Introduction to Terminal, Bash Scripting, and High-performance Computing

Vikram Manikantan[1]
[1] *Department of Astronomy, University of Arizona, Tucson*

Parts of this tutorial were adapted from Vasilis Paschalidis and Erik Wessel (2024).

Objectives

1. Use Terminal to navigate the file directory.
2. Create, edit, and execute Bash scripts.
3. Code an IF statement and a FOR loop in Bash.
4. Understand why we use high-performance computing (HPC).
5. And how to submit a basic job to the SLURM workload manager.

0. LOGGING ONTO A REMOTE COMPUTER

If you have a Windows computer, you will want to follow these instructions to log onto UA's High-performance Computing (HPC). You can also optionally do this if you have a Mac/Linux (you will have to do it later anyways):

1. Open Windows Powershell (MacOS Terminal)
2. Type: `ssh <your-net-id>@hpc.arizona.edu`.
3. If it asks you if you 'trust this computer', type yes. Enter your netid password.
4. It will ask you for some kind of DUO two-factor authentication. You should now be logged onto HPC.
5. Type `shell` and hit enter. Then, type `ocelote` and hit enter.
6. In your browser, open the web user interface for the HPC: Open OnDemand. Navigate to the `Files`.

Why use HPC/Linux? Windows has a different Terminal language to Unix based operating systems, such as MacOS and Linux. Most intensive computing is done on Linux computers. Therefore, the skills you learn on Microsoft's Powershell will most likely not be transferable to the majority of computing across academia and industry.

MacOS's Terminal is based on Unix, which makes it great for academic work. Linux is another operating system which is also widely used for academic computing work, AND it is open-source and free to install on any laptop.

I. THE LINUX TERMINAL

What is a Terminal? (Authors: Erik Wessel and Vasilis Paschalidis)

Terminals are a completely text-based method of interacting with a computer. Despite the fact that they are one of the oldest ways to use a computer, terminals remain with us today due in part to their extreme simplicity, and in part to the impressive capability of shell interfaces.

A shell is a program that interprets user input in real time. Shells provide a computer language that they interpret in real time, allowing you to describe what you want the computer to do. Essentially, they let you talk to a computer in a way that is easy to learn but also flexible enough to let you do almost anything. In fact, it is very hard to develop a graphical interface that allows you to do everything that can be done with a shell.

Here's a culinary analogy: if you want to get dinner, you can go get food from a resturaunt, or cook food yourself. If you eat out, you save yourself the trouble of learning how to cook, but you are limited by the menu options of the

resturaunts in your area. This is frustrating: even if the resturaunts around you have the ingredients and equipment to make some dish, if it doesn't appear on their menu, it isn't something you can order. However, if you put in the effort to learn to cook for yourself, then as long as you have the ingredients, you can put them together in the way you see fit. Similarly, while graphical user interfaces may have an easier learning curve, they are fundamentally limited: by learning a shell language and talking directly to your computer with it, you can transcend those limitations.

Basic Commands

Here are some commonly used Linux terminal commands:

- `pwd` - Print the current working directory.
- `ls` - List files in the current directory.
- `cd` - Change directory.
- `mkdir` - Create a new directory.
- `rm` - Remove files or directories.
- `touch` - Create an empty file.
- `echo` - Print the following text, or text contained in a variable

Exercise 1: Use the following commands:

1. Open a terminal and run (type and hit enter) `pwd`. What is the current directory?
2. Use `ls` to list the files in your current directory.
3. Create a new directory called timestep: `mkdir timestep`.
4. Run `ls` to make sure your new directory is there.
5. Change into your new directory by typing: `cd timestep`. Run `ls` and `pwd` again.
6. Finally, try the command `echo 'Hello, world!'`.

II. MAKE MORE DIRECTORIES

Throughout this tutorial you can use the HPC web interface to look at your files. If you make a file/folder through the terminal, you can hit refresh on the online portal and see your changes in real-time. For many remote computers, this will not be an option.

Now, lets make a few test directories.

```
mkdir dir1
```

You can create several at the same time by typing a sequence of directory names after `mkdir`

```
mkdir dir2 dir3
```

In addition, we can create a whole sequence of subdirectories all at once by using the -p flag

```
mkdir -p dir4/subdir/subsubdir
```

The -p flag will create not only the directory subsubdir, but also the parent directories subdir and dir3.
Go ahead and use `ls` to see what is in your current directory, you should see all 4 subdirectories.
To delete an empty directory you can type

```
rmdir dir1
```

"rm" for remove, or for multiple directories at once

```
rmdir dir2 dir3
```

Now try to remove `dir4` the same ways

```
rmdir dir4
```

Uh-oh! What happened? If you followed the earlier steps when creating `dir4` you should see an error message

```
rmdir: failed to remove 'dir4/': Directory not empty
```

This is because the `rmdir` command has a very specific function: it is only for removing empty directories. If you attempt to use it to remove anything that isn't an empty directory, it will print out an error message and refuse to do anything.

