunately, it is not as simple as simply specifying the name of the variable you plan to use.

All command line arguments are initially read in as 'strings', sets of `CHARACTER` variables with no mathematical context. Hence, it would not read the `1.5` as a number, but as the text. For this reason, a temporary variable is commonly used to first extract the command line argument, which can then be converted out to the intended proper variable. In order to illustrate this, we will use an example:

```
1  PROGRAM cmdline_input
2
3    IMPLICIT NONE
4
5    REAL :: A
6    INTEGER :: B
7    CHARACTER*128 :: arg
8
9    CALL GETARG(1,arg)
10   WRITE(*,*) arg
11   READ(arg,*) A
12
13   CALL GETARG(2,arg)
14   WRITE(*,*) arg
15   READ(arg,*) B
16
17   WRITE(*,*) A, B
18
19 END PROGRAM cmdline_input
```

We have first declared three variables, a pair of generic variables `A` and `B` which we intend to use for two numbers which will be supplied on the command line; the final variable is a 128 length `CHARACTER` variable, a 'string'. This variable '`arg`' can contain 128 characters consisting of any of the standard ascii symbols.

Having set up our variables, we can begin to read in. On line 9 we use the `GETARG` command to take all of the character symbols from the first argument (indicated by the `1` as the first argument in the brackets) and put them into our temporary '`arg`' variable. For the purpose of this example, we then have a `WRITE` command on line 10 to display what our temporary variable has read in.

The next step may appear unusual, however, it is an important one. On line 11, we now use the `READ` command on our temporary variable, as though it was an input or a file. The output of this `READ` command is to the first variable, the `REAL`, `A`. Doing this performs the conversion from text to mathematical number. If you think about it, in all of our other inputs, the situation has been exactly the same, prior to assignment into a variable, the information is all just ascii characters, not genuine numbers. When we used the interactive `READ` in Section 2.1.2 what we type when the program waits for our input is just text, *symbols* for numbers, but not actual numbers as far as the computer is concerned.

For instance, the ascii code for '`a`' is '97', everything in a computer must fundamentally be stored as numbers. But when the computer knows that it is receiving input from a keyboard or text file, and it sees '97' it knows it needs to display the character '`a`'. Similarly, codes are used for the displaying of numerical characters too, the code for the symbol '`1`' in ascii is '49'. A specific conversion will always need to be performed between user input and their use as actual numbers, or any variable type other than text. However, this is not always the case when computers use input made from output created by another (or even the same) program. In these instances, instead of outputting as ascii which is human readable, it can be instructed to output as a binary file, which will contain the raw numerical data, but is not easily human readable.

The important thing to remember is that human typed input such as '50239' is *not* actually a number, to begin with it is just a string of ASCII character symbols.

Returning to the code itself, having read the argument in on line 11, we then use the `GETARG` command again on line 13. Note that again we have used our temporary variable '`arg`' as the target of the `GETARG` command. This will overwrite

the previous content of the variable and replace it with the new content, the second variable entered on the command line when the program is run. Again, for the sake of illustration, we output the contents of the temporary variable using the `WRITE` command on line 14. Following this on line 15, another conversion is performed using the `READ` command, this time from the string contents of the temporary variable to an `INTEGER` variable, `B`, as opposed to the `REAL` used for the first input.

Finally, both of the converted variables are displayed on the same line, using the `WRITE` command on line 17.

2.3.2 Running the GETARG example

Now we can try running our program. Compile as usual, but instead of simply using '`./`' then the program name to run, use the following:

```
:> ./cmdline_input 5.21 8
```

In this case we are using the value '`5.21`' as our first command line argument, and '`8`' as our second. What happens? Hopefully the output should look something like the following:

```
 5.21
 8
   5.2100000                8
```

What do you notice that's different between the temporary variable being output, and the equivalent generic variable `A` and `B` being output at the end? The `5.2` and `8` which are on the first two lines are still in character form, exactly as they were entered on the command line, from storage in a `CHARACTER` type variable. On the final line of output, they are output as they were stored as actual numbers, in a `REAL` and `INTEGER` variable respectively.

As with previous forms of input, this method of input is susceptible to incorrectly formatted data being provided to it. Providing the letter 'a' instead of a number to either argument will cause a crash. Also, just like when only one variable was in the input file in Section 2.2.3, if only one number is written on the command line when you run the program, it will complain that it has reached an unexpected end of file. This is due to the similarity of how **`stdin`** and **`stdout`** compare to normal files.

The issue of too few variables being present in an input, and a crash resulting due to this, can be treated using 'input validation', to make sure that input is present and suitable as you go along, meaning that you can make your program able to provide specific instructions or information if something is missing or incorrectly formatted, rather than simply crashing.

2.4 Input Validation

In this section we will cover the concepts of input validation, however, due to coding requirements of implementing these concepts, actual examples and tasks related to validating input will have to wait until later in the course.

2.4.1 Basics

In commercial programming, any user input is idealised to be as user friendly and forgiving as possible. By and large, in commercial software you will likely be familiar with, if you enter something incorrectly, or try to perform an action which is not valid, the program will give you a message saying so, and give you a chance to correct the mistake, rather than simply crashing.

Scientific programming tends not to be quite as thorough, there is a certain expectation that someone will only be using a particular program if they know how it is supposed to work already, or should be able to figure it out themselves before too long. Whilst this may be the reality, it is still not a state of affairs to necessarily be desired.

However, even in scientific programming, it is important to provide at least a modest standard of input validation. The level of care which permits a user to choose a correct format for the 20th item of input after typing incorrectly, rather than crashing instantly and requiring them to write all 20 again...

2.4.2 Fundamental Validity and IOSTAT

The most basic check is simply to make sure that a particular input is of the right type and format, in other words that it is actually possible to read it into the intended variable or variables. This could cover whether there are simply enough arguments.

When reading from the command line, you can create a variable which tells you how many arguments have been provided, this is done by using the following:

```
N = IARGC()
```
Where 'N' (or whatever you might want to call your variable) is an INTEGER. The 'IARGC()' command fetches the number of arguments present. Then, if you are expecting a certain number of arguments, you could compare that number with this new N value, to see whether the user had input the correct number of arguments.

When reading from files, the most useful tool for making sure input is performed correctly is 'IOSTAT'. In the same way in which we added 'POSITION='APPEND'' to our READ statement in Section 2.2.4, this is added into a READ statement to permit various checks, and prevent the code simply crashing when something is not right with a read operation. The form of it's use might be something like:
```
READ(1,*,IOSTAT=io1)
```
In this case, we are reading from file unit '1', which may be some input file or other, and have previously created an INTEGER variable called 'io1', which the IOSTAT addition is set to. With this in place, the READ statement will complete even if a problem occurred, such as an end of file or similar. With IOSTAT, rather than crashing, the status of the READ operation, whether it worked or not, is put into the variable io1 in the form of a numerical code. After this, we can check for whether the code number in io1 is a valid one, or if it indicates something has gone wrong. For example, if IOSTAT returns a zero, it means no problems occurred. If it returns a -1, it means that the READ statement encountered an unexpected end of file, one of the types of error covered previously in this chapter. In this case, we can add code to check if the value was -1 and attempt to perform an action to remedy it, rather than letting the program crash. In the case of an end of file error, perhaps we could add code to tell the user what has happened, and then prompt them to enter the name of an alternate file which has the correct number of input lines/variables.

2.4.3 Validity of Values

Another level of checking is for once variables have already been obtained. This is the type of validation that you might be familiar with from filling in fields in any kind of online form. Checks for things like the length of a string, whether it's too long, or in the case of passwords, too short. Checks that numerical values lie within certain ranges, so you can't do things like enter an age as a negative number; checks that strings don't contain any characters which are likely to cause problems. Another important type of check is ensuring that, if an input relates to a filename, that the chosen file actually exists, or is of the right format, prior to even attempting to actually open and read from it.

Many of these validity concepts require a 'check' of some sort. A coded comparison, which will choose between whether a certain condition is true or false, and performing different actions depending on which is the case. This is a concept covered directly in next week's work.

2.5 Programming a Code to Include User Input - Assignment Material

In last weeks work we programmed a code to use the quadratic formula to provide us with the plus and minus solutions for x in a quadratic equation of the form $y = ax^2 + bx + c$. For each set of the coefficient parameters, a, b and c, it was necessary to replace them in the source code, recompile the program and then run it to get the answer. This task will expand this program to include user input permitting the values for the coefficients to be changed without needing to recompile the program every time.

Create a copy of your assign_1_2.f90 code, called 'assign_2_1.f90'; it is this file which you will be editing to include user input.

Firstly, you will need to remove the lines which assign values to the coefficients (*Note:* Not the lines *declaring* the variables for the coefficients). Having values which are defined within the code, which would require a change to the source and recompile to alter, are referred to as being 'hardcoded'; they are innate to the program and, without the source code, would not be possible to change. Following this, you are required to add suitable WRITE statements to prompt the user to input each variable, each followed by READ statements as shown in Section 2.1.2 to take the user input and give values to each of the coefficients in turn.

Then compile and run the program, using the values of '4.0' for the a coefficient; '3.0' for b and '-1.0' for c.

Hint: Remember that you can use the form 'WRITE(*,*) 'this is some text'' to output a simple line of text to the screen from within your code.

2.6 Programming a Code to Include File Output - Assignment Material

The next task is to further expand the program from the previous assignment, 2.5. First make a copy of your assign_2_1.f90 code, calling the new instance 'assign_2_2.f90'. Here, rather than outputting the resultant values of x to the screen, you will be altering your code to make it output the results to a file instead.

The intended output file will be called 'assign_2_2.out'. First you must add a suitable line of code to open such a file by using the OPEN command as described in section 2.2.2 of this week's chapter. After this, remove the existing line

which writes the final results to the screen. These then need to be replaced with a set of WRITE commands outputting several things to the file. Firstly, you need a pair of lines describing, and then outputting what the input values were for the coefficients. Then a pair of lines describing and then outputting the two result values which the program should have calculated.

Finally, it's important to add a line closing your open file unit before ending the program.

You should then compile and run your program with the same input as for section 2.5.

Hint: You will have to choose a suitable file unit number for when you open and interact with your file.

2.7 Line Length Limit

The codes so far, and likely for several more of the assignments yet (depending on how you like to code) will have had comparatively short line lengths. An oddity of Fortran, holding over from its punch-card and simple-terminal roots, is an in built limit to the number of characters allowed per line of code. If this is exceeded, the compiler stops reading the line, and everything beyond this limit is discarded. Needless to say, this will likely cause the code to either fail to compile, or to omit something intended when it runs.

FORTRAN 77 specifies a line length limit maximum of 80 characters (72 in practice, depending on the definition of a column of 8 characters not for code itself, but other information). Fortran 90/95 codes can permit 132 characters per line. Exactly how each of these limitations is implemented tends to depend on the compiler used.

If you have a line containing, say a WRITE command and a string consisting just of enough letter 'a's to exceed the line length limit, attempting to compile would provide you with the following error messages:

```
longline.f90:5.12:

 WRITE(*,*) 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
           1
Error: Unterminated character constant beginning at (1)
longline.f90:5.132:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                                                                           1
Warning: Line truncated at (1)
```

The first part of the error (`Error: Unterminated character constant beginning at (1)`) is expressing that it cannot find the final inverted comma which indicates that the string stops; this error is a symptom, rather than the root problem itself. The second part is the key, (`Warning: Line truncated at (1)`) this is the information actually informing you that a line has exceeded the limit, and the rest has been cut off (truncated). Note how the character position specified in this second message is 132, the character limit of the line.

How do we avoid this, or get around it being a problem? There are three main solutions.

1. Write shorter lines. You can generally reformat code such that the maximum required characters per line is actually reduced, and the line length limit is not exceeded. This is somewhat limiting, artificially dictating how your code should be set out and formatted, but is nevertheless an option.

2. 'Line continuation characters'. The ampersand character '`&`' can be used to split a long line across several separate lines, but still have the compiler interpret it as a single logical line. The following is an example of two instances which would be understood identically by the compiler, but formatted differently in the file:

```
f = a + 2 * b - c / 8 + 2 * (d + e)

f = a + 2 * b - c / 8&
+ 2 * (d + e)
```

   The ampersand splits the physical line into two smaller sections, while the logical line (what will actually be processed) remains the same as the original. This is the easiest 'on-the-spot'/quick fix to line length issues, especially if it only occurs in one place.

3. '`--free-line-length-none`'. This is a command line argument for the compiler, added to the line when running the command to compile your code. It specifies that the line length limit part of the standard should be ignored, and any length line permitted. This is specific to the `gfortran` compiler, other compilers may have similar arguments, but will likely be named differently. Where a normal line to compile a piece of code might look like the following:
   `:> gfortran -o longline longline.f90`
   to prevent issues with line length limits, one would use the optional argument described above in the following

manner:

```
:> gfortran --free-line-length-none -o longline longline.f90
```

The first option can be used occasionally, but is undesirable. The second option is frequently the most preferable, as it is built into the standard; in other words, any compiler correctly implementing the Fortran standards will accept that method of avoiding long lines. The third option is the nicest from a coding point of view, permitted code to be formatted as you wish; but codes written with this solution may not be operable for other compilers. Keep these solutions in mind whenever working with larger codes, or especially wherever long strings are being used (arguably the easiest way to exceed the limit).

2.8 The Command 'GOTO' and Why it Should Not Be Used (Optional)

This subsection is somewhat unrelated from the previous material of this week, and is not vital to the flow of the course at this point. Nevertheless, it is a topic worth bringing up at some point in the course, and is probably better sooner rather than later.

One of the commands in Fortran, the use of which is very strongly discouraged, is '`GOTO`'. This command, combined with lines of code labeled with numbers, permits a program to jump from the `GOTO` statement to the specified labeled line, as in the following code fragment:

```
GOTO 10
WRITE(*,*) 'This statement avoided'
10 CONTINUE
```

In this example, when the program processes the `GOTO` instruction, it searches for the line labeled '`10`', and resumes processing from that line, rather than where it was before. Here, the effect would be that the line '`WRITE(*,*) 'This statement avoided'`' would never be processed at all.

The use of `GOTO` dates back to earlier days of programming, particularly cases where codes relied quite heavily on labeled line numbers. In modern codes it has little place, and is either discouraged or outright excluded from use. Many current languages do not implement any form of this command at all. However, it is still fairly common in legacy F77 codes and, as such, you may encounter it at some point.

It could be deemed sufficient to simply state that it is not to be used and leave it at that, however it is perhaps worth a bit more of an explanation, for those who are interested.

The `GOTO` command can appear as though it is a useful tool for moving between certain tasks within a code, and in some ways it is. The issue is that it is exceptionally messy in how it does so. Unless used with extreme care you can easily cause bits of memory to be left allocated but unused, induce issues with vital code missed by accident, it is easy to break intended program structure without realising and additionally means that following the flow of the code for the programmer becomes a nightmare. The relatively structured layout possible with the standard loop and conditional statements is effectively interfered with, with links between areas of the code which will not be immediately obvious.

The final, and perhaps most major, problem with `GOTO` statements in high performance computing, is that their intention is ambiguous. Modern compilers do a substantial amount of optimisation of code when they compile. This process attempts to replace complex operations or combinations of operations with the fastest alternative available. Compilers can also do substantial work in parallelising code. This is vital in modern high performance computing, all current supercomputers are based on the concept of heavily parallel processing. Rather than try and run code linearly faster and faster, you run at a slower speed, but a slower speed on each of every one of, anything between several, to tens of thousands of processors (technically 'cores') at once (high performance computing is discussed more in Section 10).

Producing a code which is parallel to this extent, as well as most efficiently optimised and streamlined would be an exceptionally difficult, if not impossible, task to manually program, so a substantial portion of this process is left to systems of automatically accomplishing this, themselves programmed into the compiler. This is similar to how processor chips themselves are designed today, there is no person or set of people who lay out the intended location of every single transistor in a processor, there are simply too many; people create formulae and sets of instructions of how a processor should be built up, the rules for what needs to be connected to what, and in what way, and the locations are designed automatically.

This is where the `GOTO` statement causes problems. A compiler understands the intention of `DO` loops and `IF` statements and the like, and even very complex combinations of them. As such it can optimise and/or parallelise them very well. However, because the intention of `GOTO` statements is ambiguous, there is no way for the compiler to understand what was intended, or whether it can be safely parallelised around. This severely limits the potential speed of any code using `GOTO` statements.

Summary: There is no single answer to whether `GOTO` commands have their place in modern programming or not. Some believe that they provide no benefit and simply cause many problems; some consider them to be an extremely

useful tool so long as they are employed in very specific ways by skilled programmers only. Based on the complexity of implementing `GOTO` statements in the correct way, for the purposes of this course assume that you *should not* use them in your own codes. Nevertheless, it is worth being aware of how they function.

2.9 Assignments Summary

2.9.1 Programming Assignment 1

1. Create a copy of your '`assign_1_2.f90`' code called '`assign_2_1.f90`'.
2. From this new code, remove the 'hardcoded' assignments of values to each of the coefficients.
3. Add pairs of `WRITE` and `READ` statements to first prompt the user regarding what needs to be entered, and then read the users input, assigning it to the three coefficients.
4. Compile the program in the usual way, and run it, enter the values of '`4.0`' for the a coefficient; '`3.0`' for b and '`-1.0`' for c.
5. You will need to submit a copy of this code, `assign_2_1.f90`.

2.9.2 Programming Assignment 2

1. Create a copy of your '`assign_2_1.f90`' code called '`assign_2_2.f90`'.
2. Add a suitable line of code to open a file to contain your output, the file should be called '`assign_2_2.out`'
3. Add pairs of `WRITE` statements after the result has been calculated, to first output the description and values of each of the three coefficients, and then each of the two values for x calculated by the program.
4. Compile the program in the usual way, and run it, enter the values of '`4.0`' for the a coefficient; '`3.0`' for b and '`-1.0`' for c.
5. You will need to submit both a copy of this code, `assign_2_2.f90`, as well as the contents of your output file, `assign_2_2.out`.

2.9.3 Questions

For this week the questions are as follows:

1. What would an `IOSTAT` value of `-1` mean when reading from a file?
2. What type of variable is used for receiving the `IOSTAT` value?
3. What are the two main reserved file unit numbers?
4. What validation criteria might be applied to the following desired inputs?:
   Hint: For example, for age, you would want to check that it was a number, greater than zero, etc.
   (a) Time
   (b) Telephone Number
   (c) Velocity

2.9.4 Submission

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_2_1.f90`, `assign_2_2.f90` and `assign_2_2.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

3 Week Three - Conditional Logic and IF & CASE statements

3.1 Conditional Logic

3.1.1 Relational Operators

Relational operations are a concept which will most likely be familiar to you, whether you are aware of their formal definitions or not. Comparison between two values, whether one value is greater or less than another or similar, is effectively the manner in which relational operators work.

Relational operators, and their form in Fortran, are as follows:

`>`	`.GT.`	Greater than
`>=`	`.GE.`	Greater than or Equal to
`==`	`.EQ.`	Equal to
`/=`	`.NE.`	Not Equal to
`<`	`.LT.`	Less than
`<=`	`.LE.`	Less than or Equal to

These may seem fairly obvious, but there are certain ways in which they must be implemented to work correctly. For instance, if you are performing a check to see if one value is the same as another value, you must use the double equal '`==`' or '`.EQ.`', not the single equals. In Fortran, double equal is the relational check, while single equal is always an assignment, i.e. setting one value or variable to something else. Confusing these can cause issues, so be careful.

Just as all of the mathematical operators 'return' another value in one of the numerical forms (`REAL`/`INTEGER`/etc), the relational operators return a '`LOGICAL`' value. A `LOGICAL` type variable can have two values only, true or false, notated as `.TRUE.` and `.FALSE.` within Fortran. Note that the term 'return' is the proper term for whenever a piece of code 'gives back' some information to the main code or other code. So, to use a few examples to illustrate operators in general:

```
16 / (4 * 2)
```

In the above example, we have two sets of mathematical operators. In the manner of the standard algebraic rules for how equations are evaluated, we start at the 'most nested' brackets, in this case `4`, the operator for multiplication, and `2`. We can consider the statement within these brackets as being entirely separate, it will be evaluated as per the rules for the operator, and the value of `8` will be returned by that operator. The rest of the line continues to be evaluated. Having already returned the value of `8`, the next operator doesn't see either of the original two bracketed numbers, as far as it is concerned, the bracketed portion has now become the value `8`. The final division evaluation is performed, and in this case returns `2`. This new result could be used in whatever context it is in independently, it may be that it is assigned at the value for a variable, or that it is part of a much larger equation.

In the same way, relational operators are evaluated, and return a `LOGICAL` value, for example:

```
4 < 2
```

Will return '`.FALSE.`'.

```
3 .NE. 6
```

Will return '`.TRUE.`', the '`.NE.`' means 'not equal to', since `3` isn't equal to `6`, the 'answer' is true.

Relational operations are most useful when used with one or more variables, hence, you may have a variable `A` which is an age and you want to check that it is not negative. The following statement would return a `.TRUE.` or a `.FALSE.` depending on whether the value is greater than or equal to zero or not:

```
A >= 0
```

However, this on its own does not act as an actual check, it will give you a `.TRUE.` or a `.FALSE.` for your imposed condition, but on its own it won't do anything dependent on which it finds. These relational operations must be coupled with `IF` or `CASE` statements in order to achieve this.

3.1.2 Logical Operators

In the context of Fortran, logical operations are operations performed on `LOGICAL` variables, which also give a `LOGICAL` value as the output. As with relational operators, their use is probably not entirely unfamiliar, but must be used in a specific way to obtain the intended results. The logical operations can be summarised as follows:

`.NOT.`	Is not
`.AND.`	And
`.OR.`	Or
`.EQV.`	Equivalent (effectively like 'equals')
`.NEQV.`	Not Equivalent

Logical operators tend to be more difficult to get the hang of than relational operators. You are exposed to relational operations through the use of inequalities in maths. Some of you may not have come across logical operations at all before. However, these operations are fundamental to how computers work. The binary basis of computers, zero and one, is effectively identical to the logical basis for these operators, true and false. At the most base level, the *only* things that a computer actually does, are complex combinations of these logical operations. It is from these simple rules that logic based computational systems can be built up to levels capable of the functions performed by modern day machines. This makes logical operations not just useful from a programming point of view, but essential to understand from a computational science point of view, to appreciate what computers actually do in order to operate.

We will go through each of these operators, displaying their 'truth tables', illustrations of the output for each possible set of inputs.

Firstly, we will start with the '`.NOT.`' statement. Unlike the other operators, the `.NOT.` operation is 'unary', this means it acts on only a single variable. `.NOT.` inverts whatever truth value the 'operand' (variable/value being operated on) has, so, for a statement '`.NOT. A`', we have the following possible results:

A	Result
`.FALSE.`	`.TRUE.`
`.TRUE.`	`.FALSE.`

The remaining operators are all 'binary', in that they act between two variables (in the same way as many of the mathematical/arithmetical operators). Starting with `.AND.` and the statement `A .AND. B` we have the following:

A		B	Result
`.FALSE.`	`.AND.`	`.FALSE.`	`.FALSE.`
`.FALSE.`	`.AND.`	`.TRUE.`	`.FALSE.`
`.TRUE.`	`.AND.`	`.FALSE.`	`.FALSE.`
`.TRUE.`	`.AND.`	`.TRUE.`	`.TRUE.`

You can see how the `.AND.` operator only returns a `.TRUE.` if both of its inputs are `.TRUE.` as well. The truth table for the statement `A .OR. B` is as follows:

A		B	Result
`.FALSE.`	`.OR.`	`.FALSE.`	`.FALSE.`
`.FALSE.`	`.OR.`	`.TRUE.`	`.TRUE.`
`.TRUE.`	`.OR.`	`.FALSE.`	`.TRUE.`
`.TRUE.`	`.OR.`	`.TRUE.`	`.TRUE.`

In this case, either condition being `.TRUE.` will trigger the output to be `.TRUE.`. Next, the statement `A .EQV. B`:

A		B	Result
`.FALSE.`	`.EQV.`	`.FALSE.`	`.TRUE.`
`.FALSE.`	`.EQV.`	`.TRUE.`	`.FALSE.`
`.TRUE.`	`.EQV.`	`.FALSE.`	`.FALSE.`
`.TRUE.`	`.EQV.`	`.TRUE.`	`.TRUE.`

This triggers a `.TRUE.` result by the variables simply being the same, whether `.TRUE.` or `.FALSE.`. The final example, `.NEQV.` is simply a variation on this, but the results inverted as if passed through a `.NOT.` operator. For the statement `A .NEQV. B` we get a truth table of:

A		B	Result
`.FALSE.`	`.NEQV.`	`.FALSE.`	`.FALSE.`
`.FALSE.`	`.NEQV.`	`.TRUE.`	`.TRUE.`
`.TRUE.`	`.NEQV.`	`.FALSE.`	`.TRUE.`
`.TRUE.`	`.NEQV.`	`.TRUE.`	`.FALSE.`

This is clearly the inverse of the `.EQV.` statement. As such, it is important to note that the following two statements are exactly equivalent: `(A .NEQV. B)` and `.NOT. (A .EQV. B)`. As with the arithmetic operators for doing maths, these logical operators can be combined in the same way, with more deeply 'nested' or 'bracketed' statements evaluated first.

3.1.3 Linear Combination of Statements

This can be made particularly clear when we start incorporating multiple variables and combinations of statements, in order to make more complex choices; additionally, we can begin combining these logical operations with the previously discussed relational operations, in order to produce useful results. As a first example, we can show that the `.EQV.` operator itself can be constructed from the fundamental logical operators, `.NOT.`, `.AND.` and `.OR.`. If we look at the truth tables for each of these operators, we can see that three of the results for the `.EQV.` statement is already fulfilled by the `.AND.` operator, when both inputs are different, and when both are `.TRUE.`. So, we can start with `A .AND. B`, but we also need an additional criteria, in order to make the case where `A` and `B` are `.FALSE.` give the result of `.TRUE.`. What can do this is if we invert both inputs before they go into the `.AND.` operator, ie `((.NOT. A) .AND. (.NOT. B))`, for the mixed inputs, where `A` and `B` are different, the results will be the same. However, for the case where both are either `.TRUE.` or `.FALSE.` the results have been swapped around.

Now we have each of the different outcomes, we need to combine them. We can use the `.OR.` operator to do this, as it will give a result of `.TRUE.` if either of our two sub-criteria is fulfilled. This leaves us with the following:
`(A .AND. B) .OR. ((.NOT. A) .AND. (.NOT. B))`
It's not very tidy to add into code, which is why the `.EQV.` operator exists, as a shorthand for this combination. However, this statement will act exactly the same as for the `.EQV.` operation. Try going through it yourself, setting each of `A` and `B` to the four combinations of `.TRUE.` and `.FALSE.` and following through what the output will be.

3.2 The IF Construct

3.2.1 IF, ELSE, END IF

As mentioned in the previous sections, a conditional statement alone will not let you make any actual choices based on the result. To do this you need to make use of other statements. One of these statement is the `IF` statement. This is as it sounds, '`IF`' a certain condition is `.TRUE.` it can execute a particular set of commands, and if not, it can perform a different set of commands. The basic form of an `IF` statement has the following form:

```
IF (condition) THEN
  !result 1 code
ELSE
  !result 2 code
END IF
```

In this example, '`condition`' is whatever condition you wish to check for, constructed from the techniques learnt in the previous section. Alternatively this could simply be a `LOGICAL` type variable, rather than a condition itself. As described earlier, there is no difference between a conditional statement that returns a `LOGICAL` variable, and a pre-existing `LOGICAL` variable. If this condition or variable is found to be `.TRUE.` the code section indicated by the `!result 1 code` will be run, if the condition is `.FALSE.`, it runs the code in the `!result 2 code`.

So, considering one of our criteria mentioned in the section on input validation (Section 2.4), consider the case where we are reading in age values, and must ensure that all are greater than zero.

```
1  PROGRAM age_validation_a
2
3    IMPLICIT NONE
4
5    REAL :: age
6
7    OPEN(1,FILE='age.dat')
8
9    READ(1,*) age
10
11   IF (age > 0) THEN
12     WRITE(*,*) 'Age value is valid, continuing'
13   ELSE
14     WRITE(*,*) 'Age value invalid, program exiting'
15     STOP
16   END IF
17
18   CLOSE(1)
19
20 END PROGRAM age_validation_a
```

A fair portion of the program should hopefully look familiar to you now. We begin with stating the program, using `IMPLICIT NONE`, setting up our variable and reading in a single value from a file we've opened, called '`age.dat`'; all on lines

1 to 9.

Following this, we have our validation check. First we have our `IF` statement itself, on line 11. The relational check is for the `age` variable to be greater than (and not including) zero. Note the brackets around the relational check, these are mandatory, the `IF` statement will not work unless the condition contents are enclosed in brackets. If this condition is `.TRUE.`, the code on line 12 is run, using the `WRITE` command to print a message to screen stating that the value is fine.

Then we have our 'ELSE' statement on line 13, which states that the `IF .TRUE. THEN` section of the code finishes, and the alternative code is about to begin. This alternative code is on lines 14 and 15. Line 14 uses a `WRITE` statement to inform the user that the value isn't valid, and we observe a new command on line 15, the 'STOP command. This command causes the program to exit at this point if it is called, regardless of what code follows it. This means that having the `ELSE` contents of the following would *not* work:

```
STOP
WRITE(*,*) 'Age value invalid, program exiting'
```

As the `STOP` would run before the `WRITE` causing the program to exit before the `WRITE` statement can be processed.

Note that, in general, you should avoid using 'STOP' to end a program if it is otherwise possible to have it finish in the normal way. Exits before the natural conclusion of the program are mostly reserved for conditions under which a specific error or problem has occured which would prevent the code from running correctly any further.

We end our validation block of code with an `END IF` statement on line 16. It is vital to include this line, as it lets the program know that the code comprising any of the rest of the `IF` statement has now ended.

As already mentioned, the `IF` statement has no preference for whether the condition being checked is a whole relational statement, or simply a `LOGICAL` variable, as, after the evaluation of the relational statement, they are both entirely equivalent. As such, the following version of the above code is equally valid:

```
1  PROGRAM age_validation_b
2
3    IMPLICIT NONE
4
5    REAL :: age
6
7    LOGICAL :: valid
8
9    OPEN(1,FILE='age.dat')
10
11   READ(1,*) age
12
13   valid = (age > 0)
14
15   IF (valid) THEN
16     WRITE(*,*) 'Age value is valid, continuing'
17   ELSE
18     WRITE(*,*) 'Age value invalid, program exiting'
19     STOP
20   END IF
21
22   CLOSE(1)
23
24 END PROGRAM age_validation_b
```

Note how, on line 15, despite the fact that it is only a single variable it is still enclosed in brackets, this is always required. While this may seem to unnecessarily lengthen the code, for complex conditions, it can be advantageous to split them out into multiple variable comprising the check.

For example, let's consider the case where we are checking two variables simultaneously, firstly age again, but also height. Using the first method of conditional statements within the `IF` statement line, we obtain the following code. Note also that we will check for a negative, in other words, the first `IF` block is run only if the numbers *are not* valid, and we can omit any 'ELSE' statement. However, the code for the check still ends with `END IF`.

```
1  PROGRAM age_height_validation_a
2
3    IMPLICIT NONE
4
5    REAL :: age, height
6
7    OPEN(1,FILE='age.dat')
8
9    READ(1,*) age, height
10
```

```
11    IF ((age <= 0) .OR. (height <= 0)) THEN
12      WRITE(*,*) 'A variable is invalid, exiting program'
13      STOP
14    END IF
15
16    CLOSE(1)
17
18  END PROGRAM age_height_validation_a
```

In this case we are looking for if each of the numbers are less than or equal to zero. Then we combine those two relational statement with the logical operator `.OR.`, which will give a `.TRUE.` if either of the relational statements are `.TRUE.`. If this is the case, the code on lines 12 and 13 are executed. If the condition is `.FALSE.`, then the program skips along to wherever it finds an `ELSE` or `END IF` and continues from there.

The alternate way of writing this may be as follows:

```
1   PROGRAM age_height_validation_b
2
3     IMPLICIT NONE
4
5     REAL :: age, height
6
7     LOGICAL :: ageinvalid, heightinvalid
8
9     OPEN(1,FILE='age.dat')
10
11    READ(1,*) age, height
12
13    ageinvalid = (age <= 0)
14
15    heightinvalid = (height <= 0)
16
17    IF ((ageinvalid) .OR. (heightinvalid)) THEN
18      WRITE(*,*) 'A variable is invalid, exiting program'
19      STOP
20    END IF
21
22    CLOSE(1)
23
24  END PROGRAM age_height_validation_b
```

This performs exactly the same as the previous version, but the checks have been separated out. Some consider this form easier to read for complex conditional statements. In truth, this example is still fairly simple, and the addition of two further variables and lines may outweigh any improvement in readability. However, if you have conditions involving dozens of variables, this system may become preferable.

3.2.2 ELSE IF statement

There may be cases where there are multiple possible choices. For example, if you have a program which reads in assignment marks, and want the program to respond differently to different mark ranges. In this case we can use the `ELSE IF` statement, like the `ELSE` statement, it is performed if the previous check/checks has/have failed. However, unlike `ELSE`, which is a catch-all for any other possibility, `ELSE IF` also lets you impose a further criteria. As an example, we have a program designed to receive a mark value, and how many marks the assignment was out of from the command line, checks that the value is between 0 and and the maximum, and then gives a message dependent on what percentage mark it corresponds to. *Note:* This is just an example! The classmark values (3rd/1st/etc) are only representative! They do *not* necessary correspond to an actual classmark if you put any actual assignment in as the input!

```
1   PROGRAM marking_example
2
3     IMPLICIT NONE
4
5     REAL :: mark, percent, maxmark
6
7     WRITE(*,*) 'Please input mark:'
8     READ(*,*) mark
9     WRITE(*,*) 'Please input how many marks the assignment was out of:'
10    READ(*,*) maxmark
11
```

```
12    IF ((mark > maxmark) .OR. (mark < 0)) THEN
13      WRITE(*,*) 'Mark is not in valid range, must be between 0 and maximum, exiting'
14      STOP
15    END IF
16
17    percent = (mark / maxmark) * 100.0
18
19    IF (percent < 40) THEN
20      WRITE(*,*) 'Failed'
21    ELSE IF (percent < 50) THEN
22      WRITE(*,*) '3rd'
23    ELSE IF (percent < 60) THEN
24      WRITE(*,*) '2-2'
25    ELSE IF (percent < 70) THEN
26      WRITE(*,*) '2-1'
27    ELSE
28      WRITE(*,*) '1st'
29    END IF
30
31  END PROGRAM marking_example
```

Lines 7 to 10 prompt and then read in the mark from the user and what the assignment was marked out of. On line 12 we then have a check to ensure that the mark entered lies between 0 and the maximum mark received from the user (inclusive); outputting an error message and exiting if not.

Following this, we use a short mathematical statement on line 17 to calculate the percentage based on the mark and maximum mark.

We then get to the set of conditions implemented into an `IF - ELSE IF - ELSE - END IF` construct. We first check for the lowest mark range on line 19, with our condition asking whether the mark is less than 40%. If this is the case, the program uses a `WRITE` statement to output that the entered mark corresponds to a 'fail'. It would then jump straight to the `END IF` on line 29. Whenever an `IF` or `ELSE IF` criteria is found to be `.TRUE.` it skips any other `ELSE IF` or `ELSE` checks, having found the 'correct' answer.

Line 21 is our first `ELSE IF` line. Note that the form is '`ELSE IF (condition) THEN`'. Also note that for this next grade boundary, 40% to 50%, we are only checking for whether it is below 50%. This is because we implicitly know the mark must be above 40%, otherwise it would have fulfilled the original `IF` criteria and this piece of code would not have been called. As before, if the mark is in this range, it outputs a message informing of the grade.

The next two checks, on lines 23 and 25 work in the same way, assuming the lower bound to be whatever the previous check failed to find. Finally, once we have checked for all ranges below 70%, we know the mark must be above 70% and hence a first. This means we don't need another actual check, and can simply use an `ELSE` statement.

Note: Try copying and compiling this code, then running with various different input values, to observe how it responds. While it is not an assignment, you may wish to take a mental note of anything you notice, such as remaining issues where the code still fails, criteria which are unaccounted for, etc. and think about how one might go about finding a solution to these problems, or even ways in which the code could be expanded. However, prioritise the assignment work ahead of this.

Note: To reiterate the note from before the code: This shouldn't be used as a guide for actual marks! The classmark boundaries in this example are only representative!

3.2.3 Nesting of IF Statements

'Nesting' is the term used to describe the placement of a construct such as that of the `IF - ELSE - END IF` within another construct. This can simplify complex coding situations at the expense of taking up more lines. For example, consider the following generic code segment:

```
IF (condition) THEN
  !result 1 code
ELSE
  !result 2 code
  IF (othercondition) THEN
    !other code
  END IF
END IF
```

An additional decision is made within the `ELSE` code, based on '`othercondition`'. The important consideration with nesting is that it is not used excessively, nor excessive *avoidance* of nesting. Generally speaking, any particular `IF` construct

should be related to a single, or set of directly linked, decisions being made.

For example, consider the `ELSE IF` statement code in the previous subsection. The `IF - ELSE IF - ELSE - END IF` all relates to the selection based on percent. Were there to be another condition based on something else, such as output choices or similar, then it could be in a nested `IF` choice, as the alternative would be a much larger set of `IF - ELSE IF`s with every permutation of the percent selection and the other selection.

Likewise, while it would be possible to use nested `IF`s to check for related things, or few criteria, it is frequently neater to simply combine the related conditions with logical operators. For instance:

```
IF (condition) THEN
  IF (othercondition) THEN
    !code
  END IF
END IF
```

Is likely much more simply expressed as:

```
IF (condition .AND. othercondition) THEN
  !code
END IF
```

Since they have the same effect, but the latter is neater, takes up less space and is easier to follow.

While some instances can quite clearly be seen to be very much simpler expressed as one way or the other, frequently it is a matter of personal judgement as to what would be more suitable.

3.2.4 Discriminant Check - Assignment Material

In the first week (Section 1) we coded a simple quadratic equation solver, by implementing an input of three coefficients, and producing solutions using the quadratic formula. In Week 2 (Section 2) we expanded this by implementing user input to the program, to avoid needing to recompile the code each time the input was changed.

You should also recall that, for some combinations of input variables, the values of x_{plus} (the positive square root solution) and x_{minus} (the negative square root solution) were either the same, or the program produced useless output because there were no solutions to x.

A way to extend the program further is to add several `IF` conditions to check the discriminant of the quadratic formula:

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

Where the discriminant is the $b^2 - 4ac$ term within the square root. When this value is positive, there is a real solution to the square root and there are two solutions to x. When the discriminant is zero, there is only one distinct solution. When it is negative, there are no real solutions for x.

Therefore, for this assignment, you are required to create a copy of the *second* part of last weeks assignments, the program `assign_2_2` (the one which reads user input interactively *and* outputs the result to a file). Create a copy of `assign_2_2.f90` called `assign_3_1.f90`. Within `assign_3_1.f90`, rename the target output file to `assign_3_1.out`, so that it will not overwrite your old output file.

You must then implement the following conditions after the discriminant part of the quadratic formula has been calculated:

- If the discriminant is greater than (and *not* equal to) zero:
  1. Use `WRITE` to inform the user that there are two solutions.
  2. Calculate both x_{plus} and x_{minus} using the quadratic formula as previously implemented.
  3. Write both values to the screen.
  4. Write both values to your output file.
- If the discriminant is equal to zero:
  1. Use `WRITE` to inform the user that there is one distinct solution.
  2. Use one of the variables `x_plus` or `x_minus` (it doesn't matter which), to store the result of the single answer to the quadratic formula.
     Note: In this case, you may want to leave out the discriminant from the formula itself, since you already know it equals zero.

3. Write the value to the screen.
4. Write the value to your output file.

- If the discriminant is less than (and *not* equal to) zero:
  1. Use `WRITE` to inform the user that there are no solutions based on the coefficients entered.
  2. Write `'No real solutions'` to your output file.

These changes should hopefully result in a program which accounts for all normal user input (ie, other than non-numerical input which will fail to read in). Try various different values for yourself to examine whether your implementation is correct.

3.3 The SELECT CASE Construct

3.3.1 Using SELECT CASE

The `CASE` construct exists as an alternative to the `IF - ELSE IF - ELSE - END IF` construct. They take the form of the following:

```
SELECT CASE (selection_variable)
  CASE (selection_criteria_1)
    !result 1 code
  CASE (selection_criteria_2)
    !result 2 code
  CASE (selection_criteria_3)
    !result 3 code
  CASE DEFAULT
    !default result code
END SELECT
```

Just as with the `IF` statements, the case statement is designed to perform a specific piece of code depending on whether a defined condition is met. `SELECT CASE` differs from `IF` as it is more designed for matching variables to values than complex conditional statements. In the above example, some variable is used as argument for `SELECT CASE`, `selection_variable` here. For each `CASE`, the value of the variable is compared against the `CASE`s selection criteria, running the associated resulting code if a match is found, and moving on to check the next `CASE` if not.

For instance, consider the following example. Note that this example uses strings in the variable and for the criteria, while this is perfectly valid, ensuring the read-in of strings works correctly without undesired leading or trailing 'space' characters has not yet been covered. See the section on string handling if you want more information (Section 8).

```
SELECT CASE (degree_course)
  CASE ('Physics')
    noun = 'Physicist'
  CASE ('Maths')
    noun = 'Mathematician'
  CASE ('Comp Sci')
    noun = 'Computer Scientist'
  CASE ('Biosciences')
    noun = 'Biologist'
  CASE DEFAULT
    noun = 'Unknown'
    WRITE(*,*) 'Note: Degree Course name not identified'
END SELECT
```

In this example, the string variable '`degree_course`' contains the name of the degree course which the student is on. The intention is to determine the correct noun to use to describe a student on that course. The `SELECT CASE` line at the beginning specifies that the variable which will be selected with is `degree_course`, this is then compared to the strings in brackets following each of the `CASE` lines. The program will go through the `CASE`s in sequence; if there is a match, it will set the '`noun`' variable with the relevant string, and skip to the `END SELECT` line. For each `CASE` that doesn't match, it simply moves on to the next `CASE` until it reaches the `CASE DEFAULT` line, the lines following which are the code to run if none of the previous cases have matched, whatever the `degree_course` variable may be.

This works with numerical values as well. However, only `INTEGER` values may be used; `REAL` values cannot be used as the selection variable. Additionally, it can express ranges of values for the selection criteria using a shorthand syntax permitting specifying of upper or lower bounds and similar in a few characters. For example:
`CASE (1:)`
Indicates that the `CASE` criteria can be fulfilled with any value of `1` and above. The range specifiers are as follows:

`value`	Matches single value
`:value`	Matches up to and including `value`
`value1:value2`	Matches between values 1 and 2
`value:`	Matches `value` and greater

The selection criteria for `CASE` statements may also be lists. A comma separate set of values/strings, all of which will trigger the same result if matched. For example, extending the previous code returning the noun for a degree course, we can do the following:

```
SELECT CASE (degree_course)
  CASE ('Physics','physics','Phys','phys')
    noun = 'Physicist'
  CASE ('Maths','maths','Mathematics','mathematics')
    noun = 'Mathematician'
  CASE ('Comp Sci','comp sci','Computer Science','computer science')
    noun = 'Computer Scientist'
  CASE ('Biosciences','biosciences','Biology','biology')
    noun = 'Biologist'
  CASE DEFAULT
    noun = 'Unknown'
    WRITE(*,*) 'Note: Degree Course name not identified'
END SELECT
```

This version now returns '`Physicist`' for any of the strings on the `CASE` line for the various physics synonyms.

3.3.2 Purpose of SELECT CASE

The `SELECT CASE` construct is far less commonly used than that of `IF`. This is primarily due to the fact that `IF` is slightly more generic, handling instances for conditional operators regardless of what the condition relies on: strings, `INTEGER`s, `REAL`s and so on. `IF` statements are best for any criteria which is mathematical based, or generally involving ranges and combination of variables.

`SELECT CASE` is suited to instances where specific matches between a particular variable and comparatively few possible values it may have. For example, consider the following two codes; first, one based on `SELECT CASE` and then one based on `IF`:

```
1  PROGRAM yn_choice_case
2
3    IMPLICIT NONE
4
5    CHARACTER*32 :: choice
6
7    WRITE(*,*) 'Continue? y/n'
8    READ(*,*) choice
9
10   SELECT CASE (choice)
11     CASE ('Y','y','Yes','yes','YES')
12       WRITE(*,*) 'Continuing'
13     CASE ('N','n','No','no','NO')
14       WRITE(*,*) 'Cancelling'
15       STOP
16     CASE DEFAULT
17       WRITE(*,*) 'Did not understand input, please try again'
18       STOP
19   END SELECT
20
21 END PROGRAM yn_choice_case
```

```
1  PROGRAM yn_choice_if
2
3    IMPLICIT NONE
4
5    CHARACTER*32 :: choice
6     LOGICAL :: logicy, logicn
7
8    WRITE(*,*) 'Continue? y/n'
9    READ(*,*) choice
```

```
10
11    logicy = (choice == 'Y') .OR. (choice == 'y') .OR. (choice == 'Yes') .OR. (choice == 'yes'
        ) .OR. (choice == 'YES')
12    logicn = (choice == 'N') .OR. (choice == 'n') .OR. (choice == 'No') .OR. (choice == 'no')
        .OR. (choice == 'NO')
13
14    IF (logicy) THEN
15      WRITE(*,*) 'Continuing'
16    ELSE IF (logicn) THEN
17      WRITE(*,*) 'Cancelling'
18      STOP
19    ELSE
20      WRITE(*,*) 'Did not understand input, please try again'
21      STOP
22    END IF
23
24  END PROGRAM yn_choice_if
```

Both of the above codes are valid, but it is clear that the first version, using `SELECT CASE` is less long-winded and more direct. It is also more convenient as, to add or remove possible matches for `SELECT CASE`, one only needs to add or remove strings from the list. As opposed to the `IF` version, where whole relational statements and `.OR.` logic must be added or removed.

