# Working on a Linux cluster

Thales A Gutcke — 2023

This document covers a few topics to get you started doing data analysis on a cluster.
Any text within angle brackets (<>) is placeholder text that should be replaced with the correct text and without the brackets. For further reading, many sections provide links to additional resources on the web.



---

## Logging in to the cluster

Open a shell/terminal and enter:

```
ssh -A -t <username>@<cluster address>
```

It will request your password.

Now you are logged into the shell on the cluster through a secure shell (ssh) connection. Use standard Unix shell commands to navigate, create documents and run scripts.

You can change your password from the one original one by typing

```
passwd
```

It will prompt you to type your old password and then a new one twice.

## Useful commands on a shell

| | |
|---|---|
| `ls <path>` | **list** directory content |
| `pwd` | **print** working directory (where you are) |
| `cd <path>` | **change** where you are to another directory |
| `mkdir <name>` | **create** directory with given name (can include a path) |
| `cp <filepath1> <filepath2>` | **copy** file from location 1 to location 2 |
| `mv <filepath1> <filepath2>` | **move** file from location 1 to location 2 |
| `rm <filepath>` | **remove** file (you cannot undo this, so use with caution) |
| `vi <filepath>` | **open** file with vim editor |
| `emacs <filepath>` | **open** file with emacs editor |
| `python3 <filepath>` | **run** script with python 3 |
| `find <path> -name <file>` | **search** for a file (it will check nested directories in path) |
| `grep <string> <filepath>` | **search** for string within a file |
| `man <command>` | pulls up the **manual** for a command. Mostly this is a list of flags (prefixed by a dash -) and how they alter the command. Exit manual with `q` (for quit). |

**More info**: https://oit.ua.edu/wp-content/uploads/2020/12/Linux_bash_cheat_sheet-1.pdf

## Tab completion

The shell knows the commands, files and the directory hierarchy. This means it can complete commands and paths for you while you are typing. This is especially useful for long filenames.

Start typing a command or path and press the **tab button**. The shell will either complete it, or if there are multiple options, show you these. Then you can hit enter.

**More info**: https://www.howtogeek.com/195207/use-tab-completion-to-type-commands-faster-on-any-operating-system/

## Wildcard matching

Wildcards are special sequences used to match patterns in filenames. These can be used with many of the above commands such as `ls, rm, mv, grep,` etc.

| | |
|---|---|
| `*` | represents any character or any number of characters |
| `?` | represents any single character |
| `[0-9]` | any number between 0-9 |
| `[a-z]` | any lowercase letter |
| `[fgh]` | will match only either f, g, or h |
| `[!B]` | will match any character except B |

**More info**: https://www.landoflinux.com/linux_wildcard_pattern_matching.html

## The home directory and .bashrc

Every user has a home directory `/home/<username>/` and the symbol ~ is the shorthand for this path location. Your home is not readable or writable by any other user, unless you choose to give them those permissions.

There is a file called ~/`.bashrc` in your home directory. Whatever is written into this file is read each time you log in. This means you can use it to set variables and setup your work environment. For example, on your laptop, you could place this line into your .bashrc. This creates an alias (new command) that replaces a longer string.

```
alias cluster="ssh -A -t <username>@<cluster address>”
```

Then you would only need to type `cluster` and press enter to begin a session.

If you make changes to your bashrc, you can force the system to re-read it by typing

```
source ~/.bashrc
```

**Important note**: Depending on your system and the type of shell you are using, the name .bashrc will be different. On newer macs it might be .profile or .zprofile.

## Using local executables

Most clusters have certain programs/commands pre-installed (i.e. vim, python3, gcc, etc). However, we may want to use programs that are not pre-installed.

As simple users we do not have write permission in the base filesystem of the cluster where programs are usually installed. Instead, we can compile and install additional programs **locally** (meaning in our home directories where we do have write permission).
The keyword to install programs locally is `--user`.

## The PATH and PYTHONPATH variables

PATH is a variable on every linux system that defines the list of paths that will be checked when executing a command. If the executable for that command is not found in any of the paths in PATH, then the command will fail.

Check the current state of PATH with:

```
echo $PATH
```

It should look something like this:

```
/home/<username>/bin/:/usr/local/bin:/usr/bin
```

Effectively, it is a single string of paths that are separated by colons (:). To add a new path to the PATH variable, we can type:

```
export PATH="$PATH:<new path we want to add>"
```

This command can also be put into the .bashrc, so that the new path is added each time we open a new shell.

PYTHONPATH works the same way as PATH, except that it defines the list of paths where python will look for packages that we attempt to import.

