Programming Level-up

An Introduction to using Linux

Jay Morgan

7th October 2022

Outline

What is Linux?

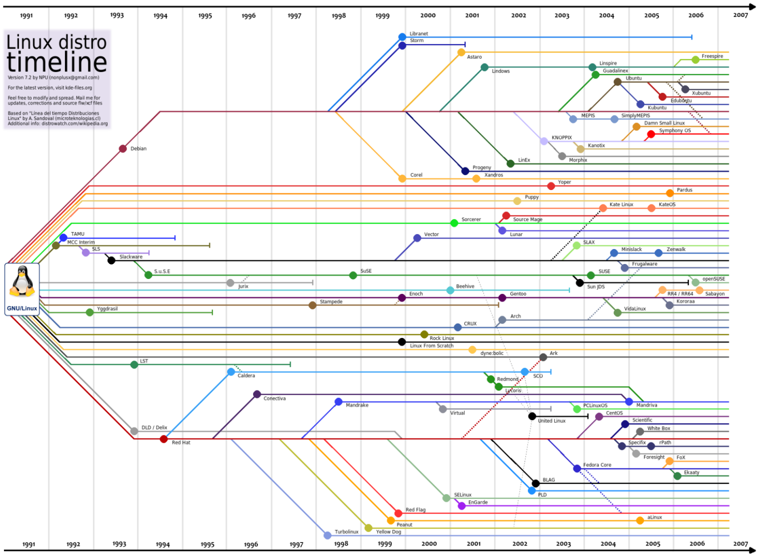
- Linux is a popular operating system (OS) like Windows, or MacOS.
- Unlike these other two OSs, Linux is open source, which means the source code is freely available to look at and modify.
- As its open source, its very possible for anyone to build their *own* version of Linux or build on top of Linux to create their own *Distribution* of Linux.

What's a Distribution?

A distribution can be considered like a *flavour* or version of Linux. There are many popular flavours that attempt to meet different needs from different users. For example:

- Ubuntu – typically the first Linux experience people will have. Attempts to be very user friendly.
- Fedora – stable and secure distribution while also providing up-to-date packages.
- Arch Linux – strong focus on customisability rather than user friendliness with bleeding edge packages.

What's a Distribution?



Defining Traits of Linux

While we have said that Linux is open source, there are many other traits that make it stand out from other operating systems:

- Complete control of how the system operates.
- The level of flexibility and automation that you can get from using the Linux command line.

While there are many other traits, these two are going to be what we're going to focus on.

What's a command?

While many recent versions of Linux makes things more accessible via GUIs, they will never be a substitute for using the command line. We're going to learn how to control the system via the command line, via a shell. A shell, like the Python REPL we've already seen, is waits for you to input commands, executes the command, and prints the output if there is output to print.

A Linux command is a call to a program optionally followed by some arguments. For example, if we want list out the files and folders in the directory, we would use the `ls` (list) command:

```
ls
```

What is a command?

The `ls` command comes with a number of optional flags and arguments that we can add onto the call. When calling a command a flag is something that begins with a `-` , for example `-l` tells `ls` to list the directory in a list format.

```
ls -l
```

What is a command?

We have supplied the `-l` flag. There are many other flags for `ls`, like for example, the human readable file systems with `-h` or show hidden files (files that start with a period) with `-a`.

When we're using multiple flags we could write

```
ls -l -h -a
```

Or:

```
ls -lha
```

What is a command?

Sometimes commands take optional positional arguments. Going back to our list directory command, where, by default, it will list the current directory. But instead we can tell the command to list a particular directory by supplying the path as an argument

```
ls images/ -lha
# or ls -lha images/ works too
```

What is a command?

How do I know how to use a command? Well that's where another command comes in. It's called `man` (short for manual). If you pass another command to the `man` command, the documentation will be shown in the terminal, e.g.:

```
man ls  # display the documentation for ls
```

The documentation should list all the different flags and arguments, describe what they mean, and sometimes give example or most common usage of a command.

When the 'man page' is display, you can scroll up and down the page using your arrow keys, and page-up and page-down. When you're done reading, just hit the 'q' character

Very useful commands

I am going to go through some of the most common commands just to make sure that you're familiar with the typical usage.