However, `IF` still retains advantage for choices involving numerical comparisons, the combination of multiple variables, logical checks, or simple single statement checks; to name some.

3.3.3 Output Mode and Check - Assignment Material

Currently, the program that you have been working on, `assign_3_1.f90`, will automatically output both to the screen, and to the file specified in the source code. A further addition is to ask for user input as to the manner in which the output is desired. For now, we will not attempt to handle user input of filenames, as this involves use of the `CHARACTER` type and strings. However, we can set up the program to support the following output options:

1. Output to Screen Only
2. Output to Screen and File in Overwrite Mode
3. Output to Screen and File in Append Mode

This will mean that we can get the program to output a set of calculations to the file sequentially, to be checked later, however deciding to do this at run-time rather than before compiling. It is this mode, after its implementation, that we will use to obtain the main output for this assignment, a file with a set of outputs for several different coefficient inputs.

As with before, you must first create a new copy of your code to contain the changes for this new version. You should copy your previous code, `assign_3_1.f90` to a new file, `assign_3_2.f90`.

In order to implement this feature, we will create a `SELECT CASE` statement. However, first, we need an additional variable. Create an `INTEGER` with a name something like '`choice`' or similar. We will use a value stored in here to represent which mode of operation we want the program to run in. Therefore, we need a piece of code to prompt the user, and then read in their decision to the `choice` variable (or whatever your own name for it is). You can place this before or after you read in the coefficients. You should ask the user to input a number: '`1`' for the mode only outputting to screen; '`2`' for the mode outputting to screen and overwriting the contents of the file; and '`3`' for the mode outputting to screen and appending to the existing file.

Using this information, we now know whether to, or in which mode, a file needs to be opened. Add a `SELECT CASE` with your `choice` variable as the selection variable. The `CASE`s it should choose between are the three number codes already mentioned. Each should use a `WRITE` statement to inform the user that it has identified that mode as being the one it is running with. Additionally, the `OPEN` statement for the output file should be in modes 2 and 3. However, mode 3 should also have the option '`POSITION='APPEND'`' within its `OPEN` statement, as described in Section 2.2.4.

The final aspect to add is a `CASE DEFAULT` informing the user that an unidentified option has been input, and then close the program using the '`STOP`' command.

By the end of the `SELECT CASE` construct (which you must remember to finish with '`END SELECT`') your program should have confirmed the selected option to the user, opened a file in the correct access mode if necessary or quit if an invalid option was entered.

Now that we do not necessarily have a file open, we need to add a condition within our output section, whereby, an output to file is only attempted if the `choice` variable is in mode `2` or `3`.

You can do this in two ways, either adding an `IF` statement at the point that the results are output, checking that `choice` is greater than `1` (which covers both modes `2` and `3`). Alternatively, you could use an `IF - ELSE IF` construct to check for modes `2` and `3` independently. This adds the ability to modify the output dependent on mode; for example, you could have an additional `WRITE` statement before the result output, which adds a short line of '`---`' symbols in mode 3, in order to separate the newly appended result from the previous one.

The extent to which the output is customised to be more useful or multifunctional is up to you, consult the Assignments Summary section (Section 3.4) for the minimum specification of your program.

3.4 Assignments Summary

3.4.1 Programming Assignment 1

1. Create a copy of your '`assign_2_2.f90`' code called '`assign_3_1.f90`'.
2. Rename the filename of your `OPEN` statement for the output file to '`assign_3_1.out`'
3. After the calculation of the discriminant, add an if statement with three criteria:
   (a) If the discriminant is greater than zero, the code must calculate both values for x as before, then output the values to the screen and to the file.
   (b) If the discriminant is zero, the code must calculate the single value for x, then output the value to the screen and to the file as well as informing the user through a message that there is only one solution.
   (c) If the discriminant is less than zero, the code must output a message to the screen, and to the file, that there are no real solutions for x.

```
IF (condition) THEN
  !result 1 code
ELSE
  !result 2 code
END IF
```

4. After ensuring that your code is working as expected, run with the following values of the coefficients: `2.0`, `-12.0` and `18.0` for a, b and c respectively.
5. Make sure to keep your output file `assign_3_1.out`, as you will be required to submit a printout of it, as well as your source code file, `assign_3_1.f90`.

3.4.2 Programming Assignment 2

1. Create a copy of your '`assign_3_1.f90`' code called '`assign_3_2.f90`'.
2. Rename the filename of your `OPEN` statement for the output file to '`assign_3_2.out`'
3. Define a new variable called something like '`choice`' or '`mode`', add a `WRITE` and `READ` statement to prompt the user for an operation mode: 1, 2 or 3; as well as informing them what they do; then read in their input.
4. Replace your `OPEN` statement, creating a `SELECT CASE` construct. This should execute different code dependent on each of the following `CASE`s:
   (a) If in mode 1, the program should simply inform the user that it is operating in that mode (This is the mode which will just output to the screen, not a file).
   (b) If in mode 2, the program should also inform of this mode, and then also have the same `OPEN` statement as was used previously.
   (c) If in mode 3, the program should also inform of this mode, and then also have the `OPEN` statement as was used previously, with the addition of '`POSITION='APPEND'` within its options.'
   (d) In the `CASE DEFAULT` the program should inform the user that an invalid value for `choice`/`mode` has been entered, and then use the `STOP` command to quit.
5. Replace each of the `WRITE` to file statements within the `IF` construct from the previous assignment (Section 3.4.1) with an if statement which only performs the output to file if the mode of operation is 2 or 3, not mode 1.
   Hint: Try 'nesting' the `IF` statement for this operation within the code sections for your existing `IF` statements.
6. After checking that your program works as expected perform the following output operations:

(a) Run in mode 2 (overwrite old file with new result), with the values `4.0`, `8.0` and `4.0` as the coefficients a, b and c respectively.

(b) Now in mode 3 (append results to existing file), run with the values `15.0`, `-8.0` and `12.0` as the coefficients a, b and c respectively.

(c) Again in mode 3, run with the values `5.0`, `6.0` and `1.0` as the coefficients a, b and c respectively.

7. Check that your output file has all three sets of results from the previous operations (file `assign_3_2.out`). Make sure to keep this output file in its current state, as a printout of it is required, along with a printout of your source code file, `assign_3_2.f90`.

3.4.3 Questions

For this week the questions are as follows:

1. What are the six relational operators?
2. What variable type is returned by a relational operation?
3. What would the result of the following relational operations be?:
   (a) `(5 > 6)`
   (b) `(7 < 2)`
   (c) `(4 /= 11)`
   (d) `(5 .NE. 5)`
4. What are the five logical operations?
5. What would be the result of the following logical operations?:
   (a) `(.TRUE. .AND. .FALSE.)`
   (b) `(.TRUE. .OR. .FALSE.)`
   (c) `(.FALSE. .AND. .FALSE.)`
   (d) `(.NOT. (.TRUE. .OR. .FALSE.))`
6. Under what circumstances are the commands within an `ELSE` statement carried out?

3.4.4 Submission

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_3_1.f90`, `assign_3_1.out`, `assign_3_2.f90` and `assign_3_2.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

4 Week Four - Loops and the DO statement

4.1 The Concept of Loops

Within programming, there are a great many situations where it is required to perform the same, or similar, actions many times. For example, based on the techniques covered so far, consider the following extension to the quadratic equation solver program. You intend to alter your quadratic equation program to read the input from a file, where each line of the file contained three values, one for each of the coefficients, a, b and c. This would mean you could list as many sets of coefficients in the file as you wanted, and it would go through and solve them all in one go, rather than typing each set into the program itself. However, with only the current techniques, this would require repeating the instructions within your code as many times as you have lines of input! This is obviously not ideal or even feasible for instances where you have a task to perform hundreds, thousands or even millions of times.

The solution to this problem is to use '**loops**'. The beginning and ending statements of a loop construct, whatever the exact statement or language is, define the segment of code which is to be repeated. It will generally also define how many times it will be repeated and has some form of variable which is used as a counter, incrementing as the loops proceed.

4.2 The DO construct

4.2.1 The Basics

In Fortran, the main form of loop uses the '`DO`' statement. The form it is used in is as follows:

```
DO counter_variable = start, finish, step
  !loop contents
END DO
```

There are several parts to this construct, lets look at each in turn. First, directly after the `DO` we need an `INTEGER` variable which will act as the loop counter. This is a variable which, for each iteration of the loop, the `DO` statement will set its value to that specified by the loop extents. This is followed by '`= start, finish, step`', this defines the extent and interval which the loop counter variable will be set with.

After the `DO` statement, as much code can be placed within the loop as desired. Finally, the loop is ended with the `END DO` statement.

To better illustrate this, lets look at a full example:

```
1  PROGRAM loop_example
2
3    IMPLICIT NONE
4
5    INTEGER :: counter
6
7    DO counter = 1, 10, 1
8      WRITE(*,*) 'hello', counter
9    END DO
10
11 END PROGRAM loop_example
```

The following code can be entered as a f90 file, compiled and run, if you wish. If you run it, it should print the word '`hello`' to screen ten times, each followed by a number increasing from one to ten. Let's examine why. Firstly, on line 5, we have created our loop counter variable, in this case simply called '`counter`'.

Our `DO` statement is on line 7. We state that the loop variable is going to be our variable '`counter`', then we state the start, end and step that the loop will iterate through. In this example, we are starting at `1`, ending at `10`, and we will move `1` value forward each time.

Within the `DO` construct we have the `WRITE` statement, for each time the loop repeats, this will write the fixed '`hello`' and the value of the variable `counter`. The loop is ended with the `END DO` statement, once the program reaches here, it starts again from the `DO` statement as long as the counter variable is within the specified range.

4.2.2 Using the Counter Variable

While loops can be used to simply repeat a simple task a set number of times, it can also be used to perform more dynamic operations. In these cases, it can be designed such that values of variables and actions change over the course

of the loop. In order to take best advantage of the more intricate uses of loops, it is important to fully understand how the counter variable is working, and how it can be used.

Consider the following example:

```
1  PROGRAM loop_example
2
3    IMPLICIT NONE
4
5    INTEGER :: sum
6
7    INTEGER :: i, start, finish, step
8
9    start = 0
10   finish = 10
11   step = 2
12   sum = 0
13
14   DO i = start, finish, step
15     WRITE(*,*) i
16     sum = sum + i
17     WRITE(*,*) sum
18   END DO
19
20 END PROGRAM loop_example
```

In this example, the extents and step of the loop have been set so as to go through all of the even numbers between 0 and 10. Also note that rather than writing the values on the `DO` line itself, we have written them as variables. Rather than `counter` as our loop counter, we have used the name `i`. `i`, `j` and `k` are extremely commonly used generic loop counter names. Because so many loops make use of the value of the counter within the loop itself, having a simple short name makes it easier to implement. It is also far more convenient when the loop counter is used as the index for an '**array**', a concept which is covered in Section 5.

Within our loop, we do two main things. First, we print the value of the loop counter `i` to the screen. We then use the value of the loop counter in a mathematical operation. In this case, we are sequentially adding the loop counter into the variable `sum`. Following the loop step by step, the exact actions are:

1. First Iteration
   - Set `i` to 0
   - `sum` is currently zero, `sum + i` will still be zero.
2. Second Iteration
   - Set `i` to 2
   - `sum` is currently zero, `sum + i` will be 2.
3. Third Iteration
   - Set `i` to 4
   - `sum` is currently 2, `sum + i` will be 6.
4. Fourth Iteration
   - Set `i` to 6
   - `sum` is currently 6, `sum + i` will be 12.
5. Fifth Iteration
   - Set `i` to 8
   - `sum` is currently 12, `sum + i` will be 20.
6. Sixth Iteration
   - Set `i` to 10
   - `sum` is currently 20, `sum + i` will be 30.

We have used the value of the counter to affect what the outcome of that loop iteration is.
Note: It is important to be aware that you cannot *set* the value of a loop counter within its own loop. In other words, within a loop using the variable `i`, you could not have the line `i = 1` or similar. It is not that the code will fail to work, or loop infinitely, but that it will simply fail to compile at all.

When you have a step with a larger value than 1, as in this example, it will keep looping for however many times necessary to fit *within and including* the final limit. In other words, if the loop above was instead defined with: `DO i = 1, 11, 2` the result would still be the same as before. Once the counter reaches the 10, it cannot increment by 2 without surpassing the 'final' value, and so will stop there. It *does not* 'round' down to 11 to include the final value, *nor* skip above it to 12 and then stop.

The observant among you will have noticed that the process this particular piece of code has performed is identical to a sum of an arithmetic progression (an arithmetic series). You may want to think of how you would code an arithmetic progression itself, or how you would go about creating a code which performed a geometric progression.

4.2.3 Using a Loop to Perform a Useful Action

The following example will use the command line argument read in method described in Section 2.3. Refer to that section if you don't follow the implementation of the use of `GETARG` in this example.
Hint: Note that, to understand the code, you should know that the command `IARGC()` returns an `INTEGER` value which corresponds to the number of arguments which have been passed to the program on the command line. ie. if you ran the program with:
`:> ./summation 1.2 6.3 8.1`
`IARGC()` would be `3`, as there are three items on the command line after the program name. This was briefly mentioned in Section 2.4.2.

Observe the following program, try to work out and understand yourself what you think it does before moving on to the subsequent text explaining its operation.

```
1  PROGRAM summation
2
3    IMPLICIT NONE
4
5    CHARACTER*32 :: arg
6    REAL :: total, value
7    INTEGER :: num_args, i
8
9    total = 0.0
10
11   num_args = IARGC()
12
13   IF (num_args == 0) THEN
14     WRITE(*,*) 'No arguments found'
15     STOP
16   END IF
17
18   DO i = 1, num_args, 1
19     CALL GETARG(i, arg)
20     READ(arg,*) value
21     total = total + value
22   END DO
23
24   WRITE(*,*) total
25
26 END PROGRAM summation
```

Have you worked out what the program does? It is handy to note that this is another clear example illustrating the need for code to be sufficiently commented. Had the previous code contained comments, it would have been instantly obvious what the purpose of it was, and what each part of the code did.

Let's look at the operation of this code step-by-step now. The initial few lines should be obvious by now, the program is started, `IMPLICIT NONE` is stated, several variables are defined and the variable `total` is initialised to zero.

This is followed, on line 11, by '`num_args = IARGC()`'. This sets the `INTEGER` variable `num_args` to the value of the number of arguments present on the command line when the program is run.

The `IF` statement starting on line 13 is an example of input validation of a sort. Since this code functions based on command line arguments given to it, if there are none, it notices, tells the user so, and quits, rather than letting the program potentially crash.

The important part of our code begins on line 18. This `DO` statement is using the `INTEGER` variable '`i`' as its loop counter, starting from the value `1`, and moving along in steps of `1`. The interesting part of this `DO` statement is the use of the variable `num_args` as the upper limit to the loop. This means that the loop will be performed as many times as there are arguments on the command line.

Now we come to the contents of the loop, on lines 19 to 21. First, we use the `CALL GETARG` command to obtain a value from the command line. However, rather than putting in a fixed number for which command line argument to read from, we have used the loop counter. This means that, as the loop counter increases, one by one, the `GETARG` statement will be reading each different argument in turn. The value `GETARG` reads is stored in the `arg CHARACTER` variable as a string. This is a temporary measure. Prior to proper conversion, the argument is only available as its component characters, not an actual value. This is then converted to a `REAL` value using the `READ` line.

Then, we have the mathematical statement '`total = total + value`'. For each command line argument, this will use it's value (in `value`) and add it to whatever the current value of the variable `total` is. Then, we end the loop on line 22.

Finally, on line 22, we write what value the variable `total` has ended up with. The operation which this program performs is to produce a sum of all of the numbers passed to the program by the user on the command line, hence the command:

```
:> ./summation 1.2 6.3 8.1
```

Would return:

```
15.600000
```

4.2.4 99 Bottles of Beer - Assignment Material

You have all likely heard of the '99 Bottles of Beer' song. The first task for this week will be to code a program which will automatically go through all of the lines of the song; however, for the purposes of this assignment, the number will be from 20 down to zero, as we are students and cannot afford 99 bottles. For those that may be unaware of the original song, it goes as follows:
"*99 bottles of beer on the wall, 99 bottles of beer, you take one down, pass it around, 98 bottles of beer on the wall*;
98 bottles of beer on the wall, 98 bottles of beer, you take one down, pass it around, 97 bottles of beer on the wall;"
etc. all the way down to zero. The lyrics to this song, with their nature of many similar lines with only linearly changing numbers, which, additionally, would be a pain to write out in full manually; is ideal for being handled with a `DO` loop.

This will be the first assignment where the majority of an entire program you will write yourself in one go, unlike the previous program which has been built up gradually. As such, make sure to remember all of the basic rules, how programs must begin and end, declarations of all variables, and so on. Additionally, as always, remember to ensure that the layout and formatting conforms as well as possible to the guidelines laid out in the Good Programming Practice section, Section 14.

You should make a new file for the code for this program, called '`assign_4_1.f90`'.

Firstly, you will need to choose and define suitable variables for the program. Don't limit yourself by this though, if you find later that an additional variable would be useful, simply add it then, you needn't try to find a work around to only use the variables you initially chose, just remember to define them!

You will need a loop going down from 20 to 1. Within this you will need to output the lines as shown above, with the correct numbers in place.
Hint: Remember that several variables or strings separated by commas after a `WRITE` statement will be printed sequentially on the same line. For example `WRITE(*,*) 'this is variable a:', a, ' and this is b:', b`, say '`a`' is `4` and `b` is `3.2`, you will get the following: `this is variable a: 4 and this is b: 3.2`
Hint 2: Also worth noting is that the items following a `WRITE` statement needn't be just variables and strings, they can be mathematical operations as well. For example the following is a valid example of a `WRITE` statement: `WRITE(*,*) 'old value =', (value-1)`

Note: You do *not* need to worry about the string formatting of the numbers, they will have extra preceding/trailing spaces, but you do not need to correct for this. If you are interested, see the section on strings and formatting (Section 8).

Try to include conditions for when the number of bottles are 1 and 0. For the former, rather than '1 bottles of beer...', plural, it should be '1 bottle of beer...', singular. In the case of zero, rather than '0 bottles of beer...' it could be 'no bottles of beer...'. This will be worth some marks, however, most important should be making the main loop operate correctly, focus on that before adding these conditions.

Your program should produce two forms of output, one should be writing to the screen; the second should be writing to an output file, as in several of the previous assignments. This output file will need to be opened before your `DO` loop, and closed afterwards. The filename should be '`assign_4_1.out`'.

4.3 Nested DO Loops

The concept of the 'nesting' of constructs has previously been described in Section 3.2.3. It is the practice of putting one construct within the confines of another of the same construct. This is also possible with `DO` loops, for which it provides an extremely powerful tool. For example, any scenario which effectively requires iteration through multiple 'dimensions' or obtaining every permutation for a combined set of parameters. A simple example of this is obtaining every relative coordinate around a cell in a 2-d grid.

4.3.1 Coordinate Listing

Consider the following subsection of a large 2-d grid, with the coordinates labelled.

31					
30					
29			X		
28					
27					
$^{y}/_{x}$	59	60	61	62	63

The cell labelled 'X' in the centre is our cell of interest in this case, at coordinates (61,29). But we want our computer program to return a list of all of the cells which are adjacent to a cell, horizontally, vertically and diagonally; for any arbitrary cell location. What is the best or easiest way of doing this?

One way is to use a nested `DO` loop. By putting one `DO` loop within another, let's say our outer loop is `DO` loop 'A' and our inner loop is `DO` loop 'B'; we will cycle through every value of the 'B' loop, for each value of the 'A' loop. We can demonstrate this with a functioning example:

```
1  PROGRAM coord_example
2
3    IMPLICIT NONE
4
5    INTEGER :: x, y
6
7    INTEGER :: i, j
8
9    WRITE(*,*) 'Please enter central x coord'
10   READ(*,*) x
11   WRITE(*,*) 'Please enter central y coord'
12   READ(*,*) y
13
14   WRITE(*,*) 'Coords are:', x, y
15   WRITE(*,*) 'Adjacents are:'
16   !Loop A
17   DO i = -1, 1, 1
18     !Loop B
19     DO j = -1, 1, 1
20       WRITE(*,*) x+i, y+j
21     END DO
22   END DO
23
24 END PROGRAM coord_example
```

Let's go through what happens step by step:
First the variables for the central cell are read in. Some confirmation output is written, then we reach the 'outer' loop, loop 'A', on line 17. Loop A is required to go from -1 to +1 in steps of 1, therefore will be the numbers: -1, 0, 1. On the first step, this means that `i`, the loop counter, is `-1`. We then move to the 'inner' loop, loop 'B' on line 19. This loop operates between the same limits, therefore, on its first pass, `j`, its counter, will also be `-1`. The `WRITE` statement on line 20, on the first iteration, therefore, will be the coordinate `x - 1` and `y - 1`.

What happens next? Loop B hasn't finished yet, the program cannot return to loop A until it has; it must reach loop B's '`END DO`', which isn't triggered until the loop has gone over its entire range. So, the next iteration is another change to loop B's counter, `j`, which becomes `0`. The `WRITE` statement will subsequently print `x - 1` and `y + 0`.

The next step also only increments loop B, this time moving its counter, `j` to `1`. Which gives a `WRITE` statement of `x - 1` and `y + 1`. Notice that this means it has gone through all of the possible y-axis values for the first x-axis value.

The next step, and loop B has reached the end of it's range, therefore, the counter of loop A has a chance to increment, and does so, setting `i` to `0`. Moving on to a new loop A iteration, however, means that another loop B is triggered. This will mean going through each of the possible y-axis values again.

By the end, we have effectively 'slowly' stepped through each possible x-axis value, and for each of those steps, 'quickly' stepped through all of the possible y-axis values. The end result is summarised in the following table, the output you would get by running this program with the coordinates from the example location X (61,29) are the values in the final two columns:

iteration	`i` (A loop)	`j` (B loop)	x coord	y coord	central x of 61	central y of 29
1	-1	-1	$x-1$	$y-1$	60	28
2	-1	0	$x-1$	$y+0$	60	29
3	-1	1	$x-1$	$y+1$	60	30
4	0	-1	$x+0$	$y-1$	61	28
5	0	0	$x+0$	$y+0$	61	29
6	0	1	$x+0$	$y+1$	61	30
7	1	-1	$x+1$	$y-1$	62	28
8	1	0	$x+1$	$y+0$	62	29
9	1	1	$x+1$	$y+1$	62	30

You will notice that this covers all of the cells in the range ±1 around the coordinates of the central cell. You may also notice that it also includes the central cell itself; this may or may not be desired, how do you suppose it could be removed if only the adjacents and not the central cell are required?

Additionally, what if we wanted to list all cells within the range of ±2 around the central cell, rather than ±1? Would this be difficult? What about in an arbitrary, user-entered, range 'R'?

The previous two questions are not assignment material, but it is recommended to have an attempt at thinking through possible answers, or even adding your own ideas to the code to see if they work. Only do this if you have sufficient time, however. You may want to leave attempting your own implementation of the problem until after having completed the set assignments. The next piece of code is an example of the previous code, expanded to exclude the originally specified central cell from the list of adjacents, and to list all of the cells within a range of ±2 on the x and y axes. The final addition of an arbitrary, user-entered, range is left up to you, should you have time at the end of the week to attempt it. However, note again that it is *not* an assignment.

```
1  PROGRAM coord_example_expanded
2
3    IMPLICIT NONE
4
5    INTEGER :: x, y
6
7    INTEGER :: i, j
8
9    WRITE(*,*) 'Please enter central x coord'
10   READ(*,*) x
11   WRITE(*,*) 'Please enter central y coord'
12   READ(*,*) y
13
14   WRITE(*,*) 'Coords are:', x, y
15   WRITE(*,*) 'Adjacents are:'
16   !Loop A
17   DO i = -2, 2, 1
18     !Loop B
19     DO j = -2, 2, 1
20       IF ( .NOT. ((i .EQ. 0) .AND. (j .EQ. 0))) THEN
21         WRITE(*,*) x+i, y+j
22       END IF
23     END DO
24   END DO
25
26 END PROGRAM coord_example_expanded
```

One additional note on the above code is the logical statement preventing the centre coordinate being display. It is important to know, and to realise why, the statement `( .NOT. ((i .EQ. 0) .AND. (j .EQ. 0)))` is *not* the same as `((i .NE. 0) .AND. (j .NE. 0))` or `((i .NE. 0) .OR. (j .NE. 0))`, as though the '`.NOT.`' had been 'multiplied' into the brackets. These statements produce different results and are not equivalent. If this does not seem apparent, try each, either with a code, or by manually determining the result for a few sample values of `j` and `i`.

4.3.2 Listing Permutations - Assignment Material

The object of this assignment will be to familiarise you with the use of nested `DO` loops for the purpose of generating a list of permutations. This concept was shown in full in the previous example (Section 4.3.1), as such, aside from the requirement specifications, guidance relating to the specific code will be limited.

Let us consider that we have a list of student numbers, and a list of assignment numbers. What we require is a list for each student, of each assignment. This would then be able to act as a table to tick off handed in work, without having to manually write out all combinations.

The name of the program will be `assign_4_2`, as such you will need to create a file '`assign_4_2.f90`'.

Student Number will be `INTEGER` values, and should be entered by the user at the beginning of the program. This should be the 'primary' column of your output, or, in other words, the list should be organised by student, and for each student repeat every assignment number, as opposed to vice-versa. The Assignment Numbers are also `INTEGER`s, and will also be entered by the user. You will also need an `INTEGER` variable to keep track of the total number of successfully outputted permutations to the output file.

After interactively asking for, and reading in, each of the input variables (which you will have to name and declare) from the user, the program must print both to screen to indicate what has been read in. Then, an output file called '`assign_4_2.out`' should be opened, this will be the location which all of the results will be printed from your `DO` loops.

Having opened an output file and obtained both input variables, you must now implement a suitable pair of nested `DO` loops which will write into the file all combinations of student number and assignment number. Additionally, each time a data line is output, you should increment your variable counting the number of output lines. The output should be something like as follows (assuming inputs of 3 for Student Number and 3 for Assignments):

Student No.	Assignment
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

The table *should* include headers such as shown above, however, note that the exact formatting of the headers and values does not matter (ie do not worry about getting columns to line up and similar). The header should *not* count towards the total number of output lines counter, though.

Hint: Since you do not want to repeat the header each iteration, make sure it is outside of the `DO` loops.

Once the `DO` loops have been performed you should then print to screen *and* to the file the value of your output line counter, in the above example, it would have been '9', indicating the 9 output permutations. Having generated the output, you should make sure to close your output file.

Once you have ensured that your program operates as expected, you should run it with the following criteria, generating an output file you will be required to submit (so make sure it is not overwritten/deleted before submitting it). Your criteria should be: 7 for the number of students and 4 for the number of assignments.

This should give you a total number of permutations of 30, and an output with 30 lines (32 including the header and number of permutations line).

Hint: Remember, if you encounter problems with your code, try working through it line-by-line, as though you are reading what the actual program will be doing. Keep track of the values of the variables as you do this, and you may be able to tell where an increment is wrong, or a mathematical statement incorrect. This line-by-line debugging process, though long winded, can be extremely effective for short codes such as this.

4.3.3 General Use

In general, the nesting of all kinds of constructs becomes a necessity with complex codes. It is possible to have `DO` loops nested which are not both varying essentially the same parameter, as in the previous example. They can be entirely unrelated parameters, which are simply suited, or necessary, to be performed in a nested fashion. For instance, in many simulations, it is necessary to use to `DO` loop to move through 'time steps'. Computers require all changes in properties to be discrete, possessing specific intervals, rather than a continuous change. As such, it is necessary for simulations which

evolve dynamically with time to go through computational steps at each time interval, these can be performed as steps through a DO loop or similar.

Within each time step, it may be necessary to process any number of quantities, the calculations of which may require their own do loops. Hence, the DO loops could be said to be nested, without being directly linked to a single set of parameters. This basically means that DO loop nesting can act together to iterate through permutations of a single set of parameters, or simply to repeat calculations for different sets of parameters a certain number of times.

For some DO loops, it may be necessary to simply continue to iterate *until* a particular occurrence, rather than a set number of times. This is covered in the next subsection.

4.4 Arbitrary length DO Loops

4.4.1 DO WHILE loops

One variant of the normal DO loop format is to let the DO loop potentially continue forever, quitting only when a certain condition is met. There are two main ways of constructing a DO loop for this purpose. The first is the DO WHILE construct, it is used in the following manner:

```
DO WHILE (logical_condition)
  !code
  !statement setting logical condition under some circumstance
END DO
```

This is telling the DO loop that until the condition `logical_condition` is `.TRUE.`, it will continue processing the loop. Therefore, it is important that under some condition, something *within* the loop can affect that logical condition. If it does not, then the logical condition will never change, and the loop can never end. This problem is referred to as an infinite loop, and will cause the program to process the loop commands until it is either forced to close by the user, or it crashes for another reason. Note that the '`logical_condition`' in the above code can be any relational or logical expression/statement, or any logical variable.

Note that it only checks the logical condition when it begins the loop. Look at the following code for an elaborated explanation of this:

```
DO WHILE (logical_condition)
  !code 1
  logical_condition = .FALSE.
  !code 2
END DO
```

In this case, in between the '`!code 1`' and '`!code 2`' pieces of code, the exit condition, `logical_condition` is set to `.FALSE.`, meaning that the exit condition is fulfilled. However, the section `!code 2` *will still be run*. The loop will only exit once all of the code within the loop has been run, and it goes to start at the beginning again, at that stage, it checks the logical condition, and the code will exit.

The following complete code shows a very simple example of a DO WHILE loop.

```
1  PROGRAM dowhile_example
2
3    IMPLICIT NONE
4
5    LOGICAL :: check
6    INTEGER :: value, count
7
8    value = 0
9    count = 0
10   check = .TRUE.
11
12   DO WHILE (check)
13     count = count + 1
14     value = value + 3
15     IF (value .GE. 100) THEN
16       check = .FALSE.
17     END IF
18   END DO
19
20 END PROGRAM dowhile_example
```

In this case, the overall purpose of the code counts how many times `3` must be added to `value`, before `value` is equal to or greater than `100`. This example uses the setting of a `LOGICAL` variable to determine whether to continue, however, it could also be written as shown in the next example, where it is a relational statement which is the criteria for whether to continue. Remember, they are both effectively the same, once any relational or logical statement has been evaluated, it returns a `LOGICAL` value, exactly equivalent to a `LOGICAL` variable of the same value.

```
1  PROGRAM dowhile_example
2
3    IMPLICIT NONE
4
5    LOGICAL :: check
6    INTEGER :: value, count
7
8    value = 0
9    count = 0
10   check = .TRUE.
11
12   DO WHILE (value .LT. 100)
13     count = count + 1
14     value = value + 3
15   END DO
16
17 END PROGRAM dowhile_example
```

Note that this time the check is for whether `value` is *less than*, as opposed to `greater than or equal to`, `100`. This is because it is effectively the opposite criteria, the conditions to *continue*, rather than the condition to *stop*.

4.5 Open DO Loops

The other method for constructing `DO` loops of arbitrary length uses the following form:

```
DO
  !code 1
  IF (logical_condition) EXIT
  !code 2
END DO
```

In this case, there is no criteria specified after the `DO` line at all. It will continue performing the actions in the loop until it meets an `EXIT` command. In this case, it is provided by an `IF` statement dependent on a logical condition, however, the `EXIT` can be included anywhere within the loop, in any manner. It is also important to note that in this variant, the loop will stop as soon as the `EXIT` command is reached, it will *not* continue to the end of the current loop before stopping as with the `DO WHILE` method. Therefore, in the example code above, the line `!code 2` would *not* be run, if the logical condition was met.

This method tends to be less popular. The `DO WHILE` loop effectively defines what the criteria for exit is at the beginning, making it clear what will end the loop. The open `DO` loop style shown here could end any time, for any reason, making it less easy to manage and understand from a programming perspective.

4.6 Assignments Summary

4.6.1 Programming Assignment 1

1. Create a new file for your code called '`assign_4_1.f90`'.
2. Variables with suitable names should be defined.
3. Using a `DO` loop structure, you should write code such that the lines to the song '20 bottles of beer on the wall' are output to the screen and a file from 20 down to zero.
4. Your output file should be named '`assign_4_1.out`'.
5. For the case where the number of bottles is one, any mention in the output should read '1 bottle of beer on the wall...' rather than '1 bottles of beer on the wall...', ie, the 'bottle' should be singular, not plural.
6. For the case where the number of bottles is zero, any mention in the output should read 'no bottles of beer on the wall...' rather than '0 bottles of beer on the wall...'.

7. Make sure to keep your output file `assign_4_1.out`, as you will be required to submit a printout of it, as well as your source code file, `assign_4_1.f90`.

4.6.2 Programming Assignment 2

1. Create a new file for your code called '`assign_4_2.f90`'.
2. Variables with suitable names should be defined.
3. Your program should ask for and then read input from the user for the number of students, and the number of assignments.
4. You should open access to an output file, which should be named '`assign_4_2.out`'.
5. You should declare and initialise a variable which will **count** the number of successful output instances.
   Hint: Starting from zero, this should increment each time a student/assignment combination is output. It should **not** simply be the product of the number of students and number of assignments!
6. You should implement a nested `DO` loop structure, to print out all combinations of student number and assignment number.
   Note: Check the section for this assignment for the manner in which the student/assignment numbers should be listed (Section 4.3.2).
7. After the output of all student/assignment combinations, you should output a line with just the value of the count of the number of output lines, using the value in the variable described earlier.
8. Make sure to close the file.
9. The program should be run using '`7`' as the number of students and '`4`' as the number of assignments.
10. Make sure to keep your output file `assign_4_2.out`, as you will be required to submit a printout of it, as well as your source code file, `assign_4_2.f90`.

4.6.3 Questions

For this week the questions are as follows:

1. What type of variable does the counter variable for `DO` loops have to be?
2. What is the default value for the 'increment' parameter of a `DO` loop if not set explicitly?
3. If you have a discrete grid system in three dimensions, x, y and z; and wish to iterate through every grid cell, how many nested `DO` loops would be required, *including* the outermost loop?
4. Look at the following code fragment; what would the output be? (*Note:* `i` and `j` are both `INTEGER` variables)

```
j = -4
DO i = -3, 6, 2
  j = j * i
  WRITE(*,*) j
END DO
```

4.6.4 Submission

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_4_1.f90`, `assign_4_1.out`, `assign_4_2.f90` and `assign_4_2.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

5 Week Five - Arrays

Important Note: The work for this week does *not* have to be handed in at the end of the week. It is due in by the end of the **first session of Week Six**. This gives you an extra weekend and workshop session to complete the assignments and questions.

5.1 The Basics of One Dimensional Arrays

5.1.1 The Concept of Arrays

An important concept in computing is that of data storage. We have already seen how single values, letters or short text can be stored in a variable, but what if we want to store a substantial amount of data?

For example, say you have a generic function, $y = F(x)$, and you want to store each value of y for one hundred different values of x. Were you to do this with regular variables in the way you have seen so far, that would require defining two hundred variables, and referencing them all one after the other manually.

This clearly is not a suitable method for the storage of large quantities of data which possesses some sort of repeated format. To solve this problem, we make use of a computational structure called an '*array*'.

A simple 1-D array can be thought of as a column in a spreadsheet, it has a certain number of spaces for data, each of which can be referenced by it's position along the column/array, this position by which the item of data can be referenced is called its '*index*'. In Fortran, array indices can run between any two integer numbers. Most commonly used is between `1` and however many elements exist. However, you could equally have an array with 5 spaces, indexed between `-2` and `2`, say.

In order to show how an array is used within Fortran, look at the following example:

```
1  PROGRAM simple_array_example
2
3    IMPLICIT NONE
4
5    INTEGER, DIMENSION(1:6) :: myarray
6    INTEGER :: i
7
8    myarray(1) = 3
9    myarray(2) = 1
10   myarray(3) = 4
11   myarray(4) = 1
12   myarray(5) = 5
13   myarray(6) = 9
14
15   DO i = 1, 6, 1
16     WRITE(*,*) myarray(i)
17   END DO
18
19 END PROGRAM simple_array_example
```

First, notice the addition to the declaration of '`myarray`' on line 5. The statement '`DIMENSION(1:6)`', means that we want to define a 1-D array with indices ranging between `1` and `6`. Having defined this, the variable `myarray` is now an array for multiple values, rather than a single value.

Referring to values within an array is done by referencing the index of the array which is desired. This works both for setting a value in an array, and reading out a value. Both are illustrated in this example. Look at lines 8 to 13. In each case we are specifying the `myarray` variable and then, in brackets, we put the index of the place in the array which we want to set as a particular value. On these lines we set each of the indices 1 to 6 with values.

This example also illustrates the versatility of arrays. On lines 15 to 17 we use a `DO` loop to read out all of the array values. This is possible because the value of the array index intended to be read doesn't need to be specified manually in the code, it too can be set via a variable. In this case, we have the line '`WRITE(*,*) myarray(i)`'. This means that it will fetch the value in the array index of whatever the '`i`' value is. In the case of this example, by using the `DO` loop, we go through each valid value of `i` from 1 to 6, therefore printing out to screen every element of the array.

Effectively, every 'place' or 'element' of an array acts as an independent variable, but rather than each having their own unique variable name, they have a single collective name for the whole array, and each element is then referenced only by a number.

For output, array elements needn't be referenced individually, one at a time. Fortran can handle output of entire arrays, this means that the final `DO` loop of the code could be replaced with just:

`WRITE(*,*) myarray`
and the code would output to screen every value of the array `myarray`. Note, however, that they will all be on the same line, not separate lines for each value.

If you want to only output a subsection of an array, this can be done using 'array slicing', specifying a range of array elements in one go. This works in the same format as when the range of the array was declared initially with `DIMENSION(1:6)`. In the brackets we specified `(minimum:maximum)` of the intended size of the array. In the same way we can select a minimum/maximum range within the total bounds of the array later on. So if we want to output the first four values of the array, but not the final two, we can use the notation:
`WRITE(*,*) myarray(1:4)`
And, so long as array elements with that range of indices exist, only those four values will be returned. Easy! The important thing to note is that the specified range of indices *must* be within the existing bounds of the array. For our array with indices 1 to 6 in the above example, the following line of code will fail:
`WRITE(*,*) myarray(4:8)`
Since the highest index is 6, and this line is trying to access elements 7 and 8, it will not work.

5.1.2 Implementation of Arrays

The following example is more complex, to aid your understanding, it is recommended that you try typing out the code, compiling and running it. It asks the user to input values of each element of an array, and then asks which value the user would like to output back:

```
1  PROGRAM simple_array_example
2
3    IMPLICIT NONE
4
5    INTEGER, DIMENSION(-2:2) :: myarray
6    INTEGER :: i, ind
7
8    DO i = -2, 2, 1
9      WRITE(*,*) 'Please input the value for element ', i, ' of the array:'
10     READ(*,*) myarray(i)
11   END DO
12
13   WRITE(*,*) 'Thank you. Please enter the index (between -2 and 2) of the value which you
       would like to read out:'
14   READ(*,*) ind
15
16   IF ((ind .GE. -2) .AND. (ind .LE. 2)) THEN
17     WRITE(*,*) 'The value of array index "', ind, '" is: ', myarray(ind)
18   ELSE
19     WRITE(*,*) 'The chosen value "', ind, '" is not a valid index for this array. Program
       will exit.'
20   END IF
21
22 END PROGRAM simple_array_example
```

Can you work out what the program does at each step? Let's look at it line by line.

Firstly, on line 5, we have defined an array, again called '`myarray`' with indices between -2 and 2, therefore five individual elements (-2, -1, 0, 1 and 2). Line 6 defines two other `INTEGER` variables which will be used.

Note: It is worth noting that array indices crossing between negative and positive numbers always include 0, therefore an array between -2 and 2 does not have as many elements as the larger minus the smaller ($2 - -2 = 4$) it will have one more to include the 'zeroth' element.

Additional Note: It may also be of interest that the array standard used by Fortran is *not* the same for all languages. For example, in the language 'C', by default *all* array indices start from zero, and go up to the value 'number of elements - 1'. This is worth keeping in mind if you ever switch to different languages, or use more than one language for a collection of tasks.

Between lines 8 and 11 we have a `DO` loop operating between -2 and 2, covering all of the valid indices of the array which we defined on line 5. For each index, the code prints a prompt to the screen for the user, and then reads in a value to that index of the array.

After this, on line 13, the code prints another prompt for the user, asking for an index value between -2 and 2, in order to display the chosen element. It then reads this in as the variable '`ind`' on the following line, line 14.

Now we have a piece of code not directly related purely to arrays, but to concepts covered in a previous section. The IF-ELSE-END IF construct between lines 16 and 20 is an input validation check. The IF statement on line 16 finds out whether the entered value for ind is greater than or equal to -2, and also less than or equal to 2, ie that it is one of the valid array indices. If this check returns .TRUE., then the code prints a message, and the value of the element at index 'ind' of the array. If the check returns .FALSE. it lets the user know that they have entered an invalid value for ind.

If you've tried writing, compiling and running the code, you should be able to check firstly that it does work, then see how it works, and possibly determine places or situations in which it doesn't work, or the result is unexpected. This code is still fairly basic from a programming point of view, and is not made to be robust for all possible inputs. As with previous examples requiring input of a value, putting a non-numeric value when the program asks for a numeric value will still cause a crash.

5.2 Input/Output of One Dimensional Arrays

5.2.1 Data Statistics Program

At the beginning of the previous subsection, we mentioned the scenario of processing a simple function $y = F(x)$ for one hundred different values of x. In this section we will look at applying a process similar to that by using arrays.

Say that we have a file of many values and want to find the average value. For this example, the file we will use contains the simulated marks out of 100 of 90 students. To obtain this sample data file in order to use it yourself, use the following command:

```
:> cp ~ph302/sample_marks.dat ./
```

This will copy the file to whichever directory you are currently in. It is a simple file with 200 lines, each line having a value between 0 and 100.

Shown in Figure ?? is the histogram of the supplied data file. As you can see, the peak in the histogram appears to lie somewhere around 60, however, we can write a code to return the specific value of the mean. Note that, while it is clear that in this example, it would be possible to do this in Excel (or similar graphical user interface spreadsheet application), what about more complex scenarios? Perhaps there are 200 students or other data points per file, but you also have 90 of those files. Going through each file one at a time manually in excel would take a substantial amount of time and hassle. If you have a code where you just specify each file in turn, or even just the list of files in one go, and all of the calculations are done automatically makes the task much quicker.

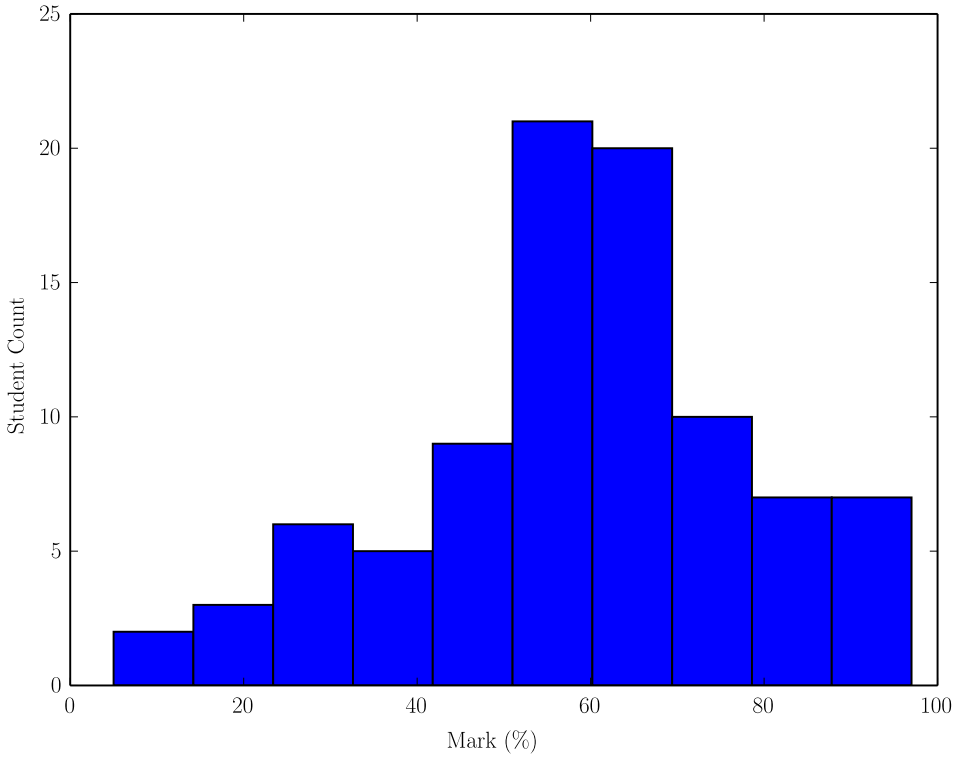


Figure 1: Histogram of sample mark data for 90 students

In the same way in which the asterisk character can be used as a 'wildcard' to refer to any file, or subset of filenames

when using commands like 'ls', it is also possible to write a program such that all of your input files are in the form 'data_#.dat' or similar (where '#' is the file number), and run you program in the following way:

```
:> ./dataprocessing data*.dat
```

To automatically process all of the results of all of your data files.

However, in this simple case we will only be using the one file.

