Getting Started With Linux and Fortran – Part 2

by Simon Campbell

[The K Desktop Environment, one of the many desktops available for Linux]

ASP 3012 (Stars)

Computer Tutorial 2

Contents

1 Some Funky Linux Tricks

Linux has some nice little features that make working with the command line much easier. Here are just a few to get you going:

1.1 Terminal (Shell) History

Linux remembers all the commands you have typed in previously. You may have noticed some commands (like those for compiling F77 code) can get quite long winded. There are two easy ways to save repeatedly typing in the same commands:

1. Use the **Up/Down Arrows** - you can move through all your previous commands. When you find the right one you can just hit <enter>.

2. Use the **!** symbol. If you remember the first letter (or two) of a command you recently typed in then typing **!** and then this letter(s) will automatically run the command. For example, if you typed in **gedit&** five minutes ago and now need another gedit window, you could just type **!ge** and it will automatically run the command.

You can also look at the whole history of commands just by typing **history** in the terminal.

1.2 Using the Tab Button

This is another time saver. Linux stores the names of all the commands and filenames it has access to, in a list. If you start typing a command or a file name (say the first two letters) and then hit **<tab>** then Linux will check its list and if it finds something that matches it will auto-complete the typing. An example might be if you wanted to run the Firefox web browser. Instead of typing the whole command, you can just type **fire<tab>** and Linux will finish the name. The only thing to remember is that there may be many commands that start with 'f'. If you just typed **f<tab>** then you'd get a whole list of commands. Just keep adding letters and hitting <tab> until you get the one you want.

1.3 More or Less

These are two programs that let you quickly look at the contents of files. They do pretty much the same thing as the **cat** command but they let you scroll through the file (using arrows and page-up/down). To use them use just type **less *filename*** or **more *filename***. If you want to find a particular word in a file you can also do a **search by typing the / symbol** and then the letters you want to search for. For example, to search for a read statement in your F77 file, just type **/read** and it will find the next place where it finds 'read'. To exit the program just type **q** (for quit). One thing to keep in mind is that more (and less) are *case sensitive*.

1.4 Man Pages - Linux Info

This is a very handy feature of Linux. Man pages are just text files with information on (almost) all of the Linux commands. Say you've heard of a command but want to know exactly how to use it. All you do is type **man *commandname*** and Linux will show you the documentation on that command (it opens the info file with the *more* program for you, so you can scroll through the manual). For example, if you wanted to know what the **lpr** command meant, just type **man lpr**. To exit just type **q** (for quit). All the man pages have the same layout. At first it is hard to understand them but after you get used to Linux they are quite handy.

2 Basic Fortran 77 Programming Part 2: Plotting Program

In Tute 1 we learnt the very basics of writing a code from scratch, compiling it, then running it. However, it didn't do alot :) In this tute we will be learning about dealing with data files and plotting graphs. We will write another little Fortran 77 code called **'plotter.f'**. If you already know how to do this, then **Sheet 3** has some more advanced computer-based questions (Q3, 4 & 5 - see John Lattanzio's webpage).

- **First of all, get a text editor up and running so we can type the code in.**

2.1 Anatomy of a Computer Code

The computer code for this tute is on the next page (Figure 1). Boxes have been put around the main structures within the code, so you can see what each part does. Computers only do what you tell them to, so the instructions (in whatever computer language) have to be organised in a logical way. Usually it is good to plan out a computer code, setting out all the parts that you will need (like a flowchart), before actually writing it.

2.2 Description of the Program Sections and Commands

As mentioned above, an overview of the code we will be using is on the next page (Figure 1). The next few subsections refer to this, so it's probably a good idea to have a look at it before reading on.

2.2.1 Declaration of Variables

This is a very important part of any program (in any language). This is where you tell the computer which variables you are going to use. Not only that but you also tell the computer what ***type*** each variable will be. There are three main types of variables in Fortran 77: **integer, real, character**. Luckily, they mean what you'd expect.

When you declare a variable, you give the ***type*** then the ***name***. For example, if you want a variable to store the value of 1.567 and you want to call it 'mynumber' then you just type: **real mynumber**. Now, there are actually two types of real variables - **real*4** and **real*8** (pronounced 'real star 8'). If you don't specify which one to use, the default is real*4. The only difference between them is the amount of memory they take up in the computer (4 or 8 bytes). Using real*8 gives a higher accuracy but we don't need to worry about that yet (just use real*4).

Two of the real variables have brackets after them (*xdata(132)* and *ydata(132)*). These are **arrays**. They can store up to 132 numbers. The way to reference the data is: xdata(1), xdata(2), ... etc. So they are one-dimensional arrays. This is where we will store all those numbers the code will read in from the data file.

