Advanced UNIX/Linux for Scientific Researchers

Workshop Nov 12th, 2012

Overview

A full day hands-on workshop that will instruct on advanced techniques of UNIX/LINUX directed towards engineers and scientists. The workshop begins with monitoring and altering programs running on a linux computer. It then builds on previous knowledge of using built-in UNIX/LINUX programs for finding, analysing, and modifying files and datasets. It then progresses into a key aspect of this workshop which is learning how to do basic UNIX/LINUX programming (i.e. shell scripts). In this, you will learn how to write shell scripts to do repetitive UNIX/LINUX tasks, perform decision making, run a series of your programs and analyse data using built-in UNIX/LINUX programs automatically, without your intervention.

Students will learn hands-on at a computer and will be instructed through a combination of tutorial style lectures and multiple practice sessions where the student will undertake practice tasks to reinforce their learning.

Prerequisites

Introduction to UNIX/LINUX for Scientific Researchers or equivalent prior experience.

Syllabus

1) Monitoring and altering the running of programs using ps, top, tail, and nice/renice.

2) Dealing with data using grep, awk, sed, redirection, and pipes.

3) Shell environment (environment variables, adding, changing, deleting) and defaults (altering default values).

4) Shell programming (Shell scripts). Writing shell scripts in the Bourne Shell. Learning how to do decision making and loops in shell scripts and the implementation of data analysis (grep, awk, etc) within these shell scripts. Running shell scripts in the background.

1. Login and password information

Login: - use your ISS login and password -
Password:

2. Monitoring programs

In this section, you will learn how to monitor, modify, and kill your jobs running on your computer.

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
3952	1	3956	3956	con	1005	20:29:36	/usr/bin/bash
1440	3956	1440	436	con	1005	20:29:41	/usr/bin/ps

If you want to delete the job then you can refer to the job by the PID and stop it with:

```
kill -9 3952
```

If you want to monitor your job continuously then many unix/linux computers will have the facility called top. Type:

```
top
```

This will update information about the jobs every couple of seconds. The important information for monitoring your jobs are the PID, the cpu and RAM memory usage, the nice value, the time the job has been running, and the program name.

To kill a job using top, type k and then specify a value such as 9 for the severity of the kill.

The priority a job is given on a computer is given by the nice value. If you want a long running job to only use a small fraction of the cpu usage whist you are working on the computer doing other task, but then use as much as possible when you leave the computer dormant, then use renice. Assign a new value for nice using Renice. Use a value of 19 to reduce the usage of the computer to a minimum. Renice in top using r followed by the PID and the new nice value.

Run a program (e.g. a.out) in the background by typing:

```
a.out &
```

If you have output that would normally go to the screen then redirect it to a file.

```
a.out >& out.results &
```

The > redirects the prints that would normally go to the screen. The >& also outputs any standard error comments as well as prints to screen to the out.results file.

If the program requires information to be entered from the terminal during running, then put the input in a file (using vi or another editor; give the input file a name such as input.file). Now type:

a.out < **input.file** >& out.results &

The last & makes the program run in the background so that you can continue to work or even log out and come back to the computer to check the program later.

Tail is used to look at the end of a file.

tail out.results allows you to look at the last lines of the file. If the file is being written to from a computer program (as in the command above), then monitor the output using tail with a –f option. This will update the tail command any time the out.results file is updated:

tail –f **out.results**

3. Data manipulation

Redirection has been introduced in the previous section (<, >) and in the introductory unix/linux workshop. In this section we will use it in conjunction with other commands.