The following code reads in the data file you have obtained, calculates min, mean and max values, and then outputs the result to the screen.

```
1  PROGRAM stats_finder
2
3    !************************
4    !code written by P. S. Eudonym, last edited 31/07/10
5    !Program to find mean and max for a file containing 90 lines of student
6    !marks between 1 and 100
7    !************************
8
9    IMPLICIT NONE
10
11   !Array for the marks
12   INTEGER, DIMENSION(90) :: marks
13   !Stats variables
14   REAL :: meanmark
15   INTEGER :: maxmark, marksum
16   INTEGER :: studentmin, studentmax
17   !generic counter variable
18   INTEGER :: i
19
20
21   !Open read in file, and iterate through all 90 entries, reading in the marks for all
       students
22   !Additional check to make sure the mark is within valid range
23   OPEN(10,FILE='sample_marks.dat')
24   DO i = 1, 90, 1
25     READ(10,*) marks(i)
26     IF (marks(i) > 100) THEN
27       WRITE(*,*) 'Error with file, mark greater than 100 found. Exiting.'
28       STOP
29     END IF
30   END DO
31   CLOSE(10)
32
33   !Initialise the stats variables
34   maxmark = -1
35   marksum = 0
36   studentmin = -1
37   studentmax = -1
38
39   !For each mark, add it to the sum and find if it is the new maximum
40   DO i = 1, 90, 1
41     marksum = marksum + marks(i)
42     IF (marks(i) > maxmark) THEN
43       maxmark = marks(i)
44       studentmax = i
45     END IF
46   END DO
47
48   !divide sum by number of elements to get mean
49   meanmark = REAL(marksum) / 90.0
50
51   !Output stats
52   WRITE(*,*) 'Maximum mark is:', maxmark, 'Obtained by student number:',studentmax
53   WRITE(*,*) 'Mean mark is:', meanmark
54
55 END PROGRAM stats_finder
```

A a slightly longer and fuller example of code than normal, you will notice that the program above has also been suitably commented. Remember to make sure that comments you write in your own programs fulfil similar functions to in this program, and the guidelines laid out in the section on good programming practice (Section 14).

The important pieces of code from the point of view of using arrays are the two loops. Firstly, each element in turn is read in. Remember that the number or variable in brackets after the name of the array is the index of the element which is being referenced. The second loop uses each element from the array in turn for determining the stats.

The process of actually finding the mean and maximum values merits some explanation. You will likely note that before the loop to determine these values, each is initialised. `marksum` to zero and, more interestingly, `maxmark` to `-1`. Can you work out why the minimum and is set like this before the loop begins? Mark sum is to be a the cumulative sum of the values of each element as they are processed. Once all elements are processed, by dividing this sum by the number of elements, we have effectively performed the following mathematical operation:

$$x_{\text{mean}} = \frac{\sum_{i=1}^{N} x_i}{N} \tag{3}$$

Where N is the number of elements, 90 in the case of this data set. We have summed each element in turn, and then, by performing the final division, we have found the mean value.

Setting `maxmark` to `-1` before the loop is for a less obvious reason. To begin with, we know that it has to be set to something. If it wasn't, then when the loop begins, and attempts to compare the first mark with the existing maximum value, it will have nothing to compare it with and fail. By setting it to `-1`, we know that whatever the first element's value is, it will be found to be the maximum. After this, any subsequent comparisons will still be valid.

5.2.2 Expanding the Data Statistics Program - Assignment Material

Currently, the program shown in the previous subsection finds the mean value and maximum value within the dataset being read in (as well as the student number of the one with the highest mark). Another useful value to determine would be the minimum value and associated student number. That aim will be the objective of the first assignment for this week.

Initially, you should create a file called '`assign_5_1.f90`', and write in the code presented above. If you have not done so already, obtain the sample data file with the command:

```
:> cp ~ph302/sample_marks.dat ./
```

You should now be able to compile the code you've typed up, and, when run, it should present you with the mean and maximum values in the data set. These should be 58.288889 for the mean, and 97 for the maximum value, and 18 for the index/student number of the maximum value.

You should be especially careful in your use of commenting for this program. Assume that you are expanding upon a proper program written by someone else; you should make sure that the information about the original author remains in the program, and that you add a comment describing in what way you are expanding it in addition to your name and the last date you edit it.

The expansion to the program which you will be performing is fairly simple, but the guidance provided will be limited, in order to test your ability to comprehend and expand code on your own.

The main task is to add a pair of variables, and set of processes through which the minimum value in the dataset is found, as well as it's index location within the dataset (ie, the student number), in much the same way that the maximum is already found.
Note: While there *do* exist intrinsic functions to find minima and maxima of an array (the functions '`MIN`' and '`MAX`' respectively); the use of these is *NOT* permitted for this assignment. While in an actual program, their use would be perfectly acceptable, the aim of this assignment is to test your comprehension of existing code and ability to adapt said code.

The specific implementation of this minimum finding is down to you, but it is broadly expected for you to mirror the existing implementation of the maximum finding setup.

The second component to this assignment is to replace the final two `WRITE` statements with suitable `OPEN` and alternate `WRITE` statements to put the same information, plus the information on the minimum, to a file to be called '`assign_5_1.out`'.

5.3 Sorting

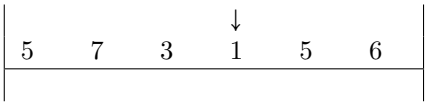
5.3.1 Concept of Sorting and Computational Complexity

One problem which frequently requires a computational solution is that of sorting. With a jumble of data in a random order, there are a number of techniques which can be applied to sort it based on some criteria. To appreciate the nature of the computational problem, try looking at the following example. How is it that you go about sorting a set of numbers? Can you make an algorithm out of that? Remember to consider that to program such a sorting procedure, every step

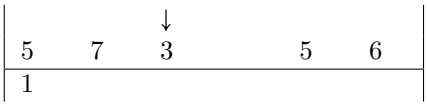
must be a specific, single action. Note that considering this problem is not an assignment, just an illustration of the nature of the problem of sorting large data sets.

Take the following values:
5, 7, 3, 1, 5, 6

It should be fairly easy for you to sort these into an order from smallest to largest, say. But how could a computer be made to do it? Do you search for the smallest value of the whole data set, then put it first in the list, then search for the next smallest and so on? So the process for the example would be the following, the top list being our data set, and the bottom being the new, sorted, list which we are building:

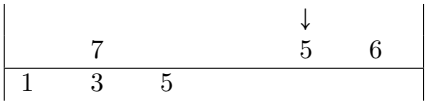


Initially, our sorted list contains nothing. We check our entire data set for the smallest value, in this case 1, highlighted by being the arrow, this is removed from the data set and added to the first element of the sorted list:

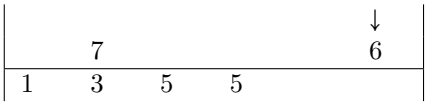


We check the remaining elements in the data set, and find the value of 3 to be the smallest; this is removed from the data and added as the next element of the sorted list:

The next smallest is 5, but there are two of them. Which is taken depends on the manner in which we have constructed the specific algorithm in our code. If we look at each element in turn, recording the location and value of the smallest element so far, move to the next element and only store its value and location if it is specifically *smaller* than the current smallest, then the first 5 and it's position is stored, and when we reach the second 5, it is *not* larger, and so the largest element remains the first 5. This would then give us:



The next check would give the remaining '5' later in the list, now that the first one which we found has been removed.



After adding the 5 it finds 6 as the next smallest, and can add that to the new list. This process continues until all of the data set has been processed resulting in a sorted list the same length as the original data set. In the case of our example, after adding the 6, we only have one element left, the 7, which we then know must be the final element for our completed list.

So, would this algorithm be efficient for a computer? To make a computer do this for a data set with 'n' values in it, initially you would need to go through the data set and make n comparisons. In other words, you first have to find the smallest value of the initial dataset by looking at every element. Having done this and recorded it, you then need to go through almost the entire data set again to find the next smallest (all of $n-1$ elements). Next time, $n-2$ and so on, until you reach $n-(n-1)$, where you have a single element left, which must be the final value. The number of times you have to go through the entire data set, or almost the entire data set, is the same as the length of the data set itself. Although the number to check against goes down, it's almost like having to go through every element, and for each of them, check every other element. This is described as an order n^2 problem (notated as $\mathcal{O}(n^2)$), as the approximate order of complexity requires n^2 computations/comparisons. Therefore, a list with 200 elements will take about 4 times as long to process as a list with 100 elements; A list with 100,000 elements will take, not 1,000 times as long as one with 100, but one million times longer.

If you have a data set many thousands of elements long, perhaps millions or even billions of elements, then the computational time may become a significant issue.

This type of sort is particularly bad because, even in a best case scenario, where all of the elements are *already* in order, it will still have to do the same number of comparisons as the worst case scenario where the elements are completely random! So, if you have a billion elements, all already in order, it could takes minutes or even hours to 'sort' them despite the fact that no further sorting is needed!

Furthermore, this is only considering the computational time. We also need to consider the stored data. Because we are building a new list as we take elements out of the old list, we need to reserve memory for a list as long as our original. Therefore, if we have a data set of 1,000 elements, we also need a second array in memory of 1,000 elements to store the new list as we build it. You may think it would be worth reallocating space as we build the new array, removing space from our first list and adding it to our second. However, this would take even more computational time. Allocating memory can be computationally expensive if done in small amounts frequently, due to the overheads. An overhead, for those that haven't come across the term before, can be thought of as the 'admin' for a computational action. For memory allocation, a certain amount of time has to be spent simply arranging to do the allocation. So if we allocate 1,000 new items at once, we have to do that organisation to allocate memory once, then the time spent for 1,000 elements. If we allocate 1,000 elements one at a time, every time we have to add the 'organisation' time, in addition to the time taken to actually allocate the single elements. As such, allocating 1,000 elements one at a time takes substantially longer than allocating 1,000 all in one go.

As such, this is clearly an unsuitable algorithm to search with for even moderately long data sets. In the following sub-section, we will look at an alternative.

5.3.2 Introducing The Bubble Sort

One of the simpler methods to perform a sort which does not require a uniform computational time regardless of how 'pre-sorted' the data set is, is that of the '**bubble sort**'. The basic concept of the bubble sort is that you move along the list comparing adjacent elements. When two elements are in the wrong order, they are switched, effectively shifting the larger value down a space 'sinking' past the 'lighter'/smaller value. Because of this shift, the next adjacent pair compares the previously shifted large value, and the next along. Once more if they are in the wrong order, the larger 'above' the smaller, they are switched again. For a particularly large value, this process of comparing adjacents causes it to 'sink' down the list until it meets an even larger value (or the end of the list), in which case no switch is necessary.

This means that after one pass through the entire list like this, the most large element is guaranteed to have reached the end of the list. Therefore, next pass, you only need to compare adjacents down to the $n-1$'th element of the list. Next time it will only be to the $n-2$'th element. This ensures the shift of large values to the bottom/end of the list, while small values gradually 'bubble' up to the top/beginning of the list.

In text, this may not be particularly clear, therefore we can look at an example bubble sort, following through the actions performed by the algorithm. After this we can attempt to estimate the computational complexity of this method compared to that shown in Section 5.3.1. Having shown whether or not it is an improvement, we can translate the process into an effective code, and create a program which performs a sort on some data.

5.3.3 Illustrating The Bubble Sort

Let's take the same data set as the previous example. We have added a line to the table to keep track of the original indices of the elements, this is the top line. Then we have the element values themselves, in the same order as the previous data set.

Element No.	—	*1*	*2*	*3*	*4*	*5*
Element Val.	—	5	7	3	1	5

Unlike our previous example, where elements were extracted from the original list to create a new list, a bubble sort is generally performed as an 'in place' operation on the array of data. The changes it makes move around the elements of the existing array, rather than gradually creating a whole new one. Two loops are required for a standard bubble sort of the form we are implementing, so manipulation of the elements will be described relative to a pair of loop counters, i and j. Our outer loop will be represented by i, and the inner loop by j. So, we start our outer loop at 1, and it will nominally go to $(n-1)$. Our inner j loop will go between 1 and $(n-i)$ for each instance of i.

For each j step we do two things. First, we compare elements j and $j+1$, if they are in the right order, we do nothing, and no change is necessary. If not, we move to the second operation, in this case we switch the elements around. In each column, the numbers in **bold** are the j and $j+1$'th elements. The arrows next to them indicate whether the elements do not require switching (straight arrows, $\rightarrow$) or do require switching (curved arrows, $\rightharpoonup$ and $\rightharpoondown$).

$i = 1$; $j = 1$ to $(n - i)(= 5)$

Index		$j = 1$		$j = 2$		$j = 3$		$j = 4$		$j = 5$	
1	**5**	→	5		5		5		5		5
2	**7**	→	**7**	↷	3		3		3		3
3	3		**3**	↴	**7**	↷	1		1		1
4	1		1		**1**	↴	**7**	↷	5		5
5	5		5		5		**5**	↴	**7**	↷	6
$(n =)6$	6		6		6		6		**6**	↴	7

After this first pass, notice how even though the 7 started near the top, it reached the end of the list. This is because on each pass of the i loop of a bubble sort, the largest element is guaranteed to reach the right place. The 'heaviest'/largest value will sink to the bottom by the end of each pass. Because the last list element is now definitely the correct value, it will not need to be considered by any subsequent pass of the algorithm. This is why the j loop goes to $(n - i)$, as you go through sequential passes and i increases, the $(n - i)$ limit will cause the j loop to go less and less far through the list. The next two tables show the second and third passes, $i = 2$ and $i = 3$.

$i = 2$; $j = 1$ to $(n - i)(= 4)$

Index		$j = 1$		$j = 2$		$j = 3$		$j = 4$	
1	**5**	↷	3		3		3		3
2	**3**	↴	**5**	↷	1		1		1
3	1		**1**	↴	**5**	→	5		5
4	5		5		**5**	→	**5**	→	5
5	6		6		6		**6**	→	6
6	7		7		7		7		7

$i = 3$; $j = 1$ to $(n - i)(= 3)$

Index		$j = 1$		$j = 2$		$j = 3$	
1	**3**	↷	1		1		1
2	**1**	↴	**3**	→	3		3
3	5		**5**	→	**5**	→	5
4	5		5		**5**	→	5
5	6		6		6		6
6	7		7		7		7

You will notice at this point that we have now fully ordered the list already. However, the algorithm does not know this, it must do one full pass without any changes or reach the end of the normal loop before it knows.

$i = 4$; $j = 1$ to $(n - i)(= 2)$

Index		$j = 1$		$j = 2$	
1	**1**	→	1		1
2	**3**	→	**3**	→	3
3	5		**5**	→	5
4	5		5		5
5	6		6		6
6	7		7		7

At this stage, there could still have been one comparison and possible switch to make, the number of elements it would definitely have moved to the right place is only four at the end of this pass. However, since the pass for this i has resulted in no changes, it is now known that the data is already in the right order, a conditional statement within the code would let the loop exit early. So although the original algorithm would want to go to $i = (n - 1) = 5$ to do the final comparison, because the elements have fortuitously ended up in the right order before then, we can exit before the normal end of the iterations. This is unlike the previous algorithm we looked at, which would always have to go through every single iteration before the list could be guaranteed to be ordered.

5.3.4 Analysing The Bubble Sort

Now let's consider the computational complexity of the process shown in the previous two subsections. Imagine the data set was already perfectly ordered, what would the bubble sort do? It would look at the first and second elements, and check if they were incorrectly ordered. Finding that they weren't, if would move on. Then it would compare the third element with the second, and again, see that no switching was necessary. It would go through every element like this,

performing only one comparison, and never needing to iteratively move elements. As such, the best case scenario for a bubble sort can be described as 'order n' or $\mathcal{O}(n)$ using the standard notation. This already is an advantage over the first sorting method we looked at, which was $\mathcal{O}(n^2)$ in even the best case. Therefore, for pre-sorted data, bubble sorting a thousand elements would take 100 times longer than bubble sorting ten pre-sorted elements; whereas, using our initial algorithm, sorting a thousand pre-sorted elements would take 10,000 times longer than sorting 10 pre-sorted elements!

We now look at the worst case scenario. In this instance, we assume the elements are in *reverse* order to that which we are sorting to. Therefore, we will have to do the maximum number of switches for each pair in turn. Every single pair will have to be compared and sorted right to the beginning of the list every time! You may notice that this is very similar to our old algorithm. First, we will have 1 comparison/switch to make, the second iteration will require 2 comparisons and switches, by the n'th element, n comparisons and switches will be needed. This means that the bubble sort, in a worst case scenario, also has $\mathcal{O}(n^2)$ complexity.

This is not an advantage over our original algorithm. However, it means that the complexity of a bubble sort scales between $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$ the more and more initially ordered the data is. As such, it provides a better process for sorting our data than the algorithm described in Section 5.3.1. Additionally, you may recall the bubble sort being described as an 'in place' sort. It moves elements around the original array, instead of building a new array. This means that it's memory requirement is simply the amount of the original array, plus any additional single variables used for the program. This is a substantial improvement over the Section 5.3.1 algorithm, which would more than double the amount of memory the bubble sort would.

If it is worked through fully, it turns out that the 'average' complexity for a bubble sort is actually still in the region of $\mathcal{O}(n^2)$. There are many other algorithms which are substantially quicker than this for an average case, however, the bubble sort is still an improvement over our initial method, and is comparatively easy to understand and program. Additionally, the process of programming a bubble sort handily brings together several of the concepts covered so far in the course, making it a useful algorithm to learn from a teaching perspective, if not from a purely computational one.

Some of the best sorting algorithms have complexities like $\mathcal{O}(n\log(n))$, a substantial improvement over the $\mathcal{O}(n^2)$ performance of the algorithms we've looked at here. The time to bubble sort 1,000,000 elements is 100,000,000 times the time to bubble sort 100 elements. For these advanced sorting algorithms, the time to sort 1,000,000 elements is about 30,000 times the time to sort 100 elements. This is clearly an enormous improvement over the bubble sort. However, these algorithms are generally substantially harder to understand and program than a bubble sort. A few algorithms with average complexities of $\mathcal{O}(n\log(n))$ which might be of interest to investigate are: the 'Merge' Sort, the 'Heap' sort (both of which also have worst case complexities of $\mathcal{O}(n\log(n))$) and the 'Quick' sort (though this one still has a worst case order of $\mathcal{O}(n^2)$). If it is still available when you are reading this, the following web site contains good descriptions, illustrations and animations of several of the key sorting algorithms and was used as a reference for the stated complexities in this section for the non-bubble sort algorithms: http://www.sorting-algorithms.com/[12].

5.3.5 Programming a Bubble Sort - Assignment Material

The previous sub-section leads us neatly, and almost inevitably, to the activity of actually going about programming the bubble sort algorithm described so far. Some help will be provided as to what needs to be done to achieve this, but, as an assignment, the implementation of the concept of the algorithm will mostly be down to you. Make sure that you have read and understood what performing a bubble sort actually does.

The nature of this assignment is ideal for describing the general process of attempting to convert a known process/algorithm into actual code. This is sometimes quite difficult. While a concept may be clear to you, being able to describe it in discrete, exact, computational steps, of the sort necessary to write a code, may still be difficult. It is an excellent skill to practice, with implications beyond programming. Converting a process that you understand into a form necessary for a particular implementation of that process occurs in all kinds of scenarios. For example, within science, being able to create an experiment to test a physical principle which you understand but wish to validate uses many of the same skills.

You should create a new file called '`assign_5_2.f90`', to act as your code file for this assignment.

We have determined the need for a nested loop which you will be required to program. Additionally, it is important to think about how elements need to be swapped. It is not possible to simply state:

```
a(1) = a(2)
a(2) = a(1)
```

Because after the first change, the original version of `a(1)` no longer exists, having already been overwritten by `a(2)`. Therefore, we require a temporary variable to put one of the values in to prevent it being overwritten and therefore unavailable to put into the other position. It would require something like:

```
temp = a(1)
a(1) = a(2)
a(2) = temp
```

Which will have successfully switched the positions of `a(1)` and `a(2)`.

We will consider more complex data later on, for now, we will use the same six element data set of our original

example. This should permit you to check that your code is doing the same thing as the example, and more easily debug your code if it does not.

You will need, as always, to start your code with suitable comments and the definitions of variables you plan to use. Two loop counters will be needed, it is recommended to make `i` the outer loop counter and `j` the inner loop counter. A temporary variable will also be needed, as described above; this must be the same type as your array. The array itself should be a single six-long `INTEGER` array. For the initial test data set, it is possible to define the whole set in the following manner:

```
array = (/ 5,7,3,1,5,6 /)
```

Which will assign all six values to the array in one go, rather than using `array(1) = 5` then `array(2) = 7` and so on. Make sure that the delimiters are the bracket and forward slash combination shown above; brackets on their own *will not* work.

The more complex part of the procedure is permitting the loop to exit if an `i` pass has been made, and no switches have occurred in the `j` loop. For this you should use a `LOGICAL` variable to track whether a change has happened, variables used like this are sometimes called 'flags'. At the beginning of each `i` loop you should set this to `.FALSE.`, every time a change is made, it should be switched to `.TRUE.`. This means that it doesn't matter if multiple changes are made, just one will be enough to switch the flag. Choose a suitable name for this variable, something like `switched` or similar. After the `j` loop, but still within the `i` loop, you will need a statement of the form:

```
IF (switched .EQV. .FALSE.) EXIT
```

This will cause the `i` loop to quit early if the data is already sorted before the final normal loop is reached.

The following is a description of the flow of the program you should write. Some items are inset to more clearly show what needs to be in which loop. You also need to include the normal `PROGRAM`, `END PROGRAM`, `IMPLICIT NONE` and any other required content such as comments.

- Define your variables, an `INTEGER` variable stating the number of elements (`n` is recommended for this); an `INTEGER` array with `n` elements; a temporary variable for when you switch elements; a `LOGICAL` variable for the check; as well as anything else you decide you need.
- Assign the values within your array, using the method described earlier.
- Begin a `DO` loop, going from `i = 1` to `i = n -1`.
  - Assign the value of your `LOGICAL` flag as `.FALSE.`.
  - Begin a `DO` loop, going from `j = 1` to `j = n - i`.
    - Have an `IF` statement checking for if element `j` and element `j+1` in the array are in the correct order. You will need to work out which should be bigger for this to be the case, remember that the array should end up with the smallest value at the beginning and the largest at the end.
    - If they are, you need not do anything.
    - If they are not, you must switch the two elements using the method described earlier, and set your flag variable to `.TRUE.`.
  - End your `j DO` loop.
- Check whether any switches have been made, so that if they haven't, the loop can end straight away rather than check the remaining steps unnecessarily.
- End your `i DO` loop.

After coding this, you should test that it works. If it appears not to, try to debug it. It is often very helpful to add temporary additional `WRITE` statements at various points. This can let you see what things are happening, and where certain numbers are or what the program is 'thinking' at certain times. For instance, you could put a `WRITE` statement at the beginning of your `j` loop to state what the array values at `j` and `j+1` actually are, and then another `WRITE` statement saying something like '`switched`' if it attempts to switch the values. Then, when you run your program, you will be able to see the pairs of numbers and whether the code switched them. If it doesn't switch pairs you know should be switched, or does switch pairs that you know shouldn't be; you can determine that something must be wrong with your criteria for switching.

Becoming familiar with this process of debugging is extremely useful. Even the best programmers will inevitably have some problem with a complex program when they first write it. Being able to find what that problem is and fix it is a vital part of developing a piece of code.

5.3.6 Modification to Your Bubble Sort - Assignment Material

The list/array which your bubble sort has processed so far is unrealistic for the typical kind of problem you may have to solve computationally. It is easily arrangeable by a person without a PC, or could be entered with ease to another

program manually. Therefore, for the next step of your bubble sort program, we will read in a file with many more elements in the list, and write out a new file containing the sorted version of that list.

Rather than have your program determine the number of elements when it runs, we will assume that the data is sufficiently well known that you can hard code the number of elements. You will have to change your value of `n` and the size of your array variable to be 1,000, rather than 6.

Then you will need to fetch the sample data file, this is obtainable by typing:

```
:> cp ~ph302/sample_random.dat ./
```

Which will copy the sample data file, called `sample_random.dat` to whichever directory you are currently in. Additionally, you must create a copy of your existing program (`assign_5_2.f90`) so as to retain a working source file of your original program. The new copy should be called '`assign_5_3.f90`'.

Then, you should add suitable `OPEN` and `READ` commands to read in your data set. Reading in values to an array is performed in exactly the same way as for regular variables. If you imagine that the array variable you are reading data into was expanded into separate variables, one for each array element, that is how a `READ` or `WRITE` command would process it. In other words, for an array '`mydata`' with 3 elements, a `READ` line like the following:

```
READ(1,*) mydata
```

Would, to the program, effectively be: `READ(1,*) mydata(1), mydata(2), mydata(3)`
With the array elements expanded out automatically. This means that it will read in separate values of data for however many elements your array has, so long as they exist in the source being read from. Therefore, if you've called your array '`array`', and have opened the sample data file on file unit `1`, your `READ` command would simply be:

```
READ(1,*) array
```

The process is more complicated, however, if you do not know how many values your program is going to read in. In that case you would have to first look through the file to determine how many were there, and then create the array after the program was already running. This more complex operation will be looked at later, but is not necessary for this assignment, as we already know there are exactly 1,000 values in the file.

With this change, your program should act in the same way as the version with 6 pre-input values, but for the much larger data set. Since there is so much more data, it is worth outputting to an output file, rather than to the screen. So you will need to add suitable `OPEN` and `WRITE` commands to do so. The file for your output should be called '`assign_5_3.out`'.

At this point you will need to run your program and check the output to ensure that the values have been correctly sorted. You should **not** print this output file though. Since there are so many values, and one line for each, it would use far too much paper and be unnecessary. Once you have ensured that your values are being correctly sorted, you should replace the statement '`WRITE(yourfileindex,*) your_sorted_array`' with two separate `WRITE` statements:

```
WRITE(yourfileindex,*) your_sorted_array(1:10)
WRITE(yourfileindex,*) your_sorted_array(991:1000)
```

Where `yourfileindex` and `your_sorted_array` are the file index and array names respectively which you chose to use. This will output just the first ten values (specified by the range '`(1:10)`' on the first line) and the final ten values (specified by '`(991:1000)`') of your sorted array. After making this change, you should re-compile and run your program, and ensure that the output file created contains only twenty values.

Important Note: Do **not** print the output file unless you are sure that there are only 20 values in the file. If in doubt, check with a demonstrator first.

5.4 Two Dimensional Arrays

The concept of arrays can easily be extended to being a 2D 'table' like construct, rather than a 1D 'list' like construct. In fact, it is possible to make n Dimensional arrays, with as many dimensions as required. The topic for the moment though is that of 2D arrays.

As mentioned, 2D arrays can be thought of as a table of data, one 'axis' of the array being the rows, and one 'axis' the columns. Such an array might be defined as follows:

```
1  PROGRAM 2d_array_example
2
3    IMPLICIT NONE
4
5    INTEGER, DIMENSION(1:3,1:3) :: myarray
6    INTEGER :: i,j
7
8    myarray(1,1) = 1
9    myarray(2,1) = 2
10   myarray(3,1) = 3
11   myarray(1,2) = 4
12   myarray(2,2) = 5
13   myarray(3,2) = 6
```

```
14    myarray(1,3) = 7
15    myarray(2,3) = 8
16    myarray(3,3) = 9
17
18    WRITE(*,*) 'Array elements first axis/second axis:'
19    DO i = 1, 3, 1
20      DO j = 1, 3, 1
21        WRITE(*,*) myarray(i,j)
22      END DO
23    END DO
24
25    WRITE(*,*) 'Array elements second axis/first axis:'
26    DO j = 1, 3, 1
27      DO i = 1, 3, 1
28        WRITE(*,*) myarray(i,j)
29      END DO
30    END DO
31
32  END PROGRAM 2d_array_example
```

The important definition is on line 5, specifically: `DIMENSION(1:3,1:3)`. Just as before the range of an axis of the array is defined by two numbers separated by a colon. However, in this case, we also separate definitions of the different axes with a comma. The array we have created here is a 3 by 3 array, therefore containing nine elements, each of which is given a value on the lines between line 8 and line 16. It is important to note the manner in which the array elements are indexed, rather than a single value for the index as before, we have two. The first is the index along the first axis of the array, and the second value for the second axis; separated by a comma, as with the initial definition of the array on line 5.

This is followed by two different loops, designed to illustrate the reading out of data from 2D arrays. Let's look at the array on lines 18 to 23 first. The nature of 2D arrays necessitates the use of nested loops for interacting with them. As one loop goes through one axis, the next loop in goes through the other axis. Therefore, every element, every permutation of array 'coordinates' is accessed. The first pair of loops has the outer loop going through the first array axis, and the inner loop going through the second axis. Remember how nested loops function, this means that for each index on the first axis, every element on the second axis will be referenced before moving on to the next index on the first axis.

When we initialised our array elements prior to the loop, you will notice that it is the first axis which varies every step, and the second axis varies only every three. This means that our first loop is reading out the values in the other 'direction' to how they were read in. What is the effect of this?

It means that, while our values were defined in numerical order, our first loop will read them out as follows:
1, 4, 7, 2, 5, 8, 3, 6, 9

This may not be entirely clear, and the structure and contents of the array is best represented by the following table:

$j \backslash i$	1	2	3
1	1	2	3
2	4	5	6
3	7	8	9

You can see how, if we look at all of the `j` indices for each `i` index, we will jump across the numbers, rather than go through them sequentially. In the second loop, though, we have made the loop through `j` the outer loop, and `i` the inner loop. This will output the numbers sequentially, or 'rows then columns' if based on the table above.

It is important to be aware which 'way around' your 2D arrays are, whenever you use them. However, if in doubt, compile one way knowing what your desired outcome is, and if it doesn't match, try the other way! Experimentation is always a useful way of ensuring you have coded something correctly.

5.4.1 2D Arrays Applied to Real Data

2D arrays as described so far have two literal axes representing generic data. However, they can easily be used to represent real properties. Consider a set of students, numbered 1 to 4, each having done two assignments. Storage of their marks could be contained in an array defined as follows:

```
REAL, DIMENSION(1:4,1:2) :: marks !referenced as marks(student number, test number)
```

As you can see, the first dimension is to provide a set of data for each student, and the second axis is for each test. This gives us an array which looks like the following:

	1	2
1		
2		
3		
4		

This obviously has the effect of working like a regular table of data, such as you might have in a spreadsheet application. However, the advantage of arrays in programming over opening tables in a spreadsheet application is that there is much more of an inbuilt limit with the capabilities of such programs compared to coding routines for data processing of your own. Spreadsheet programs are not optimised for handling very large datasets, performing complex procedures and analysis, or performing iterative processes on data. When used in a program of your own, arrays can effectively be as large as your computer's memory permits. It can also perform all manner of analysis on such datasets at a far faster rate than a commercial spreadsheet application could manage. Additionally, a far greater level of automation is permitted by coding a system yourself.

5.5 n-Dimensional Arrays

It has already been mentioned that it is possible to create arrays with more than two dimensions. Look at the following example of a declaration line and consider what it must be doing:

```
REAL, DIMENSION(1:4,1:2,1:8,1:6,1:3) :: myarray
```

In this case there are *five* different dimensions created. What does this mean in terms of what the array 'looks' like? It is possible to picture a 2D data structure easily, it would simply be a table, and a 3D data structure is conceivable if thought of as a literal third dimension of data extending back from the existing table. But what about the fourth dimension? Or fifth? It's difficult to conceptualise what an n-dimensional array actually exists as in terms of the data. It is perhaps easiest to conceive in terms of properties, rather than axes. In the previous subsection we considered a 2D array with one dimension the student number and the other dimension the test number, to store marks. Extend this idea to the multi-dimensional array, in the same way that there is a unique result for each combination of student number and test number, there will be a unique result for each combination of all of the parameters represented by an n-dimensional array.

So, can you determine how many unique elements there are in the example array above? You just need to multiply the lengths of each axis together, exactly as you would find the area of a 2D shape or volume of a 3D shape, hence: $4 \times 2 \times 8 \times 6 \times 3 = 1152$. As you can see, arrays with many dimensions rapidly take up very large amounts of space, even when the axes themselves do not have many elements.

5.6 Assignments Summary

Important Note: The work for this week does *not* have to be handed in at the end of the week. It is due in by the end of the **final session of Week Six**. This gives you an extra weekend and workshop sessions to complete the assignments and questions. The final session of this week will contain a mock test, to prepare you for the class test for next week.

5.6.1 Programming Assignment 1

1. Create a new file for your code called '`assign_5_1.f90`'.
2. Obtain the data file of marks using the command `cp ~ph302/sample_marks.dat ./`
3. Enter the provided base code, laid out in Section 5.2.1.
4. Ensure that the code operates as expected before continuing. The proper values for the statistics can also be found in Section 5.2.1, after the code is described.
5. Through adding of variables, required statements and other code; expand the existing code to find the value, and location of, the *minimum* of the data.
   Hint: The implementation of this should be fairly similar to the existing code for finding the maximum of the dataset.
   Note: Careful attention must be paid to the **commenting**. Leave the existing comments in, and add your own in addition. State your name etc, and when last modified, as well as describing the purpose and process behind any code that you add.
6. Alter the code so that, instead of outputting to screen, the output will be to a file called '`assign_5_1.out`'.
7. Ensure that your code operates as expected.

8. You will be required to hand in your modified copy of the code (`assign_5_1.f90`) and the produced output (`assign_5_1.out`).

5.6.2 Programming Assignment 2

1. Create a new file for your code called '`assign_5_2.f90`'.
2. Variables with suitable names should be defined.
3. A size 6 array should be created with the content of the sorting example: 5,7,3,1,5,6 (in that order)
4. You should then implement the algorithm for the Bubble Sort, as summarised in Section 5.3.5, and fully described in the Sections prior to that.
5. Your Bubble Sort algorithm should order the elements of your array from smallest to largest.
6. Your code must then output the new, sorted, content of the array to the terminal using `WRITE(*,*)`
7. Ensure that your code operates properly on this test array before continuing!
8. Create a copy of your `assign_5_2.f90` file called '`assign_5_3.f90`'
9. Obtain the sample unsorted data file using the command `cp ~ph302/sample_random.dat ./`
10. This file contains 1000 values, modify your array size to fit these values, remove the hard coded 6 values used to test your code, and add a suitable `READ` statement to extract the data from the `sample_random.dat` file into your array. *Note:* You can assume that the number of elements in the file (1000) is known in advance, you are *not* required to make your program adapt the array size automatically to fit what it finds in the file.
11. Erase the old `WRITE(*,*)` statement for the array from your previous version, and replace it with suitable `OPEN` and `WRITE` commands to output the sorted version of your array to a file which should be called `assign_5_3.out`
12. Compile and run your program, and ensure that it is correctly sorting the `sample_random.dat` data (you will need to check in your output file).
13. Modify the output statement to output only the first (smallest) ten values, and final (largest) ten values of your sorted array. Instructions for doing this are in this assignments section in the main text.
    Important Note: Ensure that your program is now outputting only 20 values before attempting to print. Your printed output for this assignment **must** only have 20 values. If in doubt, consult a demonstrator.

5.6.3 Questions

For this week the questions are as follows:

1. How many elements would arrays defined by the following commands have?:
   (a) `INTEGER, DIMENSION(-10:15) :: array`
   (b) `INTEGER, DIMENSION(0:20) :: array`
   (c) `INTEGER, DIMENSION(1:7,1:9) :: array`
   (d) `INTEGER, DIMENSION(-40:40,-30:30,-256:256) :: array`
2. Suggest any one purpose that a three dimensional array may be useful for; and provide a short justification (no more than a sentence or two).

5.6.4 Submission

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_5_1.f90`, `assign_5_1.out`, `assign_5_2.f90`, `assign_5_3.f90` and `assign_5_3.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

6 Week Six - Class Test

The sessions of Week Six are for continued work on the Week Five material, starting work on the Week Seven material, and revision preparing for the mid-term class test. On **Thursday** of this week you will be required to hand in *all* Week Five assignments. On Friday you should begin working on the Week Seven material.

Having progressed approximately half way through the course at this point, the Tuesday of Week Seven is reserved for attending a written class test. The content will be based on any of the programming concepts covered so far, with elements of: technical knowledge of the language, concepts of algorithms, bug finding, and some writing of code.

Good Luck!

7 Week Seven - Subroutines and Functions

7.1 Concept of 'Modular' Programming

All of the programs which have been examined so far, as well as those you have been asked to write yourself, have been fairly simple, short programs with a single basic purpose. However, frequently in programming, the requirement is for larger codes, comprising many different aspects and functions. It would be possible to write such a code in exactly the way in which we have been coding so far; however, there is a more suitable alternative. The meaning of the term 'Modular' programming is the process of dividing out a single code into smaller subsections, which can be comparitively isolated from each other, and then linked together to create the program as a whole.

There are many advantages to this method of program organisation. Firstly, modularisation helps readability; pieces of code with different purposes are separated into sections clearly designed to manage those specific purposes. It also aids the making of changes, if you want to change how a particular part of the code works, it requires only changing a single module of the code, assuming the code has been modularised sensibly.

7.2 Subroutines

7.2.1 Basics

In Fortran, subroutines are the basic unit that can be separated out to modularise a code. To illustrate how they work, we'll jump straight in with an example. Consider the following two codes:

```
1  PROGRAM errormessage_simple
2
3    IMPLICIT NONE
4
5    INTEGER :: k
6    REAL :: r
7
8    WRITE(*,*) 'Please Enter an Integer Between 1 and 10 (inclusive):'
9
10   READ(*,*) k
11
12   IF ((k.LT.1) .OR. (k.GT.10)) THEN
13     WRITE(*,*) 'An Error Has Occured, Program Quiting'
14     STOP
15   ELSE
16     WRITE(*,*) 'Your number was ', k
17   END IF
18
19   WRITE(*,*) 'Please Enter a Number to Take the Square Root of:'
20
21   READ(*,*) r
22
23   IF (r.LT.0) THEN
24     WRITE(*,*) 'An Error Has Occured, Program Quiting'
25     STOP
26   ELSE
27     WRITE(*,*) 'The Square Root of', r, 'is', SQRT(r)
28   END IF
29
30 END PROGRAM errormessage_simple
```

```
1  PROGRAM errormessage_subroutine
2
3    IMPLICIT NONE
4
5    INTEGER :: k
6    REAL :: r
7
8    WRITE(*,*) 'Please Enter an Integer Between 1 and 10 (inclusive):'
9
10   READ(*,*) k
11
12   IF ((k.LT.1) .OR. (k.GT.10)) THEN
13     CALL error
```

```
14    ELSE
15      WRITE(*,*) 'Your number was ', k
16    END IF
17
18    WRITE(*,*) 'Please Enter a Number to Take the Square Root of:'
19
20    READ(*,*) r
21
22    IF (r.LT.0) THEN
23      CALL error
24    ELSE
25      WRITE(*,*) 'The Square Root of', r, 'is', SQRT(r)
26    END IF
27
28  CONTAINS
29
30    SUBROUTINE error
31      WRITE(*,*) 'An Error Has Occured, Program Quiting'
32      STOP
33    END SUBROUTINE error
34
35  END PROGRAM errormessage_subroutine
```

In the first example, in each case where the input is invalid, there is a piece of code which determines this (the two `IF` statements) and then prints the message `'An Error Has Occured, Program Quiting'` and uses `STOP` to end the program.

In the second example, you will notice a number of differences. Instead of the code to print the message and exit, we have the single line '`CALL error`'. The second thing is that, prior to the `END PROGRAM` line, we have a sectioned off piece of code after the statement '`CONTAINS`'.

We will look at the final piece of the code, after '`CONTAINS`' first. `CONTAINS` instructs the program that the main code is over, but additional code, in the form of subroutines or functions, which the main program relies on, is about to follow. After this we have the construct:

```
SUBROUTINE subroutinename
!subroutine content
END SUBROUTINE
```

This construct acts like a mini program in itself, it begins with an opening statement (`SUBROUTINE`) and then a label for the subroutine (whatever `subroutinename` is), the same way as the `PROGRAM` statement works. It then finishes with `END SUBROUTINE subroutinename` in the same way that `END PROGRAM` works.

The code inbetween the opening and closing subroutine statements acts like the code for the mini program. In this case, we have the error message and `STOP` command.

With this defined, we now have the ability to run the content of this subroutine from any part of the main program. This is what the `CALL` command does. It is used in the form `CALL subroutinename`, and whenever it is found by the program, it runs the code found in that named subroutine.

7.2.2 Why is it Useful?

In this example, it may seem to be a pointless addition. However, say we now want to change the message that the error causes, from `'An Error Has Occured, Program Quiting'` to `'Input Invalid, Program Quiting'`; as that is more descriptive of the problem. We will also add the line '`WRITE(*,*) 'Dumping Stored Variables: k:', k, 'r:', r`', so that the values in memory for both of the variables are displayed when the program exits for this reason.

In the first example, `errormessage_simple`, we have to make these changes in two places, also making sure that we write them the same and correctly in both. In the version with subroutines, `errormessage_subroutine`, we only need to make the change in one place, the subroutine itself. Imagine that you have a code where there may be dozens of parameters to read in, and if one is invalid, you want to perform the same action whichever parameter it is that caused the problem. Perhaps you also want to open a file, write all of the stored parameters into that file, close it, and then exit.

If all of that is put in to each place that it may be needed, you end up with a very long code, with many repeated segments which need to be exactly the same to all work in the planned way. If you use a subroutine for it instead, the code becomes much more readable, as well as only needing one instance of the segment of code to be written.

If there is a task in your code which you will need to use many times, and would be the same or similar in all of those cases, putting it into a subroutine and calling that can be a much more effective way of writing it.

Most codes for complex tasks contain so many tasks to complete, and so much that needs to be repeated, that they will be very modularised to prevent the main code from becoming indecipherable. Frequently, the 'main code' part of a

program contains very little at all, just function and subroutine calls. This makes it much easier to follow the flow of the program and understand what is going on.

7.2.3 Arguments

In previous sections we have observed the use of 'arguments' to programs and functions. In a similar manner, arguments can be passed to subroutines and then used to affect the operation of that subroutine. This is shown in the very simple example below.

```
1  PROGRAM multimessage
2
3    IMPLICIT NONE
4
5    INTEGER :: k
6
7    WRITE(*,*) 'How many times would you like the message printed?:'
8
9    READ(*,*) k
10
11   CALL printmessage(k)
12
13 END PROGRAM multimessage
14
15 SUBROUTINE printmessage(num)
16
17   IMPLICIT NONE
18
19   INTEGER :: num,i
20
21   DO i = 1,num,1
22     WRITE(*,*) 'TEST MESSAGE'
23   END DO
24
25 END SUBROUTINE printmessage
```

Here we have initially written a message to the screen asking the user for a value, and then read that value to a variable `k`. On line 11 we use the `CALL` command to run the subroutine. However, unlike the previous examples, we have included the variable `k` as the argument to the subroutine call.

Looking at the code for the subroutine itself, we have also added the argument '`num`' to line 15, where the subroutine is named. Whatever is written here are whatever arguments the subroutine is expecting. Within the subroutine, we have also defined as an integer, a variable with the same name '`num`'. Therefore, the subroutine now knows it should be expecting one argument, of type `INTEGER`.

The subroutine itself will simply print a message '`num`' times, depending on the value passed to it as its argument. Note that, within the subroutine, the name '`num`' is used, but within the main code, the value passed to the subroutine is the `INTEGER k`. The program is just giving the subroutine the number stored in `k` without the subroutine 'seeing' the variable itself. When the subroutine is called, it needn't even be called with a variable, it can be given a number directly, as in:

```
CALL printmessage(15)
```

While this may seem a trivial technicality, it is important to understand that the subroutine only receives the value of a variable, not the variable itself. This means that if we have the line '`num = num + 1`' in our subroutine, and call the subroutine with '`CALL printmessage(k)`', when the subroutine runs, the change to `num` does *not* propagate back to `k`, `k` will remain completely unaffected by the actions on `num` in the subroutine, despite `num` getting its original value from `k`.

This means that the following piece of code would have *no effect*, and would not produce the output desired.

```
1  PROGRAM addition
2
3    IMPLICIT NONE
4
5    INTEGER :: num
6
7    num = 1
8
9    CALL addone(num)
10
11   WRITE(*,*) num
```

```
12
13 END PROGRAM addition
14
15 SUBROUTINE addone(num)
16
17   IMPLICIT NONE
18
19   INTEGER :: num
20
21   num = num + 1
22
23 END SUBROUTINE addone
```

Despite the variables outside and inside the subroutine both being named '`num`', they are stored in different locations. When the subroutine is given the main routines value for `num`, it creates its own new variable to store that value. When the subroutine adds one to its own `num` variable, *nothing happens* to the original `num` variable in the main routine.

There may be occasions when it is desired to have a subroutine use and/or modify the original variable passed to it. This can be done by using the `INTENT` attribute, covered in Section 7.2.4.

Also note that rather than having the subroutine inside the `PROGRAM` construct, follwing a `CONTAINS` statement as with the previous examples, these two subroutines have been defined outside of the program entirely. This is an important aspect to notice, as there is little real distinction between a piece of code being outside of the `PROGRAM` construct but in the same file, and being in another file entirely. The implication of this is that subroutines can be written in their own separate files and still called by the main program. This does rely on giving suitable instructions to the compiler to let it know where the other subroutines are, however.

7.2.4 The INTENT Attribute

In the previous section it was illustrated that if a variable is passed as simple input to a subroutine, the subroutine does not use that variable itself, but makes a copy with the same value. The result of this is that, using this method, a subroutine cannot alter any variable within the main routine. However, this method does have an alternative that permits doing just that.

