# Lab I : Introductory Linux and Shell Scripts

**Exercise 0 : Download some files and practice a few commands.**
In order to complete this lab session, you will need to download some files from web site of our course. Open a terminal and type the following :

```
hande@p439a:~$ wget www.physics.metu.edu.tr/~hande/teaching/343-labs/lab-01/files/files.tar.gz
```

`wget` is a fast way to download files from the Web directly to the directory you are in as long as you have a connection. Now pay attention to the extension of the files, `.tar.gz`. This file really has two extension. The first, `.tar` means that this files is in fact a collection of files that you combine to form one file; in computing jargon such a file is called a *tarball*. This is convenient in cases when you find yourself having to send or receive multiple files, just like in our case. The next extension is `.gz`, which means that this file is *zipped* or compressed to save space. In order to get an idea of the size reduction as a result of this operation, see the size of the file by typing

```
hande@p439a:~$ ls -lh files.tar.gz
```

As you see the size is really small. Now let us *unzip* and *untar* this file :

```
hande@p439a:~$ gunzip files.tar.gz
```

If you list the files in your directory you will see that you have a file with the name `files.tar`. Now, let us see once again the size of this new file.

```
hande@p439a:~$ ls -lh files.tar
```

As you see, the file is now several times larger. The files that are in this tarball are text files and therefore compress really well. Other file types such as binary do not compress very well since they are already in a somewhat compressed format. Now let us exract our files from the tarball. To do this, type

```
hande@p439a:~$ hande@p439a:~$ tar xvf files.tar
columns.txt
einstein1.txt
einstein2.txt
sun1.txt
sun2.txt
sun3.txt
sun4.txt
```

The options `xvf` stand for extract, verbose and file respectively. Now you have extracted all the files necessary to complete this lab exercise. The opposite of the actions described above, namely compressing and unzipping can be achieved using the commands `gzip` and `tar cvf` (`c` stands for create). Please refer to the `man` pages for details.

**Exercise 1 : Learn to work with an editor.**

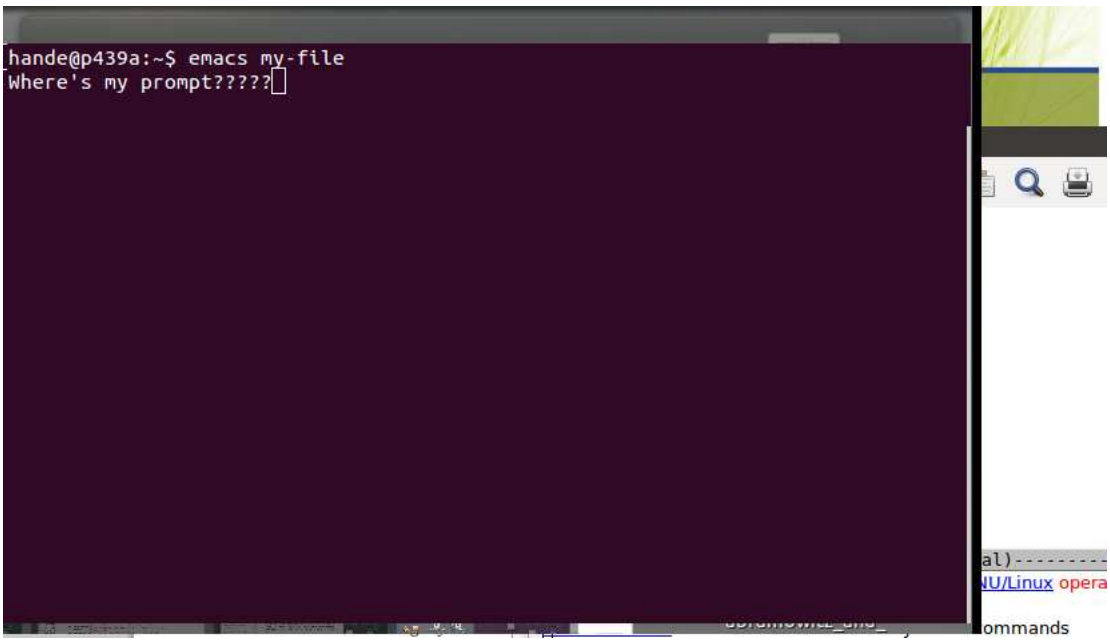
The first thing that a programmer should do is to choose a good *editor* and get to know its functionalities. An *editor* is an interface between you and the computer that allows you to create and edit files. A valid analogy in Windows would be NotePad. Conversely, Windows Word is not a good analogy since the document that you create in Windows is converted into machine language and is not human-readable as text. Editors, in contrast, allow you to read and write text files directly. While you can mix text and figures in a Word document, editors only accept text. To create a Word-like document such as a report, you need to use word-processing software such as LaTex.