3.1 Grep and Pipes (|)

As you will recall, if you have a file name **file.ex1** with

```
1     4     5
3     4     5
34    4     56
2     2     2
```

If you want to search this file using grep to isolate all lines with a 5 then type:

grep 5 **file.ex1**

This will produce output:

```
1     4     5
3     4     5
34    4     56
```

to illustrate pipes, now instead of having the results printed to the screen we can feed the results back into grep to isolate lines with 3 by:

grep 5 **file.ex1** | grep 3

and the results should be:

3 4 5
34 4 56

To isolate the line with a 6 in it from the above, use another pipe:

grep 5 **file.ex1** | grep 3 | grep 6

You can then continue this as much as you like, e.g.

grep 5 **file.ex1** | grep 3 | grep 6 | grep 5 | grep 4 | grep 3 | grep 6

This illustrates that you can use pipe to join up unix/linux command. Many commands can be used with pipes. Also, you can redirect output from the screen to an output file. Try for example,

cat **file.ex1** | grep 3 | grep 6 > out.file4

3.2 Sed - Stream Editor command

Sed is used to modify files. There are many options that can be used, and we will address a few. (For a thorough treatment of sed, see the online tutorial by Bruce Barnett and GEC).

Substitution operator, s.

In this, we will alter all occurrences of unix in a file named text1 and we will save it as a new file called text2.

sed 's/Mike/mike/' < **text1** > text2

The delimiters are /. If you are searching for unix/linux and want to replace that with linux/unix then type:

sed 's+unix/linux+linux/unix+' < **text1** > text2

As long as the delimiter is not being searched for and there is three occurrences of the delimiter then this is suitable.

Sometimes you will want to repeat what you have searched for and add characters. For example, if you want to add brackets around the work mike, ie (mike) in the file then type:

sed 's/mike/person (&)/' < **text1** > text3

Use in conjunction with pipes as:

cat **text1** | sed 's/mike/person (mike)/' > text4

Use of \(, \), and \1.

sed 's/\(unix\) is very similar to linux/\1 /' < **text1b** > text2

Also, try:

sed ‘s/\(unix\) is very similar to \(linux\)/\1 is essentially \2 /’ < **text1b** > text2

Variant operator, [ ].

Other useful regular expressions are:

cat **text1** | sed ‘s/[Mm]ike/Michael/’ > text4

To replace all upper and lower case Mike’s by Mike, or

cat **text1** | sed ‘s/\([Mm]\)ike/\1ichael/’ > text4

This will put Mike or mike as (Mike) or (mike) but leave the case.

Global operator, g.

cat **text1** | sed ‘s/[Mm]ike/Michael/g’ > text4

Not including operator, ^.

cat **text1** | sed ‘s/[^M]ike/Michael/g’ > text4

Operator *.

The * operator will match 0 or more occurrences:

cat **input1** | sed ‘s/[a-zA-Z]*//g’ > text4

Remove words and numbers

cat **input1** | sed ‘s/[a-zA-Z][0-9]*//g’ > text4

This will replace all the words in the file, but it will ignore words with characters other than letters of the alphabet.
Consider a file such as input2:

0.000 556.0
1.212 454.2
2.012 -999.99
3.340 233.4
4.132 -45.0
5.234 -60.0
7.321 -80.0
8.456 35.0
9.467 76.0

11.003 203.4

Sometimes datafiles have -999.99 when no data exists. If you need to change these values to another then you can use:

cat **input2** | sed 's/-999.99/0.000/g' > text4

You may find you need a \-999.99 since the dash is a special character. If for some reason you need all the negative numbers to be set to zero then use:

cat **input2** | sed 's/-[0-9]*/0.000/g' > text4

Option –n:

cat **text1** | sed –n ‘s/[Mm]ike/Michael/g’ > text4

Print option:

cat **text1** | sed –n ‘s/[Mm]ike/Michael/p’ > text4

Option –e:

cat **text1** | sed –e ‘s/MIKE/Michael/g’ –e ‘s/[Mm]ike/Michael/g’ > text4

Option –f

You can put the multiple commands in a file and call it match. So in file named match put:

s/MIKE/Michael/g
s/mike/Michael/g

Then type at the command line:

cat **text1** | sed –f match > text4

or

sed –f match < **text1** > text4

Applying the sed command to a limited set of lines is achieved by:

sed ’3,6 s/Mike/Michael/’ < **text1** > text4

in which the command will only change mike to Michael on lines 3 to 6. If you want it to apply to line 6 and greater then use:

sed ’6,$ s/[Mm]ike/Michael/’ < **text1** > text4

where $ is the end of the line in the file.