## Setting up python3

Update pip3

```
pip3.6 install --upgrade pip --user
```

Install a bunch of python packages

```
pip3 install numpy
pip3 install scipy
pip3 install matplotlib
pip3 install ipython
pip3 install jupyter
etc.
```

Install notebook extensions that provide a table of contents, code folding, etc.

```
pip3 install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable scratchpad/main —user
```

**More info on notebook extensions**: https://jupyter-contrib-nbextensions.readthedocs.io/en/latest

## The screen command

The Screen command in Linux allows the user to create multiple virtual terminals that can be saved by name and reopened using keyboard shortcuts. Importantly, when you having a process running in a detached screen, you can exit the cluster, close your laptop and the process will continue.

| | |
|---|---|
| `screen -S foo` | - opens a new screen called `foo` |
| `screen -R foo` | - reattaches to an open screen called `foo` |
| `screen -ls` | - lists open screens |

| | |
|---|---|
| Detach from current terminal | [Ctrl A + D] |
| Shutdown current terminal | [Ctrl S + D] |

**More info**: https://linuxhandbook.com/screen-command/

## Opening jupyter notebook on your laptop

It is possible to use the computing power, memory and data that is stored on the cluster, while still working on a jupyter notebook that is open **locally** on your laptop. The way this works is to open a notebook **remotely on the cluster**, but tell it to **forward it to a port**. Then, on your laptop you can **connect to the correct port** and open the notebook in the browser on your laptop.

**Do this once** (and again in case the screen crashes or is closed):

In a shell that is logged into the cluster:

```
screen -S <screen name>
nice jupyter notebook --no-browser --port=6666
```

You can use any screen name, but it's useful if it is memorable and descriptive.
You can use any port number, **but it needs to match the number below**!
Available ports can be any number in this range: 1024 to 49151

Copy http address shown (with long token string) and then detach [Ctrl A + D]

**Do this every time your laptop was disconnected from the internet:**

In a shell on your laptop:
```
ssh -f -N -L 6666:localhost:6666 <username>@<cluster address>
```

Paste http address into a browser on your laptop or just reload the tab if it was previously open. For convenience, the command above can be added as an alias to the .bashrc on your laptop.

**More info**: https://thedatafrog.com/en/articles/remote-jupyter-notebooks/

---

## Moving files to/from the cluster

Do this **on your laptop** to move a file **to the cluster**:
```
rsync -rv <path on laptop> <username>@<cluster address>:<path on cluster>
```

Do this **on your laptop** to move a file **to your laptop**:
```
rsync -rv <username>@<cluster address>:<path on cluster> <path on laptop>
```

These will both prompt your cluster password.

---

## ssh-keygen

Only do this if you use a cluster a lot and are getting tired of typing your long and complicated password many times. This is **not necessary** but only convenient.

On your laptop, create a public-private key pair:

```
ssh-keygen -t ecdsa -b 521 -a 100
```

It will ask for a passphrase. For added security, many clusters will require that you choose one.

Then it will ask for the name of the id file. Choose something recognizable like `id_cluster`.

Copy public key to a remote location:

```
ssh-copy-id -i ~/.ssh/id_cluster <username>@<cluster address>
```

If this fails, you can also manually copy the contents of the file `id_cluster.pub` to a new file called `.ssh/authorized_keys/id_cluster.pub` on the remote machine.

The first time you log in to the cluster with this key pair, it will ask for your password. After that, it will only ask for the passphrase, which can be much shorter and simpler.

Now you can set up `.ssh/config` on your laptop:

```
Host cluster
Hostname <cluster address>
User <username>
Identityfile ~/.ssh/id_cluster
```

Once this is done, you can log in to the cluster by just typing:

```
ssh cluster
```

And copying files on your laptop with:

```
scp cluster:<path on cluster> <path on laptop>
```

**More info**: https://www.ssh.com/academy/ssh/keygen

---

## Giving permissions to another user

Initially, no one but you will have **read, write and execute** permissions to files in your home directory. You can check the status of the permissions in a given directory path with:

```
getfacl <path>
```

You should see something like this:

```
# file: <path>
# owner:<username>
# group:<username>
user::rwx
user:<other username>:r-x
group::---
other::---
```

The line `user::rwx` tells you that you have **read (r), write (w) and execute (x)** permissions to files on this path. `user:<other username>:r-x` tells you that some other user has only **read and execute** permissions here.

You can give other users permissions. Here, you are **modifying (m)** the **read (r)** and **execute (x)** permissions **recursively (R)** for all files in the directory path for a given user.

```
setfacl -Rm user:<username of other user>:r-x <path>
```

**Recursively** means it sets these permissions within all subdirectories as well.

You are only allowed to change permissions to files and directories that you own (see `owner` above), so mainly only within the directory structure of your home directory.