There is a plethora of editors that are available for Linux. In this lecture, we will concentrate on `emacs` but feel free to explore other possibilities and find the one that is right for you. Some choices are `vim` (or `vi`), `nano`, `pico` and `nedit`. Let us practice some basic `emacs` with a step-by-step tutorial. **Important : It will take some time to get the hang of using the very extensive functionality of `emacs` but once you have, it will make your professional life a whole lot easier.**

1. Open up a terminal, and invoke `emacs`.

```
hande@p439a:~$ emacs my-file.txt
```

Now come back to your terminal and try to type something. Your terminal probably looks something like this when you try that :



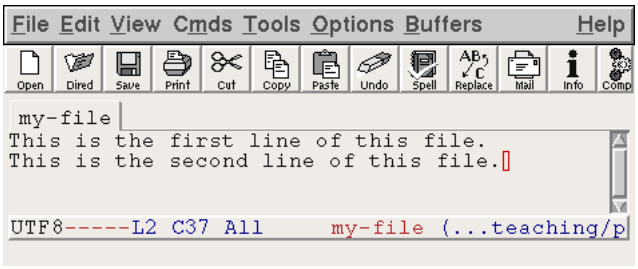

It seems like you have lost your prompt and your terminal is totally inactive. If you would like to go back and forth between your `emacs` window and your terminal and use both simultaneously, you cannot do that. You can, of course, circumvent this problem by opening a second terminal and leave the inactive terminal hanging around on your desktop but that is hardly a very elegant solution. A better way is to invoke the `emacs` window in the *background*, so that it doesn't interfere with your terminal. Now go to your inactive terminal and press `Ctrl-c` (that is hold down the `Ctrl` and while holding it press `c`). This liberates your terminal, *killing* your `emacs` window.


Now re-invoke `emacs` with the following comment.

```
hande@p439a:~$ emacs my-file &
The & sign sends a job to the background, with an external window popping up.
[1] 4771                    This is the process ID assigned to emacs.
hande@p439a:~$ jobs
[1]+  Running                 emacs my-file.txt &
```

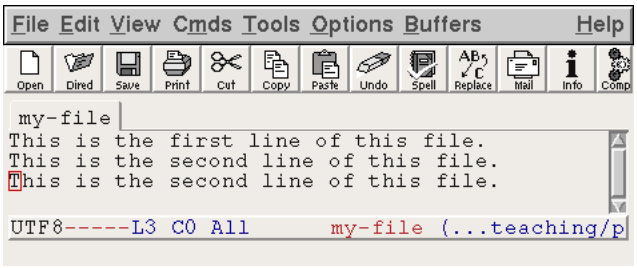
The command `jobs` allows you to list all the *processes* that you have running in the background. We will revisit this command some time later.

2. In the `emacs` window that opens, type in the text normally as if you were typing in a Word document.

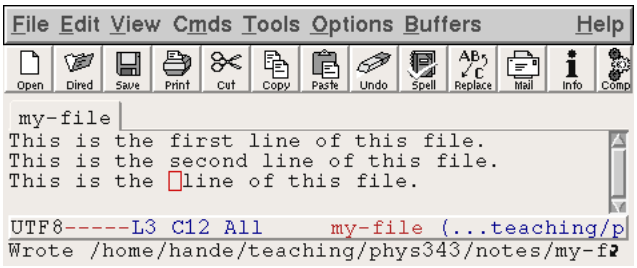



3. It's good practice to save your work often while using an editor, so click on the `File` icon on the toolbar on top of screen. Editors, in particular `emacs` usually allows tasks to be performed through *keyboard shortcuts*, which are key combinations that speed up typing. Saving your work in `emacs` can be done by the key combination `Ctrl-x Ctrl-s`. This sequence is to be interpreted in the following way : Hold down the `Ctrl` key and press first `x` and then `s`, **without** releasing the `Ctrl` key.

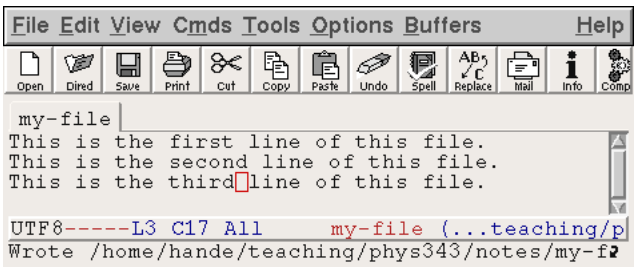
4. Next, we are going to create a third line that is almost identical to the other second line, except we'll replace the word "second" with the word " third". For this go to the beginning of the second line and type `Ctrl-k`. This will *cut* the text and store it in a sort of a clipboard. Then type `Ctrl-y` once to undo the cutting of the second line, press enter to go on to the next line and then type `Ctrl-y` again to form the third line. Save your work.




Now, you are at the beginning of the third line. Go to the end of the word "second" and type `Alt-Backspace`. This action erases entire words instead of erasing them letter-by-letter.




You can now finally type the word "third".