We've already seen `ls` to list a directory. The command to move to a directory is `cd` (change directory), that takes an argument of filepath to move to:

```
cd ~ # tilde is short-hand for the 'home directory'
cd ~/Documents/My\ Files   # go to Documents and then to "My
↪  Files"
cd   # no argument, by default goes to the home directory
```

Very useful commands – mkdir

Sticking with the them of directories, to make a new directory we use `mkdir`, whose argument takes the name of the directory we want to create:

```
mkdir my_new_directory
```

You can create a many level nested directory structure all at once using the `-p` (parents) flag, that tells `mkdir` if the parent directory of the target directory doesn't exist, create it.

```
mkdir photos/2020/01/05  # won't work unless photos/2020/01 exist
mkdir -p photos/2020/01/05  # this will work
```

Very useful commands – cp

To copy a file or directory, we can use the cp command. Here we are copying a file, where the first argument is the filepath of the file you want to copy and the second argument is the filepath where the copy should be placed.

```
cp my_old_file my_new_file
```

By default (without a flag), cp will not work with directories, for that you have to use the `-r` (recursive) flag

```
cp -r data/ data-backup
```

Very useful commands – mv

The syntax of moving a file is similar to that of cp:

```
mv old_file new_file
```

Except that it works for both files and directories without any flags. `mv` can also be used to rename files, that's all renaming is: moving a file to the same directory under a different name.

Very useful commands – rm

To remove a file us `rm`:

```
rm file_to_delete
```

If you want to delete a directory, use the `-r` (recursive) flag:

```
rm -r directory_to_delete/
```

Very useful commands – cat

cat stands for concatenate, i.e. concatenating the contents of two or more files:

```
cat file1 file2
```

The result is that the concatenation of these two files will be printed to the screen. If you wanted to put the result into its own file you would redirect the output using >

```
cat file1 file2 > newfile
```

Since cat reads the file and prints it to screen it is a very handy way to view the contents of a file, even if it was not intended for that.

Very useful commands – pwd

Sometimes you may get lost when moving directories. pwd prints the current working directory from the root directory, i.e. the path that is printed is an absolute path.

```
pwd
```

Very useful commands – find

If we want to list all files of a certain type, we can use the wildcard * that we've seen before:

```
ls *.jpg # list all files that end with .jpg
```

However, this will only list for the current directory. Perhaps the better way to find files will be using the `find` command:

```
find . -type f -name *.jpg
```

The first argument is the directory to start the search, then we define the type `f` being files, and then specify the name. Find will recursively search through directories and sub-directories to find all files that match that name.

Very useful commands – grep

How about if we want to find files that have a certain contents? For that we can use `grep`. Grep will read a file and print (by default) the lines that contains your pattern. i.e.:

```
grep 'Linux' lecture.org
```

This will print the lines that contain the word Linux in lecture.org. If we just want the matched value, we use the `-o` flag.

```
grep -o '[0-9]' lecture.org
```

This prints all occurrences of numbers in lecture.org

Very useful commands – less/head/tail

If a file is very long, we may not want to read the file using cat, as it will have to print the entire file. Instead we could use `less`, which will allow us to navigate through the file, using arrow keys to move and q to quit.

```
less filename
```

If we just want to view the first few lines, or the last few lines of a file we can use head/tail, respectively:

```
head filename
tail -n 20 filename  # last 20 lines
tail -F filename # constantly read the file
```

Very useful commands – wc

Often times we just want to count the number of something. For example, if we want to count the number of files/folders in the directory we can do:

```
ls -l | wc -l
```

We're first printing all files and folders in a list format (one per line), then passing ($_{\text{piping}}$_) the result to `wc`, which with the `-l` line flag, is counting the number of lines. Therefore we get a count of the number of files and folders. Here is another example where we're counting how many times the word bash appears in these lecture notes:

```
grep -o 'bash' lecture.org | wc -l
```

