Lecture I : Introductory Linux and Basic Algorithm Construction

I. WHY DO WE NEED COMPUTERS IN PHYSICS AT ALL?

- Conduct virtual *experiments*, which are otherwise too costly, too time-consuming or simply impossible.
- Visualize results via graphs, charts, tables.
- Study *large* systems for which theory is hard or impossible.
- Study *many* systems for statistical soundness.

II. WHAT IS LINUX?

- Linux is just another operating system much like Windows.
- It can be provided free of charge and programs are developed by volunteers.
- It is a very stable operating system and is therefore prefered in large computer systems and for long applications.
- Originally written as proprietary software under the name Unix, then rewritten by Linus Torwald and offered free of charge.

III. HOW LINUX OPERATES

A. The filesystem

The logic behind the filesystem in Linux is in some ways similar to that in Windows. Disk space is divided into *directories*, which may branch into several *subdirectories*. The smallest indivisible storage unit is a *file*, which is a concept you should already be familiar with from Windows. A *file* may contain an image format, plain text or computer code. In a Linux system, the topmost directory is usually simply called `/`. This directory is usually divided up into subdirectories each having crucial content for the operation of the system. The exact content varies among different implementations. You can view the content of the `/` directory by typing

```
hande@p439a:~/teaching/phys343/notes/lab01$ ls /
bin   dev  home  media  opt   root  srv        sys  usr  windows
boot  etc  lib   mnt    proc  sbin  subdomain  tmp  var
```

The most common of these subdirectories are [1] [2]

- `/bin` : contains utilities that are essential to system operation. Hence, the shells, file manipulation commands such as `cp` and `chmod`
- `/etc` : dedicated to system configuration. Contains configuration files for the system daemons, startup scripts, system parameters, and more.
- `/lib` : Core applications and utilities installed with Unix require the libraries in `/lib` to run.
- `/usr` : End-user applications, such as editors, games, and interfaces to system features are here, as is the library of man pages, and more. Chances are that if the file is useful, but not mandatory for system operation, it'll be found in `/usr`.
- `/var` : Repository for files that typically grow in size over time. Mailboxes, log files, printer queues and databases can be found in `/var`. It's commonplace for Web sites to be kept in `/var` as well, since a Web site tends to amass data preternaturally over time.
- `/home` : This is the most important. Contains all of the users' *home* directories. A *home* directory is where a regular user keeps his/her non-application files and modifies them. In most Linux systems, it is the default directory that you find yourself in when you log in to the system.

B. Linux commands

In contrast to the Windows operating system, actions in Linux are performed with *commands* that you write on a terminal instead of clicking on icons. The near infinite number of Linux commands that we use not only enable us to perform many different tasks but also provide flexibility within each command through *options*.

A Linux command has the following general structure :

```
 <command> [OPTIONS] [ARGUMENTS]
```

where

- `<command>` is a special instruction that causes the system to perform an action (such as `ls, mkdir, grep`)
- `[OPTIONS]` are special expressions for the particular command which increases its capabilities. Options are usually supplied by means of a single dash and a letter, e.g. `-l`, `-k`, `-a`. You might also encounter options given as two dashes and a full expression, e.g. `--escape, --ignore-backups` (for the command `ls`).
- `[ARGUMENTS]` refer to entities which we would like the command to act upon. These can be files, directories, sentences, words or even numbers.

Because it is impossible to know off the top of your head all the options to all the commands, a very handy tool has been designed by Linux developers, which is called the `man` pages. `man` stands for manual and if you would like to list all the options for a particular command, you invoke the related man pages by typing

```
 man <command>
```

The `man` command, being a command itself, also has options and a particularly useful one is `-k`, which lists the available commands for a keyword supplied as an argument. So suppose you would like to know which commands are available for manipulating, displaying and converting pdf files. You can just type

```
hande@p439a:~/teaching/phys343/notes/lab01$ man -k pdf
pdfeinitex (1)       - PDF output from TeX
texi2dvi4a2ps (1)    - Compile Texinfo and LaTeX files to DVI or PDF
.....
```

and it will list all the available commands. You can then `man` the particular command of interest and learn more about it. I find `man -k` to be very useful because you find yourself learning other commands while you are looking for a particular one, much like looking up a word in a dictionary and learning other words in the process.