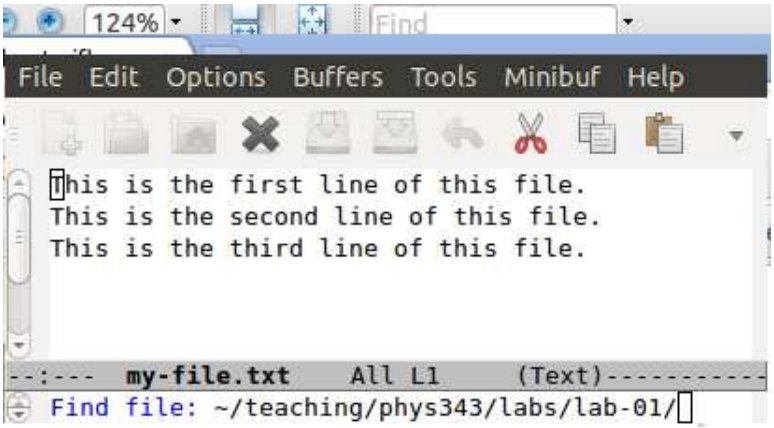


Make sure to save again! As long as you don't save your work, it won't actually get written to disk and will be lost when you close your `emacs` session.

You can alternatively do all this by using the various menus and icons on the top of the screen. However, learning keyboard shortcuts using `emacs` allows you much flexibility and speed. A complete list of keyboard shortcuts is basically impossible but a concise collection can be found at various pages on the web. Two examples :

- `http://www.cs.duke.edu/courses/fall01/cps100/emacs.html`
- `http://naveen.wikidot.com/emacs-commonly-used-shortcuts`

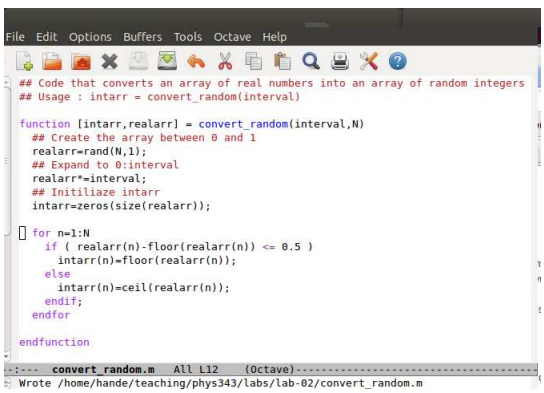
5. Suppose you are done with the first file my-file.txt and would like to open a second file. One very rudimentary approach would be to close your present `emacs` window and open another one, but as you can guess there are better ways of doing this. You can actually continue to use your current, active `emacs` window to navigate between files. We are now going to open one of the text files we have downloaded in the previous exercise, say `einstein1.txt`. To do this, use the keyboard shortcut `Cntr-f`. This will prompt you to enter the name of the file you wish to open in the small bar underneath the main text window, like so :



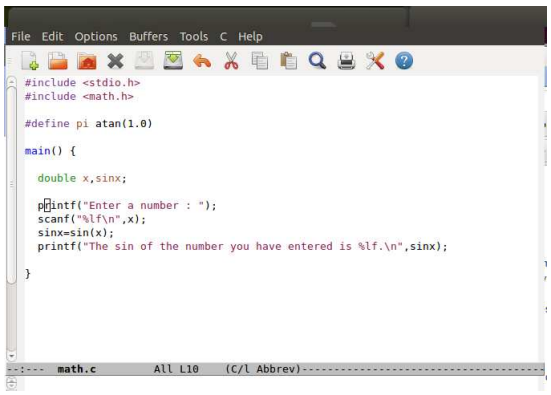

Now your cursor is in the small bar and anything you type appears there until you press `Enter`. Type the name of the file you would like to see here and press `Enter`. NOTE that Tab-completion can be used in the small bar as well. If you make a mistake or if you change your mind about opening the file, you can always use the shortcut `Ctrl-g` to cancel your operation. In fact, this key combination can be used for most errors.

6. Finally let us close the `emacs` window. Simply use the shortcut `ctrl-x ctrl-c` and watch the window disappear.

7. OPTIONAL : A big advantage of using `emacs` is that, if configured properly, it recognizes the particular language used and colors the special words of the language appropriately. In addition, it also does the *indentation* properly, which is a great help when coding. As an example, consider the following two code snippets written in `Octave` and `C`.




Octave mode




C mode

If you are interested in making use of this feature of `emacs` in addition to many others, you can download the `emacs` configuration file from the course Web site. In order for it to work correctly, you need to save it directly under you home directory with the name `.emacs`. Next time you call up an `emacs` window, you will see the changes.

Here are some similar notes on other editors. Please go through these as well by yourself and decide which one is better for you. In the next lecture, we will take a poll to see how many people decided on which editor.