You can then use

```
rm -r dir4
```

to remove directories and all their content. This is **very** dangerous if used incorrectly. Be very careful.

Exercise 1.5:

1. Remake all those directories we just deleted.

III. EDITING FILES

Directories are great. But we want to edit files and run them. There are many code editors you can use from within the terminal, here are a few: `vim, emacs, nano, gedit`. We are going to use `vim`.

First, make sure you are in your directory by running `pwd`. If you are in one of your subdirectories, run:

```
cd ..
```

`..` refers to the directory 'above' the current one.

Then run

```
touch notes.txt
```

This creates a file with that name. Type `ls` to see that the file exists. You can then enter to your code editor to edit the file:

```
vi notes.txt
```

If you try typing, it will not show up (at least not immediately). To start editing you have to hit `i`. Now you are in insert mode. You can now type whatever you like. I am going to type:

Listing 1: Text File Example

```
Hello, World!
```

To exit insert mode, simply hit `esc`. Now, you want to leave the editor and go back to the terminal. You first have to press `shift + :` then `w`. This will show up in the bottom left of your screen. Then hit `enter`. This will save your work. Now do `shift + :` then `q`, and then enter, to exit `vim`.

Congrats! You have now edited and saved your first file (at least, as far as I know).

If you want to quickly peak at the contents of your file, you can try

```
cat notes.txt
```

`cat` is a command that stands for concatenate. It is a good way to peak at what a file contains. If it is a long file and you would actually just like to look at the start or end, you can try

```
tail -n 100 notes.txt
head -n 100 notes.txt
```

where `-n` is a flag that specifies how many lines to look at. In this case, we set `-n 100` to 100 lines. `tail` and `head` look at the final and starting lines in a file, respectively. In this case, they will print the same thing.

Moving Files

You can move files (and folders) around with this command:

```
mv notes.txt dir1/
```

and

```
mv dir1/ dir2/
```

You might have to make some new folders.

Copying Files

And, I can copy files with:

```
cp dir2/dir1/notes.txt .
```

where the single period is the current directory. I can replace that with the path to another directory:

```
cp dir2/dir1/notes.txt dir4/
```

IV. DOWNLOADING AND UNZIPPING FILES

At some point, you will need data or code from somewhere else on the internet. This can come in the form of a GitHub repository (more on that in two weeks) or simply a zipped folder on the internet.

We are going to practice with a file I have put on my website. Copy the line below into your terminal:

```
curl -O https://www.vikrammanikantan.com/timestep.tar.gz
```

`curl` can be used to download anything on the internet (pretty much). Make sure this file is now in your working directory. This is a zipped folder, also known as a `tar` folder. You can unzip this folder in your terminal by running:

```
tar -xvf timestep.tar.gz
```

But WAIT. You might unzip a folder that already has the same name as another folder in your current directory. This would completely replace the existing folder (no undos in the terminal). Is there another folder with the same name? If there is, `cd` into a different directory and download and unzip the folder there.

Exercise 2:

1. Explore the directories in the new folder you have downloaded.
2. Zip up the folder `dir1` into a file called `dir.tar.gz` with the command:

```
tar -cvf <zipped-file-name> <directory-to-zip>
```

V. BASH SCRIPTING

Exercises in this section will require you to Google things for yourself. Have at it.

What is Bash?

`BASH` is a Unix shell and command language. Just like `Python` is a language, so is `BASH`. The commands you have been using so far, `mkdir, pwd, ls, ...`, are `BASH` commands.

A script is a series of commands written in a file and executed as a program. You could take the commands you have been using and just put them all into a script to execute all at once.

Writing a Simple Script

Create a script file using `touch hello.sh`. Then, open that file with `vi hello.sh`:

Listing 2: Bash Script Example

```
#!/bin/bash
# This is a comment
echo "Hello, World!"
```

Save the file as `hello.sh`, then make it executable and run it:

```
$ chmod u+x hello.sh
$ ./hello.sh
```

Do not forget the `chmod` command for future files you create. This changes the permission of the file so that you can run it in the command line.

Exercise 3:

1. Why do you need the line `#!/bin/bash`?
2. Create a file `myscript.sh`.
3. Write a script that prints your name and the current date.
4. Make the script executable and run it.

Variables in Bash

You can use variables in your scripts to store information.

Listing 3: Using Variables in Bash

```
#!/bin/bash
NAME="John"
echo "Hello, $NAME"
```

Exercise 4:

1. Modify your script to store your name in a variable and print it.
2. Add a variable that stores your current working directory (the output of `pwd`) and prints it.