Delete operator:

cat **text1** | sed ’3,5 d’ > text4

Will delete lines 3 to 5 and print the rest to text4.

Note: the commands introduced here are useful in other unix commands like, for example, the vi editor, grep, etc.

3.3 Awk programming language

The awk command is a programming language in its own right. It is a useful tool for data manipulation. The general form is:

awk '/pattern/{action}' **file**

Awk assumes that the data file is in columns and that these are separated by a delimiter, the default is a space. The pattern and action don't have to both be present. For example, we could search a file and perform an action.

awk '{print $2}' **datafile1**

This command will print the second column of data. Alternatively, we could use the pattern part,

awk '/0.05673/' **datafile1**

Together, we get

awk '/0.05673/{print $2}' **datafile1**

This will print column 2 every time the pattern of 0.05673 is matched in any column. If you want to print the whole line then either miss out the print command or use $0,

awk '/0.05673/{print $0}' **datafile1**

There is more capability that the above, for example,

awk '$3 == 0.040302 {print $1, $2, $4}' **datafile1**

or with characters,

awk '$2 == "Mike" {print $3}' **datafile2**

There are various tests that can be performed:

Test	**Summary**
==	Test to see if they are equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less then or equal to

You can also do math operations such as

awk '$3 == 0.040302 {print $3+$2+$1}' **datafile1**

The math operations are summarised in the table below:

Function	Summary
sin(x)	Sine of x in radians
cos(x)	Cosine of x in radians
tan(x)	Tangent of x in radians
log(x)	Natural log of x
exp(x)	e to the power of x
sqrt(x)	Square root of x
int(x)	Integer part of x
rand(x)	Random number between 0 and 1
srand(x)	Seed for rand()

The output can be expanded upon. If you want to output three columns of data every match then write:

awk '$3 == 0.040302 {print $3, $2, $1}' **datafile1**

Formatting output can be performed with printf. The following are useful for characterising the output to be printed:

Formatting type	Description
C	If a string, the first character of string, if an integer, the first character that matches the first value
D	An integer
E	A floating point number in scientific notation
F	A floating point number in conventional notation
G	A floating point number in either scientific or conventional notation, whichever is shorter
S	A string

awk '{printf "%s is an age of %d\n",$2,$1}' **datafile2**

Notice the new line is specified by \n. Other indicators are:

Indicator	Description
\a	Bell
\b	Backspace
\f	Formfeed
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\c	Character c

Changing the field separator:

awk –F"#" '/Mike/{print $1}' **file**

The awk command will now search for lines separated by # instead of the default space. File may have contents like:
34.00#3.244533#4.300
343.0#43.3#4.3

Wildcards or Meta characters are useful for specifying ambiguous or general situations. We have been introduced to them in the sed command. The table below lists a more complete list with definitions:

Wildcard or Meta character	Description
^	Matches the first character
$	The end of the field.
~	Matching records ($2 ~ /4$/ seeks a 4 at the end of record 2).
.	Matches any one character
|	Matches or (e.g. /Mike | mike/)
*	Zero or more repetitions of a character
+	One or more repetitions of a character
\{1,3\}	Matches between 1 and 3 repetitions
?	Zero or one repetition of a string
[Mm]	Search for M or m (e.g. /[Mm]ike/)
[^M]	Do not search for M

Task 1
Try the following commands and try some of the wildcards in the table above.
awk '/Steve|steve/{print $1}' **datafile2**

awk '$2 ~ /Steve/{print $1}' **datafile2**

awk '$2 ~ /S.eve/{print $1}' **datafile2**

awk '$2 ~ /St*/{print $1}' **datafile2**

awk '$2 ~ /^S/{print $0}' **datafile2**

awk '$1 > 40{print $0}' **datafile2**

awk '/Steve|steve/{print $1}' **datafile2**

Task 2

A program has run and is performing a looping operation and converging toward a solution. The output file has the following form:

```
Code version 3.2 run 53 iteration        654
 0.5431046          654  error   1.5290520E-03
 0.9611790
 0.5394807
-999.99
 0.3960433
 0.3017445
 0.4225559
 0.8992636
 0.9263668
 0.4498041
 0.8580322
```