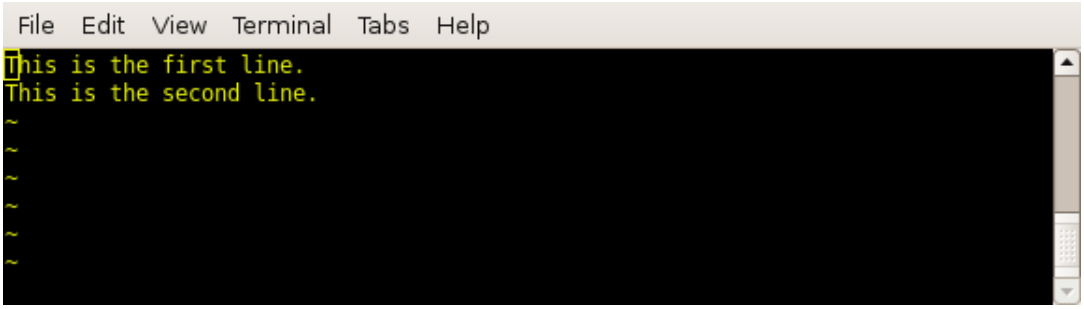
Vi : (pronounced vee-i)

1. Invoke the editor `vi`.

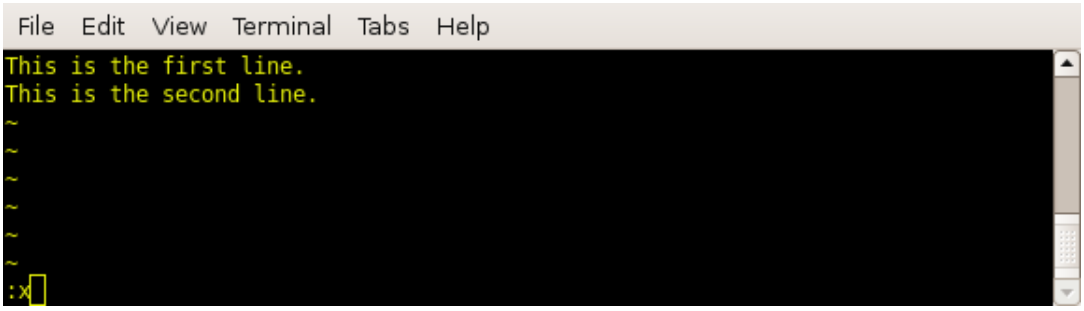
```
hande@p439a:~$ vi my-file.txt
```

   There's no need to send `vi` to the background because unlike `emacs`, it opens the editor in the *terminal* and not in an external window. As a result of this there is no option to use the process and the originating terminal simultaneously.

2. `vi` has two modes : the *insert* mode and the *command* mode. The insert mode lets you type while the command mode is for actions such as removing line, moving the cursor around and yanking text. In order to type into the file, type `i`. This will take you into the insert mode. You can then type text normally.




3. After you are done typing, you need to save your work before, exiting. For this, type `Esc`. This is allow you to exit the insert mode and go into the command mode. Now, type `:`, you will see it appear at the bottom of the screen. If you would like to save and quit the file, type `x`. If you would like to save and continue working, type `w`. If you, instead, want to quit without saving changes, type `q!`. The `!` forces quit without saving.




4. You can read more about `vi` at `http://www.cs.colostate.edu/helpdocs/vi.html`.

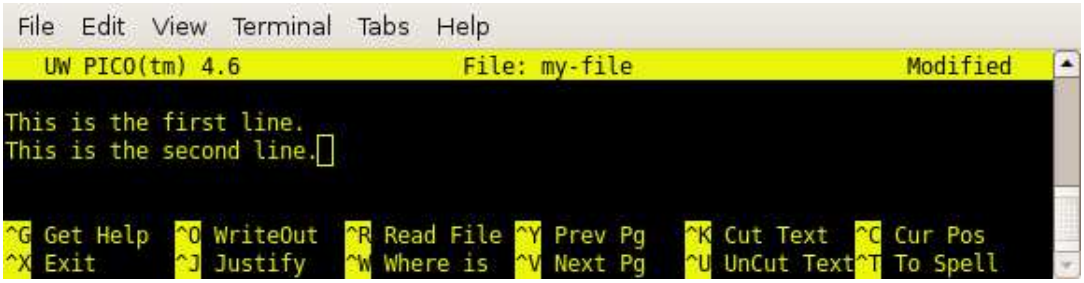
Pico :

1. Invoke the editor `pico`.

```
hande@p439a:~$ pico my-file.txt
```

   Once again, you do not need to send the editor to the background.

2. `Pico` doesn't have modes so you can type right away.




3. Some keyboard shortcuts in `pico` are the same as those in `emacs`. You can for instance use `Ctrl-k` to cut a line. Yanking a line, however, has a different binding, namely `Ctrl-u`.

4. Once you are done writing, you can either type `Ctrl-o` and save without exiting or `Ctrl-x` to exit, which will prompt you to save the file.