The alternative method is to define the variable in the subroutine using the attribute '`INTENT`'. There are three variations of the `INTENT` attribute:

`INTENT(IN)`	Use of this attribute means that the variable is being passed from the main routine to the subroutine. It is the original variable which is used, but no changes are passed back to the main routine.
`INTENT(OUT)`	Use of this attribute means that the variable is to be passed back from the subroutine to the main routine. This permits the subroutine to output values in variables back to the main routine to use.
`INTENT(INOUT)`	Use of this attribute means that the variable is being passed from the main routine to the subroutine *and* any changes made will be passed back to the main routine too.

To begin with we will modify the final example in the previous subsection using the `INTENT` attribute to make it function correctly.

```
1  PROGRAM addition_intent
2
3    IMPLICIT NONE
4
5    INTEGER :: num_main
6
7    num_main = 1
8
9    CALL addone(num_main)
10
11   WRITE(*,*) num_main
12
13 END PROGRAM addition_intent
14
15 SUBROUTINE addone(num_sub)
16
17   IMPLICIT NONE
18
19   INTEGER, INTENT(INOUT) :: num_sub
20
21   num_sub = num_sub + 1
```

```
22
23 END SUBROUTINE addone
```

In this example, the only major change is on line 18, where the attribute '`INTENT(INOUT)`' is assigned to the variable. Additionally, the names of the two variables have been altered to more easily label what happens when the program runs.

When the subroutine is called on line 9, it is passed the variable `num_main`. Although inside the subroutine, the name of the variable is different, because it represents the same argument to the subroutine, and is labelled with `INTENT(INOUT)`, it is the same 'slot' in memory being used by both. Therefore, at this point, `num_main` and `num_sub` are effectively the same variable. Any changes made to the variable in the subroutine will carry back to the main routine as well.

This has illustrated the `INTENT(INOUT)` attribute, but what of `INTENT(IN)` and `INTENT(OUT)`? `INTENT(IN)` may seem to be somewhat pointless. If not using an `INTENT` attribute at all lets you just use the value of a variable passed to the subroutine, and `INTENT(IN)` does the same, why use it? The difference between the two is that when no `INTENT` attribute is being used, effectively, a new variable with a copy of the passed variables value is being made whenever one is passed to that subroutine. If there is no need to change the value, this just means that extra memory is being used for no good reason. If the thing being passed to the subroutine is a large amount of data, then that could constitute a considerable waste of memory. Unlike this, `INTENT(IN)` causes the subroutine to reference the same location in memory, the same variable. This takes up no extra space, as no copy is created.

`INTENT(OUT)` can be used to produce 'new' output from a subroutine to give back to the main routine. This can be seen in the following example, along with use of the `INTENT(IN)` attribute as well.

```
1  PROGRAM sum_and_difference
2
3    IMPLICIT NONE
4
5    INTEGER :: A, B, summed, difference
6
7    WRITE(*,*) 'Input two integers:'
8
9    READ(*,*) A, B
10
11   CALL findsumdif(A, B, summed, difference)
12
13   WRITE(*,*) 'Sum = ', summed
14   WRITE(*,*) 'Difference = ', difference
15
16 END PROGRAM sum_and_difference
17
18 SUBROUTINE findsumdif(x, y, tot, dif)
19
20   IMPLICIT NONE
21
22   INTEGER, INTENT(IN) :: x, y
23   INTEGER, INTENT(OUT) :: tot, dif
24
25   tot = x + y
26   dif = ABS(x - y)
27
28 END SUBROUTINE findsumdif
```

This example finds the sum and difference of two `INTEGER`s entered by the user. The two input `INTEGER`s in the main routine, `A` and `B`, are given to the subroutine as the first two arguments, labelled by the subroutine as `x` and `y`. Despite their differing names, because `x` and `y` have the `INTENT(IN)` attribute (on line 22), within the subroutine they refer directly to the original `A` and `B` variables without having created copies.

The two other variables in the main routine, `summed` and `difference`, are not given a value initially, and passed to the subroutine as the final two arguments whilst uninitialised. In this function they are basically acting as 'placeholders', slots in memory known by the main rountine, waiting to be given values by the subroutine.

When the subroutine is run, the final two arguments (called `tot` and `dif` in the subroutine) are given the `INTENT(OUT)` attribute (line 23). This means that they reference the same location in memory as the original `summed` and `difference` variables in the main routine, but ignore whatever value they may already have had, they exist only for output from the subroutine.

Then the mathematical routines are performed to actually find the sum and difference of the two values (note that '`ABS`' is an intrinsic maths function, meaning '*abs*olute', which finds the 'absolute value' or modulus of the argument). Once the subroutine has ended and the flow of the program returns to the main routine, nothing has changed with `A` and `B` (regardless of anything done to `x` and/or `y` in the subroutine), but `summed` and `difference` now have the results

calculated by the subroutine.

7.2.5 Separate Files

In the previous section, the concept of splitting the main rountines and subroutines into different files was covered. This is very simple to do when using code separated out as shown in Section 7.2.4. In this section we will use the same code as before (for calculating the sum and difference between two `INTEGER` values, but house the code in two different files. The entire example process will be run through, including filenames, in order to illustrate the compiling process as well.

Our first file is called '`sumdiff_main.f90`', and contains the main program from the previous example (lines 1 to 16 of the final piece of code in Section 7.2.4). Our second file is called '`sumdiff_sub.f90`' and contains the code for the subroutine (lines 18 to 28 in the final code example in Section 7.2.4).

We now have two files, one possessing the content within a `PROGRAM` - `END PROGRAM` construct, and the other with content between a `SUBROUTINE` - `END SUBROUTINE` construct. The regular compile command, specifying the single source code file and the output name, will not work. The following would fail:

```
:> gfortran -o sumdiff sumdiff_main.f90
```

giving an error stating: '`undefined reference to 'findsumdif_'`' along with '`ld returned 1 exit status`'. `ld` is the '**linker**', which is the stage after the source is compiled to '**object code**', at which point the interacting pieces of code are connected ('linked') together. In this case, no source code for the subroutine was made into object code, and so the linker cannot find any subroutine with the correct name to link to.

Instead, when compiling multiple files, they must simply all be listed as source files when compiling, as follows:

```
:> gfortran -o sumdiff sumdiff_main.f90 sumdiff_sub.f90
```

This time, the program should compile successfully, using the content of both source files to create the final binary code. Simple!

7.2.6 Quadratic Formula Subroutine - Assignment Material

The object of this assignment is to produce a suitable subroutine independant of a main program, based on a specified set of requirements. The main program should be obtained by using the following command (while in whichever directory you plan to work in):

```
:> cp ~ph302/week7prog.o ./
```

which will copy the main program object code from the admin directory to yours. Do not try to open this file in an editor, it is not a source file. It is provided in the form of an '**object**', mentioned at the end of the previous section. It is effectively partially compiled code, waiting to be linked together with other necessary bits of code to finally produce an executable binary file. Your task is to fulfil the requirements of the subroutine such that it will work with an already existant program which has not been designed by you. Fitting your own programs into a given existing standard is a frequently required skill within programming. There will frequently be cases where it may be required to integrate code of your own design into another program, the source of which you don't wish to alter, so as to retain compatibility with its separate development.

The intention of the subroutine is to solve a quadratic equation. Your subroutine name should be '`sevensub`' (it must be called this exactly, or the main program will not be able to call it), and should be written in a file you should call '`assign_7_1.f90`'. You should be able to use some parts of your code for previous assignments based on the same mathematical processes. Your program should receive three input variables, which will represent a, b and c from the equation $y = ax^2 + bx + c$. All three are of type `REAL`. You can give the variables whatever names you consider most useful, but they must be (in order a, b, c) the first three arguments to your subroutine. When defined within the subroutine, they must have the tag '`INTENT(IN)`', as they are input variables.

In addition to receiving this input, your subroutine must provide three output variables. $x_{\text{solutions}}$, x_{plus} and x_{minus} in that order. $x_{\text{solutions}}$ is an `INTEGER` variable, indicating the number of solutions to the quadratic equation. x_{plus} should be of type `REAL` and is either the sole solution to x if only one solution exists, or the 'plus the square root' solution if two exist. x_{minus} should also be of type `REAL` and is either empty if no solutions or only a single solution exist, or the 'minus the square root' solution if two solutions exist. When defined within the subroutine, these variables must have the tag '`INTENT(OUT)`', as they are output variables.

Your resulting starting line for your subroutine should look as follows (the square brackets, and text within, *describes* what should appear at that position, neither the square brackets themselves, nor the brief description inside is what you should actually write!):

`SUBROUTINE sevensub(`[variable for 'a'],[variable for 'b'],[variable for 'c'],[variable for '$x_{\text{solutions}}$'],[variable for 'x_{plus}'],[variable for 'x_{minus}'])

To summarise the output, there are three possible combinations:

1. No real solutions for x exist: The discriminant in the quadratic formula ($b^2 - 4ac$) is less than zero, resulting in

no real value for x. Your variable for $x_{\text{solutions}}$ should equal `0`, and your variables for x_{plus} and x_{minus} can be empty/unassigned.

2. Only a single solution for x exists. In the equation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ the discriminant ($\sqrt{b^2 - 4ac}$) is zero, resulting in a single solution for x of $x = \frac{-b}{2a}$. In this case your variable for $x_{\text{solutions}}$ should equal `1`, your variable for x_{plus} should be set as the value for x and your variable for x_{minus} can be empty/unassigned.
3. Two solutions exist for x. The discriminant is greater than zero, giving two solutions for x, you variable for x_{plus} should be the $x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ solution, and your variable for x_{minus} should be the $x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ solution. Your variable for $x_{\text{solutions}}$ should be `2`.

Based on this input and output, the main program will perform a number of tasks using the information provided to it. Once you have programmed this task into your subroutine, compilation should be performed as follows (while in the directory containing both the supplied '`week7prog.f90`' file and your own '`assign_7_1.f90`' file):

```
:> gfortran -o assign_7_1 week7prog.o assign_7_1.f90
```

This will compile your source code and combine it with the object code of the main program.

Having completed this step, you should run your newly compiled executable, and if your subroutine is correctly programmed, the executable should print to the screen a message stating: '`Your subroutine appears to have run successfully`'. If your subroutine operates, but provides values which are incorrect to the main program, you will receive the response: '`Your subroutine has been called, but provided incorrect outputs`', followed by a short piece of information describing the area(s) which your program failed to provide correct values for. If any message other than these two is supplied to the screen, your subroutine has likely caused a crash. Either the subroutine itself crashed, or caused a crash in the main program by failing to be correctly called.

Once you have ensured that you subroutine works correctly, by receiving the '`Your subroutine appears to have run successfully`' message; you should also notice that an output file, '`assign_7_1.out`', will have been created in the directory you ran your program from. You will be required to hand in a printout of both your source code (file `assign_7_1.f90`) and this output file.

7.3 Functions

7.3.1 Basics

You have already come across the idea of functions whenever using any of the mathematical 'intrinsic functions' built into Fortran. `SIN`, `SQRT` and similar are examples of these intrinsic functions. User created functions operate in almost exactly the same way. They are intended to be supplied with some manner of input, in order to produce a specific output. Unlike subroutines, which are separately called, functions tend to be of the form where they can be incorporated into statements within a line (think of using the `SIN` function within a line with other maths).

To explain the operation of a function, we will examine the following example. The intention here is for the function '`magnitude`' to provide the vector magnitude of two orthogonal vector components. In other words, finding the hypotenuse in a right angled triangle based on the two other sides, `A` and `B`. To further demonstrate how functions operate, it is then used in the context of other mathematical operators on a single line.

```
1  PROGRAM pythagoras
2
3    IMPLICIT NONE
4
5    REAL :: A, B, magnitude, C
6
7    A = 3
8
9    B = 4
10
11   C = magnitude(A, B)
12
13   WRITE(*,*) 'size of hypotenuse', C
14
15   C = 2 * magnitude(A, magnitude(A,B))**2
16
17   WRITE(*,*) 'functions also work within other maths operators', C
18
19 END PROGRAM pythagoras
20
```

```
21 REAL FUNCTION magnitude(x, y)
22
23   IMPLICIT NONE
24
25   REAL, INTENT(IN) :: x, y
26
27   magnitude = SQRT(x**2 + y**2)
28
29 END FUNCTION magnitude
```

The first difference you will notice from a subroutine, is that unlike the definition '`SUBROUTINE name`', a function is defined with a variable type on the same line (line 21). This is because it is the job of a function to return a certain variable type as its answer. The variable type of the function is effectively also the type of variable that the function itself is. Since any function is expected to return a value, it is treated as a variable within the code. It will always be evaluated first (unless there is another function nested within its arguments), and from the point at which it is evaluated onwards, exists to all intents and purposes purely as a value, and hence requires a variable type.

The function name also acts as a variable within the code for the function itself. It is to this 'variable' which you must eventually assign the desired output before the function exits, as shown on line 27, where `magnitude` is set equal to the desired result '`SQRT(x**2 + y**2)`'.

This also results in the need to explicitly define the function within the main program, in the same manner in which a variable is defined. This can be seen on line 5, where the function name ('`magnitude`') is included along with the other `REAL` variables the program will use.

Line 11 shows the function being used to produce a result, with the two variables `A` and `B` acting as its arguments. Note that, as with subroutines, the names of the variables passed as arguments in the main program (`A` and `B`), do not have to match the internal names of the variables used within the function (`x` and `y`). Note that the use of '`INTENT(IN)`' on line 25, where the variables of the function are defined, is not vital, but is encouraged. This is for the same reasons as described in Section 7.2.4.

The next example of the use of the function is on line 15, where a set of mathematical operations are performed, along with an instance of nesting, where one of the arguments to the function, is the function used with two other arguments. In this case the inner function (`magnitude(A,B)`) is evaluated first, found to be `5` (the same as `C` previously was). Then this new value is used to evaluate the outer function (`magnitude(A, magnitude(A,B))`) the returned value for which is then inserted into the remaining mathematical operations in the expected order (squared first and then multiplied by two).

7.3.2 Alternatives with Functions

The method demonstrated in the previous subsection relates to the general form of including functions, which permits the functions to be scattered across files and included from anywhere. An alternative exists, which is simpler, but limits the scope over which the function can be used. This method is to use the '`CONTAINS`' command which was examined in relation to subroutines. In this layout, the code from the previous subsection would look as follows:

```
1  PROGRAM pythagoras
2
3    IMPLICIT NONE
4
5    REAL :: A, B, C
6
7    A = 3
8
9    B = 4
10
11   C = magnitude(A, B)
12
13   WRITE(*,*) 'size of hypotenuse', C
14
15   C = 2 * magnitude(A, magnitude(A,B))**2
16
17   WRITE(*,*) 'functions also work within other maths operators', C
18
19 CONTAINS
20
21   REAL FUNCTION magnitude(x, y)
22
23     IMPLICIT NONE
24
```

```
25      REAL, INTENT(IN) :: x, y
26
27      magnitude = SQRT(x**2 + y**2)
28
29    END FUNCTION magnitude
30
31  END PROGRAM pythagoras
```

In this instance, the function is included directly within the `PROGRAM` construct, after the `CONTAINS` command. Also note how it is *not* necessary to define the function name in the variable definitions on line 5. This is the only real advantage of this method, which is useful for smaller, self contained programs, where expansion to multiple files is not expected.

Nevertheless, it is a method worth being aware of, as it is quite possible that you may come across a code using this format.

The next alternative available is the form of the `FUNCTION` construct. In the previous examples, the functions have had their variable type defined on the line with the `FUNCTION` construct and function name itself. The function type can be defined within the function instead, as follows:

```
1  FUNCTION magnitude(x, y)
2
3    IMPLICIT NONE
4
5    REAL :: magnitude
6    REAL, INTENT(IN) :: x, y
7
8    magnitude = SQRT(x**2 + y**2)
9
10 END FUNCTION magnitude
```

Note how, on line 5, '`magnitude`' has been included as a regular variable definition. Also important to note is that, while the '`magnitude`' variable could be described as being 'output', it should *not* have the `INTENT(OUT)` attribute. The specifics which define variables for the use of `INTENT(OUT)` do not apply to function names.

7.3.3 Circles and Spheres - Assignment Material

There is little more to cover on the fundamentals of functions. Apart from the points described so far, their operation is similar enough to subroutines that you should be able to fill in any gaps yourself. The object of this assignment is to write several functions relating to the calculation of properties of a circle and a sphere. These will then be coupled with an example program which you will need to copy from the code segment displayed later in this section. Initially, create your own source file named '`assign_7_2_sub.f90`' within which you will write the functions as described by the specified requirements.

Important: It is necessary to know that the result of any of the intrinsic mathematical operators depends on what its operands are (the values that it is operating on). A single number, with no decimal place ('`3`' for example), will be automatically interpreted as an `INTEGER`; one with a decimal place will be assumed to be a `REAL` (`3.0` for example). So if you have an expression `3 + 5`, both operands, `3` and `5` are `INTEGER`s, therefore the result of the addition (the operation) will also be an `INTEGER`. This knowledge is vital in some circumstances. Consider the operation `3 / 4`, we would presume this would give us `0.75`, however it would not. As both the `3` and the `4` are expressed as `INTEGER`s, integer division will be performed. In other words, how many whole times `4` will go into `3`. For this example, the result of `3 / 4` will be `0`, 4 cannot be made to 'go into' 3 an integer number of times. This could be fixed by phrasing the operation as `3.0/4.0` instead; with both operands interpreted as `REAL`s, the result will also be `REAL` and give `0.75`.

- A function named '`circ_area`': This function should accept one argument, the radius, r, of a circle as a type `REAL` variable. It should return the area of a circle of that radius.
- A function named '`circ_perim`': This function should accept one argument, the radius, r, of a circle as a type `REAL` variable. It should return the perimeter or circumference of a circle of that radius.
- A function named '`sphere_area`': This function should accept one argument, the radius, r, of a sphere as a type `REAL` variable. It should return the surface area of a sphere of that radius.
- A function named '`sphere_volume`': This function should accept one argument, the radius, r, of a sphere as a type `REAL` variable. It should return the volume of a sphere of that radius.

Once you have created these functions, you are required to make a new file, named '`assign_7_2_main.f90`', which will contain your main program. The task of this program will be to calculate the following:

- The user should be prompted to enter a value for a radius, which should be read into a `REAL` type variable.
- The program should `OPEN` an output file, the name of which should be '`assign_7_2.out`'.
- Using your previously created functions, the program should then calculate the following quantities to be written to the output file in this order:
  1. The area of a circle of the specified radius.
  2. The perimeter of a circle of the specified radius.
  3. The surface area of a sphere of the specified radius.
  4. The volume of a sphere of the specified radius.
  5. The quantity $A_{\text{sphere}}(r) - A_{\text{circle}}(r)$, where $A_{\text{sphere}}(r)$ is the surface area of a sphere of radius r, $A_{\text{circle}}(r)$ is the area of a circle of radius r, and r is the radius input by the user.
  6. The quantity $V_{\text{sphere}}(P_{\text{circle}}(r))$. Where V_{sphere} is volume of a sphere, P_{circle} is the perimeter of a circle. In other words, the calculated quantity should be the volume of a sphere whose radius is as long as the perimeter of a circle of radius r.
- Having output all of these quantities to the file, the program should exit.

7.4 Assignments Summary

7.4.1 Programming Assignment 1

1. Using the command
   ```
   :> cp ~ph302/week7prog.o ./
   ```
   Obtain a copy of the object code for the main program which you will be writing a subroutine for.
2. Create a new file for your code called '`assign_7_1.f90`'.
3. Within your code file, define a subroutine called '`sevensub`' (your subroutine name must match this exactly in order to work correctly).
4. Your subroutine must accept three input variables, the a, b and c coefficient values (in that order), for a quadratic equation of the form $ax^2 + bx + c = 0$. All must be of type `REAL`.
5. The subroutine must be set up to return three output variables, $x_{\text{solutions}}$, x_{plus} and x_{minus} (in that order). $x_{\text{solutions}}$ should be an `INTEGER` variable; x_{plus} and x_{minus} should each be a `REAL` variable.
6. The subroutine must then find the discriminant of the quadratic equation $(b^2 - 4ac)$ in order to determine the number of solutions. This is used to set the $x_{\text{solutions}}$ variable to the correct value.
7. Based on the determined number of solutions, the program must subsequently do one of three things:
   - If there are **no** solutions, the program should set $x_{\text{solutions}}$ to `0`. x_{plus} and x_{minus} may be left undefined, or set to zero. It should then exit, returning these values to the main routine.
   - If there is **one** solution, the program should set $x_{\text{solutions}}$ to `1`. The single solution for the quadratic equation should then be found, and x_{plus} should be set as that value. x_{minus} may be left undefined, or set to zero. The subroutine should then exit, returning these values.
   - If there are **two** solutions, the program should set $x_{\text{solutions}}$ to `2`. Both solutions should then be found, with x_{plus} set as the value for the 'plus square root of the discriminant'; and x_{minus} set as the value for the 'minus square root of the discriminant'. Again, the subroutine should then exit, returning these values.
8. Remember to correctly end your subroutine, and to comment your code.
9. To compile using the supplied object code for the main routine, use the command:
   ```
   :> gfortran -o assign_7_1 week7prog.o assign_7_1.f90
   ```
   Note: Both the `week7prog.o` and your own `assign_7_1.f90` must be in the same directory for this to work.
10. Check that your code performs as expected using the provided output when running the compiled program. It will inform you whether you found the correct values or which value/variable did not appear to be correct for a series of test problems.
11. Running the program will create an output file `assign_7_1.out`; this, along with your source code, should be printed out to be submitted.

7.4.2 Programming Assignment 2

1. Create a new file for your main code called '`assign_7_2_main.f90`' and one for your functions, which should be called '`assign_7_2_sub.f90`'.
2. Within the file for your functions, `assign_7_2_sub.f90`, you need to define four functions, as specified below:
   Hint: Remember to consider the variable type that your functions should be.
   - A function named '`circ_area`': This function should accept one argument, the radius, r, of a circle as a type `REAL` variable. It should return the area of a circle of that radius.
   - A function named '`circ_perim`': This function should accept one argument, the radius, r, of a circle as a type `REAL` variable. It should return the perimeter or circumference of a circle of that radius.
   - A function named '`sphere_area`': This function should accept one argument, the radius, r, of a sphere as a type `REAL` variable. It should return the surface area of a sphere of that radius.
   - A function named '`sphere_volume`': This function should accept one argument, the radius, r, of a sphere as a type `REAL` variable. It should return the volume of a sphere of that radius.
3. Within the file for your main code, `assign_7_2_main.f90`, you must define your main program, which should accomplish the following tasks:
   (a) The user should be prompted to enter a value for a radius, which should be read into a `REAL` type variable.
   (b) The program should `OPEN` an output file, the name of which should be '`assign_7_2.out`'.
   (c) **Using your previously created functions**, the program should then calculate the following quantities to be written to the output file in this order:
      i. The area of a circle of the specified radius.
      ii. The perimeter of a circle of the specified radius.
      iii. The surface area of a sphere of the specified radius.
      iv. The volume of a sphere of the specified radius.
      v. The quantity $A_{\text{sphere}}(r) - A_{\text{circle}}(r)$, where $A_{\text{sphere}}(r)$ is the surface area of a sphere of radius r, $A_{\text{circle}}(r)$ is the area of a circle of radius r, and r is the radius input by the user.
      vi. The quantity $V_{\text{sphere}}(P_{\text{circle}}(r))$. Where V_{sphere} is volume of a sphere, P_{circle} is the perimeter of a circle. In other words, the calculated quantity should be the volume of a sphere whose radius is as long as the perimeter of a circle of radius r.
   (d) Having output all of these quantities to the file (`assign_7_2.out`), the program should exit.
4. You should produce output from your program using an input value of 2.5.

7.4.3 Questions

For this week the questions are as follows:

1. When passing a variable back from a subroutine to the main program, what attribute must that variable be defined with?
2. Name two ways in which a `FUNCTION` differs from a `SUBROUTINE`.
3. State two advantages to the use of `FUNCTION`s and/or `SUBROUTINE`s, rather than the equivalent written as embedded code in your main program.

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_7_1.f90`, `assign_7_1.out`, `assign_7_2_main.f90`, `assign_7_2_sub.f90` and `assign_7_2.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

8 Week Eight - Characters, Strings and Formatting

8.1 ASCII

You may have come across the concept of character encoding before. Character encoding is the system by which numbers stored by a computer can be associated with symbols intended for display to a screen or printer. A common basic encoding scheme is '**ASCII**' (American Standard Code for Information Interchange). This format is now extremely old, dating back to the 1960's in use in telegraph systems. However, due to desire for backwards compatibility and the fundamental nature of the basic character set, the ASCII system of characters still sees use today. It is also convenient as a teaching tool, as, with only 128 different character symbols, it is short enough to describe the entire set; unlike modern standards such as Unicode, which currently (Version 5.2) includes definitions for 107,361 characters[19]. Additionally, some implementations of modern standards utilise variable-length systems of encoding, making them more difficult to understand. The Unicode implementation UTF-8 retains compatibility with ASCII, the first 128 characters match the ASCII characters. Note that Unicode is the standard for the characters used, but does not constitute an implementation for computers in itself, it is standards such as the aforementioned UTF-8 which are the actual implementations of Unicode.

Below is a table illustrating the ASCII symbols:

	0	16	32	48	64	80	96	112
0	[NUL]	[DLE]	[SP]	0	@	P	`	p
1	[SOH]	[DC1]	!	1	A	Q	a	q
2	[STX]	[DC2]	"	2	B	R	b	r
3	[ETX]	[DC3]	#	3	C	S	c	s
4	[EOT]	[DC4]	$	4	D	T	d	t
5	[ENQ]	[NAK]	%	5	E	U	e	u
6	[ACK]	[SYN]	&	6	F	V	f	v
7	[BEL]	[ETB]	'	7	G	W	g	w
8	[BS]	[CAN]	(	8	H	X	h	x
9	[HT]	[EM]	)	9	I	Y	i	y
10	[LF]	[SUB]	*	:	J	Z	j	z
11	[VT]	[ESC]	+	;	K	[	k	{
12	[FF]	[FS]	,	<	L	\	l	|
13	[CR]	[GS]	-	=	M	]	m	}
14	[SO]	[RS]	.	>	N	^	n	~
15	[SI]	[US]	/	?	O	_	o	[DEL]

Table 1: ASCII 1967 Definitions. Column numbers indicate the value of the first element of that column. Rows indicate the offset from that first element. ie. 'e' would be column value 96 plus row value 5 = 101. All names enclosed in square brackets are the code names for command characters.

As can be seen, the code is fairly straightforward. For a given ASCII value, there is a corresponding symbol. The aspect which may appear confusing are those 'symbols' which are a set of letters within square brackets. These are 'command characters' which perform functions, rather than just display characters. For example, `[LF]`, ASCII value 10, is 'line feed', which machinary or computers could be coded to understand as shifting to the next line to be printed. It is still this value used to indicate where line breaks occur within Unix based systems. Interestingly `[CR]`, ASCII value 12, is 'carriage return', a concept on typewriters for example to physically move the printing carriage back to the left of the page; it is *this* ASCII value which was used for Mac PC's up until OSX. Windows uses *both* symbols to create a new line, in the order `[CR]` → `[LF]`. For this reason, when transferring text files between operating systems, the line breaks will occasionally fail to implement properly on the OS which the file was not written in.

Other notable command characters still in use are ASCII value 8, `[BS]`, which is a backspace; ASCII value 9, `[HT]`, which is a horizontal tab; ASCII value 32, `[SP]`, which is a space and ASCII value 127, `[DEL]`, which is delete. The majority of the other command characters no longer perform any function on modern computers in the context of a simple character encoding set. All have origins dating back to functions needed for teletype machines to establish connenctions and operate, not all of which found equivilent or alternate uses in text display.

The purpose of introducing you to the ASCII system is to impart an understanding of what the computer does when instructed to store text. The successful implementation of storing text as strings relies upon knowing the nature in which this storage is implemented.

For reference, as well for if you want to experiment with direct interaction between Fortran and ASCII character encoding, there are two useful intrinsic functions for this purpose. The command '`ACHAR(int)`' (where '`int`' is an `INTEGER` variable) will return the single ASCII character corresponding to the value of `int`. So '`ACHAR(97)`' would return a lowercase '`a`'. The inverse of this command is '`IACHAR(char)`' (where '`char`' is a `CHARACTER` variable). This will return the ASCII value for the character passed to the command. So '`IACHAR('a')`' would return '`97`'.

8.2 Character Variables and Creating Strings

By this point you should appreciate that all a stored letter is to a computer is a number as well as something to indicate that when this number is to be retrived, it should be processed through a pre-set code such as ASCII. The values themselves have already been described, the additional 'note' by which the computer knows to decode in a certain way is the variable type. For text, this variable type is a '`CHARACTER`' variable.

On its own, a `CHARACTER` variable will store a single ASCII value, describing the letter which is to be stored. The following code fragment illustrates how this process would work:

```
CHARACTER :: mychar

mychar = 'a'

WRITE(*,*) mychar
```

The result of this fragment would be to print the letter 'a' to the screen, equivilent to if one had used '`WRITE(*,*) 'a'`' directly, but in a manner which stores the character used. Storing single letters at a time is not necessarily useful, however. As such, it is possible to create a variable which is an entire sequence of `CHARACTER` variables. These collections are called 'strings'. Strings are not a variable type themselves as such, but a term for a certain construct of the `CHARACTER` variable itself (within Fortran, at least). Defining a string with 44 spaces for letters would be performed as follows:

```
1  PROGRAM string_simple
2
3    IMPLICIT NONE
4
5    CHARACTER*44 :: mystring
6
7    mystring = 'The quick brown fox jumps over the lazy dog'
8
9    WRITE(*,*) mystring
10
11 END PROGRAM string_simple
```

In this case, the text which was placed in the string has exactly the same number of characters as the variable had been defined with, 44. The length of a piece of text is not always known in advance, however. For the purpose of storing text, particularly text which is to be entered by the user or read from a file, a solution is to make the length of your `CHARACTER` variable as long as you anticipate needing. For example, if creating a variable which will contain a users name, you might define it as follows: '`CHARACTER*256 :: username`'. This would provide you with 256 characters of leeway for the users name, hopefully enough that it would not overrun the bound. On that matter, of text longer than a variable expects, it would typically be the case to put in place a check, ensuring that the limit for the variable is not exceeded by the input. By default for most input in fortran, if an attempt is made to read a string longer than the limit of the variable, it will simply cut off when that limit is reached. The resulting string will be truncated, but should not cause a crash.

Note that, in the example of a suitable variable to store a name, a length of 256 was used. Frequently values corresponding to powers of 2 have tended to be used for sizes (2^8 in the case of 256). This is not essential, mostly being a hold over from when the actual processing and storing of data was made easier by working in machine units.

Having decided on a length of 256 for our user name variable, we need to consider what happens when the amount of text entered is less than that total. The following piece of code asks for the user to input their name, and then writes back to the screen the contents of the `username` variable.

```
1  PROGRAM username_simple
2
3    IMPLICIT NONE
4
5    CHARACTER*256 :: username
6
7    WRITE(*,*) 'Please input user name'
8
9    READ(*,*) username
10
11   WRITE(*,*) username
12
13 END PROGRAM username_simple
```

If you code this (you can, but it's not mandatory), and input a short name, say '`anon`', the output from the code will be '`anon`' followed by 252 empty spaces (likely spanning several lines, depending on how many characters wide your

terminal is). The full 256 characters will be printed to screen, although only the first four have any symbols to display. The correction for issues such as this is with the use of string '**formatting**'.

8.3 Formatting and Manipulation of Strings

8.3.1 TRIM Command

The trailing spaces in long string variables with short text in may not be a problem, nor is it necessarily desired either. The quickest way to circumvent this issue is to use the function '`TRIM`'. This command returns a string with all trailing spaces removed. Hence `TRIM('a    ')` (the letter 'a' with five spaces) would return the string `'a'`. Applying this to the previous example with the username, instead of '`WRITE(*,*) username`', using '`WRITE(*,*) TRIM(username)`' would return just the '`anon`' entered by the user.

The issue with this method is that it is determining the length of the 'real' string purely by the offset of the trailing spaces. If you have a string which ends in a space, it will not be able to distinguish the desired space from the remaining trailing spaces. As such, it is important to properly consider the context in which the string is being/will be used. The following example code illustrates the use of strings and the `TRIM` command to pick a filename for output interactively:

```
1  PROGRAM filenaming
2
3    IMPLICIT NONE
4
5    CHARACTER*256 :: filename
6    CHARACTER*6 :: choice
7    LOGICAL :: selectionvalid
8    INTEGER :: i, strikes
9
10   WRITE(*,*) 'Please choose a filename for output:'
11
12   READ(*,*) filename
13
14   WRITE(*,*) 'The filename you have selected is:'
15   WRITE(*,*) TRIM(filename)
16   WRITE(*,*) 'Are you sure you want to output to this file? [y/n]'
17
18   CALL continuechoice()
19
20   WRITE(*,*) 'Opening file'
21
22   OPEN(10,FILE=TRIM(filename))
23
24   WRITE(*,*) 'Writing to file'
25
26   DO i=0,100,5
27     WRITE(10,*) i
28   END DO
29
30   CLOSE(10)
31
32 END PROGRAM filenaming
33
34 SUBROUTINE continuechoice
35
36   IMPLICIT NONE
37
38   CHARACTER*6 :: choice
39   LOGICAL :: selectionvalid
40   INTEGER :: strikes
41
42   selectionvalid = .FALSE.
43   strikes = 0
44
45   DO WHILE (selectionvalid .EQV. .FALSE.)
46     READ(*,*) choice
47     SELECT CASE (choice)
48       CASE ('y','Y','yes','Yes','YES')
49         WRITE(*,*) 'Confirmed, continuing with output'
50         selectionvalid = .TRUE.
```

```
51        CASE ('n','N','no','No','NO')
52          WRITE(*,*) 'Filename rejected, ending program without outputting'
53          STOP
54        CASE DEFAULT
55          WRITE(*,*) 'Input not understood, please try again:'
56          strikes = strikes + 1
57          IF (strikes .GE. 3) THEN
58            WRITE(*,*) 'Too many incorrect responses. Exiting.'
59            STOP
60          END IF
61      END SELECT
62    END DO
63
64  END SUBROUTINE continuechoice
```

This example showcases a number of the techniques that you have been shown over the progression of the course. The program begins with the conventional definitions of all of the variables.

The few lines subsequent to the variable definitions should also be fairly understood by this stage. The user is prompted to enter a filename, the filename is read and then repeated back to the screen. The user is then prompted to confirm whether to continue with the filename they have entered. This is the point at which newer material comes into play.

At this stage the subroutine '`continuechoice`' is called. The code for which starts on line 34. Within this subroutine we have three variables defined. '`choice`' is a short string `CHARACTER` variable, which is for holding the users typed response. '`selectionvalid`' is a checking variable, for the subsequent `DO WHILE` loop (this is initially set to `.FALSE.` on line 42), and '`strikes`' is an `INTEGER` variable for counting how many times the user enters an unrecognised response (initially set to zero on line 43).

The main content of the subroutine is a `DO WHILE` loop, which is set up to continue looping until the variable `selectionvalid` ceases to be `.FALSE.`. This means that for any user response that is understood, we can set `selectionvalid` to `.TRUE.` meaning that the loop will end. If the user response is not understood, `selectionvalid` will remain unchanged and hence the request for user input will repeat.

The first `CASE` instance (line 48) is for if the user does wish to continue using the selected filename. If the user response, stored in `choice`, matches any of the synonyms for 'yes' listed on line 48, the `CASE` statement is executed, writing a message confirming to the screen and switching the `selectionvalid` variable to `.TRUE.` to stop the loop from repeating again.

The next `CASE` instance (line 51) is for if the user does *not* wish to continue. Again, any response matching the synonyms for 'no' will trigger this `CASE`, which will state that the program will not continue, and then exit the program with the `STOP` command.

The final `CASE` instance (line 54) is the special '`CASE DEFAULT`' statement. If the user response stored in `choice` hasn't matched the 'yes' case or the 'no' case, it is this `CASE DEFAULT` instance which is executed. Here, a message is written to screen informing the user that the response is not understood, then it increments the '`strikes`' `INTEGER`. It then checks whether `strikes` is at or above `3`, ie the user has already entered three responses which are not understood. If so, a message is printed and the program is exited.

So, if the user types one of the supported varients of 'no', or if they type unsupported letters/words three times, the program exits. If the user types one of the varients of 'yes' the loop ends, the subroutine is allowed to exit successfully, and the 'flow' of the program returns to the main code.

Returning to the main code at the point where the subroutine was called, the remaining line of interest is line 22, '`OPEN(10,FILE=TRIM(filename))`'. In this case we open a file for output, however, rather than use a hard coded name by entering the string in inverted commas after the `FILE=` statement, we reference the `TRIM`med contents of the `filename` variable.

The final part of the code is a simple loop to output the multiples of 5 from 0 to 100 to the file, providing some output.

Important Note: Whenever referencing the contents of a string which is longer than needed to be for it's stored content, ie a filename which has been read in or similar, make sure to use `TRIM` when using that string. Using the string on it's own for further operations will otherwise *still include* the trailing spaces and could cause major problems with subsequent operations.

8.3.2 LEN Command and LEN_TRIM Command

One property which is useful to know for strings is how many actual characters the string contains. The function for performing this task is `LEN()`. Used on a string directly, it will return the entire length of the string as an `INTEGER`. When used in conjuction with `TRIM`, it can tell you the number of actual characters within the string which are not simply

trailing white space. So:
`LEN('This is text  ')`, would return 20, including the white space at the end.
`LEN(TRIM('This is text  '))`, would return 12, the length of the text only.
This combination is so useful that a single function exists to perform both tasks, `LEN_TRIM()`. This works identically to applying `LEN` to a `TRIM`med string, as in the previous example.

8.3.3 String Indexing

In many ways, strings act like arrays, both store a set of multiple values/characters such that a collection of data can be referenced/interacted with in one go, rather than through many individual variables. `CHARACTER` strings, in a similar manner to arrays, can have elements within them extracted through indexing. However, the procedure differs somewhat between the two. While, for an array, the following commands would return first a single element, and then a series of elements at once:

```
myarray(3) !returns the element of the array at index '3'
myarray(2:4) !returns a small array consisting of elements index '2', '3' and '4' only
```

`CHARACTER` strings, however, can *only* be indexed in the latter way, even if you only wish to obtain a single character from the string, in other words, the commands equivilent to the above array commands would be:

```
mystring(3:3) !returns the single character at index '3' of the string
mystring(2:4) !returns a small string consisting of the characters at indices '2', '3' and '4'
```

See how, to obtain a single character, the array notation '`(3:3)`' has been used. This effectively tells the code that you want to extract a substring starting from element 3, and ending at element 3; to perform this, the code will extract only element 3.

Indexing a subcharacter or substring can be done with `INTEGER` variables as well as hard-coded values. The following code would print to screen gradually more and more of a string; starting from the first letter on one line, then the first two on the next line, until the whole length of the string is printed. Note the use of the `LEN_TRIM` command to perform the loop over only the characters up to the end of the actual text assigned, not the four spaces which would follow. Also note the use of the 'formatting' statement in the `WRITE` statement. These are described fully in the subsequent subsection, and are vital in correctly manipulating strings when using `READ` and `WRITE`.

```
1  PROGRAM string_index_example
2
3    IMPLICIT NONE
4    CHARACTER*20 :: mystring
5    INTEGER :: i
6
7    mystring = 'This is a string'
8
9    DO i = 1, LEN_TRIM(mystring), 1
10     WRITE(*,'(A)') mystring(1:i)
11   END DO
12
13 END PROGRAM string_index_example
```

8.3.4 Format descriptors

In Section 2, the basics of input and output were covered. Within this section was the description that the `WRITE` and `READ` statements are followed by two arguments, the first of which is the file 'unit number' or an asterisk is used to output to screen; the second argument was not described, however. The purpose of this second argument is for formatting of input/output, hence being left until this section for it to receive an explanation.

The format field of the `READ` and `WRITE` commands permits you to specify explicitly the manner of formatting which the command will be required to do. The asterisk symbol used so far tells the command to attempt to work it out itself; this works for simple input and output, but can quickly become limiting with more complex interaction. One flaw you may have noticed with the use of the asterisk for formatting in `WRITE` statements is the way in which the command displays numerical values. The command `WRITE(*,*) 100` will print the number '`100`' to screen, however it will do so with many preceeding spaces before the '`100`' itself. This is because the code knows that `100` is an integer, and reserves the maximum amount of space it could anticipate needing to print an integer. On most machines this is likely 12 characters. The longest length value that most default `INTEGER` types store is 10 digits long, combined with the minus sign for negative numbers this is 11 characters to display the longest number. The additional space to total the 12 characters mentioned is to ensure that the number is separated from anything that preceeded it.

So, while this is fine for a data file, where the separation is likely only intended to be automatically processed anyway, it is not convenient when included in text. Attempting to get a Fortran code to print a message to the user of '`The task has completed in 15 seconds.`' could easily end up looking like: '`The task has completed in  15seconds.`'.

Perhaps this could be considered a trivial concern, the above example isn't nicely formatted, but supplies the intended message. Consider the case of filenames, however. Say you are generating or outputting data to a series of files, which will be automatically named and numbered. You want the names to be `data_[N].out`, where `[N]` is the `INTEGER` number of the iteration for each file. If you were to try using a string composed of components without formatting, you could get: `data_ 11.out`. This *is* an issue, as spaces in filenames can easily cause problems, not least if they were not intended.

It is the use of the `READ` and `WRITE` format field which permits the alteration of numerical fields such as outlined above, as well as further control over strings/text. To begin examining the use of formatting statements, the following example code shows an illustration of numerical formatting:

```
1  PROGRAM formatting_reals
2
3    IMPLICIT NONE
4
5    REAL :: X
6
7    X = 2.34
8
9    WRITE(*,*) X
10   WRITE(*,'(F5.3)') X
11
12 END PROGRAM formatting_reals
```

Try quickly writing this code into a file and running it. You should immediately notice the difference between the two lines of output. The first output, using the automatic formatting specified by the asterisk, prints as you have seen before, a large number of spaces before the value and to as many decimal places as is required for the accuracy of the value. The second output, however, should have no preceding spaces, and limit the number of digits after the decimal place to three. We can look at why by looking at the format statement: `'(F5.3)'`. The key part of this statement is what is within the delimiting inverted commas and brackets, for now assume that those delimiters are required. The `F` is a formatting descriptor indicating that the output is to be formatted as a *F*loating point value. The `5` states that the output is to be limited to a field 5 characters wide. The `3` after the decimal point indicates that 3 decimal places of accuracy should be used.

The necessity to explicitly define the intended field width (the '`5`' in the previous example) may seem hasslesome. The alternative is specifying the field length to be zero, this will fit however large the integer portion of the number is, plus any decimal point and decimal places specified by the second format value for the float type format.

This format descriptor is valid for `REAL` variables, but will not work for `INTEGER` variables. The equivilent of the `F` descriptor for `INTEGER`s is `I` (for *I*nteger). This descriptor requires only one value after it, the field length. For example: `'(I5)'` would be the descriptor to use for a field 5 characters wide for an `INTEGER`. Note that the use of zero for a variable-width field is valid for this descriptor as well.

The two key format descriptors for strings are `X` and `A`. `X` simply informs output to insert a space, and input to expect one. `A` is the equivilent of `I` and `F` but for a string/`CHARACTER` variable rather than `INTEGER`s and `REAL`s. `A` accepts a single value after it in the same manner as `I`, specifying the field width. Unlike `I` however, a value of zero does not work to automatically size the field. This can be achieved by using `A` alone, with no subsequent value.

Just as you can separate a list of several variables with commas following a `WRITE` or `READ` statement to output/input several at once; the formatting statements can handle a series of desired input/output statements. The following example is an illustration of this:

```
WRITE(*,'(X, I3, A, F5.2, "degrees")') myint, mystring, myreal
```

This demonstrates several concepts in the use of formatting statements.
The first stated format is an `X`, instructing the `WRITE` statement to place an extra blank space first.
After the first comma is `I3` specifying that there will be an `INTEGER` which should be formatted within a field width of three characters.
This is followed by an `A` with no value, the next expected variable is a string which the formatting should not cut down to a specific field width.
Following this is `F5.2`, a floating point type variable is expected, and will be formatted to a field width of five as well as rounded to two decimal places.
The final element of the formatting statement is an internal string; text contained within quotation marks within the inverted comma specified format are simply placed directly into the output at the point they occur.

If the text is not to be changed (ie, it is known at compile, not to be read in or similar), this is an alternative to using the `A` format descriptor, and then including your hard coded string in the list of variables. This means that the following are equivilent:

```
WRITE(*,'(F0.3, A)') myreal, ' m/s'
WRITE(*,'(F0.3, " m/s")') myreal
```

Statement	Type	Description
F[a].[b]	REAL	Floating point format, does not use exponent notation. [a] is field width; value of 0 to choose automatically. [b] is number of decimal places to display value to.
E[a].[c]	REAL	Always uses exponent format with preceding zero, ie 0.3141E+1 for 3.141. [a] is field width, value of 0 does *not* choose automatically. [c] is number of significant figures to display value to.
D[a].[c]	REAL	Same as E, but designed for DOUBLE PRECISION values, uses 'D' as exponent notation, ie 0.3141D+1 for 3.141. [a] is field width, value of 0 does *not* choose automatically. [c] is number of significant figures to display value to.
G[a].[c]	REAL	Floating point format for values within about 2 orders of 0, exponent form otherwise. [a] is field width, value of 0 to choose automatically. [c] is number of significant figures to display value to.
I[a] I[a].[d]	INTEGER	Normal integer format. [a] is field width, value of 0 to choose automatically. [d] OPTIONAL. Minimum number of digits to display. ie for [d]=5 for value 31, would display 00031.
A A[a]	CHARACTER	Format for displaying strings. [a] OPTIONAL. Field width, width based on string length when ommited, fixed width when included.
X	-	Empty space.
B[a].[d]	All	Binary Format, converts stored value/etc to binary for display. [a] is field width, value of 0 to choose automatically. [d] OPTIONAL. Minimum number of digits to display (fills space on left with zeros).
O[a].[d]	All	Octal Format, converts stored value/etc to octal for display. [a] is field width, value of 0 to choose automatically. [d] OPTIONAL. Minimum number of digits to display (fills space on left with zeros).
Z[a].[d]	All	Hexidecimal Format, converts stored value/etc to hexidecimal for display. [a] is field width, value of 0 to choose automatically. [d] OPTIONAL. Minimum number of digits to display (fills space on left with zeros).