Very useful commands – piping

The purpose of piping is to pass data around between commands. We have just seen how we can pass the output of, say, the `ls` command to the input of `wc`. This allows use to construct very sophisticated pipelines to do some quite complex things from the combination of very simple commands.

```
find . -name '*.txt' -type f -print0 | xargs -0 grep "something"
```

Very useful basic commands

In summary we have seen the following commands:

- `ls` - List a directory
- `cd` - Change/move to a directory
- `mkdir` - Make a new directory
- `cat` - Concatenate files
- `cp` - Copy a file/directory
- `mv` - Move files/folders
- `rm` - Remove files and folders
- `pwd` - Display the current absolute path
- `find` - Find files
- `grep` - Find occurrences of a pattern in a file
- `less/head/tail` - Read a file
- `wc` - Count

Very first bash script

Let's start with the classic 'Hello, World' example. We'll create a new file called 'hello.sh' and enter the following:

```bash
#!/bin/bash

echo "Hello, World!"
```

First thing to notice is that the first line contains what we call a 'shebang' or 'hashbang'. It tells Linux which shell interpreter will be used to run the script, in this case: /bin/bash

The next (non-empty) line in the file is `echo 'Hello, World'`. This is exactly the same as the other commands we've just seen.

Very first bash script

Now that we've created and saved our bash script, we will want to run it. We have two alternative methods to run this script:

```
bash hello.sh   # run the script via bash
```

The second, requires that we have executable privileges for the script:

```
chmod +x hello.sh   # add executable 'x' privileges
./hello.sh  # execute it
```

Variables

The variables we create in our bash scripts are very much the same as the environment variables we've seen before. Take for example:

```
#!/bin/bash
AGE="35"
PERSON_NAME="Jane"
echo "$PERSON_NAME is $AGE years old"
```

We create a variable `AGE` with the = assignment operator. Note we don't put spaces either side of the equals sign in bash. To refer to the variable, we use `$AGE`, using the $ dollar sign.

Interpolation in bash strings

You would have noticed in the previous example that we included the variable directly into the string we're echoing out. This is something similar to what we've seen with f-strings in Python.

When we use double quotes: `"..."` in bash, the variable will be integrated into the resulting string. We can even call bash functions from directly inside the string:

```bash
echo "I am logging in as: $(who)"
```

Bash strings – the sharp edges

You might be tempted to use a variable when generating a path:

```
TRAIN_PROCESS="training"
TEST_PROCESS="testing"

touch "./data/$TRAIN_PROCESS_error.txt"
touch "./data/$TEST_PROCESS_error.txt
```

But this will create an error as underscores can be part of the variable name, so bash will be looking for a variable named: `$TRAIN_PROCESS_error` which has never been created. To get around this, we can wrap our variable in curly braces:

```
touch "./data/${TRAIN_PROCESS}_error.txt"
```

Stopping interpolation in bash strings

We can also use single quotes for strings in bash. When we use these strings, the string itself is not interpreted, and thus it will ignore any variables or bash commands:

```
echo 'I am logging in as: $(who)'
```

Input/Output

If we want to read the input from keyboard into a variable, we use the read command:

```bash
#!/bin/bash

echo "Enter your name:"
read NAME

echo "Hello, $NAME"
```

read in this context will read in the input and create the variable with that value. As we've already seen, we can then output this value to the console using the echo command.

Booleans

Technically, bash does not have built in data types for true and false, but Linux has the commands true and false which we could use in place. The implementation of how these commands work is not important.

```
FILE_EXISTS=true

if [ "$FILE_EXISTS" = true ]; then
    echo "The file exists!"
fi
```

Conditionals

When we're creating `if` expressions, we use the following syntax:

```
if <<conditional>>; then
    # do something
else
    # do something else
fi
```

We can also use `elif`

```
if <<conditional>>; then
   # do something
elif <<conditional>>; then
    # do something else
else
    # something else entirely
fi
```

Conditionals

Writing condition expressions can be a little more cumbersome than in Python. These can be many pain points for new bash programmers, take for example:

```
FILE_EXISTS=false

if [ $FILE_EXISTS ]; then
    echo "The file exists!"
fi
```

This is because we have used the `[...]` single bracket syntax for the test. But there are others:

- No brackets: we could omit the brackets in which case it would run the false command not print the statement.
- Single paranthesis `(...)` creates a sub-shell.
- Double paranthesis `((...))` for arithmetic operation
- Single square bracket `[...]` calls `test`
- Double square bracket `[[...]]`

Conditionals

What if we write:

```
VAR_1="Mr Foo Bar"
VAR_2="Mr Foo Bar"
if [ $VAR_1 = $VAR_2 ]; then
    echo "They are the same"
fi
```

We would get an error because `test` expands the arguments into:

```
Mr Foo Bar = Mr Foo Bar
```

With the spaces included. To prevent this from happening, we have to wrap the variables in quotation marks.

```
VAR_1="Mr Foo Bar"
VAR_2="Mr Foo Bar"
if [ "$VAR_1" = "$VAR_2" ]; then
    echo "They are the same"
fi
```

Conditionals

If we use `[[` in if statement, then we can do more sophisticated things like pattern matching:

```
FILENAME="testing.png"
if [[ "$FILENAME" = *.png ]]; then
    echo "Its a png file"
fi
```

Loops

Like in Python, we can iterate in bash

```
for i in {1..10}; do
    echo $i
done
```

This iterates with i starting at 1 upto 10 (inclusive). Or we could do:

```
for (( i=1; i <= 10; i++ )); do
    echo $i
done
```

Loops

We can also iterate over a list of files/folders in a directory:

```
for FILE in ./images/*; do
    echo $FILE
done
```

Loops

Using the `while` form, we can continue looping until our conditional is false. For example, we could loop testing our internet connection, until its been established:

```bash
while ! ping -c 1 google.com; do
    echo "No internet yet"
    sleep 1
done

echo "Internet is available!"
```

Functions

To create a function, we use the following syntax:

```
function_name() {
    # do something
}
```

And to call the function, you just need to use the function name:

```
function_name # this called function name
```

Functions

Here is another example:

```
say_hello() {
    echo "Hello, $1"
}

say_hello "Jane"
```

Notice that we didn't need to include any argument list. We just used $1 for the first argument passed to the function.

```
say_hello() {
    echo "$1, $2"
}

say_hello "Hi" "Jane"
```

Functions

Returning values is 'interesting' as, coming from other languages, you think could do something like this:

```bash
say_hello() {
    return "hello"
}
RESULT="$(say_hello)"
echo $RESULT
```

This didn't work like we expected, the value wasn't returned and assigned to RESULT. So how do we return a value?

```bash
say_hello() {
    echo "Hello"
}
RESULT="$(say_hello)"
echo "This is before the printing of result"
echo $RESULT
```

What is the Cluster?

How to login

```
ssh <<username>>@saphir2.lis-lab.fr
```

Then:

```
ssh <<username>>@sms-ext.lis-lab.fr
```

How to login

Typing both these commands can become tiresome very quickly. But we can make it a lot easier by updating our `~/.ssh/config` file to include something like:

```
Host saphir2
     HostName saphir2.lis-lab.fr
     User <<username>>

Host cluster
     HostName sms-ext.lis-lab.fr
     User <<username>>
     ProxyCommand ssh saphir2 -W %h:%p
```

Then to login to the cluster, we just need to type:

```
ssh cluster
```

And we should be prompted for our password.

How to login

If you trust the machine your on, you can remove password authentication and move to key-based authentication:

```
ssh-copy-id saphir2
ssh-copy-id cluster
```

When we next login to the server, we shouldn't be prompted for a password.

How to copy files to and from the cluster

We have a number of options for transferring files to and from the cluster. Firstly, let's look at the command scp. It takes two arguments, the first argument is the file you want to send, the second argument is the destination of the sent file.

```
scp <<origin>> <<destination>>
```

Similar to commands like cp, scp by default only works for files, not folders. To send folders/directories, we use the -r flag just like cp.

```
scp -r <<origin_folder>> <<destination_folder>>
```

Copying files – rsync

One of the downsides about scp is that it will copy every file you give it. Even if the file at the destination is exactly the same. What if we only want to copy files that need to be copied, i.e. that are outdated, thus saving time? For that, we can use rsync. Rsync will copy files from one source to a destination only if the destination needs to be updated. This can save a lot of time by skipping files that already exist at the destination:

```
rsync <<source>> <<destination>>
```

The Login- and Compute Nodes

When you login to the cluster, you are logging into the *login* node. Note that no computation should be run on this node. If you want to run scripts, you will have to submit a job to the compute nodes.