5. The good thing about `pico` is that it lists relevant actions at the bottom of the screen although I find its capabilities rather limited in comparison to `emacs`.

**Exercise 2 : Learn to manipulate and list files and directories : mv, cp, mkdir, touch, ls**

The first exercise of this lab period is also going to be a lesson in good computing habits. The best way to confuse yourself about your work is by keeping your files all in one place without making use of directories. You should divide your files into meaningfully-named directories and possibly further into sub-directories. So let's start by making a directory to contain the work you'll be doing for this class and sub-directories to separate your work between classes.

```
hande@p439a:~$ mkdir phys343
hande@p439a:~$ cd phys343
hande@p439a:~$ mkdir lab-01
hande@p439a:~$ cd lab-01
hande@p439a:~$ pwd
/home/hande/phys343/lab-01
```

The command `mkdir` creates a new directory with the name you supply to it. Then you can go into the newly created directory using the command `cd`, which stands for *change directory* and does precisely that. You can equally well create the directory `phys343` and the sub-directory `lab-01` at the same time.

```
hande@p439a:~$ mkdir -p phys343/lab-01
```

You do, however, have to use the `-p` option to force the system to simultaneously create the parents of the deepest sub-directory `lab-01`. You can then make sure you are in the right place by using the `pwd` command. `pwd` gives you the full *path* or address of the current directory.

Now, we can create the files that are necessary for this lab. In most Linux systems, there are several ways to create a *file*.

1. Using the `touch` command :

   ```
   hande@p439a:~$ ls my-file.txt
   ls: my-file.txt: No such file or directory
   hande@p439a:~$ touch my-file.txt
   hande@p439a:~$ ls -l my-file.txt
   -rw-r--r-- 1 hande users 0 2007-10-02 21:22 my-file.txt
   hande@p439a:~$ touch my-file.txt
   hande@p439a:~$ ls -l my-file.txt
   -rw-r--r-- 1 hande users 0 2007-10-02 21:23 my-file.txt
   ```

   Here the command `touch` creates an empty file if the file does not already exist and it only changes its modification date if it does. The command `ls` lists all the files and directories that are under the directory you are in. The option `-l` displays different aspects of the files being listed. (See below)

2. Using the `cat` command :

```
hande@p439a:~$ ls my-file.txt
ls: my-file.txt: No such file or directory
hande@p439a:~$ cat > my-file.txt
The > sign directs the following into the file.
This is the first line.
This is the second line.          Terminate with Ctrl-d.
  ^Ctrl-d
hande@p439a:~$ cat my-file.txt       Without the > sign, it only prints contents of file.
This is the first line.
This is the second line.
hande@p439a:~$ cp my-file.txt my-file2.txt
hande@p439a:~$ cat my-file.txt my-file2.txt
This is the first line.
This is the second line.
This is the first line.
This is the second line.
hande@p439a:~$ cat my-file.txt my-file2.txt > my-file3.txt  > sign redirects into a file
hande@p439a:~$ cat my-file3.txt
This is the first line.
This is the second line.
This is the first line.
This is the second line.
```

`cat` is a command that assumes different functions in different situations. It can be used

1. to create files,
2. to con`cat`anate files, meaning to add two or more files consecutively,
3. or to just simply display a file.

**Exercise 3 : Comparing files : diff**

Suppose you have two files that are almost the same but you know that there are a few differences between them. You would like to know these differences and where in the file they occur. For this exercise, we will work on the files einstein1.txt and einstein2.txt we have downloaded from the Web site earlier. They both contain similar text with a few different words in each paragraph. First let's just see a brief account which says whether the two files differ.

```
hande@p439a:~$ diff -q einstein1.txt einstein2.txt
Files einstein1.txt and einstein2.txt differ
```

Now let's see the lines where they differ. For this we'll simply invoke the `diff` command without any options.

```
hande@p439a:~$ diff einstein1.txt einstein2.txt
9,10c9,10
<     Jack Rosenberg remembers the time Einstein's coworker asked him to
<     turn the tables and help give the famous scientist a present.
---
>     Jack Rosenberg remembers the time Einstein's colleagues asked him to
>     turn the tables and help give the famous scientist a gift.
.....
```

You'll see that the differences are given line by line together with line numbers.

Sometimes, it is useful to see the differences in context, meaning displaying the differences with some lines around it. This is especially valuable if your text is sparse and it's hard to understand the location of the differences without seeing what's around it. This function can be invoked using the `-U` option. You can decide how many lines you want to be displayed around the lines with differences.

```
hande@p439a:~$ diff -U 1 einstein1.txt einstein2.txt
--- einstein1.txt       2007-09-19 16:47:39.000000000 +0300
+++ einstein2.txt       2007-09-19 16:48:34.000000000 +0300
@@ -8,4 +8,4 @@
     Albert Einstein was well known in Princeton for his generosity. But
-    Jack Rosenberg remembers the time Einstein's colleagues asked him to
-    turn the tables and help give the famous scientist a gift.
+    Jack Rosenberg remembers the time Einstein's coworkers asked him to
+    turn the tables and help give the famous scientist a present.

@@ -22,3 +22,3 @@
     Einstein's most important findings, the theory of special relativity,
-    plans are now under way to remember the findings of the man who
+    plans are now under way to remember the discoveries of the man who
     revolutionized physics in 1905 by redefining scientists' perception of
```

**Example 4 : Extract information from files : awk, head and tail; the pipe.**

Suppose we have a file that contains several columns. We'd like to display the first three lines of items that are on the fifth column of this file. For this exercise, we will use the file columns.txt that we have downloaded earlier.