Passing Arguments

Code is useful when it is generalizable. Some times you will want to give code information, known as "arguments". You are able to do this with `BASH` as well

```
./myscript.sh hello, world
```

In this case, I am passing `hello, world` to my script. These are accessible within the script as `$1, $2`, etc.

Listing 4: Using Variables in Bash

```
#!/bin/bash
echo $1
```

Exercise 5:

1. Modify your script to print an argument passed to it (like above). What goes wrong when you pass `hello, world`?
2. Modify your script to print multiple arguments passed to it.

Conditional Statements in Bash

If Statements

Use `if` statements to make decisions in your script. Copy the sciprt below into an executable file and run it. Try to get both the possible outcomes of the `IF` statement.

Listing 5: If Statement Example

```
#!/bin/bash
if [ "$1" == "hello" ]; then
    echo "Hello to you too!"
else
    echo "You didn't say hello."
fi
```

Exercise 6:

1. Write a script that takes a user input as an argument and checks if the input is a file or directory.

Loops in Bash

For Loops

For loops allow you to repeat commands.

Listing 6: For Loop Example

```
#!/bin/bash
for i in {1..5}
do
    echo "Iteration␣$i"
done
```

Exercise 7:

1. Write a script that prints every other number from 1 to 10 using a `for` loop.
2. Write a script that prints all the files in the current directory (you might have to make some more files using `touch`.)

VI. HIGH-PERFORMANCE COMPUTING

High-performance Computing (HPC) is used when your laptop or desktop do not have the required computing power to complete a program or task. For example, I can use my laptop to run a . However, if I now want to run 1000 simulations of a star or a huge 3D simulation of black hole accretion, I need more than a few CPU cores. I might need thousands of CPU cores.

The solution? Have a shared set of computers that you can use when needed. This is HPC. Just a bunch of computers somewhere else. And they all run Linux.

How do we make use of all this extra computing power? I tell HPC to run a `BASH` script, and within that script I can ask HPC to perform a variety of tasks. If you are going to be using HPC regularly, I **urge** you to read through the documentation. It is worth your time.

First, let's make another `BASH` script called `submit.sh`:

Listing 7: Pre-slurm Script

```
#!/bin/bash

# first, go into the timestep directory
cd ~/timestep

# create a file called:
touch output.txt

# 'print' this phrase and append it to the file output.txt
echo "this␣terminal␣script␣ran␣successfully" >> output.txt
```

Now, like before, I need to change the mode to give the file run permissions, and run the file:

```
chmod u+x submit.sh
./submit.sh
```

Now, we have made sure that this script runs successfully on the log-in node. Let us try to submit this script to the HPC to run on the computing cluster.

SLURM Workload Manager

The SLURM workload manager is an **essential** part of how our HPC functions. Every user wants to submit jobs to the HPC but there is only a finite amount of computing resources. Therefore, we need some kind of system to allocate resources fairly - this is where SLURM comes in. It is a queueing system.

Submitting to SLURM

Now, we have created the script `submit.sh`. And we want to submit this to SLURM. We have to add a few lines to the script:

Listing 8: Slurm Script

```
#!/bin/bash


## the following are all settings to tell the SLURM workload manager. Depending
   on how many resources we ask for, our job will get off the queue faster.

#SBATCH -J timestep_test
#SBATCH -o job.out
#SBATCH --partition=windfall
#SBATCH -N 1 -n 1
#SBATCH --mem-per-cpu=4gb
#SBATCH --cpus-per-task=1
#SBATCH -t 0:02:30

cd ~/timestep
touch output.txt
echo "this␣HPC␣job␣ran␣successfully!" >> output.txt

sleep 120
```

At the end of the script, I have added the line `sleep 120`. This will tell the job to pause for 120 seconds, this will allow you check if it is on the queue, otherwise the job would complete instantaneously.

To submit this script as a job, enter this line into your Terminal:

```
sbatch submit.sh
```

You can check the queue with:

```
squeue
```

But, it will list every single job, by every user, on any queue. This is fun to look through sometimes. If you just want jobs that you have submitted, you can run the command:

If your job has run successfully, it should have created a file `output.txt` and added a line to it. Check whether that file exists and whether it has text in it.

```
squeue -u <your-net-id>
```

Where the `-u` is a flag that specifies to the terminal which user's jobs you want to see. You can also check other people's jobs, but that feels a little weird.

VII. ADVANCED HPC

I have not written this section yet, but the materials are available.

In the `timestep_practice` folder we downloaded, there is an `hpc_practice` folder. In there are materials from the COMPASS workshop we ran at the beginning of the year. You can go to our GitHub repository and look at Section 9: Intro to HPC, to work through the materials provided in that folder. As always, feel free to message me with any questions.