The number before the word "error" is the iteration (number of times the calculation has been done) and the number after the word "error" is the amount of error in the calculation. The error gets smaller and smaller as the run goes on. You are tasked with the following. (The file is called iteration.dat that you will use).

a) Does the error get smaller than 1.4400000E-03? Find out what iteration that the program first obtains an error less than this value. Print the error and the iteration number.

b) Remove all words in the file.

c) Convert all occurrences of -999.99 to 0.0000 in the file ready for plotting.

You can assemble various pattern matching scripts in a file and then call the file from awk.

awk –f **file.pattern datafile1**

where **file.pattern** contains:

```
$3 == 0.040302 {print $3, $2, $1}
```

Note, there is no need for the quotes.

You can also put BEGIN and END statements in the script file. These are exercised once at the start and end:

```
BEGIN { print "Beginning to analyse the file"}
$3 == 0.040302 {print $3, $2, $1}
$3 == 0.040302 {print $1}
END { print "Finished analysing the file"}
```

Built in variables:

Built in variable	Description
NR	The number of records read
FNR	The number read from the current file
FILENAME	Name of the input file
FS	Field separator (default is a space)
RS	Record separator (default is a new line)
OFMT	Output format for numbers (default %g)
OFS	Output field separator
ORS	Output record separator
NF	Number of fields in current record

You can define your own variables as well.

```
num=5
num = num+1
```

awk 'num=5 {print "The total is now ",$1+num}' **datafile2**

Increments:
num=num++ (increase by one)
num=num-- (decrease by one)

Loops (repetitive procedures)
The script file can contain, for example,

```
BEGIN { print "Beginning to analyse the file"}
{ for (i=1; i <= 3; i++){ print $i }}
END { print "Finished analysing the file"}
```

Task 3

*a) Run the above file.pattern02 using **dataf** and understand what is output. Run using*

*awk –f **file.pattern02** dataf*

*b) analyse the output of using **file.pattern03**, shown below, with **datafile1**.*

```
BEGIN{ print "Beginning"}
{
 for ( i = 1; i <= NF; i++ )
 print $i
}
END{ print "Finished"}
```

Arrays (variables used to hold multiple values)

For example (**file.pattern04**),

```
BEGIN{“Start running”}
count=4
{
for(x=1; x<=count; ++x) {
elem[x]=$x
print x,$x
}
print "elements 2 and 4 are =",elem[2],elem[4]
}
END{“Finished running the script”}
```

4. Managing your program – Makefile

This is a good reference for you if you should need Makefile. I will spend only a small amount of time discussing it in the workshop.
Make is a very useful unix/linux command work working with the compilation of programs. If you have a program and it has a lot of subroutines and/or functions then if you make a change to one of those subroutines then you don’t want to have to recompile the whole code. Make checks to see if any subroutines have been updated and it will automatically compile those subroutines and make a new executable program.

The target name is followed by ‘:’ and then the dependencies of that target. Each of those dependencies must also be listed as a target with way to create those targets. Say you have a program that you will run called goles and it has subroutines called **nnsteps.f** and **les.f**. You would then construct the **Makefile** as:

```
goles: nnsteps.o les.o
        f77 –o goles nnsteps.o les.o
nnsteps.o: nnsteps.f
        f77 –c nnsteps.f
les.o: les.f
        f77 –c les.f
```

The indented lines must be indented with a tab. To compile the program called goles you type:

```
make goles
```

You may need to change compilation options and you will not want to dig into the Makefile to find where to add them. You can do this by adding macros. If you want to use an intelf77 compile instead of the default compile then you can specify a

macro. Macros have a name and a value separated by an equal sign. Often the macro name is in upper case. Makefile now looks like: (# is a comment line)

```
# Macros
# FORT = g77
# FORT = intelf77
FORT = f77

# Target construction
goles: nnsteps.o les.o
        ${FORT} –o goles nnsteps.o les.o
nnsteps.o: nnsteps.f
        ${FORT} –c nnsteps.f
les.o: les.f
        ${FORT} –c les.f

clean: echo “Cleaning up files”
        rm les.o
        rm nnsteps.o
        rm goles
        echo “Done cleaning files”
```