First let's learn how to display the first $N$ lines of a file. You can do this using the command `head`. The default number of lines displayed with `head` is 10, however, you can override this by specifying the number of lines you would like to be displayed.

```
hande@p439a:~$  head -3 columns.txt
!    total energy              = -1090.13343774 Ry
!    total energy              = -1090.20757070 Ry
!    total energy              = -1090.24296462 Ry
```

This is almost what we want except we would like only the fifth column (i.e. the numbers column) to be displayed. We do this using a command called `awk`. `awk` is really a pattern searching language in itself, which is very helpful for certain tasks. `awk` is ordinarily used to extract columns from files in the following way :

```
hande@p439a:~$ awk '{print $5}' columns.txt
-1090.13343774
-1090.20757070
-1090.24296462
-1090.25563488
-1090.27085564
-1090.27693129
-1090.28213580
-1090.29131927
```

In awk and in a lot of shell scripting, the direction of the quotation marks is very important. Whatever is inside the $\acute{\ }\{\ \}\acute{\ }$ is interpreted as the instruction given to `awk`, which in our case is to `print` the fifth column of the file, designated by `$5`.

However, this isn't what we wanted to do. Instead of displaying the fifth column of the entire file, we are only interested in displaying the fifth column of the first three lines. A very common construct in Linux when we want to process the outcome of a command using a second command is the *pipe*, which is the vertical line `|`. Instead of having `awk` read from a file, we can *pipe* the output of `head` to `awk` directly without having to save it to a file first.

```
hande@p439a:~$ head -3 columns.txt | awk '{print $5}'
-1090.13343774
-1090.20757070
-1090.24296462
```

Note that the first part of the pipe, namely the one starting with `head` is a full command with the file name, whereas the second part does not have the file name as the argument anymore.

To add a final complication, let's say that we are interested in displaying only the fifth column of the second line of this command. We can do this by adding another pipe with the command `tail`. The usage of `tail` is very much like that of `head` except that it displays the last $N$ lines of text.

```
hande@p439a:~$ head -2 columns.txt | awk '{print $5}' | tail -1
-1090.20757070
```

The action `head -2` displays the first two lines, `awk` then selects the fifth column of the output and finally `tail -1` displays the last line of the second output. You can form a chain of pipes of arbitrary length in this way.

**Exercise 5 : Extracting information from files : ls, less, cat, grep**

In this exercise we will use the four files sun1.txt, sun2.txt, sun3.txt and sun4.txt we have downloaded earlier. First make sure that you have all the files and that they are not empty by using the `ls` command.

```
hande@p439a:~$ ls -l   -l option lists information
                       about content, permissions,
                       size, owner etc.
total 136
permissions  links   owner   group   size  modification  time  name
                                           date
drwxr-xr-x     2     hande   users    176   2007-09-18   22:03 figs
-rw-r--r--     1     hande   users    113   2007-09-18   22:03 my-file.txt
-rw-r--r--     1     hande   users  69323   2007-09-18   22:31 notes.pdf
-rw-r--r--     1     hande   users   5982   2007-09-18   22:38 notes.tex
-rw-r--r--     1     hande   users    436   2007-09-18   22:31 notes.toc
-rw-r--r--     1     hande   users   2640   2007-09-18   22:46 sun1.txt
```

Perhaps you are not interested in seeing all the files in your directory but those which start with the word "sun".

```
hande@p439a:~$ ls -l sun*
-rw-r--r-- 1 hande users  563 2007-09-18 22:46 sun1.txt
-rw-r--r-- 1 hande users 1462 2007-09-18 22:46 sun2.txt
-rw-r--r-- 1 hande users 1992 2007-09-18 22:46 sun3.txt
-rw-r--r-- 1 hande users 2640 2007-09-18 22:46 sun4.txt
```

Or equally, you want to view just those files which have a `.txt` extension.

```
hande@p439a:~$ ls -l *.txt
-rw-r--r-- 1 hande users 3048 2007-09-18 22:34 ideas.txt
-rw-r--r-- 1 hande users  563 2007-09-18 22:46 sun1.txt
-rw-r--r-- 1 hande users 1462 2007-09-18 22:46 sun2.txt
-rw-r--r-- 1 hande users 1992 2007-09-18 22:46 sun3.txt
-rw-r--r-- 1 hande users 2640 2007-09-18 22:46 sun4.txt
```

The asterisk(*) is called a *wild-card* and it is used to list files that start with, end in or contain a given pattern.