C. The shell

Every time you type a command, it instructs the system to perform an action, such as invoke an *editor* or list the files in a directory. The interpreter that converts the commands you type into actual actions is called a *shell*. There are different kinds of shells varying slightly from each other. The most commonly known shells are

1. The original Unix shell `sh`.
2. The Bourne-Again-SHell `bash`.
3. The Korn shell `ksh`.
4. The C shell `csh`.

The Linux system allows you to choose which shell you want to work with. For our purposes, any of the above shells will work.

There are two ways to use shell commands : on the *command line* (from the *terminal*) and from a file. Linux commands are a very versatile set, which means they allow you to do almost anything you can think of but it might take a while to find out how to conduct a given task through arguments and options. This can only be improved through lots of practice.

The shell not only provides us with a very useful collection of commands, it also gives us the option of putting together a set of commands in executable files and creating, in a way, our own commands. This is useful if we find ourselves repeating the set of commands several times. Such files are called shell scripts and they usually carry the extension ".sh".

Shell scripts may be quite complicated but here we shall cover only two important aspects : shell variables and control structures.

1. *What's a* variable*?*

A *variable* in a computer program is in a way similar to a variable in maths. It's basically a symbol whose content may change during the course of the program. The difference between variables in mathematics and computing is that mathematical variables are an abstract concept whose values may never be explicitly assigned whereas in most computer codes (except those which perform analytical computations), each variable has a well-defined value at any given time. This is in contrast with a *constant* whose value is assigned to a number, string or character once and for all without undergoing any changes.

Let's see an example of a variable in `Octave`, which is one of the computer languages we will be using in this course.

```
octave:1> a=pi
a = 3.1416
octave:2> b=a^e
b = 22.459
octave:3> a=(sin(a/6))^2
a = 0.25000
```

Here `a` and `b` are variables. At the beginning of the `Octave` session, `a` is assigned to the prestored number `pi`. On the next line, the second variable `b` is calculated by raising the first variable `a` to another special constant `e`. Notice here that the computer remembers the value of the variable `a` when called while calculating the value of the variable `b` and replaces in its place the number `pi`. In the final line, we observe common usage of variables available in many computer languages which is the reassignment of a variable through self-manipulation. On the right-hand side of the equation, we see an operation involving the *old* value of the variable `a`, namely `pi`. After this operation is completed, the resulting value is reassigned to the same variable `a` whereafter the value of this variable is modified to 0.25. Variables are necessary in computer programming for the flexibility of our programmes. It would be silly to write a program that knows only add 2 and 4 but not 2 and 3, much like how absurd algebraic addition would be if it were only defined for 2 and 4 and not any other number.

2. *Shell variables*

In practice, variables in different computing languages are accessed in different ways. In some languages such as `C` and `Octave`, they are *called* simply by their name, whereas in others, you might need a special additional character to access the value assigned to a particular variable. The *shell* programming language is an example to those languages where values of variables are extracted using the special character, `$`.

```
hande@p439a:~$ a="Phys343 is a great class."
hande@p439a:~$ echo $a
Phys343 is a great class.
hande@p439a:~$ b=4.3
hande@p439a:~$ echo The number that you entered is $b.
The number that you entered is 4.3.
```

Notice the difference in the way we assign and extract variables in shell. In assigning a number, a string or a character to a shell variable, we just simply use the name of the variable with the `=` sign. In accessing it however, we need to use the `$` sign. Compare this to how variables are accessed directly without the need of a special character in `Octave` as in the example above.

3. Shell scripts

As mentioned above, it's possible to work in Linux by executing command after command from a terminal. You will see very soon, though, that this proves rather inefficient if you find yourself repeating the same set of commands over and over again. For example, if you wish to calculate the terminal speed of a biker in the presence of friction, this is difficult to do from the command line for two reasons :

1. There are two many steps and you cannot take back what you have written from the command line. You have to start all over again every time you've made a mistake.
2. If you want to repeat the same calculation for different sets of parameters, say for bikers with different power outputs, you would have to type all those lines of code on the command line again.