Table 2: Table of useful formatting descriptors. Formats for REAL also apply for those of type DOUBLE PRECISION as well.

if `myreal` was `3.14159`, both would display: '`3.142 m/s`'.

These are not the only format descriptors available. A list of useful ones is provided in Table 2.

There is one further manner in which format descriptors can be altered. That is repeating arguments. This is done by prefixing any format descriptor with a number. This will repeat the following format descriptor that many times. Therefore, rather than writing the following to generate an output line:

```
WRITE(*,'("Coords (x,y,z):", F8.2, F8.2, F8.2)') xcoord, ycoord, zcoord
```

it is possible to write: `WRITE(*,'("Coords (x,y,z):", 3F8.2)') xcoord, ycoord, zcoord`

These two statements are directly equivilent.

Additionally, nesting is possible, as in the following:

```
WRITE(*,'("Coords (x,y,z):", 3(F8.2, " meters" ))') xcoord, ycoord, zcoord
```

As you can see, the '3' is outside the bracketed '`(F8.2, " meters" )`', this means that the entire bracketed statement, a float formatting statement then the string 'meters' is repeated three times. So, for `x = 1.000`, `y = 2.000` and `z = 3.000`, the output would be:

```
Coords (x,y,z):    1.00 meters    2.00 meters    3.00 meters
```

8.3.5 Use of Formatting Statements for Strings

Particular note must be taken of the use of formatting statements when using READ and WRITE to operate on strings. When using READ, with automatic formatting (the asterisk used so far), the command attempts to divide the supplied text being read from into sequential fields to assign to the variables listed as being read in to. Consider the following line of text (it could be within a file or entered by the user interactively):

```
1.521 35.62 1.22
```

It is read in from file unit `n` (which, again, could be '6' for `stdin` input from the user interactively, or an actual file), with the following command:

```
READ(n,*) myreal_a, myreal_b
```

The REAL variables `myreal_a` and `myreal_b` will then contain `1.521` and `35.62` respectively. The third item will then be skipped, and subsequent READ commands would read from the next line onwards.

However, this separation of a line into fields also applies when reading text into a CHARACTER variable. For the following line of text and READ statement:

```
One word followed by another
READ(n,*) mystring
```

Where `mystring` is a CHARACTER variable with a length at least 28 (enough for all of the words on that line). After the READ command is performed, the string would only contain the following:

```
One
```

Having used the same process of splitting the line into fields, and assuming that variables will be supplied for each field, or the value will be ignored. To correctly get a READ statement to take multiple words in one go, including the spaces between them, the correct formatting statement must be used. For the same line of text to read in as before, you would use the following:

```
READ(n,'(A)') mystring
```

This specifies that the only thing being read in is a character variable of arbitrary length (to specify a specific length, usually the length of the variable itself, you would use `'(A28)'` for example, assuming the CHARACTER variable's length was 28). Now that the READ command is aware that it is an arbitrary length string being read in, it will continue to read and add characters to the variable `mystring` until either the variable is 'full', or a new line character is reached on the line.

A smaller issue is encountered when using the WRITE statement with strings. For a command such as the following (where mystring is again a CHARACTER variable):

```
mystring = 'Some text'
WRITE(n,*) mystring
```

The output will not be:

```
Some text
```

but instead:

```
_Some text
```

Where the underscore represents a space. The string in the latter case is preceded by this additional space. When automatic formatting is used for output, additional spaces are included before and between variables being output, to act as buffers. However, when outputting strings in certain circumstances where the exact content of the string is important, this can be the cause of problems. Specifying the formatting statement for what you are outputting (in this example `'(A)'` would work) prevents the command from inserting extra spaces.

8.3.6 Concatenation

Concatenation is the process of joining two strings together. So the concatenation of the strings `One Two ` and `Three Four`, the result would be `One Two Three Four`. To use this operator in Fortran, you use double forward slash '`//`' between the two operands (the two strings you wish to join). Multiple strings can be put together in series as well, as in:

```
'One Two '//'Three Four '//'Five Six'
```

which would result in the single string: `One Two Three Four Five Six`

Note that the line of code used to concatenate the strings had spaces in the component strings already, one after '`Two`' and one after '`Four`'. If concatenation is used without these already in the string, they will be joint regardless, connecting the words themselves. As in: '`'One Two'//'Three Four'//'Five Six'`' would give `One TwoThree FourFive Six`

This is a particularly pertinent problem, due to the manner in which the TRIM command acts on strings. Consider the following fragment of code:

```
1 CHARACTER*40 :: a,b,c,d
2
3 a = 'Aperture Science '
4 b = 'Weighted Companion Cube'
5
6 c = a//b
7 WRITE(*,*) c
8
9 d = TRIM(a)//TRIM(b)
```

```
10 WRITE(*,*) d
```

Our first string, '`a`' has a space at the end. The assignment of the variable '`c`' does not use `TRIM`, this means that it keeps all of the empty spaces caused by the `CHARACTER` variable having been defined with length 40, and the 'real' content of the string being only 17 characters long (including the space). Therefore `c` only contains '`Aperture Science`', and spaces. It uses all 40 characters including spaces of the `a` string, and then tries to add all 40 characters of the `b` string. But because '`c`' itself is only 40 characters long, it fills up after adding `a` and so only contains that content.

When `d` is assigned, both component strings are `TRIM`med. However, this will result in '`Aperture ScienceWeighted Companion Cube.`', missing the space after '`Aperture Science`', which was cut along with the other trailing spaces by the `TRIM` command.

This should be kept in mind when manipulating strings, as it can cause problems if the strings are used for more than just text display, such as keywords, calling commands or filenames.

8.4 Encryption/Decryption - Assignment Material

The objective of this assignment is to develop you own program for encrypting and decrypting text. Encryption and decryption is a huge topic in itself, with constant development on new ways to secure data both in place, and in transit. The algorithm used to perform encryption is referred to as a '**cipher**' (or, less commonly, 'cypher'). A cipher is applied to the '**plaintext**' (the original, unencrypted, text or data), with a '**key**'. Based on the content of the key, the cipher generates a encrypted version of the plaintext (the '**ciphertext**'). To reverse the process, one must possess (or reverse engineer, for the more nefarious cases) the specific key used with the specific cipher to generate the ciphertext.

One of the simplest cipers is the 'Caeser Cipher'. This is a type of substitution cipher; each letter of the alphabet is replaced by a different letter. In this method, the basis for encryption is shifting the alphabet by a number of characters, and replacing the plaintext letters with the new, shifted, letters. The key for a Caeser Cipher is the number of characters which the alphabet is shifted by.

So, for a key of 2, the following change would take place:

Plain:	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Cipher:	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b

As you can see, *a* would be replaced by *c*, *b* by *d* and so on. This is not a particularly robust cipher. A vital clue in breaking an encrypted piece of text is, with a sufficiently long piece of text, to examine how often each letter turns up. With all languages, there is generally a very distinct pattern of frequency of each letter. With a Caesar Cipher encrypted text, you would see exactly the same frequency, but shifted along by a number of characters. That number would tell you what the key was. Additionally, if you guess or know that it is a Caesar Cipher, you can simply try all 26 possible keys, and see which one results in sensible text!

So, what is a more reliable, yet similarly easy to implement, way of encrypting text? The method we will look at, and you will end up coding, is also a substitution cipher. In this case however, our key is going to be an entire word, or word with numbers.

We have looked already at ASCII code. This will be used as the basis for our cipher. It is useful because all of the characters already have an assigned value, so any shift is easy to implement. With the cipher you will program, rather than there being one value to shift by across the whole plaintext (the single number key of the Caesar Cipher), characters will be shifted by different values depending on where they appear within the text. This method is effectively what is formally called a 'Vigenére cipher'. However, rather than assigning each letter a number, we will simply use the numerical assignments for all of the characters provided by the ASCII encoding.

Consider the following example, in each case we show both the letters involved and their ASCII character values.

First, we have the 'plaintext', the piece of text that is intended to be encrypted:

H	e	l	l	o	,		m	y		n	a	m	e		i	s		A	n	o	n	2
72	101	108	108	111	44	32	109	121	32	110	97	109	101	32	105	115	32	65	110	111	110	50

Next, we choose our key, this will form our cipher. Note how, in both cases, we have been able to use a combination of upper and lowercase letters, numbers and even punctuation (the comma in the plaintext above), as they all have ASCII values:

S	p	y	0	0	7
83	112	121	48	48	55

To create a full key text, we repeat the key as many times as is necessary to be as long as the plaintext:

S	p	y	0	0	7	S	p	y	0	0	7	S	p	y	0	0	7	S	p	y	0	0
83	112	121	48	48	55	83	112	121	48	48	55	83	112	121	48	48	55	83	112	121	48	48

We use this long version of the key to act as the value by which we shift each of the characters of the plaintext. Shifting the original ASCII values by the ASCII value of the key character at the same location (same number of characters along in the strings). So, for the capital 'A' in 'Anon', ASCII value 65, its matching key character would be the capital 'S' in the final 'Spy', ASCII value 83. So we would want to shift the value of 65 by 83 giving 148. This would give us the ASCII value of the new character which would be the encrypted version of that letter.

However, because the ASCII code stops at 127, and only up to 126 are display characters (127 is the code for 'delete'), we need to loop back around if a value exceeds 126. Additionally, display character only start at ASCII value 32 (the value for a space), so rather than looping back to zero, we loop back to 32 (by subtracting 95). To make this work, we need to ensure that the key value won't skip too far. If we shift character 121 (letter 'y') by 122 (letter 'z'), looping if the answer is above 126, but starting at 32 when we loop back around, we have: 121 + 5 from the key, reaching 126, then with the remaining key value, we have 32 + 117 = 149; still greater than 126. This would cause a problem. As such, instead of adding the value of the key character, we add the value of the key character minus 32. This ensures that it will never overrun after the addition. We are making it represent the range of characters available.

For our key, we have:

S	p	y	0	0	7
83	112	121	48	48	55
-	-	-	-	-	-
32	32	32	32	32	32
↓	↓	↓	↓	↓	↓
51	80	89	16	16	23

We can then apply an encryption across the whole of the plaintext, for each plaintext character, we add the value of 'key text - 32', then subtract by 95 if the result was greater than 126 (to loop back to 32). This gives us the following:

H	e	l	l	o	,		m	y		n	a	m	e		i	s		A	n	o	n	2
72	101	108	108	111	44	32	109	121	32	110	97	109	101	32	105	115	32	65	110	111	110	50
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
51	80	89	16	16	23	51	80	89	16	16	23	51	80	89	16	16	23	51	80	89	16	16
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
{	W	f	|		C	S	^	s	0	~	x	A	V	y	y	$	7	t	_	i	~	B
123	87	102	124	32	67	83	94	115	48	126	120	65	86	121	121	36	55	116	95	105	126	66

In a more readable layout, 'Hello, my name is Anon2' has become '{Vf| CSŝ0~xAVyy$7t_i~B'; quite unreadable, and extremely difficult, if not impossible, to convert back to the plaintext without knowing the key.

The reason that this encryption, simple though it is, could potentially be impossible to crack, is that the length of the original key is fairly similar to the length of the plaintext. Consider a key which is the same length as the plaintext, every character has been moved by an entirely different amount, purely dependant on the key. When a key is repeated for a long piece of plaintext, that repetition can leave patterns which are detectable, combined with analysis of letter frequency, it can still be broken. But with a key that doesn't need to be repeated, there is no repetition which might give away the underlying algorithm and key. This concept is called a 'One-Time Pad' ('Pad' referring to key, in this context). Additional criteria on the key in this scenario do exist though, the key must itself be entirely random. If the key contains repetition or patterns such as regular words which are potentially recognisable, then the possibility exists to break the code without the key.

Hopefully this has provided some insight into how this Encryption method works, the assignment instructions also include an enumerated series of points to describe it fully again. The decryption proceedure is almost identical. 32 is still subtracted from the key values; rather than adding the plaintext values and key values, the modified key values are subtracted from the plaintext values; and rather than checking for if the resultant value exceeds 126, the check should be for if it goes below 32. If it does go below 32 it must loop back around from 126 by *adding* 95.

With luck, this task will prove to be of particular interest, it is the first topic involving the construction of a larger scale program performing an entire useful task, unlike the previous assignments where small, or artifical, tasks are performed in isolation. Encryption and decryption itself represents an enormous topic, with a myriad of concepts and procedures. Covered here is only a tiny part of the ideas behind the topic, and outside of formal notation and explanations of an actual course on encryption.

In particular, the two main methods described here (Caesar ciphers and Vigenére ciphers) are what are referred to as 'symmetric key algorithms'. This means that the same single key encrypts one way, and when applied in reverse, decrypts. An extremely important field is that of '*asymmetric* key algorithms'. In this case, there are two keys, paired to one another, key 1 and key 2. A message encrypted with key 1 is *not* created such that applying the same key in reverse will decrypt it; only its partner, key 2, can perform the decryption. Vice-versa, a message encrypted with key 2,

can only be decrypted with key 1. This is the system used by the RSA public-key cryptographic system, used commonly for transmission of electronic data over the internet. Very frequently in emails, for example.

It is, however, a fascinating topic; both encryption/decryption procedures themselves, and the methods used to crack encryption. If one reads of the German '*Enigma*' machine of World War Two (an electro-mechanical encryption tool used by the German forces during that time), and of the, first Polish, and later collaboration of Allies, work in understanding and decrypting Enigma messages; it is almost impossible not to marvel at the sheer ingenuity and exceptional intelligence shown to achieve that goal. At least, that is this author's opinion.

8.5 Assignments Summary

Important Note: There is only one programming assignment for this week; however, it contains two parts (the encryption, and the decryption) and as a whole is more complex than previous individual programming assignments.

8.5.1 Programming Assignment

1. Create a new file for your code called '`assign_8.f90`'.
2. Using the command
   `:> cp ~ph302/plaintext.txt ./`
   Obtain a copy of the text which you must encrypt.
3. The program must initially interactively ask the user whether they wish to perform encryption or decryption. It should set a `LOGICAL` variable based on the response received to store this choice. (We suggest a `LOGICAL` variable, but you are free to choose your preferred way of storing the choice.)
4. If the user selects to Encrypt, the following must occur:
   (a) The program should read in the text of the file `plaintext.txt` to a `CHARACTER` variable of sufficient length (this should be more characters than the file itself has, to ensure that it, or any other similar or slightly longer length file, would be processed correctly).
   (b) The program should then interactively ask the user for a key-phrase. This key phrase should be stored in a `CHARACTER` variable the same length as that used for your plain text variable.
   (c) The key-phrase must be repeated over the entire length of the plain text variable, so that your key variable contains the same number of stored character values as your plain text.
   Hint: The intrinsic command '`REPEAT(text,times)`' will be useful here, `text` should be a character variable, and `times` should be an `INTEGER`. The command returns a string which is the characters in the `text` variable, repeated `times` times. Hence, for:
   `newstring = REPEAT('test',5)`
   `newstring` would be '`testtesttesttesttest`'
   (d) Having now obtained both the text to be encrypted, and the full-length repeated key, the encryption itself must be performed.
   (e) You should create a suitable loop structure which will go through each of the characters in the plain text variable.
   (f) For each character, it should take the ascii value of that character (`IACHAR(character)`) and modify it based on the ascii value of the key phrase variable at the same location. (ie if your text was 'thisismytext' and your repeated key 'keykeykeykey', the 'm' of 'my' would match with the third 'k' of the key)
   (g) With the ascii values of both the plain text character and key character, the modification is as follows: (plain text character value) + (key text character value - 32). If the resulting value is greater than 126, you must also then subtract by 95 (to loop back to 32).
   (h) You should either update the original plain text variable with each modified/encrypted character (converting back from ascii with `ACHAR`), or have another string variable of the same length to store the encrypted message, so as not to alter the original variable.
   (i) Once you have encrypted the entire string, it should be output to a file '`assign_8_encrypt.out`'.
   Note: You must make sure that, when you write your string to the file you use the correct string formatting descriptor, `'(A)'`. By default, when printing strings without any format specification, a single space in inserted before the string is printed. This space is then assumed to be the first character to decode by the decryption algorithm, and means that the key is incorrectly displaced by one character for the entire message; preventing the decryption from being performed correctly (See Section 8.3.5).
5. If the user selects to Decrypt, the following must occur:

(a) The program should read in the text of the file `assign_8_encrypt.out` to the `CHARACTER` variable you created for the plain text in the encryption part.

(b) The program should then interactively ask the user for a key-phrase. This key phrase should be stored in the same `CHARACTER` variable used for the key in the encryption part.

(c) The key-phrase must be repeated over the entire length of the encrypted text variable, in the same manner as for the encryption stage.

(d) The loop which you have already created for the encryption phase need not be changed, you must still go through each character in the original string as before.

(e) However, a check must be added for if decryption is selected. In the case of decryption the modification must be as follows: (plain text character value) - (key text character value - 32). If the resulting value is less than 32, you must also then add 95 (to loop back from 126).

(f) This should update the original string, or write a new string in the same manner as for encryption.

(g) Once you have decrypted the entire string, it should be output to a file '`assign_8_decrypt.out`'.

6. You should test that your code correctly encrypts and then decrypts text for the correct key. You can try changing the text in `plaintext.txt`.

7. Once you are satisfied that your code works correctly. Make sure that the `plaintext.txt` file is the one obtained using:
```
:> cp ~ph302/plaintext.txt ./
```

8. Run your program in encrypt mode using the key phrase '`Fortran 2013/2014`' *exactly as stated*, case sensitive and correct spacing (a single space between 'Fortran' and '2013/2014'), numbers and the forward slash. This will generate the `assign_8_encrypt.out` which you will be required to submit.

9. Then run your program in decrypt mode, using the same key phrase, '`Fortran 2013/2014`'. This will generate the `assign_8_decrypt.out` which you will be required to submit.

10. You should print off your code (`assign_8.f90`) and both outputs (`assign_8_encrypt.out` and `assign_8_decrypt.out`).

8.5.2 Questions

For this week the questions are as follows:

1. What *exactly* would the following code write to screen (assuming suitable `PROGRAM`/`END PROGRAM`/etc were included around this segment of code):
```
CHARACTER*256 :: namechar
namechar = 'Roger'
WRITE(*,*) namechar
```
Note: You don't need to write the output in full, you can just describe what it would be.

2. For a simple Caesar Cipher with a key of '2', what would the following sentence transform to?
   - 'I wandered lonely as a cloud'

3. For a `REAL` variable, '`a`', with a value of 1.567000, what would the following `WRITE` statements produce?

   (a) `WRITE(*,'(F0.2)') a`

   (b) `WRITE(*,'(F8.1)') a`

   (c) `WRITE(*,'("Value:",X,F0.3)') a`

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_8.f90`, `assign_8_encrypt.out` and `assign_8_decrypt.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

9 Week Nine - Advanced Functionalities and Concepts of Fortran

9.1 Complex Numbers

By this point, many of you should be familiar with the mathematical concept of complex numbers. If you are not, refer to additional documents covering the topic provided on Moodle, or ask a demonstrator. The Fortran language is designed to intrinsically handle complex numbers and operations on complex numbers. The data type used to accomplish this is called, perhaps unsurprisingly, 'COMPLEX'. The COMPLEX type is effectively two REAL datatype numbers back-to-back. One of which represents the real component of the value, and the other the imaginary component. There exist several intrinsic functions which operate on COMPLEX numbers.

Declaring a COMPLEX variable should be done using the function CMPLX(a,b), where a is a REAL variable representing the real component and b is a REAL variable representing the imaginary component. So, displaying mathematically, the complex number $c = a + ib$ would be declared as c = CMPLX(a,b).

Of the two most basic COMPLEX functions, the first of these is 'REAL(c)', where 'c' is your COMPLEX variable. This returns the real component only of the complex number. The second is 'AIMAG(c)', again for a COMPLEX variable c. This returns just the imaginary component of the complex number.

There are some functions specifically for more in-depth interaction with COMPLEX variables. 'CONJG(c)' will return the complex conjugate of a COMPLEX variable c. ABS(c) will return the absolute value of a COMPLEX variable; unlike the implementation of ABS() for non-COMPLEX data types (which simply return the modulus of the argument variable), for a complex number c = CMPLX(a,b), d = ABS(c) is equivilent to d = SQRT(a**2 + b**2).

The majority of regular mathematical functions also operate on COMPLEX variables. This includes functions such as EXP, COS, SIN and other mathematical processes which can generally be applied to complex numbers. See Section 16.3.2 for a more complete list of the available mathematical functions which use COMPLEX variables.

A simple example of the use of a COMPLEX variable being given a value and used within a calculation is as follows:

```
1  PROGRAM complex_example
2
3    IMPLICIT NONE
4    COMPLEX :: val1, val2
5    REAL :: realcomponent, imaginarycomponent
6
7    realcomponent = -1.0
8    imaginarycomponent = 0.0
9
10   !assign a complex number with a real component of -1 and no imaginary component
11   !ie, this is a complex representation of just the value '-1'
12   val1 = CMPLX(realcomponent, imaginarycomponent)
13   WRITE(*,*) 'Components of val1 are: ', val1
14
15   !val2 is the square root of val1, ie, the square root of minus one
16   !if this was simply entered as 'SQRT(-1.0)' there would be an error
17   !for complex output, the function needs complex input, even if there is
18   !no imaginary component to the input
19   val2 = SQRT(val1)
20
21   WRITE(*,*) 'Components of val2, the square root of -1, are: ', val2
22
23 END PROGRAM complex_example
```

Most of the above program is annotated using comments which should make it's operation clear. The output that would be obtained from the program would be:

```
( -1.0000000     ,   0.0000000     )
```

for val1, (ignoring the additional string preceding the output), and would be:

```
(   0.0000000     ,   1.0000000     )
```

for val2. In other words, for val1, the first component (the real component of the complex value) is -1.0, and there is no imaginary component. For val2, there is no real component to the value, and a value of 1.0 for the imaginary component. This is as expected, as we know that $\sqrt{-1} = i$. 1 in the imaginary axis, with nothing on the real axis.

Key things to note are that the SQRT function will still not work if expressed simply as SQRT(-1.0), even if it is providing the value for a COMPLEX variable. When the function SQRT is run, it determines from it's argument (the value passed to

it) what manner of maths it should perform, and what type of variable it should return. If called as `SQRT(-1.0)`, it finds it's argument to be a `REAL` variable and intends to return a `REAL` variable. As `REAL` square root of a negative number does not have a solution, it will act the same regardless if the variable being assigned with the answer to the computation is of type `COMPLEX`.

9.2 Allocate

9.2.1 Basic Use

We have previously covered the use of arrays in Fortran as a method to store data together as a single structure, rather than many individual variables. These structure are suitable to then be iterated through to accomplish some task on the entire set, or a subset, of the data. The examples which were examined in Section 5 were all based on cases where we already knew how much data needed to be stored. The sizes of the arrays were hardcoded into the source code; once compiled, it would not have the capability to store a different quantity of data. Consider the following:

```
1  PROGRAM array_limitations
2
3    IMPLICIT NONE
4    INTEGER, DIMENSION(100) :: myarray
5    INTEGER :: i
6
7    OPEN(UNIT=10,FILE='input.in')
8
9    DO i = 1, 100, 1
10     READ(10,*) myarray(i)
11   END DO
12
13   CLOSE(10)
14
15   WRITE(*,*) myarray
16
17 END PROGRAM array_limitations
```

This program creates an array, `myarray`, of 100 `INTEGER` values; then opens the file `input.in` and reads one value per line into the array until the array is full; finishing by closing the file, and printing the newly obtained contents of the array to the screen.

However, what if the number of records is not known in advance? A solution is to use `ALLOCATE`. In this case we would also make an alteration to the input file; we make the specification such that, however many records an input file has, the first line must be a number stating that number of records. So, consider a file with 104 records, rather than 100. Our file would start with the line '`104`', then subsequent lines would contain the records themselves. With this file format implemented, we could write the following code to read in the data of an arbitrary length file:

```
1  PROGRAM array_allocate
2
3    IMPLICIT NONE
4    INTEGER, DIMENSION(:), ALLOCATABLE :: myarray
5    INTEGER :: i, num_records
6
7    OPEN(UNIT=10,FILE='input.in')
8
9    READ(10,*) num_records
10
11   ALLOCATE(myarray(1:num_records))
12
13   DO i = 1, num_records, 1
14     READ(10,*) myarray(i)
15   END DO
16
17   CLOSE(10)
18
19   WRITE(*,*) myarray
20
21 END PROGRAM array_allocate
```

Let's look at the important lines. The first major difference from the first example is on line 4; instead of specifying a size of dimension, we have simply put a colon indicating an open range of values, then we have the additional parameter

'ALLOCATABLE'. This parameter tells the program that the variable specified on this line is an array, but it does not yet have an allocated size, which will be determined later.

The next change is that on line 9 we read a single line of data to get the value for the number of records. The most important statement follows this, on line 11. Here, we perform the array allocation itself, we have the ALLOCATE statement, then an ALLOCATABLE array as it's argument, this time with a specific range we wish to use given in brackets after the name of the array. In this case, we want the ALLOCATABLE array `myarray` to have '`num_records`' number of elements, ie, the number specified at the head of the file.

After this, we can perform the read in almost as before, this time with the correct number of records as the upper limit, rather than a hard coded value.

Note: It is extremely important to note that, once you have allocated an array in this manner, you *cannot* use another ALLOCATE command to change the size of the already allocated array. In other words, the following example fragment is **invalid!**

```
ALLOCATE(myarray(1:mylength))
!I want to extend the length of myarray by one extra element
ALLOCATE(myarray(1:mylength+1))
!This will fail!
```

As you can see, the intent of the above fragment was that, after having assigned the array `myarray` to have a length `mylength`, the programmer wishes to add an additional one element to the end. This method is invalid, arrays **cannot** be allocated additional space whilst assigned in Fortran. This bring us to the counterpart of the ALLOCATE statement, DEALLOCATE.

9.2.2 The DEALLOCATE Statement

Once an ALLOCATABLE array has been allocated, it is no longer possible to modify it's size, with one exception. The only change that can be made is to take apart the entire array, releasing the stored memory space back to the computer. Therefore, the following sequence of commands is valid:

```
ALLOCATE(myarray(1:mylength))

DEALLOCATE(myarray)

ALLOCATE(myarray(1:mylength+1))
```

An original array is created with the length `mylength`, it is then deallocated and allocated again with one additional element.

This method, however, does **not** keep any of the data previously stored in the array, so it is not possible to use this directly to alter the size of an existing array containing data. For example, using ALLOCATE to create the original size, putting data into the array, then DEALLOCATE and another ALLOCATE with a larger size to make the array bigger. While the array *will* be bigger after this, it will **not** retain the data that was put into the original array.

The only way to effectively 'reallocate' an array, extending an array with data in to accept more elements, is by using a temporary array, as follows: This code has also been labelled with suitable comments describing the code flow and intent.

```
1  PROGRAM reallocation
2
3    IMPLICIT NONE
4    INTEGER, DIMENSION(:), ALLOCATABLE :: myarray, temparray
5    INTEGER :: I, mysize
6
7    mysize = 10
8    !This creates an array with ten elements
9    ALLOCATE(myarray(1:mysize))
10
11   !This loop populates the array with basic data
12   DO I = 1, mysize, 1
13     myarray(I) = I
14   END DO
15
16   !This shows the current contents of the array
17   WRITE(*,*) myarray
18
```

```
19    !We now want to add an additional two elements to the array, making it twelve elements
        long
20
21    !First, we allocate a new array the same size as our original
22    ALLOCATE(temparray(1:mysize))
23    !Then, we copy over the data from the original array into this temporary one; this can be
        done in a single step as follows:
24    temparray(:) = myarray(:)
25    !Note the use of the array notation (the bracketed colons) to indicate that all elements
        of myarray, should be copied into the equivilent spaces of all of the elements in
        temparray
26
27    !Now, we deallocate the original array, with it's data backed up in the temporary
28    DEALLOCATE(myarray)
29    !Then, allocate it again with the desired new size:
30    ALLOCATE(myarray(1:mysize+2))
31
32    !Now our original array is the correct size, but no longer contains any data. We now copy
        back the data from the temp array into our resized new array
33    myarray(1:mysize) = temparray(:)
34    !Note how this time, we must specify the space in my array explicitly, as they are no
        longer the same size, so would not be able to perform a straight copy
35
36    !We can then deallocate the temporary array in order to free the memory it occupies
37    DEALLOCATE(temparray)
38
39    !We initialise the two new elements of the original array, giving them the following
        values
40    myarray(11) = 98
41    myarray(12) = 99
42
43    !Finally, we print out the array again, this time it should appear with twelve elements
        rather than ten
44    WRITE(*,*) myarray
45
46  END PROGRAM reallocation
```

This process may appear convulted and somewhat long winded, however, due to the manner in which Fortran manages memory use, there are limited alternatives without resorting to more complicated structures involving items called 'pointers'.

In any code where such a task would need to be performed more than once, the entire proceedure would almost certainly be made into a subroutine which could be called to perform this whole task whenever needed.

When transferring information back from the temporary array to the space it originally occupied in the resized array, the use of the array element range notation is vital. Using a form which does not specify an equal number of elements on both sides of the assignment will fail.

Something to consider though, is that memory allocation is not computationally cheap. Each time memory is allocated, a certain amount of time is used simply organising for the allocation to be performed. When one large block of memory is allocated, this extra time, the 'overhead' will only be performed once. However, if lots of smaller allocations are made instead, the overhead time will be used for every single of those allocations. This means that allocating a large amount of memory in one go is substantially more efficient than allocating the same total amount of memory in smaller chunks. Therefore, whenever possible, the amount of memory required should be determined in advance of allocation, in order to reduce the instances where reallocation is necessary.

9.3 IOSTAT

9.3.1 Input Validation

We have previously looked at the IOSTAT parameter in Week Two, albeit briefly. Since the concept of IOSTAT has been covered, that it 'monitors' an input/output operation, and returns a value based on whether it was successful, or what went wrong if it wasn't, we can skip straight to an example. Try to examine the code and determine what it may be doing before reading the description which follows it.

```
1   PROGRAM iostat_example
2
3     IMPLICIT NONE
```

```
4   LOGICAL :: success = .FALSE.
5   INTEGER :: myint, myiostat
6
7   WRITE(*,*) 'Please input an Integer:'
8   DO WHILE (.NOT. success)
9     READ(*,*,IOSTAT=myiostat) myint
10    IF (myiostat .NE. 0) THEN
11      WRITE(*,*) 'Input failed, please ensure that you enter an Integer, try again:'
12    ELSE
13      WRITE(*,*) 'Thank you, Integer accepted'
14      success = .TRUE.
15    END IF
16  END DO
17
18  WRITE(*,*) 'My integer was: ', myint
19
20 END PROGRAM iostat_example
```

For this code we have utilised three variables, a `LOGICAL` checking variable, initially set to `.FALSE.`, a generic `INTEGER` variable which represents a parameter we desire to read in, and an `INTEGER` to act as the variable to receive the `IOSTAT` value.

The code initially prompts the user for input (line 7). Following this we begin a `DO WHILE` loop, with a criteria for exiting based on our checking variable, `success`. Within the loop, we first attempt to read in a value entered by the user, with the `IOSTAT` variable set as `myiostat` (line 9).

If the user does provide an input which can be successfully interpretted as an `INTEGER`, `IOSTAT` will return it's default value of 0. If, for any reason, the provided input cannot successfully be converted into an integer, `IOSTAT` will return a different value, dependant on the exact type of problem.

Therefore we have two conditions, implemented using an `IF-ELSE-END IF` construct. If the `IOSTAT` did not return zero (and hence the value of `myiostat` is not zero), then we know that the input has failed. A `WRITE` statement informs the user of this, and no change is made to the `success` checking variable. This means that when the `DO WHILE` next checks to see whether the exit criteria has been met, it will still not have been (`success` will have it's original `.FALSE.` state) and loop will try again, attempting another `READ` operation. Another incorrect input would continue this repetition.

When a value is entered by the user which *can* be successfully read as an `INTEGER`, the `ELSE` condition of the `IF` statement is triggered, as the successful `READ` operation will have returned an `IOSTAT` value of 0 for the variable `myiostat`. In this case, the user is informed that the value has been correctly entered, and the `success` checking variable is switched to `.TRUE.`. This means that, when the `DO WHILE` checks again for it's exit criteria, this time is *has* been met, the loop ends and the program will exit after showing the final `WRITE` statement with the user's value to screen.

This manner of input checking is vital in the implementation of reliable user interfaces. If a program crashed every time the user did something which was not expected, it would be almost impossible to interact with most software.

9.3.2 Testing File Sizes

We have now seen how `IOSTAT` can be used in practice, and in the previous subsection we saw how `ALLOCATE` can be used to choose the size of an array when a program is run, rather than having to decide in advance. Here, we will combine the two concepts to make a program which can read in a file of arbitrary length, determine that length, and allocate the necessary space to store the data.

In Section 9.2.1 we saw a program which did a similar task, however, it required a file format where a number indicating how many lines are in the file would always be contained on the first line of that file. This is not always plausible or desirable to implement. A more useful and adaptive approach is to have the program determine how long the file is by itself, without needing to be told the information.

The following code is quite long, and contains suitable commenting which should illustrate both how the code works, and hopefully better highlight the purpose and use of comments as you should use them in your own code. Note how they describe the *purpose* of code segments, not simply what a line/collection of lines *does*.

```
1  PROGRAM automatic_array_sizing
2
3    IMPLICIT NONE
4    !main array for data
5    REAL, DIMENSION(:), ALLOCATABLE :: myarray
6    !check var for end of file
7    LOGICAL :: eoffound = .FALSE.
8    !dummy var for searching through data file
```

```
  CHARACTER :: dummyvar
  INTEGER :: myint, myiostat, linecount, i

  !initialise line counter and open datafile
  linecount = 0
  OPEN(10,FILE='mydata.dat')
  !keep looping until the eoffound var is switched from .FALSE. to .TRUE.
  DO WHILE (.NOT. eoffound)
    !test line by reading a single character
    READ(10,*,IOSTAT=myiostat) dummyvar
    !if the iostat returned by the test read is 0, then data is on the line, increment count of valid lines
    IF (myiostat .EQ. 0) THEN
      linecount = linecount + 1
    !if the iostat is -1, the read command reached the eof character at the end of the data file,
    !switch eoffound var to .TRUE. and write number of lines found, close original access to data file
    ELSE IF (myiostat .EQ. -1) THEN
      WRITE(*,*) 'End of file reached.'
      WRITE(*,*) 'Total of ', linecount, ' data lines.'
      eoffound = .TRUE.
    !if iostat is anything other than -1 or 0, then different error has been encountered, abort program
    ELSE
      WRITE(*,*) 'Non End Of File error occured in file read, with IOSTAT value of ', myiostat
      WRITE(*,*) 'Aborting.'
      CLOSE(10)
      STOP
    END IF
  END DO

  !if loop ended successfully, but with zero lines found, then file was either empty or didn't exist
  !note that, having used 'OPEN' for that filename, if it didn't already exist, it was created
  IF (linecount == 0) THEN
    WRITE(*,*) 'No valid lines of data present, file was empty or did not previously exist.'
    WRITE(*,*) 'Aborting'
    STOP
  END IF

  !allocate memory space for array necessary to store all of the lines of data and rewind data file
  ALLOCATE(myarray(1:linecount))
  REWIND(10)

  !go through all lines, reading data to correct location within array
  !reuse myiostat var to again test IOSTAT, this time looking for other errors (non valid characters etc)
  DO i = 1, linecount, 1
    READ(10,*,IOSTAT=myiostat) myarray(i)
    IF (myiostat .NE. 0) THEN
      WRITE(*,*) 'Error trying to read data from file at line ', i
      WRITE(*,*) 'Aborting.'
      CLOSE(10)
      STOP
    END IF
  END DO

  !once read process complete, close file and inform user
  CLOSE(10)
  WRITE(*,*) 'All lines read in from data file'

END PROGRAM automatic_array_sizing
```

So, having read the contents of this program, it's operation should be clear. The intial step uses a 'dummy' variable, a variable not designed to store any content in particular, just to be used to facilitate performing a separate task. The

DO WHILE construct will repeatedly try to read a line from the file, storing a single character in the dummy variable. If there is data on the line, this will work without issue, and IOSTAT will return a value of 0 to the myiostat variable. For as long as this is the case, the counter for the number of lines (linecount) will be incremented and the loop will move to the next line.

At some point, the final record will be read in from the file, and the next read operation will not succeed, instead hitting the 'End of File' character which indicates the location a file stops containing data. This gives an IOSTAT value of -1, and an associated IF statement catches this, switching the LOGICAL variable eoffound to .TRUE., which will stop the DO WHILE loop from attempting to continue.

Having performed this first task, determining the length of the file, the array can now be allocated to the correct size with the ALLOCATE statement. We must also make sure the read operations start from the beginning of the file, not continuing from where it left off at the end of the file after determining the number of lines. For this we can use the REWIND command. REWIND expects a file number as a required argument, so REWIND(10) jumps back to the beginning of file 10.

Finally, the data can be read into the array. This final read in of the data has been written to also contain IOSTAT for other possible errors which may occur with the data file. For instance, if a line, instead of containing a number which can be interpreted as a REAL variable, contains an alphabetic character; rather than instantly causing a crash, the program will be able to inform the user as to which line in the input file caused the problem before exiting.

Note: It *is* plausible to combine the check for the number of lines with the check for whether the data is of the valid format. However, this can be trickier to implement in a robust fashion, and it is certainly easier to display the concepts involved with the format illustrated in this example.

9.3.3 Complex Data Read In - Assignment Material

We have seen both how to use the COMPLEX variable type and how to use IOSTAT to test the length of, and read in an arbitrary length file. This assignment will combine those two techniques. You will be required to fetch a data file from the following location:

```
:> cp ~ph302/complex_data.dat ./
```

This file contains a number of lines of data in two columns, the first column represents the real component of a complex number, and the second represents the imaginary component. While it *is* possible to open the file and determine the number of lines manually, assume that you will be dealing with many files of varying length. Not knowing file lengths in advance means that you will need to have your program determine the length of the file when it runs.

You should write a code (using a filename 'assign_9_1.f90') which first scans through the file to determine it's length. Having done this, a suitable COMPLEX array must be used to actually read in the data.

Once the data of COMPLEX values has been read in, your code must determine the result of two calculations. The first is the sum of the absolute values (moduli) of the COMPLEX numbers read in. So, for an array of n COMPLEX numbers, C_i, where i is the index of a given element, from 1 to n; you should calculate:

$$\text{Value 1} = \sum_{i=1}^{n} |C_i| \,. \tag{4}$$

The second is the product of the first element and the final element, ie the first COMPLEX value multiplied by the final COMPLEX value. So, for the same array C_i as described above, the calculation is:

$$\text{Value 2} = C_1 \times C_n. \tag{5}$$

Note that this should be done *after* having read in the data to an array. You may note that storing all of the data is not strictly necessary; as each element is only required once, it is possible to calculate the sum whilst reading the file, storing only the first element permanently in order to multiply it by the final element. However, the idea of the task is to illustrate reading in to an array an arbitrary quantity of data to later perform some task on it; not specifically to find this summation and this product, as there are easier ways of performing those tasks alone.

The value of the sum should then be output to a file called 'assign_9_1.out', along with suitable text describing what the number is (eg. 'the sum of the absolute values of a set of complex numbers' or similar). Followed by the product of the first and final elements, as well as a short line of text describing that.

Remember: Consider very carefully what data it is that you are reading in and when. Think about when it is that you will need to allocate your array with the size necessary to store the data.

You will be required to submit both assign_9_1.f90 and assign_9_1.out.

9.4 TYPEs

9.4.1 Vectors

So far several variable types have been used to facilitate the tasks we have performed with the programming concepts covered so far. However, the limited available variable types are not always entirely suitable for all aspects of a task. Consider a case where we are tracking multiple particles in a gravitational simulation. Each of these particles will have properties like location and speed, possibly acceleration and force. Each of those properties will also be vectors, with as many components as dimensions being simulated. Assuming a 3D simulation, that's three components for each vector parameter. One way to define the properties for the particles would be as follows (spread across multiple lines for ease of reading):

```
REAL :: location_x, location_y, location_z
REAL :: velocity_x, velocity_y, velocity_z
REAL :: force_x, force_y, force_z
```

However, having all of these parameters can become quickly unmanagable, or, at best, tricky to keep track of and follow program flow. An alternate method is to use a special construct called a `TYPE`. A type is effectively a custom variable type, consisting of a collection of existing variable types. So, an example of this would be as follows:

```
TYPE myvector
  REAL :: x, y, z
END TYPE myvector

TYPE(myvector) :: location, velocity, force
```

In this example we have defined a new type of variable called '`myvector`', with three internal `REAL` number components, one for each of the x, y and z axes. Then, we have defined each of the three variables from before, location, velocity and force, as variable of this type. This means that we have the three properties, each with the correct three vector components, without needing to define separate variables for each component.

Accessing any given component would be performed as follows:

```
!The y component of velocity:
WRITE(*,*) velocity%y
!The z component of location:
WRITE(*,*) location%z
!All three components of force at once:
WRITE(*,*) force
```

This may not appear to be especially useful, but it boasts a number of advantageous qualities. Say we wish to add two parameters which are vectors together. For the example where we did not use `TYPE`, we might do the following to get all three dimensions:

```
vector_result_x = vector_a_x + vector_b_x
vector_result_y = vector_a_y + vector_b_y
vector_result_z = vector_a_z + vector_b_z
```

If this must be done many times it would become inconvenient, and it would be difficult to write a function to do this, as we need to return values for three different variables (each of the dimensions). A subroutine could be used, but it would require looking something like the following, being equally unweildy:

```
CALL add_vectors(vector_result_x, vector_result_y, vector_result_z, vector_a_x, vector_a_y,
    vector_a_z, vector_b_x, vector_b_y, vector_b_z)
```

This is still a long piece of code, and would prove just as inconvenient in complicated tasks. However, considering now the example where we *do* use a new `TYPE`, and all three dimensional components are encapsulated within single variables, we could use a function as simply as follows:

```
vector_result = add_vectors(vector_a,vector_b)
```

This is much more compact than the previous examples, and since it is a function, it can also be nested, so adding three vectors could be:

```
vector_result = add_vectors(vector_a,add_vectors(vector_b,vector_c))
```

Doing the same without a type would span at least two lines, or require extra functions for however many combinations of adding/multiplying/dividing/subtracting you might want to use.

For this example of adding vector properties using a TYPE, the function for adding itself could look as follows:

```
TYPE(myvector) FUNCTION add_vectors(a, b)
  IMPLICIT NONE
  TYPE(myvector), INTENT(IN) :: a, b
  add_vectors%x = a%x + b%x
  add_vectors%y = a%y + b%y
  add_vectors%z = a%z + b%z
END FUNCTION add_vectors
```

Here you can see clearly how we have again used the statement 'TYPE(myvector)' in the manner we would usually have used the name of an existing variable type. Note how the actual mathematics within the function is clearly the same as the original set of lines for adding two vectors together without using types. However, in this case, we are able to arrange the code in a function such that we only need to have it written in full once, rather than every time vector addition is required.

A representation that may make the structure of variables and TYPEs more clear is the diagramatic. Consider the section of code which follows, we define a custom TYPE called MyType, which has three sub components, Component1 as an INTEGER, Component2 as a REAL and Component3 as a LOGICAL. After defining this, we define two variables, one regular INTEGER variable, and one of the new TYPE we have written:

```
TYPE MyType
  INTEGER :: Component1
  REAL :: Component2
  LOGICAL :: Component3
END TYPE MyType

INTEGER :: MyVar
TYPE(MyType) :: MyVarNew
```

The following diagram can be thought of as a representation of the two variables defined. The first is the regular INTEGER, 'within' the variable is contained a single value with type INTEGER. On the other hand, the second variable is of type TYPE(MyType). You can see how, rather than containing a value, it effectively contains three sub-variables of it's own, each with a value of the type for that component.

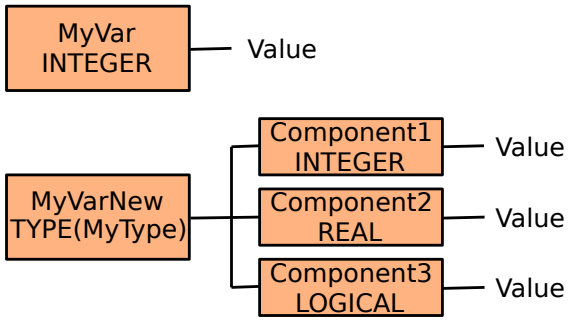


Figure 2: Diagramatic illustration of two variables, one of type INTEGER, and one using a custom TYPE structure.