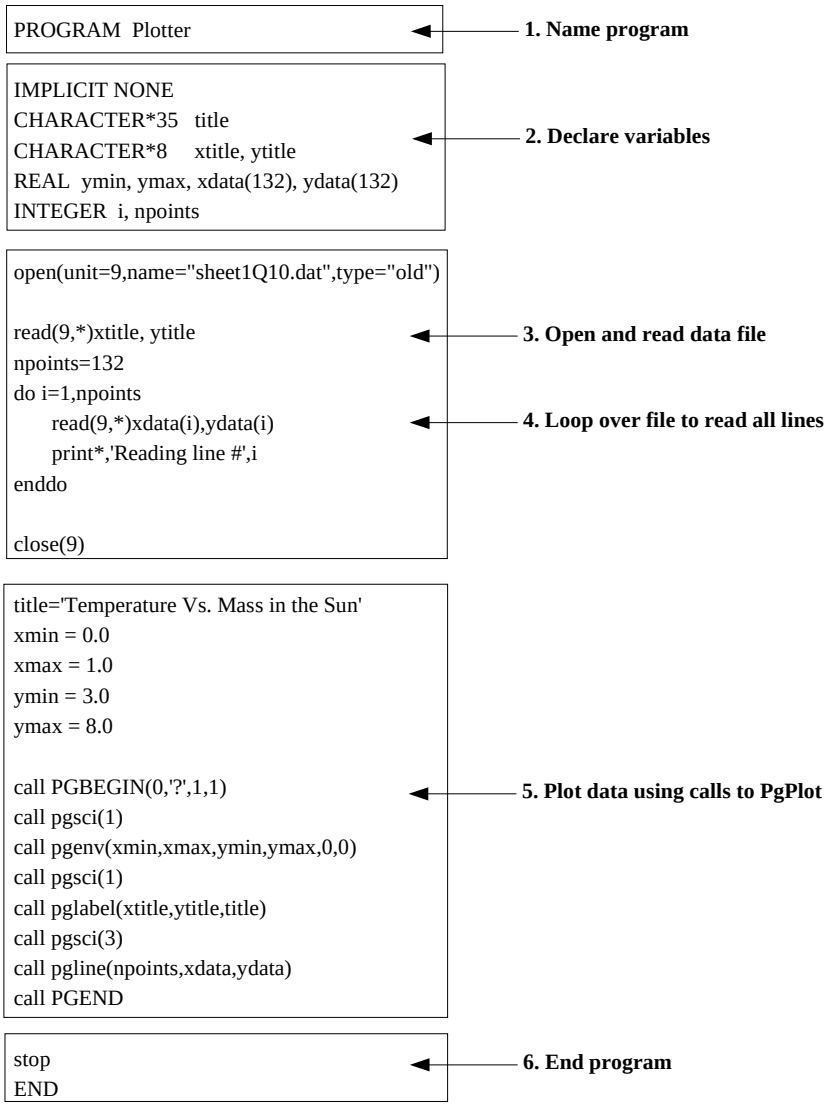

Figure 1: This computer code has been broken up into its main functional pieces. See text for details on particular commands.

Declaring **integers** is the same, except no need for *4 or *8.

Character variables (or 'string variables') are a little different. Here you need to tell the computer how many characters long you need the variable to be. For example if you want to store the word 'hello' later in the program in a variable called 'message' then you'd just declare it like this: **character*5 message**. That way your variable is 5 characters long. Of course you can always make it longer (say *50) if you don't know how big the word is going to be later in the program.

You may have noticed the command **'implicit none**' at the very start of the program (Figure 1). This tells the compiler that you are going to declare all variables - there will be no implicit variables. It is good practice to declare everything.

2.2.2 Reading in Data from a Text File

This is a task that comes up often in science - you have some data and you want to load it into a computer program so you can manipulate it or plot it. Here we just want to plot the data from Question 10 on Problem Sheet 1.

To do this we need to **open** the file. The open command has a number of extra options we need to fill in. The first is to assign the open file a **unit number**. In plotter.f I use **unit = 9**. You could use almost any number but remember from the last tute that 5 and 6 are already taken (reserved input/output unit numbers for the keyboard/screen). The next most important thing is the **name** of the file. This is the name that you saved it as. If you need to, do an **ls** in the terminal to check what your file is called. Don't forget to put the **double quotes** around the name. The **type** of file is unimportant at this stage, just use **"old"**. Note that when you open a file you must **close** it after you've finished reading it (here we would use **close(9)**).

Once the file is opened we can read data in from it. To do this we just use the **read(9,*)** command. The 9 is the unit number that we specified above, so the computer knows which file to read if there's more than one open. The * tells the computer about the formatting of the read statement. A * just means 'don't worry about the format, just read the numbers'. Sometimes, if the data file is complicated, you can't use a * - you have to use a format statement - but we won't worry about that right now.

On the right of the first read statement there are two character variables. When the computer reads the file it will store the strings (words) in these variables (*xtitle* and *ytitle*).

2.2.3 Do Loops

If you have a look at the data file you will see that there are lots and lots of numbers, with two numbers to a line. Fortran reads files line-by-line so we are going to have to do lots of read statements! That's alot of typing... However, there is an easier way.