To compile goles, type:

make goles

To start over and compile all subroutines again, type:

make clean
make goles

There are two internal macros that are defined that may be of use, $@ and $?. The first specifies the target and the second specifies all dependencies that needed updating.

```
# Macro
OBJS =  nnsteps.o les.o

# Target
goles: ${OBJS}
        ${FORT} –o $@ ${OBJS}
```

Use the $? to store updated files. If you are working on a project with another person and they want to know what subroutines you have updated then use:

```
# Macro
OBJS =  nnsteps.o les.o

# Target
update: ${OBJS}
        cp $? /public_html/storehouse
```

Where /public_html/storehouse would be a directory that both could view.

5. Shell

When you log in, you are in a unix/linux environment. But in addition to that, you are specifically in a unix/linux shell. Most of the commands that you would use when doing basic unix/linux will be the same in any shell that you are in. However, the use shell variables and of shells for scripts and programming requires choosing a shell and adhering to its commands and way of forming scripts.

There are various shells that you can choose. The main three are the Bourne shell (named sh) (written by Steve Bourne) the C shell (named csh) (written by Bill Joy), and the Korn shell (named ksh) (written by Dave Korn). The C shell has commands for programming that are similar to the computer language C. The Bourne shell is excellent for programming, however, it is a bit limited in terms of the user interface. The Bourne shell was extended to have an excellent user interface and was called the Bourne Again shell (bash). We will use the bash shell for our programming.

Whenever you open a terminal window it will be in the default shell (which is set by the system administrator). If you want to determine your shell type then type:

env (stands for environment)

You can search through this for csh or bash or ksh, or you could type:

env | awk '/SHELL/'

You can see that there is a shell variable called SHELL which has a value of

SHELL=/bin/csh

So the default is the C Shell on feeble.

You will be working in the bash shell so you should type

bash (to get be in the bash shell).

If you wanted to always use the bash shell then you can alter this in the .cshrc file. The default is the C Shell and some initial commands are stored in the .cshrc file. You can add commands in that file to do things such as switch shell. So at the bottom of .cshrc add bash. Now if you log out and log in again you will see that the terminal has a prompt with bash in the name. You are now in the bash shell.

Alias

If you want to modify your commands so that a specific option is always used then you can use aliases to do that job. If, for example, you always want long formats when using ls, i.e. ls –l then alias it. For bash, use:
alias ls="ls –l"
(For csh use alias ls "ls –l")

If you are happy with that then you can put it in your profile, .cshrc or for bash it is either .profile or .bash_profile.

5.1 Environment variables

If you type env then you will see a list of variables that are set for the system. A selection including some of the main ones is shown below.

HOSTNAME=maxima
USER=lecsjed
MACHTYPE=sparc-sun-solaris
SHELL=/bin/csh
HOME=/home/maxima01_d/lecsjed
PATH=/adm/sge/bin/solaris64:/opt/globus2/bin:/opt/globus2/sbin:/usr/local/bin:/apps/local/bin:/usr/bin:/opt/SUNWspro/bin:/usr/ccs/bin:/usr/openwin/bin:/usr/dt/bin:/usr/ucb:/opt/SUNWhpc/bin:/opt/sfw/bin:/opt/SUNWsamfs/bin:/etc:/etc/local:/apps/SRB/bin:.:/home/maxima01_d/lecsjed:/home/maxima01_d/lecsjed/bin:/apps/explorer/bin:/apps/explorer/lib:/usr/local/grace/bin:/apps/mpichg2/bin

HOSTNAME is the name of the computer, USER is the login name, MACHTYPE is the machine type, SHELL specifies the default shell, and HOME specifies the home directory. The last one is PATH. This is a list of directories that the computer will search (in order) when you issue a command. The directories are separated by a colon (:). So, if you compile a program and called it a.out in your home directory and you are ready to run the program so you type a.out (and hit return) the computer will search for a.out first in the directory /adm/sge/bin/solaris64 then if not found it will look in /opt/globus2/bin, and so on. Notice that eventually it will come to :.:. The "." in unix/linux for directories specifies the current directory. When some linux versions are installed, the "." Is not in the PATH variable and so if you run a program it isn't able to find it in the current directory and you have to specify the path, which is ./program to run it.