9.4.2 Using Vectors in Code

We have seen in previous weeks some of the properties and forms we are able to assign to variables. One way in which we have used variables is making them into an array, to store an arbitrarily large number of records within a single construct. It is also possible to do this with variables which are non-standard types, such as those declared by you with the TYPE construct. Consider the following example, it is the first full functioning program example using the 'myvector' type described in the previous section:

```
1  PROGRAM vector_example
2
3    IMPLICIT NONE
4
5    TYPE myvector
6      REAL :: x, y, z
7    END TYPE myvector
8
9    TYPE(myvector) :: vsingle
10   TYPE(myvector), DIMENSION(1:5) :: varray
11   INTEGER :: i
```

```
12
13    vsingle = myvector(2.,-4.,6.)
14    WRITE(*,'(A,F0.1,X,F0.1,X,F0.1)') 'vsingle values:   ', vsingle
15
16    DO i = 1, 5, 1
17      varray(i) = myvector(i,i,i)
18      WRITE(*,'(A,I1,A,F0.1,X,F0.1,X,F0.1)') 'varray ',i,' values: ', varray(i)
19    END DO
20
21    WRITE(*,*) 'Adding vector vsingle to each vector in varray'
22
23    DO i = 1, 5, 1
24      varray(i) = vect_add(varray(i),vsingle)
25      WRITE(*,'(A,I1,A,F0.1,X,F0.1,X,F0.1)') 'varray ',i,' new values: ', varray(i)
26    END DO
27
28    WRITE(*,*) 'Scaling the vector vsingle by a factor of 2.0'
29
30    vsingle = vect_scale(vsingle,2.0)
31
32    WRITE(*,'(A,F0.1,X,F0.1,X,F0.1)') 'vsingle values:   ', vsingle
33
34    WRITE(*,*) 'Reset all elements of vector array to the value of vsingle'
35
36    varray(:) = vsingle
37
38    WRITE(*,'(5("(",F0.1,X,F0.1,X,F0.1,")",X))') varray
39
40    WRITE(*,*) 'Change the z component of the first three vectors to zero'
41
42    varray(2:4)%z = 0.0
43
44    WRITE(*,'(5("(",F0.1,X,F0.1,X,F0.1,")",X))') varray
45
46  CONTAINS
47
48    !Function which adds two myvector types together and returns the result as a myvector type
49    TYPE(myvector) FUNCTION vect_add(a,b)
50      IMPLICIT NONE
51      TYPE(myvector), INTENT(IN) :: a, b
52      vect_add%x = a%x + b%x
53      vect_add%y = a%y + b%y
54      vect_add%z = a%z + b%z
55    END FUNCTION vect_add
56
57    !Function which scales a myvector type by a scalar in the form of a REAL type variable
58    TYPE(myvector) FUNCTION vect_scale(vector,scalar)
59      IMPLICIT NONE
60      TYPE(myvector), INTENT(IN) :: vector
61      REAL, INTENT(IN) :: scalar
62      vect_scale%x = vector%x * scalar
63      vect_scale%y = vector%y * scalar
64      vect_scale%z = vector%z * scalar
65    END FUNCTION vect_scale
66
67  END PROGRAM vector_example
```

This program performs a number of operations, not with any particular goal, but just to demonstrate the use of this new type of variable within actual code. On lines 5 to 7 we have the definition of the `myvector` `TYPE` which was shown earlier. This is followed by definition of several variables. Firstly, on line 9, a single variable instance using the new type; next (line 10) we have created a variable which is an array this new type. Finally we have a regular integer to use in the loops.

The first operation performed is to assign the values for the single vector, on line 13. Note how this has been done, the name of the variable type has been used in the same manner as a function: `myvector(2.,-4.,6.)`. This defines, in one line, all three elements for the vector, in the order they appear written in the definition for the `TYPE` (x, y, z in this instance).

The following line writes this to the screen, note how the variable name has been passed to the `WRITE` statement as one name, rather than each component, however the formatting statement for the `WRITE` operation specifies the format of each

of the three components individually (the `'(A,F0.1,X,F0.1,X,F0.1)'` could be more concisely written as `'(A,3(F0.1,X))'` using the repeating format statements described at the end of Section 8.3.4).

On line 16 we have a `DO` loop to interact with the array of vectors. For each element `i`, we set the values of the three components to the same as the value `i` (hence element 1 of the array will be the vector `(1.0,1.0,1.0)`, element 2 of the array will be `(1.0,1.0,1.0)`, etc.). The result of this is also printed out to screen.

In the `DO` loop starting on line 23 we make use of one of the defined `FUNCTION`s, `vect_add`. To each of the vectors in the array, we add the vector `vsingle`, then `WRITE` the result to screen.

On line 30 we make use of the other `FUNCTION`, `vect_scale`. Here, we scale the single vector variable `vsingle` by a factor of two, again printing the result to screen.

The final section of the code is especially important, as it illustrates the ability of Fortran to operate on multiple array elements without manually iterating through them. The content of line 36 is the key statement:
`varray(:) = vsingle`
This is an example of implicit iteration, or array 'slicing', it is instructing the code to set every element of the array (indicated by the ':' within the parentheses which would usually contain the index of a single array element) to the value of `vsingle`. This acts on the whole array, setting all five vectors to have the same value as `vsingle`.

The subsequent `WRITE` statement, on line 38, is also interesting:
`WRITE(*,'(5("(",F0.1,X,F0.1,X,F0.1,")",X))') varray`
It makes use of a complex formatting statement to permit a formatted output of the entire array of vectors in one go. The initial '5' repeats the main statement five times, meaning once per expected array element. Within that, there is a sequence of a fixed string (the `"("`) which will provide a left bracket, then three floating point statements separated by spaces (`F0.1,X,F0.1,X,F0.1`), finally closed with a right bracket (`")"`) and an additional space.

This means that, when the entire array of five vector elements, each with three components, is passed as the argument to the `WRITE` statement, it will display each vector in separate brackets, with all three components of the vector within each bracket.

One further array based operation is performed, where the array slicing is combined with the `TYPE` components to perform a complicated task in one go. This case is displayed on line 42:
`varray(2:4)%z = 0.0`
This assignment is applying to a different slice of array elements, instead of all elements '`(:)`', it is applying to a particular range. In this case it is applying the assignment to the elements 2 to 4 inclusive '`(2:4)`'. The action being carried out is to set the z vector component only, of these three vectors, to zero. In one line, the z components of vectors 2, 3 and 4 in the array have been set to zero, leaving the x and y components as they were before, as well as not affect elements 1 and 5 at all.

This shows the utility of being able to simultaneously perform actions on multiple elements of an array at once. However, this technique must be used carefully. There are fixed rules about what can be implicitly iterated over, and in what manner it will do that iteration. The process requires some getting used to in order to make use of it most effectively.

9.4.3 More Advanced TYPEs

We have seen how it is possible to create a `TYPE` for some parameter which contains multiple properties, such as the vectors dealt with so far. However, other than vectors, what else can `TYPE`s be used for?

The examples for vectors have included such physical properties as location and velocity, properties which a physical object or particle could be considered to have. Extending this concept, we can nest `TYPE`s to create more complicated structures to more effectively write programs. The following fragment of code is the definition of a `TYPE` designed to store information about a simulated particle:

```
TYPE myvector
  REAL :: x, y, z
END TYPE myvector

TYPE myparticle
  TYPE(myvector) :: location, velocity, force
  REAL :: mass, charge
END TYPE myparticle
```

In this case we have created a `TYPE` called '`myparticle`', it contains two regular `REAL` variables, for the mass and charge of the particle, and contains several variables of the `myvector` type for the location, velocity and force for that particle.

So, in a simulation, we might set up and reference several particles in the manner described in the following code fragment:

```
TYPE(myparticle), DIMENSION(6) :: particles
INTEGER :: i

DO i = 1, 6, 1
  particles(i)%location = myvector(0.0,REAL(i),0.0)
  particles(i)%velocity = myvector(0.0,0.0,0.0)
  particles(i)%force%x = 0.0
  particles(i)%force%y = 0.0
  particles(i)%force%z = 0.0
  particles(i)%mass = 1.0
  IF (i <= 3) THEN
    particles(i)%charge = -1.0
  ELSE
    particles(i)%charge = 1.0
  END IF
END DO
```

On the first line we create an array of the new `myparticle` variable type, called `particles`. We then loop through the valid elements of the list (1 to 6) assigning the particles their initial properties.

The location of a particle is assigned with `particles(i)%location = myvector(0.0,REAL(i),0.0)`. You can see how, using the `%` extension, we are referencing the `location` property of that particle; because the `location` property is a vector, we use the `myvector` constructor to make a vector out of zeros and the counter value (this will put all six particles in a line along the y-axis).

The velocities of the particles is assigned similarly, but given a zero initial value in all axes. Force is also assigned zero on all axes, but has been done in a different manner. Instead of using the `myvector` constructor directly, we are referencing the sub components of the force vector individually. Note that this is exactly equivilent to the process for assigning zero for velocity, it is just to illustrate how sub components may be referenced.

A mass of `1.0` is then assigned (arbitrary units, at this point, as no formal unit conventions have been determined for whatever simulation this may be part of). Finally, we have an `IF` statement, checking whether the particle being assigned is one of the first half of the list, or last half of the list; for the first half, a charge of `-1.0` is given, and for the second half, a charge of `+1.0`.

Extending this idea of a `TYPE` for all of the properties of a particle, these variables can also be passed to functions or subroutines for processing as a whole. This permits an even greater degree of separating out your code into modular blocks, each performing a set task, linked together through your main program. The main program, in this case, need do little more than consist of a sequence of calls to subroutines and functions. This makes it particularly easy to see and understand the flow of a program code.

For instance, the following loop could be the main content of an 'n-body' simulation. An n-body simulation is one which simulations a collection of an arbitrary number of particles ('n') undergoing some manner of interacting series of forces, for instance gravity.

```
DO timestep = 1, maxtimesteps, 1
  CALL calculate_forces(particles,timestep)
  CALL calculate_acceleration(particles,timestep)
  CALL calculate_velocities(particles,timestep)
  CALL update_positions(particles,timestep)
END DO
```

Even without comments, this clearly shows what happens and in what order, all of the actual calculations for performing the simulation would be within the stated subroutines. The equivalent code written without functions, subroutines or `TYPE`s would be an enormous block of continuous code, making it extremely difficult to follow and comprehend the flow of. It is for this reason that the use of `TYPE`, functions and subroutines are so important in properly designing a program.

9.5 Modules

So far we have seen several types of code encapsulation, the main program, within a `PROGRAM-END PROGRAM` construct; the `SUBROUTINE` and the `FUNCTION`. A further structure exists called a '`MODULE`'. These are effectively a way to collect together several other structures to reference as one unit. The following is an example of a `MODULE`:

```
1  MODULE mymodule
2
3    TYPE newtype
4      REAL :: a, b
5    END TYPE newtype
```

```
6
7  CONTAINS
8
9    REAL FUNCTION newfunction(var)
10     IMPLICIT NONE
11     REAL, INTENT(IN) :: var
12     newfunction = var
13   END FUNCTION newfunction
14
15 END MODULE mymodule
```

This module contains one example of a `TYPE` and one of a `FUNCTION`. This could be in a separate file for use by any program. The following main program code could then use the module as shown:

```
1  PROGRAM moduleexample
2
3    USE mymodule
4    IMPLICIT NONE
5
6    TYPE(newtype) :: t
7
8    t%a = 0.1
9    t%b = 0.4
10   WRITE(*,*) t
11
12   WRITE(*,*) newfunction(4.588)
13
14 END PROGRAM moduleexample
```

The important statement is just before the `IMPLICIT NONE` line, '`USE mymodule`'. The `USE` statement tells a program to use content from an external module. Having applied the `USE mymodule` line (which *must* precede the `IMPLICIT NONE` statement), all of the content within the module is included for the program as though it was contained in the program itself. This permits us to use the `TYPE` '`newtype`' and the `FUNCTION` '`newfunction`' without any further definitions within the main code (no variables defined for the function, no repeat of the original `TYPE` definition). The use of modules allows the packaging of portions of related codes and `TYPE`s together for inclusion in a program or multiple programs which may need them.

9.5.1 Vector Mathematical Functions Module - Assignment Material

For this assignment you will be required to code a set of functions to perform vector mathematical operations; all of which will be housed within a `MODULE`. You should name this file '`assign_9_2_mod.f90`'. The first thing that your module must contain is the definition of the vector `TYPE` you will be using. This should be defined as follows:

```
TYPE myvector
  REAL :: x, y, z
END TYPE myvector
```

(The exact name of the type and the name of the components can be selected by you.) Following this, you should have the `CONTAINS` statement, with the functions you will be programming written below that.

The simple operations will be to add two vectors, and to scale a vector with a `REAL` (which can be adapted from the example earlier) and you must add a function which subtracts one vector from another. This new function should be declared as:
`TYPE(myvector) FUNCTION vect_sub(a, b)`
where your function should subtract `b` *from* `a`. It is important to note the order specified for the subtraction, make sure that it is vector `a` minus vector `b`.

The more complex task is to write suitable functions for performing the vector 'dot product' and the vector 'cross product'. These two functions should be declared as follows:
`REAL FUNCTION vect_dot(a, b)`
and
`TYPE(myvector) FUNCTION vect_cross(a, b)`
In mathematical form, for vectors $\mathbf{a}$ and $\mathbf{b}$, they should perform the calculations: $c = \mathbf{a} \bullet \mathbf{b}$ and $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ respectively. (For the latter of these, you may be more familiar with the alternate notation $\mathbf{c} = \mathbf{a} \wedge \mathbf{b}$ for representing the cross product.)

This may require you to read your notes on vector mathematics from other modules. Remember that the dot product returns a scalar (hence the function type being `REAL`) and cross product returns another vector (hence it's function type being `TYPE(myvector)`).

Your module should now contain: the initial definition of the vector `TYPE` you will be using and each of the functions `vect_add`, `vect_scale`, `vect_sub`, `vect_dot` and `vect_cross`.

Having constructed this module, you should now write a main program which will make use of the module, and perform several calculations on a set of vectors given to you. You should create a file '`assign_9_2_main.f90`', you will need to make use of the `USE` statement before `IMPLICIT NONE`, with the name of your module, in order to include it within your program.

It will be necessary to obtain a file containing the values for the vectors you will be performing the mathematical operations on. This file can be obtained with the following command:

```
:> cp ~ph302/vectors.dat ./
```

Which will copy the file to your active directory.

Within your code you should define an array of your vector `TYPE` with 5 elements. Your code should then open the supplied `.dat` file to read in the five vector values. The file is arranged such that each line contains three elements, x, y and z in that order for each of five vectors. So reading in a single line for a `TYPE(myvector)` might read:

```
READ(10,*) varray(1)%x, varray(1)%y, varray(1)%z
```

to read in a vector to the first element of the array `varray`. This example reads each of the x, y and z components separately. However, so long as the components are listed in the correct order in the definition of the vector `TYPE`, the following will read in all three components simultaneously:

```
READ(10,*) varray(1)
```

The code assumes that it should read each component sequentially from the line in the file, reading in x, y and z at once.

Instead of reading in each vector into the array separately, you should instead use a loop from 1 to 5 to read all of the required vectors. This will leave you with a vector array, which will be notated as $\mathbf{V}_i$, where i represents the index of the particular element of the array. You should then perform, and output (with suitable notes output indicating which output is which) the following calculations:

$$\begin{aligned} \mathbf{L} &= \mathbf{V}_1 + (\mathbf{V}_2 - \mathbf{V}_3) \\ M &= \mathbf{V}_2 \bullet \mathbf{V}_4 \\ \mathbf{N} &= \mathbf{V}_1 \times \mathbf{V}_5 \end{aligned}$$

Where, $\mathbf{L}$, M and $\mathbf{N}$ are the values you are trying to find. The answers should be the following:

$$\mathbf{L} = \begin{pmatrix} 13.0 \\ -3.8 \\ 0.0 \end{pmatrix} \qquad M = 22.6 \qquad \mathbf{N} = \begin{pmatrix} 7.0 \\ -2.0 \\ -17.0 \end{pmatrix}$$

If they are not these, then you should check your functions, and the maths used within your functions and main program to determine the problem. The answer for each value (or set of 3 values, for the vectors), should be output to a file called '`assign_9_2.out`'.

9.6 Assignments Summary

9.6.1 Programming Assignment 1

1. Create a new file for your code called '`assign_9_1.f90`'.
2. Using the command
   ```
   :> cp ~ph302/complex_data.dat ./
   ```
   Obtain the data file your program will need to read in.
3. You cannot preprogram your code with the expected number of lines to read in.
4. You must make use of a `DO WHILE` loop and the `IOSTAT` parameter of `READ` to determine how many lines of data the file contains.
5. Having determined this, you must `ALLOCATE` a suitable `COMPLEX` array to store the data from the file.
6. The code should then read the contents of the file into the array.
7. The program should determine:
   - The sum of the absolute (modulus) values of the data read in. ie, for a complex array C_i containing n elements, where i is the index of a given element, the calculation (in mathematical notation, not Fortran) should be:
     Value 1 = $\sum_{i=1}^{n} |C_i|$

- The product of the first and final `COMPLEX` values. ie, for a complex array C_i containing n elements, where i is the index of a given element, the calculation (in mathematical notation, not Fortran) should be: Value 2 = $C_1 \times C_n$.

8. Your code should then open an output file called `assign_9_1.out`, into this file should be output a short line describing each value is (what it was calculated from), and the values themselves.
9. You will need to print off your program file (`assign_9_1.f90`) and it's output (`assign_9_1.out`).

9.6.2 Programming Assignment 2

1. Create a new file for your vector module called '`assign_9_2_mod.f90`', your `MODULE` must also be given a suitable name.
2. Into this file should be coded the vector `TYPE` described in Section 9.5.1.
3. You should then copy the two `FUNCTION`s '`vect_add`' and '`vect_scale`' found in 9.4.2 to be included in your module.
4. It is then necessary to write a new `FUNCTION`, called `vect_sub`, which should perform the action of subtracting a vector `a` from another vector `b`, and return the result as a vector.
5. Two further `FUNCTION`s should be coded, one to calculate the vector 'dot' product, and the other to calculate the vector 'cross' product. (See Section 9.5.1 for more information on the exact format these functions should take.)
6. You should then create a new file for your main program, called '`assign_9_2_main.f90`'.
7. Within this program, you should use a `USE` statement, along with the name of your `MODULE` in order to give your program access to the vector type and vector functions previously written.
8. Using the command
   `:> cp ~ph302/vectors.dat ./`
   Obtain the data file your program will need to read in.
9. The data file can be assumed to contain five vectors, one per line. You do *not* need to have your program determine the length of the file itself.
10. Within your main program you should create an array of length 5 of your vector `TYPE` for storing the vectors listed in the file.
11. The program should `READ` the file in to this array using a `DO` loop from 1 to 5.
12. Following this, your program should perform a sequence of three calculations on the vectors. For an array of vectors '$\mathbf{V}_i$', where i is the index of the element within the array, the following values should be calculated:

$$\begin{aligned} \mathbf{L} &= \mathbf{V}_1 + (\mathbf{V}_2 - \mathbf{V}_3) \\ M &= \mathbf{V}_2 \bullet \mathbf{V}_4 \\ \mathbf{N} &= \mathbf{V}_1 \times \mathbf{V}_5 \end{aligned}$$

13. The answers to each of these calculations ($\mathbf{L}$, M and $\mathbf{N}$) should then be output to a file called '`assign_9_2.out`', along with a short line of text for each stating the calculation which resulted in the value.
14. You can look at the values at the end of Section 9.5.1 to determine whether your code has performed the calculations correctly.
15. To submit the assignment you are required to print both code files (`assign_9_2_mod.f90` and `assign_9_2_main.f90`) and the output created (`assign_9_2.out`).

9.6.3 Questions

For this week the questions are as follows:

1. What would happen when attempting to compile and run each of the two following lines of code, if included within a program:
   `a = SQRT(CMPLX(-1.0,0.0))` where `a` is a variable of type `COMPLEX`
   `a = SQRT(-1.0)` where `a` is a variable of type `COMPLEX`
2. If you have a `REAL` array '`A`' containing 20 elements, and you wish to resize the array to contain 30 elements *without losing the exising data*, what would be the sequence of commands? You can assume that `A` is `ALLOCATABLE`, and you have an additional `ALLOCATABLE` array '`B`' which has not yet been allocated.

3. For a vector `TYPE myvector` as described in previous subsections, and a variable `C` of type `TYPE(myvector)`; how would you initialise the variable `C` to have an x component of 1.00, a y component of zero and a z component of -2.00?

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_9_1.f90`, `assign_9_1.out`, `assign_9_2_mod.f90`, `assign_9_2_main.f90` and `assign_9_2.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

10 Week Ten - Introduction to Scientific Programming

10.1 Problem Solving with Programming

10.1.1 Stochastic Methods

A process described as 'stochastic' is one which is based on some manner of randomisation. These methods can be used to solve a variety of physical and non-physical problems. Generally the problems targetted with such techniques are either based on some manner of probabalistic process, or as a way to bypass otherwise extremely complex non-linear mathematical problems.

10.1.2 Finding 'Pi'

At this point in your degree you should be more than familiar with the mathematical constant π, whenever a problem relates even tangentially (no pun intended...) to circles, π will inevitably be in the equations. You may know it's value off the top of your head to a few significant figures, 3.14159 is a typical precision to remember, for use in by-hand calculations and estimations. However, how does one actually find the value in the first place? There are a few approximations which take the form of an equation, such as 22/7, which equals 3.142857 to seven significant figures. However, this isn't a calculation for the value, nor is it accurate to a great degree, 0.04% for a fundamental mathematical constant could still be problematic given the prevalance of π. So, it is this accuracy which we will attempt to beat using a stochastic computational/numerical method.

One method of finding an approximation for π is using a random number generator. You would start by drawing a square with sides of length $2 \times l$, within this square, draw a circle with a radius of l, such that the circle touches the middle of each of the four sides of the square, as shown in Figure 3.

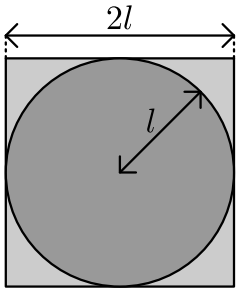


Figure 3: Geometry for stochastic determination of π.

This geometry gives two distinct areas, the darkly shaded area within the circle, given by $A_{\text{circle}} = \pi l^2$ and the total area of the square, lightly shaded plus darkly shaded, $A_{\text{square}} = 4l^2$. If we take the ratio of these two areas, we get:

$$\frac{A_{\text{circle}}}{A_{\text{square}}} = k_{\text{ratio}} = \frac{\pi l^2}{4l^2} \tag{6}$$

The l^2 terms cancel, and, solving for pi, we can see that:

$$\pi = 4 \times k_{\text{ratio}} \tag{7}$$

So, how does this help us determine π? This is where the advantage of a computer in its ability to process many thousands of pseudo-random numbers comes in handy. Imagine drawing such a geometry in real life on the floor. Onto it you throw small objects, ball bearings or similar, that will stay where they land. You throw more and more ball bearings; then, after a while, you count how many are inside the circle, and how many are inside the total square (i.e. how many you've thrown, assuming they all land within the bounds of the square). These two values approximate the areas of the two shapes, the ratio of which, as was shown in the equations above, should equal $\pi/4$. The issue with this is that, to get a reasonable approximation, you would have to throw a great many ball bearings. How much easier it would be to have the ball bearings 'thrown' randomly by a computer, and automatically find the areas and value of π as it goes.

This is far from the best way of finding π; many more efficient methods exist, however, it displays clearly how answers to problems can be determined with stochastic methods. With increasingly fast and powerful computers, using 'brute force' methods, to numerically calculate an answer rather than solve mathematically, are becoming increasingly popular. See the later subsection on High Performance Computing for more information on this topic (Section 10.2).

So, how can we go about writing a program to do this for us? We must consider the geometry in computational terms; the situation shown in Figure 3 can be thought of as having a horizontal x-axis, and vertical y-axis. For simplicity, we

will centre those axes on the mid-point of the circle. Since the absolute value of l does not affect the resulting value of π, it can simply be 1. This gives us a computational layout as shown in Figure 4.

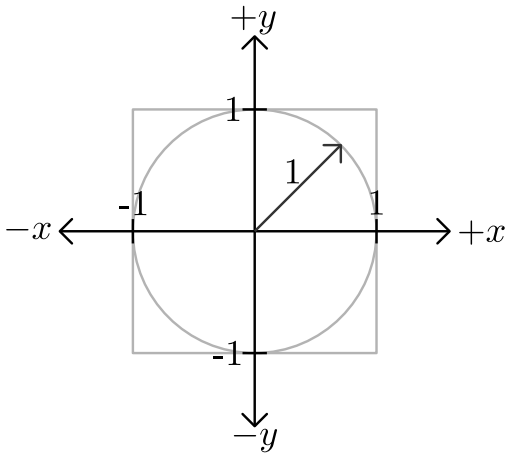


Figure 4: Geometry for stochastic determination of π.

The equation for the edge of a circle is $r^2 = x^2 + y^2$. So, for a radius of 1, any point where $x^2 + y^2 < 1$ must be inside the circle. This gives us the basis for our program. We start with two counters N_{square} for counting points within the square, and N_{circle} for counting points within the circle. If we generate a point at a random location between x of -1 to 1 and y of -1 to 1, it is definitely within the area of the square, so we increment N_{square}; then, we use the criteria that $x^2 + y^2 < 1$ if a location is within the circle to find out whether the random point is inside our circle or not. If it is, we also increment N_{circle}. After a user defined number of iterations, we stop placing particles and take the value of $4 \times \frac{N_{\text{circle}}}{N_{\text{square}}}$, which should give us an estimate of π.

10.1.3 Writing Your Finding 'Pi' Program - Assignment Material

The previous subsection has described the theory behind the program you will write, but you may still be unsure how to go about implementing it. The most important aspect is the generation of random numbers. To do this, you must use a Fortran function '`RANDOM_NUMBER()`'. This function is not technically a Fortran 90 intrinsic function; it is standard from Fortran 95 onwards, and implemented for Fortran 90 in the GNU *gfortran* compiler. To use `RANDOM_NUMBER()`, it must be called in the following manner:

```
CALL RANDOM_NUMBER(a)
```

Where `a` is a `REAL` or `DOUBLE PRECISION` variable, into which a random number in the range $0 \leq x < 1$ will be placed by the function. To place this newly created random number in the range $-1 \leq x \leq 1$, we could do the following: `a = (a * 2) - 1`, which would shift the $0 \leq x < 1$ range to $-1 \leq x < 1$. However, it is also possible to take advantage of the symmetry of the geometry, and simply leave the range as the original. The ratio of the areas for one quadrant is the same as the ratio of the areas for all four quadrants. It is important to note that the range of values is from and *including* zero, up to but *excluding* one. For the purposes of this task, this slight asymmetry can be considered to have sufficiently little of an effect as to be negligable; however, this is not always the case, and may sometimes need to be taken into account.

However, an additional consideration is that of the 'seed' for the random number generator. The seed is effectively a value given to the random number generator which characterises the numbers created. As the generator is deterministic, a given seed will always produce the same 'random' numbers. This is frequently desirable as, although you want the numbers to be random in terms of their distribution, you can also require the results to be repeatable. So that, for a given set of input you will always receive the same set of output. The complexity of the seed also contributes to the nature of the random numbers, on the test computer used above, the seed also required 8 starting integers from which it would use rules to generate random numbers.

To initialise the seed to a particular set of values, you will require an array to store eight integers. Into this array go each of the values you wish to use to set the seed, after this is done, the following command should be run to send the values to the random number generator:

```
CALL RANDOM_SEED(PUT = seed)
```

Where '`seed`' would be your length 8 array of values. In formal terms, this is not the correct way to perform this process. The number of integer elements for the random seed varies from machine to machine, and the subroutine `RANDOM_SEED` *does* include a method with which the program can determine how many values are needed when it runs. However, to reduce the complexity of the random number implementation, as the system on which it will be run is known, you can assume that the number of elements is eight. Should you so wish, you can use research of your own to determine the fuller method of creating a system-independant, more portable, code.

Now we have all of the necessary specifications and components to write the required program. Use the following

bullet points to help you plan the programs layout:

- Your program should be written in a file called `assign_10_1.f90`.
- When your program starts, it should request that the user enter, either all eight values to use for the seed, or a single value, from which you generate eight numbers using some fixed rule (eg. for a value `a`, the eight values could be `a`, `a+1`, `a+2`, `a+3`, etc.).
- You should make use of a length eight array with the seed values, and the command `CALL RANDOM_SEED(PUT = seeds)` (where '`seeds`' is your array name), to initialise the random number generator.
- You will need a loop of a length n (which can be hardcoded, you do not have to read it in). This value will implicitly be the total count of objects within the square, as it is not possible for a coordinate pair to be outside of the square. *Note:* Ensure that the step prior to this, setting the random number seed, is done before/outside of the loop. Otherwise, it will reset the seed on every iteration, resulting in only one 'random' number!
- Within your loop, you will need to use the `CALL RANDOM_NUMBER()` command to create a random x coordinate, and a random y coordinate (you should not use the same number for both coordinates every time, though! You will have to use `CALL RANDOM_NUMBER()` twice if you are using separate variables for x and y)
- You must determine whether the random location falls within the radius of the circle, if it does, increment the counter for the circle.
- After the loop ends, you must first find the value of π given by your analysis $\left(\pi = 4 \times \frac{N_{\text{circle}}}{N_{\text{square}}}\right)$. This should be output to the screen.
- Then you should find the percentage difference between your value and the stated value of π. To obtain a stated value, use `4.0 * ATAN(1.0)` (as arctan(1) or $\tan^{-1}(1)$ is $\pi/4$ radians.) This should also be output to the screen.
- When you run your program, run it with the command:
  `:> time ./assign_10_1`
  This will output timing statistics when it finishes running.
- You should run your program with a number of iterations n of 1,000, 10,000, 100,000 and 1,000,000. Each time record the number of iterations used, the '`usr`' time from the output of the `time` command, the value of π and the percentage error. These should be written into a document `assign_10_1.notes`, followed by brief comments on anything you notice regarding the times, or percentage errors, as n was varied. (At least a couple of sentences, no more than a couple of paragraphs.)
- Keep your source code file, `assign_10_1.f90`, and your file with values and notes, `assign_10_1.notes`, as you will be required to submit printouts of both.

10.1.4 Analysing The Stochastic Program

You should be careful about how many iterations you permit your program to run for. The maximum recommended number of iterations is 1E6, one million. For reference, several sample runs of a test implementation of this program were done on a machine a few years old (Intel Core 2 Duo E700, 2.93GHz per core; 2009). For a run with one million iterations, it took 0.09 seconds to complete, and provided a value for π with a percentage error of 0.0014% of the stated value. Using one hundred million iterations, it took 5.75 seconds to complete, with a percentage error of 0.0012%. One billion iterations took approximately 57 seconds, with a percentage error of 0.0002%. A final test of ten billion iterations took 9 minutes 40 seconds and provided a percentage error just smaller than 0.0002%.

This shows a failing of this method, that there is clearly a limit in accuracy that can be attained. As the time taken for a given number of iterations is basically linear, ten times more iterations would take approximately 110 minutes, however, it would likely not improve in accuracy by a great degree. The method is also limited by the extent to which the random number generator is actually random. Almost all random number generators in computers are only pseudo-random; some manner of algorithm to give values with a random distribution, but that are fundamentally deterministic. In the above figures, a fixed random number generator was used, with a fixed 'seed'.

An additional limit which you will find with your codes is that of 'precision'. A regular `REAL` variable only gives precision of about 7 decimal places. For many purposes, this is insufficient for these highly mathematical processes, and will result in poorer margins of difference between your calculated values and the reference values.

Whilst remaining within a million iterations will likely only provide you with an error of the order of the 22/7 approximation; the task illustrates the concept of using stochastic computational techniques for a mathematical or scientific purpose.

10.1.5 Computational Simulation

It is difficult to adequetely cover the concepts of scientific computing in an introductory course, as the methods and applications are many and varied. One aspect is to write a program such that a simulation of a physical phenomena is performed. Observing how this simulation acts to various starting conditions and effects gives an insight into how the real physical system may act. A great many variations in this field exist with interlinking and intertwining between the methods, techniques, physics and maths.

One catagory of computational simulation is that of 'n-body simulations'. These simulations can vary from fairly simple to highly complex, however their basis is in the fact that multiple bodies interacting (any more than 2 bodies, hence 'n') can be evaluated at intervals in time (known as 'time steps') in order to gradually observe approximately how these bodies should move with respect to one another.

An example of a physical situation this may simulate is a planetary system. When multiple planets orbit a star, they are not necessarily stable (in fact, cannot be proved to be stable). Not only is each planet orbiting the star, through that star's gravitational effect on it; but it is pulled about and perturbed very slightly by every other planet orbiting the same star. This means that, while a single planet orbiting a single star may trace a permanent Keplerian orbit, in reality there are multiple sources of gravity pulling on the planet, and at different strengths at different points of it's orbit. Through this the planets will deviate from fixed orbits over time. Some systems will fall into a basically stable configuration (such as the solar system), some may instead engage in large resonances or close encounters which thrown objects inwards or outwards, potentially scattering some systems completely, or plunging some of the planets into their host star.

Using an n-body simulation, such situations can be evaluated to varying degrees of certainty. One issue with n-body problems is that they are chaotic, the results are highly non-linearly tied in to the starting conditions. An absolutely tiny change in initial conditions could make barely any difference later on, or it could just as easily completely change the outcome. Such problems, while not entirely certain with computational modelling, are exceptionally difficult to model analytically, using the mathematics of the system alone. In particular with the advent of faster and more powerful computers, this makes simulations incredibly useful, not just for n-body problems, but for any similarly dynamically varying problem (where the change to any given variable with respect to another, is highly dependant on the changes and states of many other variables at once).

Another catagory is fluid dynamics, this is the simulation of fluidic motion and similar effects. The applications for this are numerous, from engineering (flow of air past an object such as a wing or car), to the internal structure and motion of the sun ('magneto-hydrodynamics' where magnetic effects as well as fluidic are accounted for) and far more.

Attempting to summarise all of the catagories of simulation here would be almost impossible, due to the variety inherent in study of so many physical phenomena; however, hopefully this gives you some idea of the scope and utility of the programming techniques you have been learning within this field.

10.1.6 Numerical Methods

Numerical methods, or numerical analysis, is the process of applying computational, discrete analysis, to solve (or, rather, to obtain *approximate* solutions for) mathematical analytical problems. Numerical methods have a wide range of applications, any problem involving an analytical mathematical problem where solving it by hand may be excessively time consuming or even impossible. Solutions to simultaneous equations, differential equations, matrix problems, determining eigen vectors/values, as well as many other areas. Strictly speaking, numerical methods can't be formally catagorised as a sub catagory of scientific computing, alongside the previously described topics of stochastic methods and simulation, as elements from within these modes of analysis constitute numerical methods themselves.

Numerical methods could, therefore, be described as the field of study which provides translation between pure mathematical methods, and the capabilities of a computer.

The subject of numerical methods being this broad means that providing a brief introduction, like some of the other aspects of this week's work, would be difficult. As such, we will be focusing down on a single aspect of the topic, with limited analysis of the underlying mathematics. For that, there is an entire module later on in the course.

10.1.7 Newton-Raphson Method

The topic for consideration in this subsection, as well as the assignment associated with it, can be related to a questions which appears simple: How do you find the square root of a value? A calculator or computer would be the likely choice, these too, however, need a method to find a square root of their own. So, how is it calculated by a computer or calculator?

In fact, there are a variety of methods for finding the square root of a number, effectively all of which make use of some method or practice which falls into this topic of 'numerical methods'. The method we shall examine is called the 'Newton-Raphson Method'. The maths will be briefly outlined; however, as it is not the maths which is the focus of the course, it will not cover the necessary detail to understand the full background to the problem, and the manner of the

solution.

The Newton-Raphson Method is an iterative application of a mathematical equation, intended to converge on an approximate solution to a chosen mathematical problem to be solved.

For a suitable equation to be solved (there are criteria regarding the mathematical nature of the problem, not all problems can be analysed), we have some function of x, $f(x)$; this function is illustrated by the red line in the three sub-plots of Figure 5 (next page), where it is plotted as $y = f(x)$, we wish to identify the point it crosses the y-axis, ie where $f(x) = 0$.

We start at the first iteration, shown by Figure 5a, where x_0 is an initial guess at the value. At the y value of $f(x_0)$, if we plot a straight line from here, at the same gradient as the function $f(x)$ has at that point, we obtain the line shown in green. This line would have an equation of the form $y = mx + c$. In this case, our gradient is $f'(x_0)$ the same gradient as our original function has at the location x_0. This means that $c = f(x_0) - f'(x_0)\, x_0$ (which can be determined by geometry and the known y position of $f(x_0)$). This gives an equation for the line of:

$$y = f'(x_0)\, x - f'(x_0)\, x_0 + f(x_0). \tag{8}$$

Finding the location at which a straight line crosses the y-axis is easy, we set Equation 8 equal to zero, and solve for x. This new x which we are solving for is the next x in the series following x_0, and as such is notated x_1. This results in the equation:

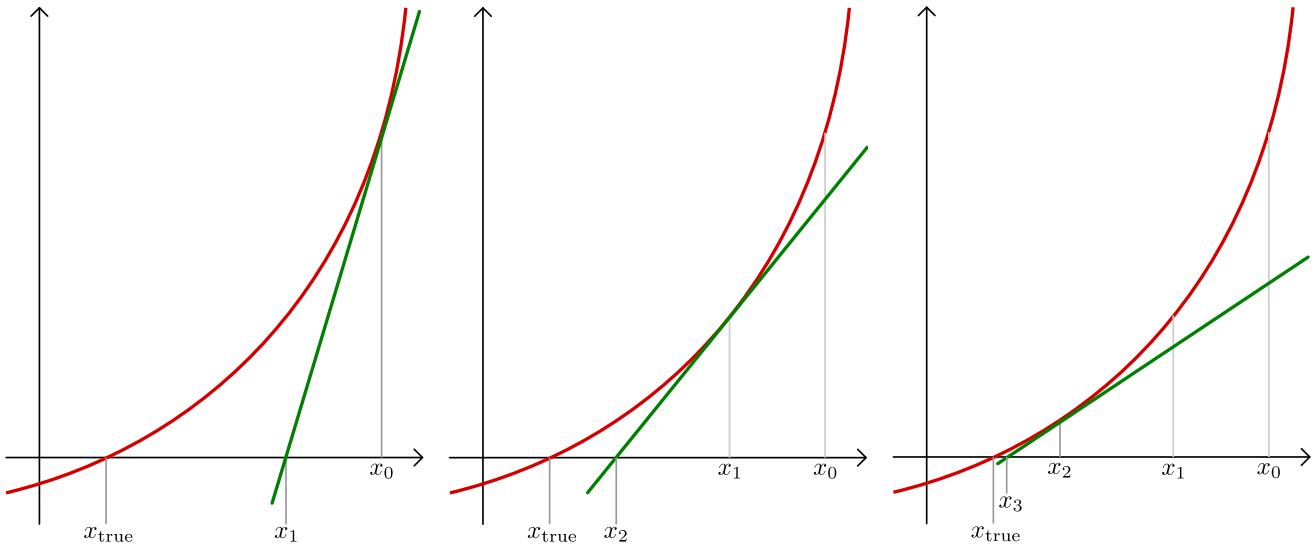
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{9}$$

Note how, in Figure 5a, the solution to $y = 0$ for this line results in a value of x which is closer to the solution for x of our original function, $f(x)$ than our guess at the solution.

What this means is that, after applying this straight line projection from the known point on our original function, we will obtain a new value of x which is a better approximation of the solution than before. What we are doing is attempting to find a value of x closer to the real solution with each step we take.

Having calculated the new value of x, this can be plugged back in to the same equation again to obtain the next version of x. This is illustrated in Figure 5b. We have found the value for the original function at our new x value, x_1, giving $f(x_1)$; we perform the same procedure of projecting a line from this point with the gradient of our original function, finding x_2, an even better approximation of the true solution:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}. \tag{10}$$



(a) First Iteration (b) Second Iteration (c) Third Iteration

Figure 5: Illustration of several iterations of the Newton-Raphson Method.

The final panel, Figure 5c, shows one further iteration of the method, reaching an x value which is closer still to the value we wish to find. The important generalisation to make from these specific instances is the equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{11}$$

In each case of x with a subscript, x_i, the 'i' represents the iteration number for that version of the x value. In other words, x_0 is before our iterations begin, the guess value; x_1 is the result of x for the first iteration/application of the

equation, and so on. By repeatedly applying this process, any equation with suitable criteria upon which the method can be applied may be solved to an arbitrary precision.

10.1.8 Newton-Raphson Method with Square Root - Assignment Material

Using the material from the previous subsection, we will now attempt to apply the Newton-Raphson Method to obtaining a solution for the square root of a number, as posited in the introduction to that section. So, for our case of the square root the method is the following: We are trying to find x, which is the square root of a, ie $x = \sqrt{a}$. Arranging this as a suitable function of x, we can get '$0 = x^2 - a$'; it is this form of the equation which we will attempt to solve with Newton-Raphson. This equation needs to be plugged into the form expressed in Equation 11 above. Doing this, we get:

$$x_{n+1} = x_n - \frac{{x_n}^2 - a}{2\,x_n}, \tag{12}$$

where the '$2\,x_n$' term is the differential of our function '$x^2 - a$'. To start the process, we use an initial guess at what x may be, the first x is that guess value. Each time the equation above is calculated, the resulting value of x is a better approximation of the square root of a than the previous one (subject to a couple of caveats, though those are beyond the scope of this assignment).

The following sequence is an applied example of this repetition, the problem we wish to solve is the square root of 200; in this example we make an initial guess of 20 as our value of x (ie $x_0 = 20$). For reference the real solution, to 8 significant figures, is 14.142136. So for our first step we have:

$$x_1 = (20) - \frac{(20)^2 - 200}{2 \cdot (20)} = 15.000000.$$

So, for our next iteration, we are trying to find the next x, x_2, using the x_1 value we have just found:

$$x_2 = (15) - \frac{(15)^2 - 200}{2 \cdot (15)} = 14.166667,$$

then,

$$x_3 = (14.166667) - \frac{(14.166667)^2 - 200}{2 \cdot (14.166667)} = 14.142157.$$

and finally,

$$x_4 = (14.142157) - \frac{(14.142157)^2 - 200}{2 \cdot (14.142157)} = 14.142136.$$

which gives us the full 8 significant figures of accuracy we desired (more, in fact, if we were tracking the value to a higher precision). As you can see, even after only 3 iterations, we already had a value accurate to 6 significant figures. As another example, the following sequence starts with a much poorer initial guess, 100:

$$\begin{aligned}
x_1 &= (100) - \frac{(100)^2 - 200}{2 \cdot (100)} &&= 51.000000 \\
x_2 &= (51) - \frac{(51)^2 - 200}{2 \cdot (51)} &&= 27.460784 \\
x_3 &= (27.460784) - \frac{(27.460784)^2 - 200}{2 \cdot (27.460784)} &&= 17.371949 \\
x_4 &= (17.371949) - \frac{(17.371949)^2 - 200}{2 \cdot (17.371949)} &&= 14.442381 \\
x_5 &= (14.442381) - \frac{(14.442381)^2 - 200}{2 \cdot (14.442381)} &&= 14.145257 \\
x_6 &= (14.145257) - \frac{(14.145257)^2 - 200}{2 \cdot (14.145257)} &&= 14.142136
\end{aligned}$$

Despite a very poor initial guess, six iterations of the method resulted in all eight significant figures. As it stands, the reason for this method working may not be clear. However, for the purposes of this assignment that is not an issue, we are concerned with the computational method, rather than the mathematics behind it (which, as mentioned, is dealt with in depth in a later module). There are plenty of additional sources and reference books which explain the method in full, should you be interested or unsure about the way it works.

Your task is to implement this method for finding a square root as a program, which should be written in a file called '`assign_10_2.f90`'. The following bullet points specify the requirements:

- The program should ask the user what value they would like to determine the square root of, which should then be read in as a `REAL`.
- The program should ask the user to input a value to use as the initial guess for the solution, which, again, should be a `REAL`.
- The program should finally ask how many iterations it should perform to calculate the value, which should then be read in as an `INTEGER`.
- Using the above values, a loop should be performed between 1 and the total number of specified iterations.
- Each step of the loop, the current value of the 'solution' should be used to calculate the next value of the solution, in the form specified by Equation 12.
  Hint: Remember, the assignment of a value for a variable evaluates the right side of the equals before performing the assignment itself. This means only one variable for x is needed to calculate each new value; you don't need '`old_x`' and '`new_x`' or similar. If you are still unsure, see the worked solution for the mid-term mock class test for an example of updating the value of a variable within a loop.
- After each new estimation of the solution is calculated, it should be output to screen.

After checking that your program works, you should modify it such that it outputs to a file, '`assign_10_2.out`'. the criteria of the problem (value you are trying to find the square root of, the initial guess, and the number of iterations), followed by each sequential new solution during the course of the loop (as was output to screen before). All these values should be accompanied by suitable text describing what they are.

The values that you should finally run your program with to produce a final output are as follows:
Value to find the square root of: 131.28
Initial guess of: 200.0
Number of iterations: 10
This should converge to the true value of 11.457748 (to eight significant figures) after about 8 iterations (this may vary with exact implementation and rounding errors or similar).

10.2 High Performance Computing (Optional)

This subsection is optional, it is here to read if you have an interest in the topic; but if not, or if you have yet to finish the assignments for the week, it is worth leaving it until another time.

High Performance Computing is a catchall term for the use of what might generally be termed supercomputers. More specifically it is the development and use of codes for, generally, highly parallelised machines, either supercomputers with many cores, or a clustered/distributed computer system.

The current paradigm of HPC is parallelisation. It is becoming increasingly difficult to increase the speed of a single processor core, this increases their complexity and cost with only gradually increasing speed. Therefore, rather than going for especially high core speeds, a slower core speed can be used, but you set up code such that you can have many cores processing different parts of the same program at once. By doing this you can obtain huge effective speeds.