Now that we are convinced that the files exist and they are not empty, let's take a look at one of them. When you try to use the command `cat` like we did before, you will see that the file is too long to fit into a single screen. What would be nice is to be able to control the portion of the file that is shown on screen. This can be done with the command `less`.

```
hande@p439a:~$ cat sun3.txt
......      Runs off the screen!
of the Moon is an awesome experience. For a few precious minutes it gets
dark in the middle of the day. The stars come out. The animals and birds
think it's time to sleep. And you can see the solar corona. It is well
worth a major journey.
```

```
hande@p439a:~$ less sun3.txt
......
total eclipse of the Sun. Partial eclipses are visible over a wide area of
the Earth but the region from which a total eclipse is visible, called the
path of totality, is very narrow, just a few kilometers (though it is
sun3.txt lines 1-23/32 69%  Scroll using up and down arrows.
```

Next, let's count the number of lines, words and bytes in the file sun2.txt. This can be done using the `wc` command, which stands for *word count*.

```
hande@p439a:~$ wc sun2.txt
  96 1086 5976 sun2.txt
```

Doing research, we often find ourselves looking for a particular word or a pattern in a given file. This could be, for example, `energy` or `result`. While this search can be done by visual inspection if the file is small, for larger files, this would be impossible. Linux has a very powerful command, `grep`, for conducting pattern search. It takes as arguments the pattern being searched and a file name. If called without any options, it prints lines containing the pattern on the screen. Suppose we want to know the number of times the word "sun" occurs in the file sun2.txt.

```
hande@p439a:~$ grep Sun sun2.txt
The surface of the Sun, called the photosphere, is at a temperature of
about 5800 K. Sunspots are "cool" regions, only 3800 K (they look dark only
.....
```

Calling `grep` with the option `-n` causes the number of the line to be displayed where the given pattern occurs.

```
hande@p439a:~$ grep -n Sun sun2.txt
1:The surface of the Sun, called the photosphere, is at a temperature of
2:about 5800 K. Sunspots are "cool" regions, only 3800 K (they look dark only
.....
```

If we are only interested in the number of occurrences of the pattern, we can use the `-c` option.

```
hande@p439a:~$ grep -c Sun sun2.txt
33
```

You might have noticed that `grep` not only finds the instances of the word "sun" but also words which "sun" is a part of, such as "Sunspot" (Notice also that `grep` is not case-sensitive, but that can be modified through options). Suppose now that we would like to count all occurrences of the word "Sun" but not Sunspot. This can again be achieved by piping the output of the `grep` from above to a second `grep`, this time using the `-v` option, which matches *nonoccurences* of the given pattern. Because we are only interested in the number, we can do a further pipe to `wc`, as before.

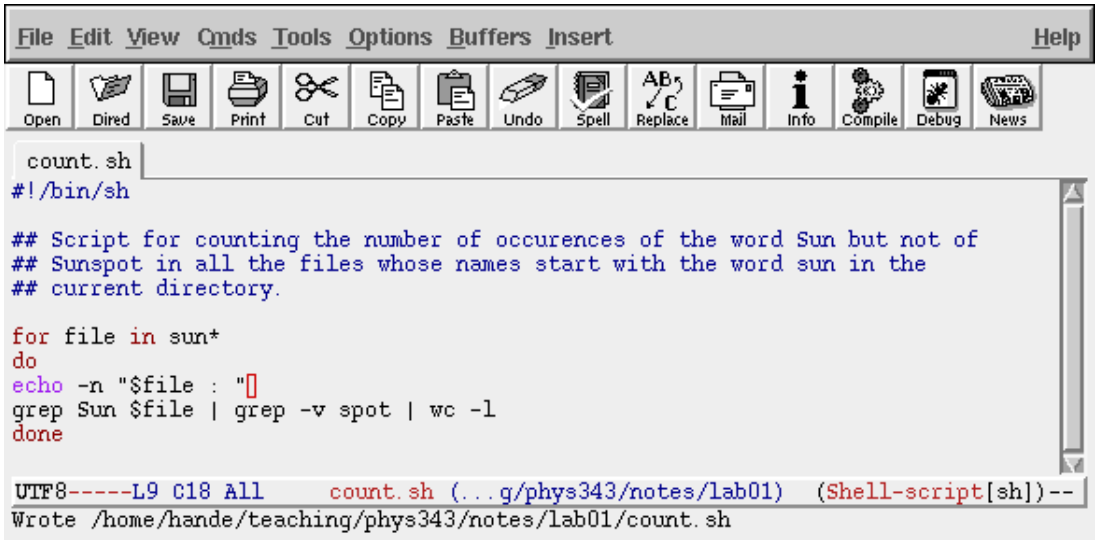
```
hande@p439a:~$ grep Sun sun2.txt | grep -v spot | wc -l
     24
```

**Exercise 6 : Learn to write Shell scipts**.

As mentioned previously, the shell not only provides us with a very useful and versatile set of commands, it also gives us the option of collecting a set of commands in executable files and creating, in a way, our on 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 : the control structures and shell variables.