You can use the shell variables. For example, if you want to show the value of a shell variable then use echo and a dollar sign to specify the value of the variable. Consider:

echo SHELL (this would echo the work SHELL)

echo $SHELL (this would echo the value of the variable SHELL which is /bin/csh).

If you are off in a directory and you want to return to the home directory then you can specify cd $HOME. (This is what cd is doing when you don't give it a directory).

If you want to modify the shell variables when you are in a bash shell then type

Whereis bash (this computer has it stored in /bin/bash)

export SHELL=/bin/bash

export PATH=$PATH:.:

If you want to find a program (and a shell is a program that runs), then type which bash or whereis bash. This will search for bash in the PATH and list the directory if it finds it. If not, it will tell you which directories it searched.

If you want to add to a shell variable, such as the PATH, a directory off your home directory that you will store your programs in (~/programs) then type:

export PATH=$PATH:$HOME/programs/

5.2 Shell scripts and programming

The shell is an environment in which you can execute commands and even program (just like with fortran, C, etc.). If you are going to execute a series of the same commands then it is useful to put them in a file and run it like a program (called a shell script). Within the shell script, you can enter programming details that allow you to make decisions, use variables and arrays, and do repetitive operations (loops).

5.2.1 Shell scripts

Say you have a list of commands that you want to do. For example, when a program called **shel.f** (compiled executable called **shel**) is run it produces files called output1, output2, output3, output4. You want to store them away in a directory called runout (not created yet) and list the contents of the files and store that in a file called summary, then perhaps gzip the files to save space. To do this, create a file called **shell1** and in it put the following:

```
#! /bin/bash
# run the program
shel >& out.run
# create an output directory
mkdir runout
# store the files there
mv output* runout
```

```
# copy the print to screen file to runout
cp out.run runout
# go into the directory
cd runout
# assess the files
ls –l output* out.run > summary
# compress the data
gzip output*
echo “done”
```

The top line with the #! /bin/bash is not a comment. The combination of #! followed by the path to the shell will ensure that this shell is used when running the script.

To run the script, you must first make it executable.

chmod u+x **shell1**

Now it can be executed like a series of commands by typing:

shell1

If you plan to be running this **shell1** over and over again and each time you will be modifying the input read by **shel** to give different results. It will always store the data in the same location and so this will cause a problem. It will overwrite your data. You can get around this by entering a variable in at the script running command line.

This is in script:

```
#! /bin/bash
# run the program
shel >& out.run$1
# create an output directory
mkdir runout$1
# store the files there
mv output* runout$1
# copy the print to screen file to runout
cp out.run$1 runout$1
# go into the directory
cd runout$1
# assess the files
ls –l output* out.run$1 > summary
# compress the data
gzip output*
echo “done”
```

You will notice the $1. When you run the script at the command line, redirect a value into the script that will be $1. For example,

shell1a 662

So this run will be called 662. Everywhere in the script, when it is run, the $1 will be replaced by the 662 (the value of the first input value). You can do more than one value, i.e.

shell1b 622 543

Where script would be the following:

```
#! /bin/bash
shel >& out.$1
mkdir runout$2
cp output* runout$2
```

The above script makes use of two input variables to make the output and the directory for output unique.

Running jobs in the background

If you are running a script and suddenly realize that it is not working right or is not going to do as you wanted then you can hit control and C at the same time and it will stop the script.

You may wish to run the script and let it run even whilst you have logged off and gone home. This is achieved by including an "&" at the end of the command. So

shell1b 622 543 &

will run this process in the background. This script will run quickly and so there is no benefit of running it in the background. If a script is running for a long time then you can monitor it using ps, it will show up as another shell and can be terminated by kill -9 PID, where PID is the PID number of the shell. If you use top to kill the task that the shell is doing (i.e. kill the a.out4 program) then the script will go onto the next command.