On the login node there is a system installed called 'SLURM'. SLURM is a job scheduler program that receives your requests for executing scripts, it will queue them and assign them to available compute nodes.

We will take a look at how to request and manage jobs using the various commands that SLURM provides.

How to launch a job – srun

The first command we will look at it is `srun`. This command will run request a job for execution in 'real-time'. By real-time, we mean that the shell will wait until the job has been submitted.

```
srun <<compute node options>> <<command to run>>
```

Let's take a look at an example where we want to run an interactive bash shell on the compute shell (similar to ssh'ing into the compute node).

```
srun --time=00:10:00 --pty bash -l
```

This will request a job on any available compute node for 10 minutes. When a node becomes available, `bash` will execute, dropping you into the shell. You will notice that the shell prompt has changed from `sms` to the name of the node.

How to launch a job – salloc

There is another method we can use to create an interactive job. We could use the salloc command to allocate resources for a task. After the resources have been allocated and our job is ready, we can ssh into the node with the allocated resources.

```
salloc --time=10:00:00 &
ssh <name>
```

How to launch a job – options

In the previous command, we used the `--time` option to specify how long the job will run for. But there are other options we can use to be more specific about the jobs we want to run.

`--cpus-per-task` can be used to request more than one CPU to be allocated. This is especially helpful when we have a multithreaded process we want to run.

`--mem` specifies how much memory should be allocated to the job. For example: `--mem=16G` tells SLURM to allocate 16 GB of memory.

How to launch a job – GPU allocation

If we need to use a GPU, we need to use a few options. Firstly, we can specify that our job is on a compute node with GPU. There will usually be a group of nodes in a 'GPU' group or partition, and thus we can specify to use one of these partitions:

```
srun --time=00:10:00 --partiton=gpu --pty bash -l
```

But you will notice that you still do not have access to a GPU. You're running on the GPU node, but you haven't actually requested a GPU be allocated to your job. For that you will use `--gres`:

```
srun --time=00:10:00 --partition=gpu --gres=gpu:1 --pty bash -l
```

Here we are requesting one GPU, but if we use `--gres:gpu:2` we are requesting 2 GPUs etc.

How to launch a job – GPU allocation

There are many different types of GPUs available, some older than others. If you wanted to allocate a job with a specific type of GPU you can use the `--constraint` flag:

```
srun --time=00:10:00 \
     --partition=gpu \
     --gres=gpu:1 \
     --constraint='cuda61' \
     --pty bash -l
```

This command requests that our job run on the GPU partition, with 1 GPU allocated that has the capability of running CUDA compute 61.

Or we can specify the type of GPU in the gres option:

```
srun --time=00:10:00 \
     --partition=gpu \
     --gres=gpu:2080:1 \
     --pty bash -l
```

Learning more about nodes

To understand what each compute node has we can use the `scontrol` command.

```
scontrol show nodes
```

Will list out all nodes and all capabilities of each node. Or just one node:

```
scontrol show node lisnode2
```

How to launch a job – sbatch

It can be quite inconvenient to launch an interactive job to run some compute, and wait for the job to be allocated. If, instead, you have a long running experiment that you want to run without any intervention from you, you can use `sbatch`.

Sbatch will require us to write a small bash script that specifies how to run a job and what to do once its allocated.

```
#!/bin/bash

#SBATCH --time=00:01:00
#SBATCH --job-name=my_new_job
#SBATCH --output=my_new_job.out
#SBATCH --error=my_new_job.err

echo $HOSTNAME
```

And run it:

```
sbatch my_job.sh
```

How to launch a job – sbatch

Notice that instead of supplying options to sbatch, we can instead record them directly into the script using the `#SBATCH`. SLURM will examine this file, looking for lines starting with this comment, and infer that the rest of the line contains the options.

