# Introductory LINUX/command line training

Steinchneider Research Group, Cornell University

# Motivation

In sort of priority order:

- **Only** way to interface with HPC resources, either locally (e.g. Hopper) or external (e.g. ACCESS) [HPC training]
- Easiest way to interact with Git repositories, although GUI methods exist [Git training]
- Many spatial geophysical datasets (e.g. reanalysis, gridded hydrology, forecasts, etc) most easily accessed/processed through command line tools [Spatial data training]
- Task automation through shell scripting

# Objectives

- Familiarize with basic LINUX terminology and derivations
- Familiarize with basic LINUX shell command structure and commands
- Introduce some LINUX shell operations useful for later training (HPC, git, spatial data)
- Introduce some useful resources, motivate the utility of LINUX proficiency, pave the way for self study

*This training assumes no working knowledge of command line!*

# Overview

- Terminology
- Basic command line construction and commands
  - Directory management
  - Interacting with files
- Slightly more advanced command line stuff
  - Text editors
  - Shell scripting
  - Running an R script
- High level (e.g. R) -> shell functionality
- Resources

# Terminology

LINUX: The Linux operating system is an extremely versatile Unix-like operating system, and has taken a clear lead in the High Performance Computing (HPC) and scientific computing community.

There are two major components of Linux, the kernel and the shell:

- The **kernel** is the core of the Linux operating system that schedules processes and interfaces directly with the hardware. It manages system and user I/O, processes, devices, files, and memory.
- The **shell** is a text-only interface to the kernel. Users input commands through the shell, and the kernel receives the tasks from the shell and performs them. The shell tends to do four jobs repeatedly: display a prompt, read a command, process the given command, then execute the command. After which it starts the process all over again.

It is important to note that users of a Linux system typically *do not* interact with the kernel directly. Rather, most user interaction is done through the shell or a desktop environment.

# Demystifying terminology…maybe

So Linux is basically the newer, open source version of UNIX that allows us to interface somewhat directly with the computer's operating system. But what is the 'shell', 'command line', 'bash', 'bash scripting'?

In a nutshell, these are terms for how you interact with LINUX/UNIX kernel and tell the computer what you want it to do:

- 'shell' is the most basic form; **shells** interpret commands for the operating system to execute
- 'bash' (Bourne again shell) is a **specific form** of shell very commonly used
- 'command line' or 'command line interface' is a more generic descriptor for something that interacts with the computer's operating system
- 'shell scripting' is simply the automation of a bunch of shell tasks, just like one would automate a bunch of high-level language (R, Python) with a script

As a user (ie not a software engineer or computer scientist), these nuanced differences really aren't that important; it's just useful to know that all these terms refer to roughly the same thing in many contexts…I will probably use these terms somewhat interchangeably going forward

# Command line operations on non-LINUX OS

For non-LINUX operating systems (Windows PC), there are a number of LINUX-like interfaces to the OS (powershell, Cygwin, R-terminal, etc).

These interfaces often allow command line tasks in line with LINUX/bash conventions. However, they can be fairly limited in their LINUX functionality and are often unable to execute higher level command line executables/packages.

For instance, many command line tools for spatial data processing (CDO, NCO) can be challenging to get to work on PCs, but will work great on Mac/Ubuntu as well as LINUX based HPCs

# Construction of shell command (LINUX/bash)

*Note: these examples largely follow the Cornell CAC Linux tutorial*

Structure of a command:

```
§ <command> <option(s)> <argument(s)>
```

**command** – the executable to be run

**option(s)** – (aka 'flags'), optional arguments that alter command behavior (some commands have them, some don't)

**arguments** – depend on the command; one common example of an argument is 'file.in file.out'

*bash can be case sensitive and is also sensitive to spaces

**common convention is '$' to refer to a bash command and '>' to refer to a higher level language command (R, Python)

# Getting info on a bash command

To get more information on a specific command, the ‘man’ command can be used in actual LINUX machines;

```
man <program or command>
```

LINUX-like interpreters may use a ‘<program> --help’ kind of command:

$ mv –help

Help pages will include the following items:

1. **NAME** – a one-line description of what it does.
2. **SYNOPSIS** – basic syntax for the command line.
3. **DESCRIPTION** – describes the program's functionalities.
4. **OPTIONS** – lists command line options that are available for this program.
5. **EXAMPLES** – examples of some of the options available
6. **SEE ALSO** – list of related commands.

# Getting info on a bash command

For the ‘mv’ [move] directory management executable, you get this for instance:

```
$ mv --help
Usage: mv [OPTION]... [-T] SOURCE DEST
  or:  mv [OPTION]... SOURCE... DIRECTORY
  or:  mv [OPTION]... -t DIRECTORY SOURCE...
Rename SOURCE to DEST, or move SOURCE(S) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.
      --backup[=CONTROL]       make a backup of each existing destination file
  -b                           like --backup but does not accept an argument
  -f, --force                  do not prompt before overwriting
  -i, --interactive            prompt before overwrite
  -n, --no-clobber             do not overwrite an existing file
If you specify more than one of -i, -f, -n, only the final one takes effect.
      --strip-trailing-slashes  remove any trailing slashes from each SOURCE
                                  argument
  -S, --suffix=SUFFIX          override the usual backup suffix
  -t, --target-directory=DIRECTORY  move all SOURCE arguments into DIRECTORY
  -T, --no-target-directory    treat DEST as a normal file
  -u, --update                 move only when the SOURCE file is newer
                                 than the destination file or when the
                                 destination file is missing
  -v, --verbose                explain what is being done
  -Z, --context                set SELinux security context of destination
                                 file to default type
      --help     display this help and exit
      --version  output version information and exit

The backup suffix is '~', unless set with --suffix or SIMPLE_BACKUP_SUFFIX.
The version control method may be selected via the --backup option or through
the VERSION_CONTROL environment variable.  Here are the values:

  none, off       never make backups (even if --backup is given)
  numbered, t     make numbered backups
  existing, nil   numbered if numbered backups exist, simple otherwise
  simple, never   always make simple backups

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation <https://www.gnu.org/software/coreutils/mv>
or available locally via: info '(coreutils) mv invocation'
```