Do loops are very very handy. All they do is repeat a set of commands between **do i = n,m** and **enddo**. The variable **i** can be any variable but it must be an **integer** - it's just for keeping count in the loop. The **n** and **m** tells the loop to count from **n** to **m**. They can be integer variables or just numbers (usually we start at 1 and go to some other number).

Inside this do loop (Figure 1) we can see a read statement and a **print** statement. The do loop tells the computer to do these two commands 132 times! It's 132 because that's how many lines there are in the file.

Now we have all the data read into the computer we can play with it - and make a graph!

2.2.4 Plotting the Data with Calls to PgPlot

The first thing to do is set up some parameters for the graph, so we set a title name and the maximum and minimum values for the x and y axes of the graph (*xmin, ymax, etc.*).

Next we call PgPlot into action by issuing the **'call pgbegin()'** command. Further calls to PgPlot apply colours and plot the graph, as listed below:

Command	Effect
pgbegin	Starts up PgPlot
pgsci(n)	Sets the drawing colour (n=1 =>white, 2 => red, 3 => blue, etc.)
pgenv()	Sets up the details of the plotting window.
pglabel	Tells PgPlot to use labels on the graph.
pgline(n,x,y)	Draws a line between the points (n = no. data points, x = x data array, y = ydata array)
pgend	Closes PgPlot

That's it! That's all you need to do to plot a data file. If you want more info on PgPlot, try a Google search or go to their home page:

http://www.astro.caltech.edu/~tjp/pgplot/

2.2.5 Stop and End Statements

Never forget to put these in at the end of you program, otherwise it won't compile properly.

2.3 Compiling Code with PgPlot Calls

In the last tutorial we learnt how to compile an F77 program with the **f77** compiler. We ended up with a file that the computer could understand and execute. The current program uses some features of the plotting program **PgPlot**. Pgplot is basically a collection of **libraries** - code that someone else has written that you can use in a Fortran program. When we compile an F77 program that contains references to PgPlot routines we need to tell the compiler (f77) exactly where to find those routines. This is done on the command line by adding some options like this:

f77 -lpgplot -o *program.exe program.f* **-L/usr/X11R6/lib -lX11**

So, to compile the *plotter.f* code we've just made, just replace the *program.exe* and *program.f* with **plotter.exe** and **plotter.f**. Hopefully it compiles without any error messages :) Remember you can always use the up arrow to save typing in the long compile command again and again.

2.4 Running the Code

Run the code in the normal way, with a **./** ('dot forward slash') in front of the executable's name, like so:

./plotter.exe

If all is working well you will get a message like this:

"Graphics device/type (? to see list, default /XS):"

This is a message from PgPlot. It is asking how you would like the graph displayed. If you type **/XS** then hit <enter> it will put the plot up on the screen. If you want it to make an image file you just type **/GIF** or **/PS** for a gif file or a postscript file. Another good one is **/CPS** for a colour postscript file. PgPlot will save the image as **pgplot.gif** or **pgplot.ps**. It's wise to rename the file straight away because it will overwrite it the next time you make a plot. To rename just type:

mv pgplot.gif mypic1.gif

To look at your saved image try:

display mypic1.gif

or open it in a graphics program like **gimp**.

2.5 Printing Your Masterpiece

When you're happy you've got the best plot and want to make a hard copy, try using the **lpr** command to print. I haven't tested this in the computer labs yet but give this a go:

lpr -P*printername* **mypic1.gif**

There should be no blank space between the -P and the printer name. The printer name should be on the printer itself (I hope!).

You can also print your code using the cool **a2ps** command:

a2ps -P*printername* **plotter.f**

This gives a nice formatted and syntax-highlighted output of your code to the printer. **Warning:** only print text files like this - if you try a2ps with an image file or postscript file you'll probably get 5 million pages of rubbish!

3 Bored?

If this is all too easy for you then you may want to go on and try the computer related questions on **Sheet 3**. Questions 3, 4, and 5 are about solving Differential Equations with various numerical schemes (see John Lattanzio's webpage).

ps. John Lattanzio's webpage is wrong on sheet 1. Here it is:

http://www.maths.monash.edu.au/~johnl/astro/ASP3012/stars.html

Mine also has some useful info:

http://www.maths.monash.edu.au/~scamp/tutes/asp3012/index.htm

4 Want Linux for Your Computer?

If you want to practice with Linux (and programming) at home, or just want to give it a go, I can recommend these ‘distros’, which you can download for free:

- **http://fedora.redhat.com/** (Fedora Core - I have this on my laptop)
- **http://www.ubuntulinux.org/** (Ubuntu)
- **http://www1.mandrivalinux.com/en/ftp.php3** (Mandriva (was Mandrake))
- **http://www.debian.org/** (Debian)

If you’re really keen - or don’t have a broadband connection to download the installation cds - I can burn some copies for you :)