Let's go back to our earlier example of the sun.txt files and let's say that we would now like to do the operation of counting the occurrences of the word "Sun" in all the files that start with "sun" and also separate the occurrences of Sunspot from them. In addition, we would also like to know how many occurrences we get for which file. Foreseeing that we might need to do this several times, it's better to write the shell instructions in a file. Let's go to `emacs` again.

```
hande@p439a:~$ emacs count.sh &
```

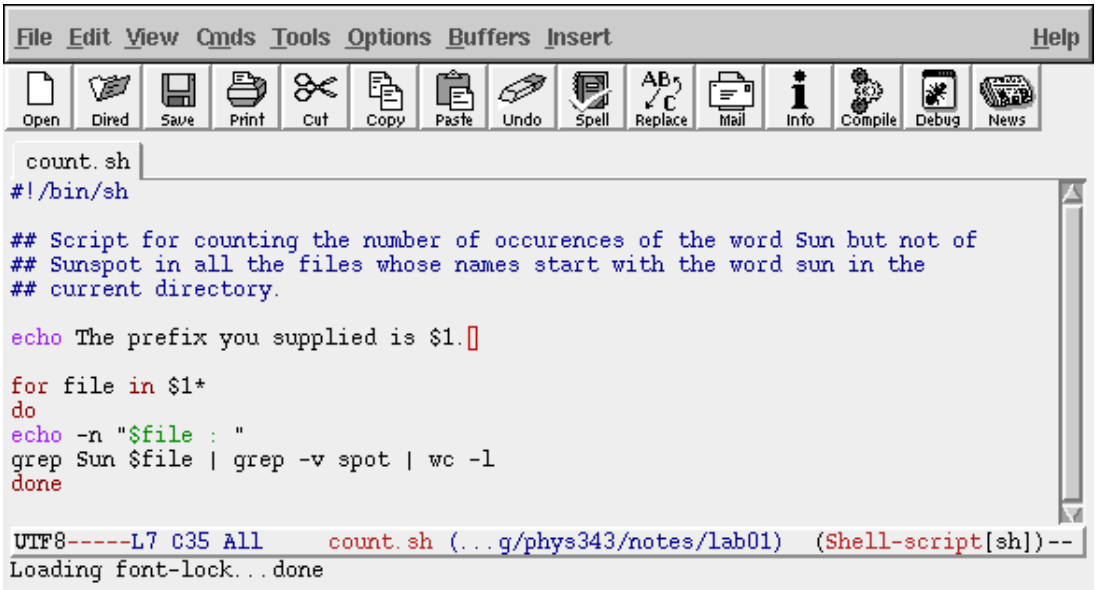



Shell scripts are run from the command line (or terminal) using the command sh.

```
hande@p439a:~$ sh count.sh
sun1.txt : 5
sun2.txt : 24
sun3.txt : 8
sun4.txt : 8
```

So we see that there are 5, 24, 8 and 8 occurrences respectively of Sun but not Sunspot in these files. In the simple shell script above, we recognize the line `grep...` from the earlier example. The only difference is that we know define a *shell variable*, called `file` and let it run over all the files starting with the word `sun` using the `for, do, done` construct. This, as mentioned before, is made possible by the asterisk, *. In shell, once a variable is assigned, it can be recalled using the dollar sign, $. The `for` loop assigns one by one all the files that start with the word `sun` to the variable `file` and between the keywords `do` and `done`, we specify what we'd like shell to do with these files. For each `$file`, we first display the name of the file using the `echo` command. The `-n` option suppresses the newline. The next line is already familiar from the previous example and we finally close our loop with the keyword `done`.

In fact, we can make our shell script even more similar to a real shell command by supplying a command line argument. Let's say that we would like to have the flexibility to tell the shell script the prefix of the file, in this case "sun". We can modify the script in the following way.




```
hande@p439a:~$ sh count.sh sun
The prefix you have supplied is sun.
sun1.txt : 5
sun2.txt : 24
sun3.txt : 8
sun4.txt : 8
```

Here `$1` refers to the first command line argument supplied to the shell script. Clearly, you can have any number you like. Once supplied, command line arguments are treated in exactly the same way as regular variables, using the $ sign whenever they need to be expanded.

IMPORTANT : A good programmer always writes sufficient *comments* in their programs. A *comment* is a line or lines that start with a special character (# for shell scripts) which are not interpreted by the shell but which provide explanation of the usually cryptic program. This is as much for the readability of the code by other people as by the programmer because it's guaranteed that to months later he will have forgotten most of what he's written.