A word of caution, a shell script is very powerful and can do operations independent of you like moving around within your directories storing files, running programs, analyzing, making new directories, etc. If, however, the commands in the shell script are not doing exactly what is expected then the shell script can do a lot of damage to your files. It may store files in the wrong place and over write work, it may delete files in the wrong place. I always use a terminal and step by step check what the script will do before letting it lose to run on its own.

5.2.2 Graphing data from within a script

In dealing with data, it is common to want to process the data using scripts and have plots ready prepared for analyzing. There are many plotting packages available that produce publication quality graphs, such as IDL, matlab, gri, etc. These can be run

from a script but require quite a short course to understand them. I will introduce gnuplot which is useful for producing reasonable plots. If you type:

gnuplot fileinst

And in the fileinst you put the plotting instructions then it will produce the desired plot without interaction by a person during the plotting (so it can be run from a script). The fileinst we will consider contains the following commands (called fileinst2):

```
set terminal postscript
set output "output.ps"
set title "Energy vs. Time for Sample Data"
set xlabel "Time"
set ylabel "Energy"
plot "input.dat" with lines
pause -1 "Hit any key to continue"
```

This produces a plot using the data from input.dat. You can view the graphics file using standard unix viewers for postscript files such as gv output.ps or gs output.ps.

This is contained in **shell3**:

```
#! /bin/bash
# run the program
shel >& out.run$1
# create an output directory
mkdir runout$1
# store the files there
mv output* runout$1
# copy the print to screen file to runout
cp out.run$1 runout$1
# go into the directory
cd runout$1
#-------------------------
# plot the data
cp output1 input.dat
gnuplot fileinst2
cp output.ps output1.ps
#second output file
cp output2 input.dat
gnuplot fileinst2
cp output.ps output2.ps
# third output file
cp output3 input.dat
gnuplot fileinst2
cp output.ps output3.ps
# fourth output file
cp output4 input.dat
gnuplot fileinst2
cp output.ps output4.ps
#-------------------------
```

```
# assess the files
ls –l output* out.run$1 > summary
# compress the data
gzip output*
echo “done”
```

The plotting section enclosed by the #---------- could be replaced because it is a repetitive operation. We will address this in the Shell programming section below.

5.2.3 Shell programming

If Statements

Decisions can be made in shell scripts by using “if statements”. The general form is:

```
if [ expression ]
then
        “commands”
elif [ expression2 ]
then
        “commands”
else
        “commands”
fi
```

For example, consider: **script4a**

```
if [ 3 -gt 2 ]
  then
  echo "Integer example"
else
  echo “It was not true”
fi
```

It is essential that there are spaces on either side of the expression. The expressions or tests can take the following forms for integers.

More generally, the if statement can be thought of as:

```
if “condition is true or gives a value”
then
        “commands”
elif “2nd condition is true or gives a value”
then
        “commands”
else
        “commands”
fi
```

Let us consider some examples, **script4** is

```
if grep 0.4 d1.dat
  then
  echo "I found 0.4 in d1.dat"
fi
```

Using variables in the tests, consider **script5**:

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
 echo expression evaluated as true
else
 echo expression evaluated as false
fi
```

The following table lists the tests:

Tests - Integers	**Description**
int1 –eq int2	Returns true if int1 is equal to int2
int1 –ge int2	Returns true if int1 is greater than or equal to int2
int1 –gt int2	Returns true if int1 is greater than int2
int1 –le int2	Returns true if int1 is less than or equal to int2
int1 –lt int2	Returns true if int1 is less than int2
int1 –ne int2	Returns true if int1 is not equal to int2

The tests that may be used for strings are as follows:

Test - Strings	**Description**
str1 = str2	Returns true if str1 is equal to str2
str1 != str2	Returns true if str1 is not equal to str2
Str	Returns true if str is not null
-n str	Returns true if length of str is greater than zero
-z str	Returns true if length of str is equal to zero

Test for interrogating files and directories are as follows:

Test – files and directories	Description
-d filename	Returns true if file, filename is a directory
-f filename	Returns true if file, filename is an ordinary file
-r filename	Returns true if file, filename can be read by the process
-s filename	Returns true if file, filename has a nonzero length
-w filename	Returns true if file, filename can be written by the process
-x filename	Returns true if file, filename is executable

Case command