There are a few other options we've included that are very useful when running non-interactive jobs. Firstly, we've given the job a name (`my_new_job`). This is so we can different between many jobs that we might run at the same time. To list out the jobs we currently have running we use `squeue`.

```
squeue
```

By default, squeue will list all of the active jobs, even other peoples. To specify only your jobs user the `--user` option:

```
squeue --user=jay.morgan
```

How to launch a job – sbatch

The other two options, `--output` and `--error` specify where the printed output and printed errors will be stored. Since the job is being run on a different node, by a non-interactive process, if you didn't include these lines, you wouldn't be able to see what was being printed by `echo` or by any other process such as `print` in Python.

Job Management – squeue

When we list the jobs using `squeue` it will give us multiple columns of information, such as:

- JOBID – the referable id of the job.
- PARTITION – the partition on which the job has been requested for.
- NAME – the name of the job.
- USER – the user who submitted the job.
- ST – the status, is the job currently running, waiting, or exiting?
- TIME – how long the job has been running for.
- NODES – how many nodes have been allocated to the job.

Job Management – scancel

Let's say that we've submitted a job, but we've noticed that there was an error in the code, and want to stop the job. For that, we use `scancel` and specify the id of the job we wish to cancel:

```
scancel 158590
```

After running this command, we should see, using `squeue`, that either the job is finishing, or that its disappeared from our list (meaning that its completely stopped).

Job Management – sacct

If our job has finished, or exited and is no longer in `squeue`, we can use `sacct` to get a history of the jobs.

`sacct` will list all of your jobs within some default window of time. If we want to change this window we can use the `--starttime` and `--endtime` options.

Valid time formats are:

- HH:MM[:SS][AM|PM]
- MMDD[YY][-HH:MM[:SS]]
- MM.DD[.YY][-HH:MM[:SS]]
- MM/DD[/YY][-HH:MM[:SS]]
- YYYY-MM-DD[THH:MM[:SS]]
- today, midnight, noon, fika (3 PM), teatime (4 PM)
- now[{+|-}count[seconds(default)|minutes|hours|days|weeks]]

Job Task Arrays – motivation

Task arrays allow you to submit many jobs of the same type. Why might this be useful? Suppose you have a list of files that take a long time to process:

- `file_0.txt`
- `file_1.txt`
- `file_2.txt`

Or you have some computation script, such as deep learning training script, that takes uses a hyperparameter which can be tuned to achieve different performance results:

```
python train.py --learning-rate 0.001
```

Instead of a creating a sbatch script for each value of hyperparameter, or sequentially enumerating the values, you can use a job task array to spawn multiple jobs with slightly different values.

Job Task Arrays – how to

First, we will look at how to actually submit an array of tasks. To create an task array, you will need to add the --array options to your sbatch script:

```
#!/bin/bash

#SBATCH --job-name=my_task_array
#SBATCH --array=1-5

...
```

Here we are creating an array of tasks numbered from 1-5. When you submit this script, you will see five tasks submitted to the queue.

Job Task Arrays – how to

Now that we know how to create an array of tasks, we will want to do something useful with it. When you create an array, each individual task will have a unique variable called `SLURM_ARRAY_TASK_ID`. So for example, if we launch an array of 5 tasks, the first task will have the value `1`. Why is this useful? Well, we can use this variable to alter the program slightly. Take for example our list of files we need to process:

```bash
#!/bin/bash
#SBATCH --job-name=my_task_array
#SBATCH --array=0-4
#SBATCH --time=00:10:00

FILENAME="file_${SLURM_ARRAY_TASK_ID}.txt"
python process.py $FILENAME
```

This will create a new bash variable called `FILENAME` by concatenating `file_` the current task's (i.e. 0, for the first task, 1 for the second task, etc) and `.txt`.

Job Task Arrays – how to

If we run the previous example, we will see that we have five jobs named exactly the same thing `my_task_array`. This is okay, but we can be a little bit more clear as to which task is running, i.e. which task is processing which file?