Instead, you can collect all those commands in a file and call *that* file from the command line with different arguments.
For example, suppose we want to assign a number two a variable, multiply this variable by two and three and assign each result to different variables and finally display all three variables. Compare the following two ways of doing this.

- On the command line

```
hande@p439a:~$ a=3
hande@p439a:~$ b=‘expr $a \* 2‘
hande@p439a:~$ c=‘expr $a \* 3‘
hande@p439a:~$ echo The three numbers are $a, $b and $c.
The three numbers are 3, 6 and 9.
```

- From a file (a shell script)

```
hande@p439a:~/$ cat multiply.sh
a=3
b=‘expr $a \* 2‘
c=‘expr $a \* 3‘
echo The three numbers are $a, $b and $c.
hande@p439a:~/$ sh multiply.sh
The three numbers are 3, 6 and 9.
```

Notice that shell scripts are generally run by using the interpreter `sh`. When you run the set of commands from the script, it becomes very easy to change the value of parameters and rerun. You only need to go into the file `multiply.sh` and change the line `a=3` to, say, `a=4`. In fact you can even do better by making only a minor modification to the code and supplying *command line arguments* to your customized shell script. These arguments are very much like arguments you would give to an ordinary shell command and this way you wouldn't even have to modify your file.

```
hande@p439a:~/$ cat multiply.sh
a=$1
b=‘expr $a \* 2‘
c=‘expr $a \* 3‘
echo The three numbers are $a, $b and $c.
hande@p439a:~/$ sh multiply.sh 3
The three numbers are 3, 6 and 9.
hande@p439a:~/$ sh multiply.sh 4
The three numbers are 4, 8 and 12.
```

In shell, the construct `$<n>` designated a special function, which accesses the value of the `n`th argument that you supply to your function when you call it from the command line. You can call your script with different arguments each time and you can use as many arguments as you wish although it is in general good practice to keep the command line arguments to a minimal set.

4. Control structures

As you have seen in the above example, the shell interpreter `sh` interprets commands in a shell script in a sequential manner, i.e. in the order as they appear in the file containing the shell script, in our case `multiply.sh`. Often, however, we need to follow a different order than the apparent one for a number of possible reasons. We might for example want to

- execute certain commands only if certain conditions are met
- skip some commands for the moment and come back to them later
- repeat a certain comment several times before moving on to the next one
- or terminate the programme at a certain point.

In any mainstream programming language, there exist special structures that enable this modification in the flow of the commands in the source file. These structures are called *control structures* and the operation of modification of the order of execution of commands is called *flow control*. The syntax and associated keywords of control structures vary among different programming languages. At this point, we will only concentrate on the shell syntax. The most commonly used control structures are

- The conditional `if` structure : `if...then...else...fi`

```
hande@p439a:~$ a=3
hande@p439a:~$ if [ $a = 3 ]
> then  echo Success
> else echo Failure
> fi
Success
```

- The conditional `case` structure : `case...<n>)...;;...esac`

```
hande@p439a:~$ a=2
hande@p439a:~$ case $a in
> 1) echo Morning;;
> 2) echo Afternoon;;
> 3) echo Evening;;
> *) echo "Not a valid choice";;
> esac
Afternoon
```

- The `for` loop structure : `for...  do...done`

```
hande@p439a:~$ for n in $(seq 1 3 10)
> do
> echo n=$n
> done
n=1
n=4
n=7
n=10
```

- The `while` loop structure :

```
hande@p439a:~$ a=1
hande@p439a:~$ while [ $a -lt 10 ]
> do a=`expr $a \* 2`;
> echo a=$a;
> done
a=2
```

```
a=4
a=8
a=16
```

[1] condensed from *Linux Magazine*, A Tour of the Linux System by Martin Streicher, Thursday, August 23rd, 2007
[2] You can find a nice comparison of the Linux filesystem to the Windows filesystem at `http://polishlinux.org/first-steps/filesystem-and-disks/`