```
case string1 in
        str1)
                commands
                ;;
        str2)
                commands
                ;;
        *)
                commands
                ;;
esac
```

You may put as many lines of commands as you like. String1 is tested to see if it matches str1 and str2 and if not then it matches *.

An example is given by (**script6a**)

```
#! /bin/bash
var="STEVE"
str0="STEVE"
str1="SE"
case $var in
      $str0)
           echo "got here 0"
         ;;
      $str1)
           echo "got here 1"
         ;;
      *)
           echo "got here else"
         ;;
esac
```

There is no limit of commands that you can put in each section.

Loops
There are various forms the loops can be specified. We will illustrate with a couple of types:

```
while expression
do
   commands
done
```

The following is an infinite loop until the file called filename10 comes into existence. This is in script6. (Open a second window and be prepared to make the file called 'filename10' to get the script6 to stop or hit 'Ctrl' and 'c' at the same time to stop the script running).

```
#! /bin/bash
var="STEVE"
str0="STEVE"
while [ $var=$str0 ]
do
  if [ -r "filename10" ]
  then
   exit
  fi
  echo "still running"
done
```

The second type of loop involves a list:

```
for var1 in list
do
    commands
done
```

A simple example is given in **script7**:

```
#!/bin/bash
for d in 1 2 3 4 5;
do
  echo Doing run number $d
done
```

A more elaborate example is given below. This is a script that I use in my research.

```
#! /bin/bash
# Script to allow the LEM to continue with a run.
# You need to specify how many more restarts will be
# needed in order to complete the run.
# Clear up the directory ready for a run
rm -f RUN*
rm -f filea
rm -f fileb
rm -f filez
rm -f fort.15
rm -f leminput.dat
rm -f mesg
rm -f mesgres
rm -f run_info
rm -f runout*

echo 'FAILSAFE' > mesgres

#Start up
rm -f mesg
cp mesgres mesg
cp nmlsetup nmldata
./goles >& runoutdg1

# Continue the chain run
for d in 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34;
do
  echo Doing run number $d
  rm -f mesg
  cp mesgres mesg
  cp nmlchain nmldata
  ./goles >& runoutdg$d
done
```

Task 4

Task 4 is appended after the references section.

References

- Bruce Barnett & GEC
- Time Parker, Linux Unleashed, Sams Publishing, 3rd edition1998.

Task 4

This task is challenging and you may not complete all of the sub-parts.

You are given a program called rtcode. When the program runs, it asks for a value of the size of the aerosol particles in the layer in the atmosphere. Specify a size value from 5 to 50. It then requests a value of the solar angle from 0.1 (grazing) to 1 (overhead). The code then runs and outputs the results into rtoutput. The output is the size, solar angle, reflectance, and transmission.

Your task is to:

1) Use awk to check if any values of reflectance and transmission are greater than 1 (which would not make sense in this example).

2) Use awk to calculate the absorptance. The absorptance is defined by:

Absorptance=1-Reflectance-Transmission

3) Write a shell script to loop over solar angle from 0.1 to 1 in steps of 0.1. Use a fixed value of size of 10. This will be output to the file rtoutput. Be sure to remove the file rtoutput before you run the shell script since it will have previous values in it.

4) Modify the shell script in part 3) so that it can be run for different values of size. Select size ranging from 10 to 50 in 10's. Have the shell script automatically output the results to a new directory and go into that directory and compress the results and then return to your main directory.

5) Write a shell script that will go into each of the directories and use sed or awk to extract the value of the reflectance and transmission for a solar angle of 1. Have the script then automatically plot the data of reflectance verses size to the screen.