For reference, as of originally writing this (mid-to-late 2010), the number one[20] supercomputer in the world based on the 'Linpack' benchmark[22] was the Cray '*Jaguar*' machine at Oak Ridge National Laboratory in the US[23]. It housed 224,162 cores, each running at 2.6 GHz, a total of 300 TB of memory, and had a performance of over 1.75 petaflops, and peak ideal performance of 2.3 petaflops. A 'flop' is a 'floating point operation per second', in other words one flop would be the ability to do one basic mathematical operation on a floating point value per second, 'peta-' is the SI prefix for values meaning 10^{15}. Therefore the computational ability of *Jaguar* is to perform over 1.75×10^{15} or 1.75 million billion floating point operations per second! It is almost inevitable, given the rate of advance of computational technologies, that in future, looking back at this document, this value will either appear run-of-the-mill, or be a parameter which has ceased to be relevant.

The latest update to the 'Top 500' supercomputers list (the November 2013 listing)[21] is the '*Tianhe-2*' machine at the National Super Computer Centre in Guangzhou. In the intervening time between *Jaguar* and *Tianhe-2*, the same list (published at six month-intervals) has given the top spots to another computer in China, one in Japan and two different machines in the US. The *Tianhe-2* possesses 3,120,000 cores, 1,024 TB of memory, a performance of over 33.9 petaflops and peak ideal performance of 54.9 petaflops. This constitutes a twenty-fold increase in performance in approximately three years. Fast enough that updating this information every time a new machine takes the lead would be quite a hassle (hence retaining the original description of *Jaguar* at the beginning of this paragraph, as well as it giving an impression as to how dynamic the field can be). The Cray *Jaguar* described at the beginning is no longer running, but in its original configuration (it underwent an upgrade in early 2012) it would place 16^{th} in the current listing.

Writing code designed to be run in parallel can be a difficult task. It must be very carefully considered which sections can be parallelised and which can't, as well as whether multiple 'threads' (the name for any one of the parallel instances

of the program) will ever be in a situation where they would try to read or write from the same location in memory at the same time (which would cause at least unexpected behaviour, and possibly simply crash). Given these difficulties, it is not part of this course to write a parallelised computer program. However, as such a central topic within scientific modelling or computational work of any sort, it is worth covering the basic concepts of parallel computing.

The majority of tasks which are not sequential are likely to be candidates for being made parallel. Consider a large list of numbers, and you wish to perform a simple mathematical operation on all of them where, for each element, no other element is involved. For example just to multiply each by two. This is easily made parallel; each element could be sent to separate processors, along with the code required to multiply it by two. All of the processors would perform the task concurrently, and the entire list would be completed in one step.

In the case of our earlier problem of stochastically determining π, the order in which the points are 'dropped' onto the square/circle is irrelevant, you could do them in batches, and then simply count the ratios across both of the batches. This would mean that you could calculate the same approximation of π across as many machines as desired. Even given this, the method we have used is still not the most efficient for finding π, a great many more mathematically subtleties can let a better value be produced.

10.3 Recursion (Optional)

This subsection is optional, it is here to read if you have an interest in the topic; but if not, or if you have yet to finish the assignments for the week, it is worth leaving it until another time.

The inclusion of the topic of recursion within this section is something of a side note. It can be a powerful tool within scientific programming, however, it also finds much use outside of those confines in computing in general. You may recall the mention at the beginning of the course that the compiler you have used, '*gfortran*', is written by the '*GNU*' group. It was mentioned that this name *GNU* was a recursive abbreviation standing for '*G*NU's *N*ot *U*nix'. What this means is that the abbreviation itself is one of the words being abbreviated. Were you so inclined, you could continue to expand the abbreviation indefinitely. While this is somewhat of a trivial example of recursion, it obeys the same concepts as the recursion used within computing.

We have already seen how you can call functions from within a program; additionally, functions themselves can call other functions. The following code is not formal fortran, rather a fortran-like pseudo-code, to illustrate the concept. For a program '`main`' calling a function '`funcA`', which in turn calls a function '`funcB`', we have the following:

```
PROGRAM main
  y = funcA(x)
END PROGRAM main

FUNCTION funcA(x)
  funcA = funcB(x)
END FUNCTION

FUNCTION funcB(x)
  funcB = x**2
END FUNCTION
```

The `x**2` in `funcB` simply represents some manner of process performed on the variable `x` which isn't reliant upon another user defined function. So, having established this process of stringing together functions; what happens with the following example:

```
PROGRAM main
  y = funcA(x)
END PROGRAM main

FUNCTION funcA(x)
  funcA = funcA(x)
END FUNCTION
```

This time there is only one function, but it calls *itself* rather than another function. This process is called recursion; when `funcA` is called, it calls another instance of itself, which in turn calls *another* instance of itself, and so on. If calling itself was all that funcA did, it would enter an infinite recursion, constantly creating new instances of itself with no criteria for stopping. It is essentially the same as pointing a webcam at a monitor displaying the images from the webcam itself; this creates a nest of images on the screen as each frame from the webcam contains the monitor showing the frame of the webcam and so on.

Infinite recursion, with no stopping criteria, is seldom, if ever, to be desired. However, if used properly, with conditions which permit a particular instance of the function to be the final one, not calling another instance at that point, recursion can be an extremely powerful tool. The following is a working example of a recursive function. The task it performs is

simply to keep adding '1' to a value until 20 is reached. Normally this would not be a situation where recursion would be the ideal solution, there are much easier and faster ways to accomplish this. However, it illustrates the key elements of making a recursive routine.

```
1  PROGRAM recursion_example
2    IMPLICIT NONE
3    INTEGER :: x, y
4    INTEGER :: addone
5
6    x = 1
7
8    y = addone(x)
9
10   WRITE(*,*) y
11
12 END PROGRAM recursion_example
13
14 RECURSIVE INTEGER FUNCTION addone(varin) RESULT(varout)
15   IF (varin .LT. 20) THEN
16     WRITE(*,*) 'Adding One'
17     varin = varin + 1
18     varout = addone(varin)
19   ELSE
20     varout = varin
21   END IF
22   RETURN
23 END FUNCTION addone
```

Several things have been introduced in this stage of going from pseduo-code to actual f90 code, so we will look at it step-by-step. Firstly, we have the main program, `recursion_example`. On line 4 we have defined a variable to represent the function in the same manner as we would for a normal function. There are also two `INTEGER` variables defined for storing our values, `y` and `x`. We set `x` to `1` to begin with, and then set `y` equal to the result of calling the function `addone` with `x` as its argument.

This brings us to the recursive function itself. Line 14, where the function is defined, has two key differences to regular functions. The first is that it is preceded by the statement `RECURSIVE`, this is required to let the code know that the function should be allowed to call itself. The second is the addition of `RESULT()` following the name and arguments of the function itself. This specifies the name of a variable to be created to send the result of the function back through; conceptually, it is similar to using a variable defined with `INTENT(OUT)` in a subroutine.

Within our function, the main construct is a single `IF-ELSE` statement. One line 15, there is a check to see whether the input variable, `varin`, is now the desired target, `20`. If it is less than this, we `WRITE` a statement to the screen, and add one to the input variable (line 18). After this, we call the function again, with the, now increased by one, input variable. As the next instance is called, it is as though the current instance is paused, it still exists, but is waiting for the next instance to give back a value. The next instance will again check if the input has reached `20`, calling more instances until it does.

When an instance is called where the input variable *is* 20, rather than call the function again, we set the output variable `varout` to be equal to the input variable on line 20, which is now the desired number. Then we tell the function to `RETURN`. At this point, it is like all of the instances of the function nested within one another, fold back out or unravel, passing the final output back up the chain until the original instance of the function passes the result back to the main program.

10.4 Assignments Summary

10.4.1 Programming Assignment 1

1. Create a file '`assign_10_1.f90`' for your program code.
2. Your program should request that the user enter the seed for the random number generator. This can be either all eight values to use for the seed, or a single value, from which you generate eight numbers using some fixed rule (see this assignments Section for more details).
3. You should make use of a length eight array with the seed values, and the command `CALL RANDOM_SEED(PUT = seeds)` (where '`seeds`' is your array name), to initialise the random number generator.
4. You will need a loop of a length n (the value for n may be hardcoded, you do not have to read it in). This value will implicitly be the total count of objects within the square, as it is not possible for a coordinate pair to be outside of

the square.
Hint: Ensure that the step prior to this, setting the random number seed, is done before/outside of the loop. Otherwise, it will reset the seed on every iteration, resulting in only one 'random' number!

5. Within your loop, you will need to use the `CALL RANDOM_NUMBER()` command to create a random x coordinate, and a random y coordinate (they must not be the same number though! You will have to use `CALL RANDOM_NUMBER()` twice if you are using separate variables for x and y)

6. You must determine whether the random location falls within the radius of the circle, if it does, increment the counter for the circle.

7. After the loop ends your program must find the value of π given by $\pi = 4 \times \frac{N_{\text{circle}}}{N_{\text{square}}}$. This should be output to the screen.

8. Then you should find the percentage difference between your value and the stated value of π. To obtain a stated value for π, use `4.0 * ATAN(1.0)`. This percentage difference should also be output to the screen.

9. When running your program, use the command:
```
:> time ./assign_10_1
```
This will output timing statistics when it finishes running.

10. You should run your program with a number of iterations n of 1,000, 10,000, 100,000 and 1,000,000. Each time record the number of iterations used, the '`usr`' time from the output of the `time` command, the value of π and the percentage error. These should be written as a short table into a document `assign_10_1.notes`, followed by brief comments on anything you notice regarding the times, or percentage errors, as n was varied. (At least a couple of sentences, no more than a couple of paragraphs.)
Note: You may also want to experiment with different seeds, commenting on what changes you notice with the percentage errors with different values.

11. Print both `assign_10_1.f90` and `assign_10_1.notes`.

10.4.2 Programming Assignment 2

1. Create a file '`assign_10_2.f90`' for your program code.

2. The program is to use the Newton-Raphson method to find the square root of a value a such that $x = \sqrt{a}$.

3. The program should first ask the user for the value to be square rooted, a in the form above, which should be read in as a `REAL`.

4. It should then ask the user for an initial guess of the value of x, which should also be read in as a `REAL` variable.

5. Finally, it should ask the user for how many iterations it should perform, to be read in as an `INTEGER`.

6. The program should open a file to be called '`assign_10_2.out`', and output the values entered by the user, *along with suitable text to explain which value is which.*

7. You should code a `DO` loop to go through the specified number of iterations.

8. Each iteration, the program should calculate a new value of x, x_{n+1}, such that $x_{n+1} = x_n - \frac{{x_n}^2 - a}{2x_n}$, as shown in Equation 12 of Section 10.1.8.

9. For each newly calculated x, the program should output that value to both the screen *and* the output file opened prior to the loop.

10. After the loop finishes, ensure that the output file is closed before the end of the program.

11. Your output to hand in should consist of the values produced by your program for the following settings:

    - Value to find the square root of: 131.28
    - Initial guess of: 200.0
    - Number of iterations: 10

10.4.3 Questions

There are no questions for this week's work. However, it is recommended that you begin reading through and working on the Week 11 work as soon as it is available having finished this week's work. It is more project-like than previous assignments, requiring greater application of problem solving skills to determine what needs to be done to achieve the stated goal.

We require a hard copy version of the code for each of the weeks assignment programs, this is to provide us with a copy which we can mark and give you feedback and suggestions of where to improve if necessary. You will also need a hard copy of the answers to the weeks questions, these may be hand written or typed up and printed. All paper work handed in *must* have your name on, and preferably be suitably bound (stapled, preferably).

For this assignment, you will be required to submit (in addition to the answers to the questions): `assign_10_1.f90`, `assign_10_1.notes`, `assign_10_2.f90` and `assign_10_2.out` . To print the required files from PCSPS07, you should transfer the relevant files to your own machine using WinSCP or similar (described in Section 1.2.5) then open them with an editor like wordpad to print.

11 Week Eleven - Final Assignment

Important Note: You have more time than usual to complete this assignment. In addition to the workshops of this week (Week Eleven), you will have the Thursday and Friday sessions of next week (Week Twelve) to work on this material. This means that the deadline for this work is on **Friday the 11$^{\text{th}}$ of April** (the end of that day's workshop session).

Ensure that you read the content of this chapter carefully. There are several new concepts introduced, such as time step iteration and use of a different unit system; you must make sure you have understood their descriptions in full to avoid making mistakes.

11.1 Topic and Overview

The focus of this final assignment is the programming of a simple computational simulation. The scenario which you will be programming is based on the Rutherford Scattering Experiment. You may already be familiar with the set of experiments where, in the early 20$^{\text{th}}$ century, α particles (helium nuclei produced from radioactive decay) were fired at a sheet of gold foil. Through knowledge of electrostatic interactions the experiments showed Ernest Rutherford that, rather than the positive and negative charges of an atom occupying the same diffuse space (the 'Plum Pudding' model), the positive charge was in fact concentrated in a far smaller volume than that occupied by the negative charges of the electrons. This eventually led to the idea of a tiny nuclei, containing positive particles (the protons) surrounded by orbiting electrons (literally orbiting, until the advent of Quantum Mechanics showed this to not be the case).

Using the knowledge of the charge and mass of a helium nucleus and the charge of a gold nucleus, a simulation involving two particles will be constructed. A fixed gold nucleus will be assumed to exist at the origin and the motion of an incident α particle modeled with discrete advances to the simulation time. The two particles will be required to interact through the force produced by a simple Coulomb potential $U = \frac{1}{4\,\pi\,\epsilon_0}\frac{q_1\,q_2}{r}$ (where ϵ_0 is the permittivity of free space, r is the distance between the particles, and q_i the charge of particle i). The terms $\frac{1}{4\,\pi\,\epsilon_0}$ are frequently replaced by the single term k_{e}, the Coulomb constant (as the components of set of terms it replaces do not change). The simulation should permit selection of two properties for the α particle, its initial kinetic energy, T, (and hence its initial velocity) and its initial offset of approach to the gold nucleus.

Output will be the positions of the α particle at each step in time, produced in a file of fixed format. The trajectory of the alpha particle relative to the gold nucleus can then be plotted from this file using a series of commands given later in this document. The material required for submission will be:

- Any written code (your main code and any external modules/functions/subroutines)
- Plots of several runs of the simulation with specific starting criteria
- The first and final ten lines from a run of the simulation with specific starting criteria.
- The answers to several questions relating to the nature of the simulation.

11.2 Background

What is known as the Rutherford Scattering experiment were the specific set of experiments carried out by Hans Geiger and Ernest Marsden, working with Rutherford, the initial results of which were published in 1909[25] and 1910[26]. The experimental setup used was to generate a beam of α particles using a radium source, targeted at a sheet of metal foil. By placing scintillator detectors at positions around the metal foil, the number of α particles deflected by the foil in the direction of the scintillator could be determined. Various metals were used to determine the effect that materials of different atomic weights had on the amounts and angles of reflection. Gold foils were used for the subsequent tests within the experiment, as production methods permitted creation of extremely thin foils ($\approx 0.5\mu$m). These gold foils were used to observe the effects of different thicknesses of material as well as further examining the proportions of α particles reflected[25]; it is these results with the gold foil that became the most well known.

The results of these experiments were then given further physical meaning by Rutherford himself, adapting the 'Plum Pudding' model of the atom determined by J.J. Thompson a number of years before. J.J. Thompson's model of negatively charged particles (originally 'corpuscles', though later 'electrons') had managed to explain some aspects of deflection of electrically charged particles[27], but permitted only small accumulative deflections though many sequential interactions. This limit to very small deflections only was implied especially for particles which were heavy and had high velocity such as α particles, suggesting that their scatter should be extremely limited. The experiment conducted by Geiger and Marsden would be problematic in this regard, as they observed much higher scattering, and to much greater angles, even back-scattering with the α particles reflecting back from the foil entirely.

Rutherford derived that the observed scattering could be accounted for by a tiny compact positive charge at the center of the wider space of negative charge created by the electrons. The interaction with such a compact and condensed

positive charge would support scattering of the nature observed with the α particles. This was contrary to the idea that the positive charge was diffuse, occupying the same physical space as the electrons[28]. This was a seminal result, having effectively determined the existence of a nucleus based atom, additionally finding that the effective charge of this nucleus was approximately proportional to the atomic mass. Rutherford's theory laid the groundwork for subsequent models of atoms, and for discovery of the component particles of the nucleus.

11.3 The Simulation

11.3.1 Scenario

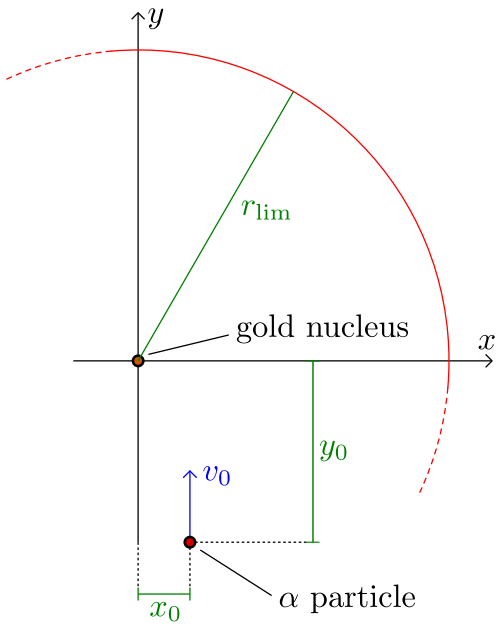


Figure 6: Starting geometry for the simulation.

The simulation you will code will mimic the interaction between a single α particle incident on a single gold nucleus. The simulation will be in two dimensions as, when only two particles are considered, any interaction and movement can be considered as being within a single plane. Figure 6 illustrates the setup of the simulation you will be required to code, as well as labeling the important parameters with a notation which will be used for all further descriptions.

In Figure 6 we see all of the key components necessary to define the initial condition of the simulation. The labeled circles are: the gold nucleus, which will be fixed at the origin of the coordinate system $(0, 0)$, and the moving α particle. The initial location of the α particle are given by the two parameters x_0 and y_0. y_0 being its 'vertical' distance from the gold nucleus (which can remain fixed for all of the tests), and x_0 being the 'sideways' displacement of the α particle, which will be one of the parameters which can be set for each test. The other kinematic property of the α particle is its velocity, the initial velocity $\mathbf{v_0}$ having the components $v_{0x} = 0$ and $v_{0y} = |\mathbf{v_0}|$, ie its initial velocity is only in the positive y direction. The final parameter of importance is r_{lim}, this will be the boundary of the computational simulation and act as the criteria for the simulation ending. If the α particle is determined to have crossed this boundary, the simulation should finish outputting results and end successfully (ie, it is not an error).

11.3.2 Calculation Process

The previous subsection gives us the starting condition for the simulation, but gives none of the process for the simulation to actually run. Several concepts must be considered, which will be described here to an extent, and will require you to complete and implement.

Firstly: Discretisation. Computers fundamentally deal with discrete numerical steps, they cannot deal with truly continuous systems. Any system to be dealt with computationally must be broken up into chunks of specific sizes, each chunk to be dealt with one at a time. As such, converting a continuous problem (ie, most mathematical equations) into a computational simulation to approximate a solution, involves determining a manner in which the system can be made into discrete steps without interfering too much in the resultant behaviour.

For dynamic systems like this which vary with time, we need to consider the concept of a 'time step'. A time step in a computer simulation is the unit of time by which the simulation advances with each iteration. It is the smallest time in the simulation that any property can change over, when a time step is calculated all physical properties are assumed to stay constant for that duration. Because it is not the case that they *will* all stay constant, it is a potential source of

inaccuracy. However, for the purposes of this assignment, you need only understand how they work, not why they do, or the criteria for which they remain valid.

Force

So, in a given step of time for the system what needs to be calculated, and what does it result in? We must consider what properties the moving particle possesses, and what effects are acting on that particle. The main effect acting on the particle will be the force from the Coulomb interaction with the gold nucleus. We wish to determine a new location for the particle based on its old location and this Coulomb force acting on it. So, for our α particle at an arbitrary location of $\mathbf{r}_i = (x_i, y_i)$, with charge q_α, interacting with the gold particle at the origin $(0,0)$ and possessing charge q_{Au}, we wish to determine a location at a new time step $i+1$, $\mathbf{r}_{i+1}$. The force on the α particle is:

$$\mathbf{F}_{i+1} = k_\text{e} \frac{q_{\text{Au}}\, q_\alpha}{|\mathbf{r}_i|^2} \hat{r}_i = k_\text{e} \frac{q_{\text{Au}}\, q_\alpha}{{x_i}^2 + {y_i}^2} \hat{r}_i, \tag{13}$$

where the distance between the particles, $\mathbf{r}_i$ is the hypotenuse of the x and y components of the α particle's location for the step i. (*Note:* This is because the location is always in reference to the origin, if the two particles interaction were at two different arbitrary locations, $\mathbf{r}_i$ would have to be one location minus the other.) The remaining parameter, k_e, is the Coulomb Constant, a consolidated term for the expression $\dfrac{1}{4\,\pi\,\epsilon_0}$ which appears in the Coulomb potential and force equations.

Remember, the program thinks in specific time steps, all of the indices indicate the property at that step. If you'd prefer, you can think of the index as a kind of time itself, ie $\mathbf{F}_{i+1}$ is the force for our new 'time' $i+1$, $\mathbf{r}_i$ is the particle's current location.

Note that the force calculated is still a vector quantity (indicated by the variable symbol being in bold, in case that notation is unfamiliar), in the direction pointing from the α particle to the gold nucleus. To simplify considering the components individually, we can determine the magnitude of the force separately to its direction. It is not necessary to do this, however, if you write your program to make use of a vector format, with suitable vector operations. The magnitude of the force is:

$$|\mathbf{F}_{i+1}| = F_{i+1} = k_\text{e} \frac{q_{\text{Au}}\, q_\alpha}{{x_i}^2 + {y_i}^2}. \tag{14}$$

To make a vector in the same direction as $\mathbf{r}_i$, but with the magnitude calculated in Equation 14, we need to calculate the unit vector $\hat{r}_i$. This is:

$$\hat{r}_i = \frac{\mathbf{r}_i}{|\mathbf{r}_i|}, \tag{15}$$

the vector components scaled by one over the magnitude of the vector. So, since r_i is composed of x_i and y_i, combining this with the equation for the force, we have:

$$F_{x\,i+1} = F_{i+1} \cdot \frac{x_i}{\sqrt{{x_i}^2 + {y_i}^2}} \tag{16}$$

$$F_{y\,i+1} = F_{i+1} \cdot \frac{y_i}{\sqrt{{x_i}^2 + {y_i}^2}} \tag{17}$$

Remember that, while here the vector components have been resolved, if you write functions to work on vector types, the vector form can be used with no further conversion required. It is this method which is recommended, rather than the resolution of the vectors and the treatment of their components separately.

Having determined the components of the force acting on the α particle, we can get its acceleration. From $\mathbf{F} = m\mathbf{a}$ this is simply:

$$\mathbf{a}_{i+1} = \frac{\mathbf{F}_{i+1}}{m_\alpha}, \tag{18}$$

where m_α is the mass of the α particle. Note that this equation is in vector format, if you are working in separately resolved components rather than vectors, this will have to be split into its x and y components as well.

Motion

We have mentioned the concept of a time step, over which the program advances through time. Within a given time step, newly calculated properties are assumed to stay constant. So from the beginning of the time step to the end, acceleration will stay the same within these confines. As such, we can use the fact that a velocity can be defined as:

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_{i+1}\, \Delta t, \tag{19}$$

that, in our period of time Δt, for a specific acceleration $\mathbf{a}_{i+1}$, the velocity at the end of that time step, $\mathbf{v}_{i+1}$, is the velocity the particle currently has, $\mathbf{v}_i$ (the previous value of the velocity), plus the product of the acceleration and the length of time, $\mathbf{a}_{i+1}\, \Delta t$. This means that over the course of the time step, the velocity will have changed from $\mathbf{v}_i$ to $\mathbf{v}_{i+1}$.

In turn, with this new velocity, we can calculate the change in location. What we assume is that, for the whole of the time step, the velocity is already at the value $\mathbf{v}_{i+1}$ which was just calculated, it does not gradually change, it jumps. This is what is meant by the size of the time step being the smallest amount of time any change occurs over. So, it is the

new velocity $\mathbf{v}_{i+1}$ which we will use to calculate the next location $\mathbf{r}_{i+1}$, the former location $\mathbf{r}_i$ add the contribution of the time step of our new velocity:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_{i+1}\,\Delta t. \tag{20}$$

Note that this is not formally correct, you may notice that it can be considered to be 'out of step' in terms of when the update to the velocity is applied compared to the location. Implemented here is a simplification of the problem introducing a non-negligible error. More formal derivations are required for the application of a particle motion where the accuracy of numerical results are relevant for quantitative analysis of the simulation, such as genuine research. These alternate derivations frequently include higher order approximations and would be more difficult to implement as a first step into the field of computational simulation for this assignment.

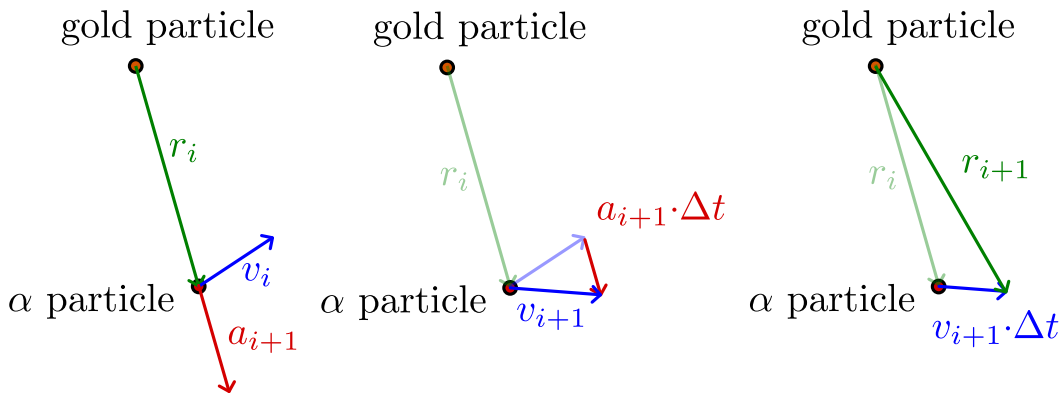
So, this means that we now have a process by which, accounting for the force on the particle, we can determine how the particles velocity should change, and therefore how the particle's location should change. To reiterate, our key equations of motion are as follows:

$$\mathbf{a}_{i+1} = \frac{k_e}{m_\alpha}\frac{q_{Au}\,q_\alpha}{|\mathbf{r}_i|^2}\hat{r}_i \tag{21}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_{i+1}\,\Delta t \tag{22}$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_{i+1}\,\Delta t \tag{23}$$

Again, these are expressed in their vector form. As mentioned, they can be implemented directly in this form if suitable vector operations and variables are defined. If you wish to work with the individual components, then the resolved components of the force shown in Equations 16 and 17 can be used instead, with adjustments to the velocity and location equations to operate on the components separately (ie '$v_{xi+1} = v_{xi} + a_{xi+1}\,\Delta t$' etc.) Also note that the equation for force (Equation 13) has been incorporated directly into the equation for acceleration (Equation 21) rather than calculate force separately. If calculating force as its own 'step' then calculating acceleration using that, is preferred by you, that is an equally valid method. Figure 7 illustrates the progression of the steps shown.



(a) First Step (b) Second Step (c) Third Step

Figure 7: Illustration of the process of updating the location of a particle. r_i and v_i are the current location and velocity vectors of the α particle respectively. r_{i+1}, v_{i+1} and a_{i+1} are the new location, velocity and acceleration vectors respectively; Δt is the time step. Step 1 shows the calculation of the new acceleration (note that the acceleration vector is parallel to the location vector). Note how the new acceleration is *not* dependent on the previous acceleration, unlike location and velocity. Step 2 shows the velocity vector being updated using the new acceleration and the time step. Step 3 shows the location vector being updated using the new velocity and the time step.

Iteration

Having defined our equations describing the motion for a time step, we must consider the iteration through time steps. By looping until the end criteria is met (to be discussed later), with a particular time step Δt, updating the velocities and locations in the manner described, the motion of the particle over time will be calculated. After N steps have been performed, the particle's motion for a time interval $N\,\Delta t$ will have been determined.

Ending Criteria

While it is possible to set a fixed limit of iterations for the simulation, performing the operations described above that many times, it is more convenient to use a limiting boundary. This boundary will be defined as a fixed distance away from the central gold nucleus, if the result of an iteration of the time step places the location of the particle further away from the gold nucleus than this, then the simulation will end. The boundary should be set as follows:

$$r_{\text{lim}} = 1.1 \cdot |\mathbf{r}_0| \tag{24}$$

In other words, the boundary is 1.1 times the distance away from the nucleus that the α particle starts at. For reference, in resolved components this becomes:

$$r_{\text{lim}} = 1.1 \cdot \sqrt{{x_0}^2 + {y_0}^2} \tag{25}$$

11.3.3 The Coding

We have covered the concept of the assignment, the background physics and the maths required to implement the scenario as a simulation. This subsection will be concerned with the nature of what must be actually coded. As the task of the assignment is to write the program, limited direct guidance will be given here, only a basic outline of the process.

It is recommended that you use `DOUBLE PRECISION` for all floating point variables, rather than `REAL`, as this will stave off any problems which may be caused by rounding errors.

The following elements are key to successfully completing this task:

- Variables; consider what physical properties and parameters you must start with and track to fit the equations previously described:
  - Initial setup, offset and starting velocity
  - Properties of the α particle which will vary with time
  - Any constants or similar which may be needed repeatedly
- `FUNCTION`s and `SUBROUTINE`s; consider which processes and actions suit being separated out in order to simplify the flow of the main code:
  - Vector `TYPE` and functions to perform vector operations
  - Any tasks which may take up many lines, and would benefit from being separated to a `SUBROUTINE`/`FUNCTION`
- Coding format and style:
  - Suitable programming practice is especially important for this assignment
  - Ensure that you comment your code, use indentation and choose suitable variable names
  - Consult Section 14 for further details

11.3.4 Production of Output

The form of the output must be two columns, one of the x coordinates of the α particle for each time step, and one the y coordinates. If you wish to additionally output the time and/or velocity/velocity components at each step, these should be put in subsequent columns (ie. any but the first two). You will be required to produce plots of several runs of the simulation, for this it is recommended you use the program '*Gnuplot*' a command-line based plotting utility. The basic version of the Gnuplot script is provided for you and can be obtained using the command:

```
:> cp ~ph302/coulomb_plot.gnu ./
```

You will need to make some changes to adapt it for your own files. The code contained in the script file is shown below:

```
1 set terminal png
2 set size square
3 set xlabel "X-Axis ()"
4 set ylabel "Y-Axis ()"
5 set xrange [*:*]
6 set yrange [*:*]
7 plot 'output.dat' using 1:2 with line title "Alpha Path", \
8 "< echo 0 0" with points pointtype 7 pointsize 2 linecolor rgb 'gold' title "Gold Nucleus"
```

The purpose of the commands are as follows:
Line one sets the output ('terminal') to png (Portable Network Graphics) mode, to create `.png` image file code when run. Line two instructs the code to produce a plot where the axes are square (ie. the same length along both sides). The third and fourth lines set the labels to appear on the x and y axes (the empty open brackets are intended for you to fill in the units used for your output). The fifth and sixth lines set the ranges of the axes, the asterisks instruct the script to work out the limits itself, however, these can be replaced with numbers if you wish to alter the limits manually yourself.

Line seven is perhaps the most important, as it performs the plotting of the data itself. The string directly after the `plot` command instructs the script which file to plot, in this instance the file `'output.dat'` is specified, however, you will have to replace this with whatever your output file is named. The '`using 1:2`' makes it use columns `1` and `2` from the file. '`with line`' tells it to make this dataset plot a line (as opposed to points or similar). The `linecolor rgb 'red'` statement makes the colour for this dataset red (note the use of the American spelling 'color' rather than 'colour', only

the American spelling will work). Finally, `title "Alpha Path"` instructs what the dataset should be referred to as, it is this label which turns up in the plot legend.

The final line is for plotting the location of the gold nucleus at the origin, it is a continued line from the previous `plot` command (note the comma and backslash on the previous line). The "< echo 0 0" is a way of creating a single datapoint at coords $(0, 0)$; `with points` makes it use points rather than a line; `pointtype 7` makes it use a type of point which is a filled in circle; `pointsize 2`, `linecolor rgb 'gold'` and `title` should be self explanatory.

To run the script, once edited to your satisfaction, you must use the following command:

```
:> gnuplot coulomb_plot.gnu > filename.png
```

where '`filename.png`' should be whatever name you wish to give your output file image.

Figure 8 shows an example of the output of the script. Note that this specific plot was produced using the postscript output format to a `.eps` file, rather than in png format. All other script commands used were identical to those provided.

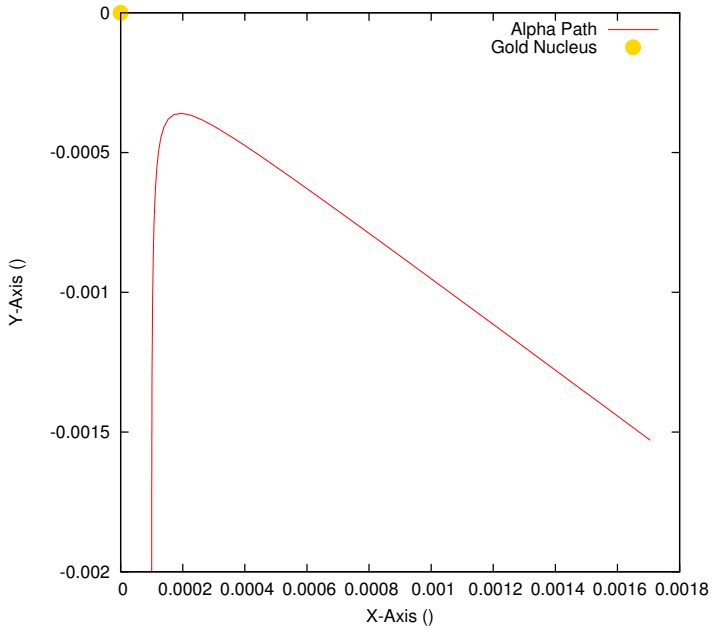


Figure 8: Sample of Gnuplot output.

11.4 Important Values

Within this simulation, rather than work in SI units, it is convenient to choose a different unit system for the calculations that better suits the scenario. Generally, this involves selecting units such that as many of the values used are in the region of '1' or low orders of magnitude. In SI units, for subjects on an atomic scale, many of the values have large positive or negative powers. The speed of light is $\times 10^{8}$, while the distances are of the order of $\times 10^{-11}$; this means that the timescales involved have to be of the order of $\times 10^{-17}$; it also means that the charges of the particles involved are around the order of $\times 10^{-18}$; the mass of an α particle in SI units has the order $\times 10^{-27}$! Using such varied values with such different orders of magnitude can cause problems computationally, with issues with precision and rounding taking effect.

To avoid this, as stated, other unit systems tend to be used within simulations. In the case of this simulation we will use 'atomic units', this is a system predicated on making units based on multiples of several fundamental physical parameters. All masses are in units of electron mass m_e, charge in units of the charge of an electron, e; and several units result from setting the Coulomb constant to 1. In particular, as the Coulomb constant is used for the force calculation, a unit system where that parameter is 1 effectively factors it out of the calculation.

This change in units necessitates no alteration to any of the basic process, nor the equations used. The equations described in the previous sections are independent of the unit system used, so long as that unit system is internally consistent. All it requires is setting starting values which are also based on the chosen unit system. The remainder of this section describes and explains the values which are necessary.

Use of atomic units results in velocities of the order of $\times 10^{1}$, distances of the order of $\times 10^{-3}$, times of the order of $\times 10^{-4}$, masses of the order of $\times 10^{3}$, charges of the order of $\times 10^{1}$; to list several of the key parameters. As you can see, these values are all more manageable than the range of values used within SI.

The unit symbols we will use within atomic units are as follows (ie. these are the a.u. equivilents of meters/seconds/etc.):

Dimension	Symbol	Additional Description
Length	a_0	Bohr Radius
Mass	m_e	Mass of an electron
Charge	e	Magnitude of charge on an electron
Angular Momentum	$\hbar$	Planck's Constant over 2π
Energy	E_h	'Hartree' energy, given by $E_\mathrm{h} = (m_\mathrm{e}\, e^4)/(4\,\pi\,\epsilon_0\,\hbar)^2$
Time	t_A	Corresponds to $\hbar/E_\mathrm{h}$
Velocity	$a_0 \cdot {t_A}^{-1}$	Corresponds to c multiplied by the fine-structure constant α

11.4.1 Physical Constants

The following table contains the values to assume for several physical constants and parameters which you may require to write your program. Note, both SI units and atomic units (a.u.) are listed. However the SI values are simply there for context and comparison, your code should be written to use atomic units only.

Name	Symbol	Value (SI)	Unit (SI)	Value (a.u.)	Unit (a.u.)
Charge of an α particle	q_α	3.204353×10^{-19}	C	+2	e
Charge of a gold nucleus	q_Au	1.265719×10^{-17}	C	+79	e
Mass of an α particle	m_α	6.646478×10^{-27}	kg	7294.3	m_e
Coulomb Constant	k_e	8.987551×10^{9}	$\mathrm{N \cdot m^2 \cdot C^{-2}}$	1	k_e
Speed of Light	c	2.99792458×10^{8}	$\mathrm{m \cdot s^{-1}}$	137.035999	$a_0 \cdot {t_A}^{-1}$
One 'length' in a.u. (Bohr Radius)		5.29177×10^{-11}	m	1.0	a_0
One 'time' in a.u.		2.41889×10^{-17}	s	1.0	t_A

11.4.2 Simulation Run Specifications

To ensure reasonable values are used for important parameters, the following are provided:

- **IMPORTANT:** The time step, Δt, should be set as $1.0 \times 10^{-5}\ t_A$ in atomic units (**Important:** Remember, the t_A is the unit symbol, it does not mean that you need to multiply by the SI value for 1 t_A, the same goes for all other unit symbols as listed in the first table of this subsection (a_0, $a_0 \cdot {t_A}^{-1}$, etc)). Incorrectly scaled time steps very easily cause problems with the simulation. If you wish, you can experiment with varying the time step to observe the manner in which the results change.
- Initial y distance should be $-0.005\ a_0$ in atomic units.
  Note: The minus sign is intentional, if the velocity is positive, the starting location in y should be negative, such that the α particle is heading towards the gold nucleus, not away from it.
- The range of x offsets to select should generally be of the same order of the initial y distance, but can go as low as zero. Both negative and positive values could be used, however, due to the symmetry of the problem, negative values will simply produce a mirror image of the positive value.
- Initial y velocity should be in the region of $0.07 \times c$, $9.592520\ a_0 \cdot {t_A}^{-1}$ in atomic units. However, it is easier to have a variable for the speed of light in atomic units, $c = 137.035999 a_0 \cdot {t_A}^{-1}$; this means that velocities can simply be set as a proportion of c, automatically giving the correct value for velocity in atomic units. (Note that direct specification of velocity is not necessarily physical, the energy possessed by an α particle is dependent on its radioactive source, hence too its velocity.)
- The range of velocities you use should generally not exceed $0.1 \times c$, velocities which are much lower than $0.04 \times c$ or so *will* work, but the results would likely be uninteresting. Setting a velocity of zero may be a way to consider to test aspects of the code. Velocities which are too high become increasingly nonphysical due to the lack of treatment of relativistic effects and decreasing accuracy of the assumptions used to set the time step.

11.5 Assignment Summary

The following list of steps are required for completion of the assignment, take careful note of what needs to be accomplished:

1. Create suitable `.f90` files for source code which you will write, the main program as well as any additional files you may wish to use for `MODULE`s/`TYPE`s/`FUNCTION`s/`SUBROUTINE`s should you want to. All of the source files you create and use must be submitted if you use more than one.
2. You should define suitable variables and layout within the program and any external content (the `MODULE`s, etc).

3. Two properties must be read in when the program runs, either from an input file or through interactive user input: the x axis offset of your α particle starting location, and its velocity. All other parameters may be fixed within the code, refer to section 11.4 for values you may need. All should be in atomic units.

4. The algorithm described in Section 11.3.2 should be implemented. All values used should remain in atomic units. Refer to section 11.4 for important values in both SI and atomic units to determine the relevant conversions.

5. Your program should iterate through time steps until the location of the α particle is determined to be further from the central gold nucleus than $r_{\text{lim}} = 1.1 \cdot |\mathbf{r}_0|$, ie 1.1 times the distance away that the α particle starts at.
Note: While your program is still being tested, it is advised to have an additional condition for exit limiting the total number of steps performed to $\approx 10,000$, to prevent excessive run times and generation of excessive output if something is wrong. This may be left in the program even after testing, if you would like.

6. While iterating, your program should output to a file the x and y coordinates of the α particle for each step calculated. The name of this file is your choice.

7. When the iterations are complete, you should ensure that your program closes all open files and exits successfully.

8. You can plot your simulated particle trajectory using the script and process described in Section 11.3.4.

9. You should produce plots to hand in with your work for the following set conditions (in all cases the initial y distance, y_0, and the time step, Δt, are as defined in Section 11.4). **Important:** All velocities are relative to the speed of light, c, you will either need a variable containing the value of the speed of light in atomic units or to convert these proportions of c into atomic units yourself. Distances are given in atomic units directly:

   - Velocity: $0.07 \times c$; x displacement: $0.002\ a_0$.
   - Velocity: $0.07 \times c$; x displacement: $0.001\ a_0$.
   - Velocity: $0.07 \times c$; x displacement: $0.0001\ a_0$.
   - Velocity: $0.1 \times c$; x displacement: $0.0001\ a_0$.

10. Write answers for the following questions:

    (a) Think about the conditions which your simulation would *not* adequately model; write *one* set of conditions which would cause a *computational* problem, and *one* which would *not* cause a computational problem, but *would* cause the simulation to be incorrect with regards to the *physics*.

    (b) Consider the limitations of your simulation. Describe *two* ways in which it could be expanded to include more useful results/more physics or similar.

    (c) If you wanted to have the simulation conduct an entire series of tests for a large number of different x displacements, how would you go about doing this? **Note:** You do *not* have to actually implement this in your simulation.

    (d) Can you think of any way to test the validity of your simulation? Write a brief (a few sentences) description of *one* method by which you think you could, at least partly, show whether your simulation was physically accurate or not.

Requirements for submission are shown on the next page.

11.5.1 Submission

The deadline for submission is **Friday the 11th of April**. Ensure that your name is on *all* pages submitted and that your work is stapled or otherwise bound together.

You must submit the following for the purposes of this assignment:

- Printouts of *All* source code files written for the assignment.
- Printouts of the plots produced for the four specified sets of test conditions.
- Your answers to the questions (these may be typed up or hand written).

12 Week Twelve - Class Test

The sessions of Week Twelve are for continued work on the Week Eleven Material, at the end of which you will be required to hand in *all* of the Week Eleven assignment (Friday).

By this point you will have reached the end of the course, and should have gained a reasonable standard of programming skill. On Tuesday of this week is the end of term class test. This counts for a reasonable portion of your marks for this module; as such it is worth putting in some time preparing for.

The end-of-term class test takes the same format as the mid-term class test; covering material throughout the entire term (i.e. weeks 13-18 in addition to weeks 19-24).

* * *

Closing Notes

Hopefully you will have found this course useful, if perhaps not actually fun; but with luck the content covered should prepare you for any programming based task, module or job that you undertake in the future.

Should you want to continue learning this topic beyond this course, the best recourse is to simply continue programming. Choose projects to undertake to complete simple or even complex tasks through the use of programming. As you do this, you will automatically encounter difficulties in certain areas. Overcoming these difficulties is almost always possible, and attempting to do so continues to provide more knowledge and familiarity with programming.

Additionally, for further reading, make sure to have a look at the references section at the end of this document (Section 17), in particular those books mentioned in the section for the first week which are available from the Templeman Library (Section 1.3). The Internet, while not necessarily a source of information possessing a great deal of consistancy; is an incredibly valuable tool when used correctly. An online search for a short phrase describing your problem will likely provide dozens or even hundreds of pages of information. When using the Internet for this purpose, generally try checking a few references before attempting to implement a solution based on what you find. If multiple pages appear to describe the same idea independantly, there's a fair chance it will work. If only a single page describes it, it may still work, but there may be a better or more accepted alternative.

Good Luck One Final Time!

13 Unix Based Systems

The advantage of Unix based systems like Linux for learning programming is that they operate an environment specifically designed to permit writing, compiling and running a wide variety of programming languages. They also give sufficient control over the computer itself to better make use of programming knowledge to complete useful tasks.

The main system that you will be expected to use for this course is **Linux**. There are several varieties of Linux (called '**distributions**'), which differ in their exact workings, but all follow the same basic operating structure. As such, the information provided here should be valid for most, if not all, distributions. Probably the main distributions of Linux are: '*Ubuntu*', '*Debian*', '*Fedora*', '*SUSE*' and '*Solaris*'. For those who would like to experiment with running a Linux system yourself, arguably the best distribution to act as an introduction and provide the easiest, most user friendly, setup, is Ubuntu. The specifics of setting up a Linux system for yourself will not be covered here, but there is a substantial amount of information on how to do so online. For the main site of the Ubuntu OS, see http://www.ubuntu.com/[13].

Linux heavily makes use of a 'terminal' or 'shell', a command line interface to the computer. From a terminal you can run programs; view directories; make, move, copy and delete files; open files; as well as performing programming like sequences of commands to accomplish a string of tasks in one go. Here we will go through the basic set of commands necessary to navigate around the terminal environment and perform key tasks.

13.1 Navigation and Directories

When you open a terminal, you will typically be met with a text cursor following some form of prompt. For the purposes of this document, anything intended to be typed into the terminal will be represented with the following symbol as the prompt:

```
:>
```

Into this prompt you type commands to perform actions. A terminal window always has an associated directory which the terminal is effectively sitting in. Directories are essentially the same as 'folders' on Windows systems. The terminal is not stuck in this directory though, it can move through any directories which your user account has access to.