We can use some special variables in our bash script to make this more clear. These are `%A` that is the main job id, and `%a` that is the task array id.

```
#!/bin/bash

#SBATCH --job-name=my_task_array.%A_%a
#SBATCH --output=my_task_array.%A_%a.out
...
```

Now, every task in our array will have a slightly different name because of the `%a` and therefore we will be able to determine which job is processing which file.

Job Task Arrays – how to

Let's move on to the second example, where we have a Deep Learning training program and we want to try different parameters. In this case, we can again use a task array.

```
#!/bin/bash

#SBATCH --array=1-10
```

We could either pass the SLURM_ARRAY_TASK_ID as a command line argument to the script:

```
python training.py --learning-rate $SLURM_ARRAY_TASK_ID
```

But in this case, we could have to properly calculate the correct learning rate from the SLURM_ARRAY_TASK_ID value (remember that in my sbatch script I set --array=1-5). But bash only performs integer arithmetic, therefore we will need to calculate the correct learning rate in something else

Job Task Arrays – how to

Instead of passing the learning rate via a command line argument. We can get the value directly from our python script and calculate the value.

```python
import os

task_id = int(os.environ["SLURM_ARRAY_TASK_ID"])
learning_rate = task_id / 100
```

Here we are using the builtin os module in Python, getting the environment variable from the dictionary `environ` and parsing the value as an integer. Then we can calculate the appropriate learning rate using this value. So for example, if `SLURM_ARRAY_TASK_ID` is set to 1. Our learning rate would be 0.01 for this task.

Job Task Arrays – how to

If you're creating a job task array, you may want to create hundreds of jobs. And of course, you don't want to use up the entire cluster leaving no resources for anybody else! Therefore, you will only want a maximum number of tasks to run at any one time.

```
#!/bin/bash

#SBATCH --array=1-100%5
```

This will create a job task array of 100 jobs numbered from 1 to 100. But we have added an additional argument %5 which means that only 5 jobs can run at any one time for this task array. If you have five tasks running, the other 95 tasks will wait.

If, at any point, you want to change how many jobs can run simultaineously, you can update this 'throttle' value using `scontrol`:

```
scontrol update ArrayTaskThrottle=<count> JobId=<jobID>
```

Job Task Arrays – how to

So if we've already launched a job task array with the job id of 50602 that has a throttle value of 5 (only 5 tasks will run at once), we can change it to 10 using:

```
scontrol update ArrayTaskThrottle=10 JobId=50602
```

A guided walk through – environment

In this section we're going to give an example walk through of working with the HPC cluster. In this example, we're going to write our scripts locally, including the slurm submission script, and when they're ready, we'll send them to the cluster to perform the actual computation.

Let's imagine we're starting a new project, and are programming our scripts in Python. Now is a good time to create a new conda environment to install our packages we're going to use for our research. We'll create this environment with (replacing `<env-name>` with whatever we want to call this environment):

```
conda create --name <env-name>
```

and then activate it:

```
conda activate <env-name>
conda install python=3.9
```

Writing our scripts

Let us also image we've just wrote the following script to create a lorenz attractor: lorenz.py

The specific implementation of this script is not particularly important for this walk through. Just know that we're importing a few packages such as numpy and matplotlib. Then, we're performing some computation, and saving the results to analyse later. As this script uses external libraries, we need to install them:

```
conda install numpy matplotlib
```

Writing our job submission script

Since we want our calculations to be performed on the cluster, we will need to also write a job submission script (let's call this `submit-job.sh`) in bash to pass to SLURM.

```bash
#!/bin/bash

#SBATCH --job-name=lorenz_attractor
#SBATCH --output=lorenz_attractor.log
#SBATCH --error=lorenz_attractor.log
#SBATCH --time=00:10:00

python lorenz.py
```

Replicating our environment on the cluster

As we've installed external packages in our local development environment, we will want to ensure that when we run the calculations on the cluster, it will be using the same versions of packages. Conda makes this a lot easier. First, we export our environment to a recipe file:

```
conda env export --no-builds > environment.yml
```

Sending our scripts to the cluster

All of our scripts are ready! We can now transfer them from our personal computer, to the cluster. The files we need to transfer are:

- `lorenz.py`
- `environment.yml`
- `submit-job.sh`

While we can send a folder (and the containing files), let's send them one at a time:

```
scp lorenz.py <hostname>:<destination-path>
scp environment.yml <hostname>:<destination-path>
scp submit-job.sh <hostname>:<destination-path>
```

where `<hostname>` is the hostname/IP address that you've used to connect to the login node on the cluster before. `<destination-path>` is the path to where you want to save the files.

Logging into the cluster

Now that our files are on the cluster, we can login:

```
ssh <username>@<hostname>
```

At which point, we've logged into the login node, and then we need to change directory to where we saved the files:

```
cd <destination-path>
```

Re-creating our development environment

Now that we're in the same folder as our scripts, we're almost ready to submit our job. First, we need to recreate our development environment from our `environment.yml` file.

```
conda env create -f environment.yml
```

And activate our newly created environment:

```
conda activate <env-name>
```

Submitting our job

Now we can submit our job:

```
sbatch submit-job.sh
```

We can check the progress of our job with `squeue`, or its already completed, look at the job history with `sacct`.

Downloading the results

If our job runs successfully, a `data.pkl` file will be created. Back on our local computers, we will need to run the following to download it:

```
scp <hostname>:<destination-path>/data.pkl ./
```

This will download the file into the current directory.

Analysing the results

With the `data.pkl` file downloaded, we can visualise the results using `plot_lorenz.py`: `https://pageperso.lis-lab.fr/jay.morgan/resources/2021-programming-level-up/lectures/week-5/plot-lorenz.py`

If everything has been run correctly, you should see a plot of the lorenz attractor.

Useful features – X11 Forwarding I

If we're performing analysis interactively using the cluster, we'll often want to visualise the results, using matplotlib for example. To *see* our plots display like they would on our local machine when we call `plt.plot` or `plt.show()`, we will need to ensure that we're using something called `X11 Forwarding`. To enable `X11 Forwarding`, we use the `-X` option when `ssh`'ing into the cluster and compute nodes (i.e. it will need to be enabled on every 'hop' so to speak).

```
ssh -X <<remote-host>>
```

If want to enable it by default, we can enable it in our `ssh` config file:

Useful features – X11 Forwarding II

```
Host saphir2
     HostName saphir2.lis-lab.fr
     ForwardX11 yes
     User <<username>>

Host cluster
     HostName sms-ext.lis-lab.fr
     FowardX11 yes
     User <<username>>
     ProxyCommand ssh -X saphir2 -W %h:%p
```

After setting up `X11 Forwarding` correctly, and when logged into the remote host, we should be able to echo a variable called `$DISPLAY`.

```
echo $DISPLAY
```

Useful features – X11 Forwarding III

If `$DISPLAY` has a value, we know that X11 Forwarding has been setup correctly, and we're ready to do some plotting!

Useful features – Jupyter Notebooks I

We've talked about how good jupyter notebooks are for performing analysis and exploration. But often times, we will need a lot of compute resources (more than our local computers can handle) to do this analysis. This is where using jupyter notebook on the supercomputers comes in handy. However, it is not as simple as starting the jupyter notebook server and opening up your web browser. First, we will need to setup a 'reverse ssh tunnel'.

In a nut-shell, a reverse ssh tunnel allows you to redirect data on a remote port to a local port.

Therefore, we can, using a reverse ssh tunnel, start a jupyter notebook on the supercomputer and access it using the web browser on our local computer!

To begin, we can create an interactive job on the cluster:

```
srun --time=01:00:00 --pty bash -l
```

Useful features – Jupyter Notebooks II

And start our jupyter notebook, specifying a port that will not be in use:

```
jupyter notebook --port 30333
```

With our notebook server now started on the 30333 port, we will want to create an ssh tunnel from our local computer, to the cluster's login node, and then a tunnel from the login node to the specific compute node where the job is running:

```
ssh -L 30333:localhost:30333 <<cluster-login-node>> ssh -L
↪ 30333:localhost:30333 <<cluster-compute-node>>
```

If everything goes well, we should now be able to open up our web browser, navigate to `localhost:30333` and see our jupyter notebooks.