13.1.1 The 'ls' Command (and 'man' Command)

Generally when you first open a terminal, you will be in the 'home' directory of your user. Lets look in the current directory, and list the files and sub-directories present. This is done using the command '`ls`' as follows:

```
:> ls
```

This will list on screen all files and sub directories in your current or 'working' directory. Depending on system/terminal settings, these may be colour coded depending on what type of file/directory they are.

Many commands accept further text after the main command. Any other words, letters or numbers following the command are referred to as 'arguments'. In the case of the `ls` command, the default argument which can be given to it is a 'filter' to limit the files or directories which will be listed. This function tends to be used along with the asterisk symbol '`*`', which is a 'wildcard character'. What this means is best demonstrated by an example:

```
:> ls *.txt
```

This command, with the argument `*.txt` tells the terminal to list files which begin with anything (represented by the wildcard character), but which end with the extension '`.txt`'. When used for this purpose, there are no limits to where wildcards and actual letters/numbers/symbols are used. So, for example:

```
:> ls *ten*
```

Will list any file or directory which can have anything at the beginning and anything at the end, but must have the word 'ten' somewhere in their filename. The following is also valid:

```
:> ls *once*ten*
```

This will list any files which have the words 'once' and 'ten' somewhere in their name, in that order. Files with the following names would all be listed by this command:

```
r5once7ten.txt
450onceten3
53once435ten39
onceten
```

Note how the wildcard character can also be empty, with nothing between the 'once' and the 'ten' in the example '`450onceten3`'.

Many Linux commands like `ls` have special types of arguments called 'options'. These options are specified by following the command with a space, then a hyphen and a letter. Which letters represent which options depends on the command. For example `ls` has the option '`-l`'. This option means '*l*ong output' and, when run with `ls` as follows:

```
:> ls -l
```

Will produce a more detailed list of the files and sub-directories, containing information like file sizes and permissions (the information shown may look confusing, more of it will be described in a moment). To find out what options a command

has, most commands have a 'manual' page, a short piece of documentation explaining how the command works, to access the manual page for `ls`, you would type:

```
:> man ls
```

Where '`man`' is the command which requests the manual page for the chosen command (frequently abbreviated to 'man page'). Any command name can be put after the `man` command in order to see its man page. If the command doesn't have a man page, it will state that there is no manual entry for that command. Man pages can be navigated with the up and down arrow keys to scroll through the text, and exited with the 'q' key.

Returning to the `ls -l` command, much of the information displayed will not be needed for the purposes of this course, and is all explained in the man page for the command. Probably the most important column of information though, is the size of the files, which is the fifth column along, just before the date and time (which is the date and time that the file was last altered). These file sizes are in bytes, which is not easy to work in when you are used to file sizes varying between kilobytes and megabytes (KB and MB). However, there is another option for `ls` which makes this easier to read.

This option is '`-h`', standing for '*h*uman readable'. This option can be appended following the original `-l` option as follows:

```
:> ls -l -h
```

This should now display the file sizes with a 'K' or 'M' suffix, indicating whether the value is in kilobytes or megabytes (or even 'G', for gigabytes). Single letter options can also be strung together as a single joined up option, such as:

```
:> ls -lh
```

Which is quicker to write. These are a couple of simple examples of options, there are many different variations for the various different commands. More complex options can expect their own arguments too, in addition to any arguments being given to the main command. In a number of commands, there are some options which are written as whole words, and preceded by two hyphens rather than one as with single letter options.

13.1.2 The 'cd' command

The next important command to look at is the 'change directory' command, '`cd`'. This command requires one argument, telling it which directory you want to move to. If this command is run without an argument, some implementations will cause it to jump back to the home directory for you, from whichever directory you were in. The correct use of this command would be something like the following:

```
:> cd subdirectory
```

Which would move you from the current directory, to a subdirectory in the current directory, in this example called '`subdirectory`'. You can also string together directory locations using the forward slash symbol, '/'. So, if your home directory had a subdirectory called '`subdirectory`', and that had a subdirectory of its own called '`subsubdirectory`'. Then, while in your home directory, you could type:

```
:> cd subdirectory/subsubdirectory
```

and this would take you straight through `subdirectory` and into `subsubdirectory`.

To go backwards, to a higher directory than you are currently in, say you are in `subsubdirectory` and want to go back up to `subdirectory`, you can use a special name consisting of two full-stops '`..`', as here:

```
:> cd ..
```

Which could take you from `subsubdirectory` back up to `subdirectory`. These can also be strung together with forward slashes. So, say you are in `subsubdirectory` and want to go all the way back up to the home directory, you could type:

```
:> cd ../..
```

Which would accomplish that task. Note that the directory names '`subdirectory`' and '`subsubdirectory`' have just been chosen for this example, and actual directory names can consist of any standard characters. However, it is considered good practice to give directories names which are not very long, contain no spaces, and preferably lower case.

Another special name in directory structures is a single full stop '`.`', which represents your current directory. It is of limited use with the `cd` command (as you would have no need to change to the directory you are already in) but is useful in other contexts. Many people prefer to refer to all directories being described relative to the current directory explicitly by using the full stop, such as:

```
:> cd ./subdirectory
```

This is because there are other ways of beginning a directory name that do not necessarily start from your current directory. Starting with tilde-slash: `~/` refers to a directory relative to your home directory; and starting with just a slash: `/` refers relative to the 'root' directory, the highest up directory on the file system.

13.1.3 The 'mkdir' command

So far we have looked at how to look at the contents of a directory and move between directories. However, how do you create them in the first place? This is done with a simple command called '`mkdir`' (for '*m*a*k*e *dir*ectory'). This command expects one argument, which should be the name you would like to give your new directory. So, for example, if you are in your home directory and type:

```
:> mkdir myfiles
```
A new directory will be created within home, called '`myfiles`'.

Note: Do not attempt to create directories (or, indeed, files) with spaces in the name. It is technically possible, but simply writing the name with spaces after the `mkdir` command, it will interpret the separate words as separate arguments, *not* creating a single directory with a name with spaces in. In general, within the file/directory structures, use of spaces should be avoided as a rule.

13.2 Basic Interaction with Files

13.2.1 The 'mv', 'cp' and 'rm' commands

Now we can create and move around directory structures, as well as view their contents, we can look as interacting with files. The basic commands for files are those to move, copy and delete them. The first we will look at is to move files, this uses the command '`mv`'. `mv` expects *two* arguments, the first of these is the name of the file you want to move, and the second is the location you want to move it to (and optionally, a new name for the file). The place (+ optional new name) you want to move the file to is called the 'target'. In order to avoid confusion, whenever moving a file into a directory, without changing the file's name, the name of the target directory should be ended with a forward slash. An example:
```
:> mv myfile.txt myfiles/
```
This would be the command to move a file called '`myfile.txt`' from the current directory into a subdirectory called '`myfiles`'. However, you could also choose to do the following:
```
:> mv myfile.txt myfiles/newname.txt
```
This would move the file from the current directory, into the sub directory `myfiles` and also change your file's name to '`newname.txt`'. By always using a forward slash at the end of a directory name if you are only moving a file there, it prevents confusion regarding whether you want to move a file into a directory, or change the name of the file to the same as the directory name!

The copy command '`cp`' works the same way as the `mv` command, however rather than moving the file, it leaves the original copy of the file in place. Note that, using `cp`, you can still change the name of the copied file in the same way as with the `mv` command, but the original file's name will remain unchanged.

For both of these commands, you can also fetch files from another directory or subdirectory, and bring them (either moving or copying) to the directory you are currently in. For example:
```
:> mv myfiles/myfile.txt ./
```
Will bring the `myfile.txt` file out of the directory `myfiles` and up to your current directory, represented by the full stop. The use of the full stop as a special directory character was described in Section 13.1.2.

The final command in this set, and the one with which you must be most careful, is the delete or '*rem*ove' command. The command for remove is '`rm`', any arguments other than options preceded by '-' or '–' it interprets as file or directory names which you would like to delete. It is run by doing the following:
```
:> rm myfile
```
Where, in this case, it will attempt to delete the file called '`myfile`' if it exists. By default, `rm` will only work on files, and not directories. To delete directories, you need to add the option '`-r`' representing '*r*ecursive'. **HOWEVER**, be **extremely** careful when doing this, running `rm -r` on a directory will delete that directory as well as any and all files and sub directories within it.

Another option which you may find useful for the `rm` command is the '`-i`' option. This option represents '*i*nteractive' mode. When you run `rm` with `-i` it asks you to confirm before deleting the file(s) selected. This option does not 'toggle' however, running `rm -i` once does not mean it will always ask you to confirm deleting a file in the future. However, getting into the habit of *always* writing `rm -i` may help, though many may prefer otherwise.

All of these commands have arguments which accept wildcard characters. For `mv` and `cp`, the first argument (the file to be moved/copied) may have wildcards in, for example:
```
:> mv *.txt myfiles/
```
Will move any file ending in '`.txt`' from your current directory, into the `myfiles` directory.

The remove command can accept wildcards in any of its filename arguments. However, again, be **very** careful with this as it becomes quite easy to delete files unintentionally. The following example would delete *all* files ending in `.a` and `.b` from your current directory:
```
:> rm *.a *.b
```
Typing something such as:
```
:> rm *
```
Will attempt to delete *every* file in that directory and is to be used with **extreme caution**.

13.2.2 The 'less' Command

Any plain text file, such as a regular `.txt` file or code, can be handily previewed by using a command called '`less`'. `less` expects a single argument, which is the file you would like to preview in the terminal. For example:

```
:> less myfile.txt
```

This would show the contents of `myfile.txt` in the terminal window. It can be scrolled through much like a man page, using the up and down keys, and quit from by using the 'q' key. This is useful if you would like to just quickly see a small part of a file without wanting to change anything, rather than open it with a proper editor.

13.2.3 The 'head' and 'tail' Commands

These commands are somewhat similar to the `less` command. Each takes a small portion of the specified file (the top few lines when using `head` and the final few lines when using `tail`) and prints them into the terminal window. Unlike `less` it is just these few lines and no more, and they remain in the terminal window for the next command prompt, permitting you to see their contents whilst you are typing your next commands. This can be useful for various reasons, for example, you may have a file you've written with instruction of how to call/run a certain command or program, which you would like to see while you type that command in.

13.3 Editing Text/Code Files

There is no single method for how to edit text files in Linux. There are a wide variety of different programs which will let you open plain text files such as code in order to edit or write them. We will cover a few of the popular ones here, but it is by no means an exhaustive list. Most people find an editor which they work well with through some degree of experimentation and exploration of the options. Some of these editors have many, many more features and functions than can be explained here, so it is recommended to research them in other places if it is of interest to you. For this purpose there are also links provided to useful online resources for several of the editors. Additionally, all of the editors described here should have `man` pages (explained in section 13.1.1), containing explanations of their various features.

13.3.1 'gedit'

'gedit' shortened from 'gnome edit' is a simple, graphical text editor designed primarily for the 'Gnome' window manager/desktop environment. It is recommended for this course because of the combination of similarity to common Windows-style editing (`ctrl+c`/`ctrl+v` for copy/paste, `ctrl+s` for save, etc.; and the same dropdown-menu formatting), and integration with programming languages provided by the context highlighting system and parenthesis matching. 'Context highlighting' highlights key coding words, comments and similar, in different colours to more easily follow a piece of code, much as highlighting is used in the code example segments in this document. 'Parenthesis Matching' highlights parenthesis (bracket) pairs, this permits you to see which opening/closing bracket is paired with which closing/opening bracket. This is extremely useful when writing complex expressions which many nested bracketed expressions, a frequent cause of errors is down to incorrectly laid out brackets.

13.3.2 'vi' or 'vim'

'vi' and 'vim' are part of one family of text editors. 'vi' came first, named with respect to the fact that it was a '*vi*sual' text editor. 'vim' stands for '*vi* i*m*proved'.
Some useful links:
http://www.vim.org/[14]

13.3.3 'emacs'

'emacs' is another editor, making very heavy use of powerful keyboard shortcut commands to do everything from finishing standard forms of code, to compiling the program from within the editor. The graphical version of emacs available in the desktop environment has menus to perform most operations. Particularly useful features which are recommended include selecting, under the 'options' menu, 'Syntax Highlighting' and 'Paren Match Highlighting'.
Some useful links:
http://www.gnu.org/software/emacs/[15]
http://www.emacswiki.org/[16]

13.3.4 ‘nano’ or ‘pico’

‘nano’ and ‘pico’, so called for being minimal editors, with very few features, are two editors which operate within the terminal window, rather than opening a new window for them to run in. ‘nano’ is the version available on PCSPS07.
Some useful links:
http://www.nano-editor.org/[17]

14 Good Programming Practice

14.1 Comments

Commenting is the practice of include short descriptions of what the code is doing. Comments themselves are a special case within files of code. In F90, they can be started with an exclamation mark character. When the program is compiled, the exclamation mark tells the compiler to ignore any text after it and not interpret it as code. This way notes and descriptions can be left in the code file without confusing the compiler when you try and run it. The inclusion of comments in code is vital if the code is the be read and understood by someone else, or even just yourself at a later date. The key aspect of commenting is that it is *not* simply stating in words what the next line of code does or similar. The use of comments is to describe the *purpose* of a section of code. For example:

```
PROGRAM loopcount
  IMPLICIT NONE
  !makes integers 'i' and 'total'
  INTEGER :: i, total
  !sets total to zero
  total = 0
  !Do loop from 1 to 10 in steps of 1
  DO i=1,10,1
    !adds i to total
    total = total + i
  !ends the do loop
  END DO
  !writes the total
  WRITE(*,*) total
END PROGRAM loopcount
```

Does not help the reader in understanding what the code is actually for. Stating that '`WRITE(*,*) total`' '`!writes the total`' does not provide any useful information. These comments do not tell the reader any more than the line of code contains already. A better example of commenting would be the following:

```
PROGRAM loopcount
  !************************
  !code written by A. Nonymous, last edited 24/03/10
  !program to sequentially add up the numbers from 1 to 10
  !************************

  !Set up variable, 'i' will be the variable for the do loop, and 'total' the running sum or
     counter
  INTEGER :: i, total

  total = 0 !initialises the running sum

  !Loop steps through 1 to 10, adding them to the sum, then outputs the total afterwards
  DO i=1,10,1
    total = total + i
  END DO
  WRITE(*,*) total

END PROGRAM loopcount
```

In this case, the comments explain what the parts are *for*, as opposed to simply what they *do*. Also note the lines at the beginning. First the author of the code and date when the code was last edited is stated, and then some text describing the purpose of the code as a whole. It is generally considered standard practice to include such a description at the top of your own code files. The commented lines of asterisks before and after this introduction are simply to help visually split up the code and are just a stylistic choice, they don't perform a special purpose or function. The filename alone is unlikely to convey what the code is for. '`program1.f90`' does not tell you what it does, hence a description is useful for letting people understand your code without working it out line by line, relying on an explanation from you, or for helping *you* to remember what it does, if you return to it at a later date and can't recall!

Note: For the purpose of programs written as part of any assignments for this course, the header comments at the top of the file should include: Your name, the date it was last edited, the module code, the assignment number, and a description of what the code does.

Note that you can place comments on a line of their own, or at the end of a line of actual code. Which you do is a combination of what circumstance the comment is for, and personal preference. The code above shows examples of both.

14.2 Variable Naming

While in older codes, the limit to line length meant that variable names tended to be kept as short as possible. With modern code, however, this is not an important concern, and it is far more beneficial to use longer, and more descriptive variable names. While variables to be iterated over in loops are conventionally single letters (commonly '`i`', '`j`' and '`k`'), unless there is a good reason for other variables to be only a single letter, the use of only one character as a variable name should be avoided.

Variable names like '`A`', '`B`' or '`C`' may be used in examples, where their purpose is generic. When you are writing code for something specific, the variable names should describe what they actually represent. The following example is for solving the 'suvat' equation $s = ut + \frac{1}{2}at^2$, first asking for the values for initial velocity, time and acceleration; and then printing to screen the distance travelled. The following code is valid, but difficult to follow:

```
PROGRAM suvat
  REAL :: A, B, C, D

  WRITE(*,*) 'Please enter initial velocity, time and acceleration separated by spaces:'
  READ(*,*) A, B, C

  D = A * B + 0.5 * C * B * B

  WRITE(*,*) 'Distance travelled is : ', D
END PROGRAM suvat
```

Much better would be to write it as follows (suitable commenting has also been added):

```
PROGRAM suvat
  !code by R. Andom, last edited 24/03/10
  !program to do suvat calculation for distance

  REAL :: vel, time, acc, distance

  !perform user input
  WRITE(*,*) 'Please enter initial velocity, time and acceleration separated by spaces:'
  READ(*,*) vel, time, acc

  !calculate s = ut + 1/2 at^2 then output result
  distance = vel * time + 0.5 * acc * time * time

  WRITE(*,*) 'Distance travelled is : ', distance
END PROGRAM suvat
```

In this case, it is unambiguous what each of the variables is supposed to represent. It is now clear which variables are which parameters of the equation. This way, if they are used many times in a long code, it is easier to keep track of what means what, without constantly referring back. In this case '`vel`' and '`acc`' were used, rather than longer names like '`velocity`' and '`acceleration`'. For the sake of saving time writing and preventing lines being far longer than necessary, abbreviations are still used. However, they should still be as descriptive of their purpose as possible.

14.3 Indentation

The practice of indenting sections of code with either several spaces or a 'tab' is another important concept when it comes to making the code easy to read and follow. Whenever a new 'level' of code is entered, the contents of a `DO` loop, or an `IF` statement, the code should be indented. This makes it clear which loops or statements certain sections of code belong to. The following is an example of how not to organise code:

```
PROGRAM indentation
IMPLICIT NONE
INTEGER :: count, i, j
count = 0
DO i = 1, 10, 1
DO j = 1, 100, 1
IF (j == (10*i)) THEN
count = count + 1
END IF
END DO
IF (count == 2) THEN
WRITE(*,*) 'Two numbers found'
END IF
```

```
END DO
END PROGRAM indentation
```

It is very difficult to follow which bits of code are parts of which statements or loops. Such a piece of code should be organised as follows (here, the comments are just labels to refer to in the subsequent description):

```
PROGRAM indentation
  IMPLICIT NONE
  INTEGER :: count, i, j
  count = 0
  DO i = 1, 10, 1 !do loop A
    DO j = 1, 100, 1 !do loop B
      IF (j == (10*i)) THEN !if A
        count = count + 1
      END IF
    END DO
    IF (count == 2) THEN !if B
      WRITE(*,*) 'Two numbers found'
    END
  END DO
END PROGRAM indentation
```

This makes it clear that `DO` loop 'A' is in the main code; that `DO` loop 'B' and `IF` statement 'B' are both part of `DO` loop A; and that `IF` statement 'A' is part of `DO` loop 'B'.

15 Error Messages and Likely Solutions

The error messages given to you by the computer when compiling or running any of your programs are vital in helping to determine what the bug in your code is, and what needs to be fixed to make it work. This section of the documentation gives an overview of a few of the more common error messages, and what they are trying to tell you. The general form of the error messages, what each part of any given message contains, is described in Section 1.6.2, this section deals with specific messages.

The following errors in this section were obtained using the gfortran compiler, gcc version 4.4.1 20090725. Not all compilers are the same and will not necessarily provide the same information, or in the same manner, as the errors listed here. However, it is gfortran which you are expected to use within this course, so the following examples should hopefully provide a good overview of the general type of errors which you may come across during the course.

15.1 Variables

15.1.1 Symbol has no Implicit Type

This is an error of the form displayed below, stating that a particular named variable, found at a particular described location, has no implicit type. It means that either through a typo, or omission, the variable in question has not been declared at the beginning of your code. See Section 1.7.1 for the introductory material on declaring variables.

Possible Solution: If the intended variable has been declared, the one highlighted by the error may simply be a typo, check the spelling of the variables in the variable defining section of your code. If it is not a typo, you may have forgotten to initially declare the variable you are attempting to use at the beginning of your code.

```
test.f90:5.2:

 A = 1
  1
Error: Symbol 'a' at (1) has no IMPLICIT type
```

15.1.2 Can't Convert 'Variable Type One' to 'Variable Type Two'

This example refers to something like the case below, explaining that an argument of the first type ('CHARACTER', in this case) cannot be converted to the second type ('INTEGER', in this case). This will come about if you have declared a variable as one type (in this example the variable A was previously declared as an INTEGER) and tried to set it as another type (in this case we have tried to set A to be the letter 'a').

Possible Solution: Determine what variable type your variable is, depending on context, either change the type of the variable itself, or change the value being assigned to it at the point highlighted by the error.

```
test.f90:7.5:

 A = 'a'
     1
Error: Can't convert CHARACTER(1) to INTEGER(4) at (1)
```

Note: This issue is not always flagged as an error, depending on what the two types are. As explained in Section 1.7.1, if you try to set an INTEGER variable as a REAL value, it will automatically round the REAL value into INTEGER form, *regardless of whether that was your intention.* To prevent the possibility of undetected problems with your codes, make sure to check that the variables you are assigning values to are the correct types.

Note 2: Also of interest is the number in brackets after each variable type, ie the 'code1' in 'CHARACTER(1)'. This refers to how many bytes long the variable type is; in this case a single byte CHARACTER, and a four byte normal length INTEGER. It is worth being aware that this will not refer to synonyms for different length versions of the same fundamental variable type. So for example a 'REAL' variable will be described as 'REAL(4)'; and a 'DOUBLE PRECISION' variable will simply be described as 'REAL(8)', not 'DOUBLE PRECISION(8)' or similar.

15.1.3 Argument of 'Function' must be 'Variable Type'

This example is a case such as the example below, describing that a particular input for a function must be of a certain type. In this instance, the code attempts to pass the CHARACTER variable 'a' to the mathematical square root function, 'SQRT'. The function SQRT expects REAL or COMPLEX arguments only.

Possible Solution: Check only to pass a variable of a valid type to the function.

```
test.f90:7.10:

 A = SQRT('a')
          1
Error: 'x' argument of 'sqrt' intrinsic at (1) must be REAL or COMPLEX
```

Note: This error is similar to the previous one, 15.1.2, in that it relates to a variable with the wrong type being used as the argument for some operation. As such, the first note from that error also applies; that it will not flag incorrect types between `REAL` and `DOUBLE` types as errors, it will simply round them automatically.

15.1.4 Symbol Already Has Basic Type

The example below is the result of having the lines '`INTEGER :: A`' followed by '`DOUBLE PRECISION :: A`'. If a variable name has already been used for a different variable type, or is reserved for some reason, this is the resulting error. In other words the same name has been given to two effectively different variables, and one must be changed.

Possible Solution: Change the name of one or other of the variables which conflict.

```
test.f90:7.22:

 DOUBLE PRECISION :: A
                     1
Error: Symbol 'a' at (1) already has basic type of INTEGER
```

15.2 Functions

15.2.1 Function has no Implicit Type

The following example consists of a situation where a piece of code purporting to be a function does not appear to exist as a proper type. This may be a valid function which is not recognised due to a library issue; or a valid function name has been misspelt, and the compiler thinks it is supposed to be a different function, one which it does not understand.

Possible Solution: Check whether it is a typo, check that any requisite libraries have been included.

```
test.f90:7.5:

 a = SQRTT(a)
     1
Error: Function 'sqrtt' at (1) has no IMPLICIT type
```

15.2.2 Missing Actual Argument in Call to 'Function'

This example is where the compiler has found a call to a function, and either has none of the required arguments for the function, or too few. In this case, the function `SQRT` requires a number to take the square root of, but has not been provided with one.

Possible Solution: Check the required arguments for a function.

```
test.f90:7.5:

 a = SQRT()
     1
Error: Missing actual argument 'x' in call to 'sqrt' at (1)
```

15.3 General Structure

15.3.1 Expected Label 'label' for END PROGRAM statement

This error consists of an indicated location at an '`END PROGRAM`' statement, where the program name label is not the one expected based on the label used for the initial '`PROGRAM`' statement at the start of the code. The second part of this error, the '`Unexpected end of file in 'test.f90'`' relates to the fact that, due to the lack of a valid `END PROGRAM` statement, the compiler believes there is yet to be one, continues reading, and reaches the end of the file without one being found.

Possible Solution: There is a typo or other mistake in either the name chosen at the beginning for the `PROGRAM` label, or in the label for the closing statement of the '`END PROGRAM`' statement.

```
test.f90:9.15:

END PROGRAM tes
               1
Error: Expected label 'test' for END PROGRAM statement at (1)
Error: Unexpected end of file in 'test.f90'
```

15.3.2 Unexpected End Of File

This issue is the same as the latter portion of the previous example error, 15.3.1. The compiler has reached the end of the file without receiving the specifically expected statements to one or more sections of the code. For example, if you have left out the `END PROGRAM` statement, or if you leave out both the `PROGRAM` *and* `END PROGRAM` statements; to compile code, it expects at least one program to exist (or equivalent, such as in the case of modules). There is no indication of where the problem has occurred, because the compiler cannot know where you intended to put statements.

Possible Solution: This error covers quite a lot of possible problems. Basically, some standard statement, expected to be there, is not. As such, checking all of the locations you would have intended to have these manner of statements should indicate which may be missing.

```
Error: Unexpected end of file in 'test.f90'
```

15.3.3 Unclassifiable Statement

This error describes a situation where a piece of code which is of the format of a statement is not recognised. In this context, 'statement' is meant in the way that '`IMPLICIT NONE`' is a statement, as opposed to an assignment, such as giving values to variables. In the example below, the random text '`sdgfasd`' has accidentally been put on an empty line.

Possible Solution: Examine the line indicated by the error. It could be accidentally placed text, or a typo of a real command.

```
test.f90:5.1:

 sdgfasd
 1
Error: Unclassifiable statement at (1)
```

15.4 Run-Time Errors

Run-time errors are those which are returned when a problem occurs while the program itself is running, rather than a problem found in the code when the compiler is run. These tend to be less descriptive, and more difficult to determine the cause. Some compilers permit programs to provide more information at run-time than others. For instance, when an error occurs while running the code for a particular reason on a particular line, you may find compilers which cause the program to state both the type of problem and the line in the code which it is thought to originate. Some will only cause the program to describe the general type of problem, but will not know where it is, and some will not be able to provide any information at all.

This is because, while a compiler is (by its very nature) looking at each line in turn, once the program is compiled as binary, these lines are no longer really there, it is all a sequential series of specific computational instructions. A compiler has to put extra information into the program for it to be able to realise what code line certain tasks must correspond to. Adding all of this extra information is not always desirable, it can cause the size of the compiled program to be larger, or even cause it to run slower. It is partly for this reason that so many errors in commercial programs end up being reported to the screen as potentially indecipherable series of numbers, rather than specific information about the source code.

15.4.1 Bad 'Variable Type' for Item in List Input

In this case the example states that the variable type being passed as the first item in the list passed to the input is not valid. This program example was attempting to use the `READ` command to obtain an `INTEGER` value from the user at run time, instead, the letter '`a`' was given. It is possible to tell it was a command line input, as it lists the unit and file as '`5`' and '`stdin`' respectively, the unit and name of the command line input. Were it being read from a file, the unit number

and file name would be different. This error is similar to the compiler error 15.1.2, where a variable assignment does not have the correct type.

Possible Solutions: Ensure that the variable you are attempting to read into is the correct type for your data, or that it is the correct data being entered.

```
a
At line 7 of file test.f90 (unit = 5, file = 'stdin')
Fortran runtime error: Bad integer for item 1 in list input
```

16 Fortran Command Overview

16.1 Variable Types

`INTEGER`	A variable of type `INTEGER` is a 'whole' number, usually 4 bytes long, permitting values between -2147483648 and +2147483647.
`REAL`	A variable of type `REAL` is a 'floating point' or 'decimal' number taking the form $x \times 10^y$, usually 4 bytes long, permitting a precision of about 7 digits, and minimum and maximum exponents of -126 and +127
`DOUBLE PRECISION`	A variable of type `DOUBLE PRECISION` is a 'floating point' or 'decimal' number taking the form $x \times 10^y$, usually 8 bytes long, permitting a precision of about 16 digits, and minimum and maximum exponents of -1023 and +1024
`COMPLEX`	A variable of type `COMPLEX` is essentially a pair of two `REAL` type variables, one representing the real component of the complex number, and the other representing the imaginary component of the complex number.
`LOGICAL`	A `LOGICAL` variable is a boolean value, with a value of either `.TRUE.` or `.FALSE.`.
`CHARACTER`	A variable of type `CHARACTER` is a 'letter' variable. It can have a value equivalent to any of the 'ascii' characters.
'String'	A string is not a variable type itself, but rather a combination of characters. These are expressed as `CHARACTER*<n>`, where `<n>` is an integer value representing how many characters long the 'string' of characters is. This is covered in more detail in the section of the course on strings.

16.2 Operators

16.2.1 Mathematical

`*`	Multiply
`+`	Add
`-`	Subtract
`/`	Divide
`**`	'To the power of', ie `a**b` is equivalent to a^b

16.2.2 Relational

These are the operators which effectively 'check' for a condition. Their use would be something of the form: `C = (A .GT. B)`; the logical variable `C` would then either be '`.TRUE.`' if `A` is greater than `B`, or '`.FALSE.`' if `A` is not greater than `B`. Note that the 'equals' here is for this context *only*

`>`	`.GT.`	Greater than
`>=`	`.GE.`	Greater than or Equal to
`==`	`.EQ.`	Equal to
`/=`	`.NE.`	Not Equal to
`<`	`.LT.`	Less than
`<=`	`.LE.`	Less than or Equal to

Note: See Section 3.1.1 for more information on using relational statements specifically, and the entirety of Section 3 for conditional statements and their use in code.

16.2.3 Strings

`//`	Concatenate. Links strings/characters together into a longer string. ie `'once '//'upon'` would give a single string `'once upon'`

16.2.4 Logical

These are the operations which are used to 'combine' other `.LOGICAL.` variables.

`.NOT.`	Is not
`.AND.`	And
`.OR.`	Or
`.EQV.`	Equivalent (effectively like 'equals')
`.NEQV.`	Not Equivalent

Note: See Section 3.1.2 for more information on using logical statements specifically, and the entirety of Section 3 for conditional statements and their use in code.

16.3 Intrinsic Functions

16.3.1 Variable Casting Functions

`INT(<x>)`	Returns a `INTEGER` value converted from `<x>`, where `<x>` was a `DOUBLE PRECISION`, `REAL` or `COMPLEX` (when the argument is `COMPLEX` the value returned is the integer rounded real part of the complex number)
`REAL(<x>)`	Returns a `REAL` value converted from `<x>`, where `<x>` was a `DOUBLE PRECISION`, `INTEGER` or `COMPLEX` (when the argument is `COMPLEX` the value returned is the real component of the complex number)
`DBLE(<x>)`	Returns a `DOUBLE PRECISION` value converted from `<x>`, where `<x>` was a `REAL`, `INTEGER` or `COMPLEX` (when the argument is `COMPLEX` the value returned is a double precision version of the real component of the complex number)
`CMPLX(<x>[,<y>])`	Returns a `COMPLEX` value converted from `<x>`, where `<x>` was a `REAL`, `INTEGER` or `DOUBLE PRECISION` representing the real component of the complex number. `<y>` is an optional second argument, of the same type as `<x>`, which represents the imaginary component of the complex number.

16.3.2 Maths Functions

`SQRT(<x>)`	Returns the square root of the variable `<x>`, ie. $\sqrt{x}$.
`ABS(<x>)`	Returns the absolute value of the variable `<x>`, ie. the modulus $\|x\|$.
`EXP(<x>)`	Returns the exponential of `<x>`, ie. e^x
`LOG(<x>)`	Returns the natural logarithm of `<x>`, ie. $\log_e(x)$
`LOG10(<y>)`	Returns the base 10 logarithm of `<y>`, ie. $\log_{10}(y)$
`SIN(<x>)`	Returns the sine of `<x>`, where `<x>` must be in radians, ie. $\sin(x)$
`COS(<x>)`	Returns the cosine of `<x>`, where `<x>` must be in radians, ie. $\cos(x)$
`TAN(<y>)`	Returns the tangent of `<y>`, where `<y>` must be in radians, ie. $\tan(y)$
`ASIN(<y>)`	Returns the inverse sine (or 'arc sine') of `<y>`, where `<y>` must be between -1 and 1, and the result is in radians, ie. $\sin^{-1}(y)$
`ACOS(<y>)`	Returns the inverse cosine (or 'arc cosine') of `<y>`, where `<y>` must be between -1 and 1, and the result is in radians, ie. $\cos^{-1}(y)$
`ATAN(<y>)`	Returns the inverse tangent (or 'arc tangent') of `<y>`, where `<y>` must be between -1 and 1, and the result is in radians, ie. $\tan^{-1}(y)$
`SINH(<y>)`	Returns the hyperbolic sine of `<y>`, where `<y>` must be in radians, ie. $\sinh(y)$
`COSH(<y>)`	Returns the hyperbolic cosine of `<y>`, where `<y>` must be in radians, ie. $\cosh(y)$
`TANH(<y>)`	Returns the hyperbolic tangent of `<y>`, where `<y>` must be in radians, ie. $\tanh(y)$

Figure 9: In all of these cases, `<x>` is any variable of types `REAL`, `DOUBLE PRECISION` or `COMPLEX`; `<y>` is a variable only of types `REAL` or `DOUBLE PRECISION`.

16.3.3 String Functions

TRIM(<string>)	Returns a version of the CHARACTER variable <string> with trailing spaces removed.
LEN(<string>)	Returns the length of the CHARACTER variable <string> as an INTEGER.
LEN_TRIM(<string>)	Performs LEN and TRIM simultaneously, returning the length of the TRIMmed variable <string> as an INTEGER.
ACHAR(<x>)	Returns a CHARACTER with the ASCII value given by an INTEGER x.
IACHAR(<c>)	Returns the ASCII value as an INTEGER for the CHARACTER given in <c>.
REPEAT(<string>, <x>)	Returns a CHARACTER string containing the contents of the CHARACTER variable <string> repeated INTEGER <x> times.

16.4 Control Structures

16.4.1 'DO' Loops

Syntax is:

```
DO <variable> = <starting value>, <ending value>, <increment value>
  <Instructions to perform on each loop>
END DO
```

Where <variable> is a variable of type INTEGER which will be iterated over (frequently this is i, j, k or similar). <starting_value> is the initial number, <ending_number> is the value at which the loop will stop and <increment_value> is the size of the step at which it will count. Note that in the section of code <Instructions to perform on each loop> you cannot set a new value of the chosen variable for <variable>, the value of <variable> must *only* be controlled by the loop itself. However, the value of <variable> *can* be used to set another variable. See the following example, where the value of <variable> is used as the argument to a WRITE statement.

Example :

```
DO i = 0, 10, 2
  WRITE(*,*) i
END DO
```

In this example, the loop goes through 6 steps, sequentially giving the variable i the values of 0,2,4,6, 8 and 10, with the action of printing that value of i on each loop.

16.4.2 'IF-ELSE IF-ELSE-END IF' Statements

Syntax is:

```
IF <Condition A> THEN
  <Code A>
ELSE IF <Condition B> THEN
  <Code B>
ELSE
  <Code C>
END IF
```

Where <Condition A> is a conditional statement, such as '(a .NE. b)'. If the conditional statement is True, then the code in the <Code A> code block will be performed. If <Condition A> is not True, it proceeds to check the conditional statement '<Condition B>'. As before, if this is True, the code in <Code A> will be performed. If not, it proceeds to the 'ELSE', statement and performs <Code C>. The contents of the ELSE statement, <Code C> is the code to be performed if none of the previous conditionals have been True.

It is possible to include as many ELSE IF conditions and code blocks as desired (Although normal coding practice dictates that this be limited, if possible, to prevent the code being confusing). When multiple ELSE IF conditions are include, an ELSE statement will still only run if none of the previous conditions has been True. While as many ELSE IF statements may be used, only one ELSE is permitted.

Example :

```
IF (a == b) THEN
  WRITE(*,*) 'a matches b'
ELSE IF (a == c) THEN
  WRITE(*,*) 'a matches c'
ELSE
  WRITE(*,*) 'no match found for a'
END IF
```

In this example, a variable 'a' is checked first against 'b', and then against 'c'. If a matches b it prints a message to screen saying so, it would then exit the displayed block of code. If it doesn't match, if checks c, again, printing a message if it does and then exiting this code block. If neither matches, it prints the message 'no match found for a' and then exits.

It is important to note that if a matches b *and* c, the code will **only** print the message 'a matches b'. This is because the ELSE IF condition is only checked if the original IF condition (or any other preceding ELSE IF condition) **has already failed**.

16.4.3 'Case' Statements

Syntax is:

```
SELECT CASE (<variable>)
  CASE(<variablevalue1>,<variablevalue2>)
    <Code A>
  CASE(<variablevalue3>:<variablevalue4>)
    <Code B>
  CASE DEFAULT
    <Code C>
END SELECT
```

Where '<variable>' is the name of the variable which will be checked to control the case statement. The code proceeds through the CASEs, and checks variable against the various variable values for that particular CASE.

In the first CASE here, variablevalue1 and variablevalue2 are the conditions, they must be the same type as <variable>, so if <variable> was an integer, they could be stated as '3' and '7', for example, or the names of two existing variables, so long as they were INTEGERs. Separation by a comma indicates that this CASE is True if variable matches any of the comma separated values. If this is the case, then the code contained in the '<Code A>' block will be executed, and the CASE statement exited.

In the second case, the colon separating variablevalue3 and variablevalue4 indicates a 'range', in this case, the statement will be True if <variable> is any value between variablevalue3 and variablevalue4. If this is the case, then the code contained in the '<Code B>' block will be executed, and the CASE statement exited.

The final CASE statement is a special one which much always come last (if it is included). It is the default case that will be run if none of the prior checks were True. If this happens, the final section of code, <Code C> will be executed. This is essentially equivalent to the 'ELSE', in IF statements.

Example :

```
testnumber = 8
SELECT CASE (number)
  CASE(0,testnumber)
    WRITE(*,*) 'Your number was zero or eight'
  CASE(1:7)
    WRITE(*,*) 'Your number was between zero and eight'
  CASE(9:)
    WRITE(*,*) 'Your number was above eight'
  CASE DEFAULT
    WRITE(*,*) 'Your number must have been negative'
END SELECT
```

In this example, an INTEGER variable 'number' has been defined or read in earlier in the code. We also define another INTEGER variable as 8. This first check run by the SELECT CASE command is whether the value of number is the same as (0) or testnumber, if it is, the subsequent WRITE statement is executed and the CASE statement exited. The next check is a range, looking for whether testnumber is between 1 and 7 (inclusive). Again, if this is True, as message is written.

The next CASE statement is more unusual, in this instance, the lack of a number *after* the colon implies that this case will be True if number is 9 or any value higher than 9, it implies that the range is open ended beyond 9.

Finally, if none of these checks have matched `number`, the `WRITE` statement following `CASE DEFAULT` is performed.

16.5 Conditional Statements

A conditional statement is a piece of code which acts as a 'check' for some condition, and gives back either 'True' or 'False' (effectively a `LOGICAL` variable).

Conditional statements can be single checks, or include multiple conditions combined with logical operators. Conditions can 'check for negatives', not just positives. For example, `IF (1 .NE. 0) THEN`
the conditional in the brackets is asking whether the value '1' is different to the value '0', this statement is True, as it is True that they do not equal one another, and hence the conditional with return True, and follow the `IF` statement.

Example :

```
LOGIC1 = ((a .NE. b) .AND. (c .EQ. d))

LOGIC2 = (((a .NE. b) .OR. (c .EQ. d)) .AND. (e .LE. f))
```

These examples show two different composite conditionals, each assigned to `LOGICAL` variables `LOGIC1` and `LOGIC2`. In the case of `LOGIC1`, there are two sub component checks. This means that effectively three checks will be performed in order to return an answer. Firstly, it checks whether `a` is different to `b`, then checks whether `c` is the same as `d`. The final check is from the `.AND.` statement. This asks whether **both** of the sub component checks are True. If only one sub check is True, or neither are True, the variable `LOGIC1` is set to `.FALSE.`. IF both checks are True it is set to `.TRUE.` (note the 'full stops' either side of `.TRUE.` and `.FALSE.`, these are how the True and False 'values' are referred to within F90).

In the case of `LOGIC2`, several conditionals are again performed, this time totalling 5 checks. The first two checks are the same as for `LOGIC1`, if `a` and `b` are different, and if `c` and `d` are the same. However, instead of checking whether both of these are true, using an `.AND.`, it asks whether either of them are, using '`OR`'. In this case if one, the other, or both are True, then the result of the `.OR.` check is True. After this check, another pair of checks are performed, first using the 'less than or equal to' check, `.LE.` (which can also be written as the less than symbol, '`<=`') to see whether `e` is less than `f`. The final check is the `.AND.` check, between this less than or equal to check and the check described previously.

The important point is that the order of operation is from within the lowest/deepest nested bracketed conditions first, up to the broadest checks. This is the same as the way when performing algebra, statements within brackets are evaluated first.

Conditionals return a `LOGICAL` variable if assigned to a variable, but it is also important to remember that these `LOGICAL` variables are equivalent to the 'condition' required for `IF-ELSE IF-ELSE` statements. The following two examples would operate identically:

```
LOGIC1 = ((a .NE. b) .AND. (c .EQ. d))

IF (LOGIC1) THEN
  WRITE(*,*) 'Statement True'
END IF
```

```
IF ((a .NE. b) .AND. (c .EQ. d)) THEN
  WRITE(*,*) 'Statement True'
END IF
```

In the first case we do not need to *check* that '`LOGIC1`' equals `.TRUE.`, we simply need to supply the required conditional with just a `.TRUE.` or `.FALSE.` directly. While one could instead write '`IF (LOGIC1 == .TRUE.) THEN`', this is unnecessary. All that the conditional `(LOGIC1 == .TRUE.)` returns is a `LOGICAL` value, since `LOGIC1` is already a `LOGICAL` variable which is the result of our conditional, it can be used as in the example.

Note, however, that the brackets around the `LOGIC1` variable in the `IF` statement **is** necessary. A code will **not** run if the conditional for an `IF` statement is not enclosed in brackets.

17 References

[1] xming Sourceforge Web Page, http://sourceforge.net/projects/xming/. (Accessed June 2010)

[2] R. M. Stallman & the GCC Developer Community, *Using the GNU Compiler Collection; for GCC version 4.5.0.* GNU Press, 2008.

[3] The gfortran team, *Using GNU Fortran; for GCC version 4.5.0.* Free Software Foundation, 2008.

[4] gfortran Web Page on GNU http://gcc.gnu.org/fortran/. (Accessed August 2010)

[5] GNU Project Web Page, http://www.gnu.org/. (Accessed June 2010)

[6] T. M. R. Ellis, Ivor R. Philips & Thomas M. Lahey *Fortran 90 Programming* Addison-Wesley, Wokingham, 1994.

[7] Michael Metcalf & John Reid, *Fortran 90/95 Explained.* Oxford University Press, Oxford, 1990.

[8] David R. Will, *Advanced Scientific FORTRAN.*

[9] Loren P. Meissner, *Fortran 90.* PWS Pub.Co., Boston, 1995.

[10] William H. Press *Numerical recipes in FORTRAN : the art of scientific computing* Cambridge University Press, Cambridge, 2nd Edition, 1992.

[11] William H. Press *Numerical recipes in fortran 90 : the art of parallel scientific computing Vol.2, Fortran numerical recipes.* Cambridge University Press, Cambridge, 2nd Edition, 1996.

[12] Sorting Algorithm Animations, http://www.sorting-algorithms.com/. (Accessed August 2010)

[13] Ubuntu OS Web Page, http://www.ubuntu.com/. (Accessed August 2010)

[14] vim Web Page, http://www.vim.org/. (Accessed June 2010)

[15] GNU Web Page on emacs, http://www.gnu.org/software/emacs/. (Accessed June 2010)

[16] emacs wiki Web Page, http://www.emacswiki.org/. (Accessed June 2010)

[17] Nano Web Page, http://www.nano-editor.org/. (Accessed June 2010)

[18] Hewlett Packard *Fortran 90 Programmer's Reference.* Hewlett Packard, 1st Edition, October 1998

[19] The Unicode Consortium. *The Unicode Standard, Version 5.2*, Mountain View, CA: The Unicode Consortium, 2009. (http://www.unicode.org/versions/Unicode5.2.0/)

[20] TOP500 Supercomputers List Web Site, June 2010 list, http://www.top500.org/. (Accessed September 2010), (Presented formally at a meeting of the ISC[24])

[21] TOP500 Supercomputers List Web Site, November 2013 list, http://www.top500.org/. (Accessed November 2012),

[22] Linpack High Performance Computing Benchmark Web Site, http://www.netlib.org/benchmark/hpl/. (Accessed September 2010)

[23] 'Jaguar' Supercomputer Web Page at the National Center for Computational Sciences Web Site, http://www.nccs.gov/computing-resources/jaguar/, (Accessed September 2010)

[24] International Supercomputing Conference Web Site, http://www.supercomp.de/isc11/. (Accessed September 2010)

[25] H. Geiger & E. Marsden, Proceedings of the Royal Society A, Volume 82, Pages 495-500, July 1909.

[26] H. Geiger, Proceedings of the Royal Society A, Volume 83, Pages 492-515, April 1910.

[27] J.J. Thompson, Cambridge Literary and Philosophical Society, Volume 15, Part 5, 1910.

[28] E. Rutherford, Philosphical Magazine, Series 6, Volume 21, Pages 492-515, May 1911.

[29] The LaTeX 3 Project Team, *LaTeX 2ε for Authors.* 2001