# Command line training exercises

# Directory navigation

First, get into your machine's version of terminal/shell, then-

1. Where am I? Use the 'pwd' **p**rint **w**orking **d**irectory. Prints the full path to the directory you are in, starting with the root directory

```
$ pwd
/home1/05574/<username>
```

**absolute** path: preceded by '/', full address to a location/directory

**relative** path: path from where you are to a location/directory

2. What shell am I using?

To determine which shell you are currently using, you can type the `echo` command followed by the system environment variable `$SHELL` as follows:

```
$ echo $SHELL
/bin/bash
```

2. What's in this directory? Use of the 'ls' **ls**t command

```
$ ls
test1.txt  test2.txt  test3.txt
```

3. I want to go somewhere…use 'cd' **c**hange **d**irectory command

-I want to go home: 'cd ~'

```
[zpb4@hopper mv_swm]$ cd ~
[zpb4@hopper ~]$
```

-I want to go to target directory X: 'cd ./path-to-directory X' or 'cd path-to-directory X'

-'cd ../' to go up one directory

```
[zpb4@hopper ~]$ cd ./mv_swm/out/
[zpb4@hopper out]$
```

```
[zpb4@hopper ~]$ cd mv_swm/out/
[zpb4@hopper out]$
```

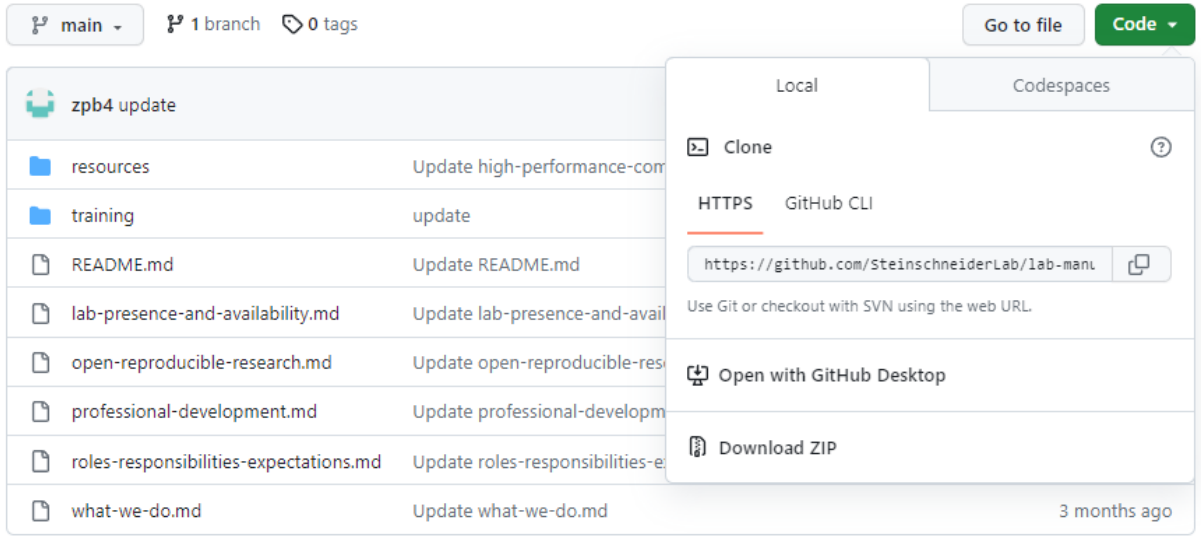
*Check out the documentation for each of these commands and 'tab completion' familiarization

# A little Git primer

Now we are going to clone the Steinschneider lab manual training repository from GitHub

1. Navigate to an open directory using cd command
2. Go to https://github.com/SteinschneiderLab/lab-manual
3. Get the clone link '*git-clone-link*'
4. In your terminal execute:

'git clone *git-clone-link*'




You will now have a copy of the shared repository on your local machine

We will learn more about Git in a later training. Now that you have this Git Repository on your local machine, a simple 'git pull' operation from this local 'lab-manual' git repository will update your local repo to the latest and greatest!

# A little Git primer

Note, for the purposes of this training, we will be using command line operations in a git repository. This is okay in this situation because we are using git in a solely ‘pull’ mode and we aren’t going to ‘push’ anything back to the git repository.

Importantly, git has it’s own set of command line type operations (e.g. ‘git mv’ instead of ‘mv’) that maintain internal accounting of files and directories in the git architecture. Using command line tools in git can cause problems…we’ll discuss later in the git training.

# Manipulating files & directories

`mkdir` – **mak**e a new **dir**ectory of the given name, as permissions allow.

```
$ mkdir Newdir
```

`mv` – **mov**e files, directories, or both to a new location.

```
$ mv file1 Newdir
```

- This can also be used to rename files:

```
$ mv file1 file2
```

- Use wildcards like `*` to move all files of a specific type to a new location:

```
$ mv *.c ../CodeDir
```

- You can always verify that the file was moved with `ls`.

`cp` – **cop**y files, directories, or both to a new location.

```
$ cp file1 ~/
```

- You can give the copy a different name than the original in the same command:

```
$ cp file1 file1_copy
```

- To copy a directory, use the `-r` option (**r**ecursively). In this case, both the source and the destination are directories, and must already exist.

```
$ cp -r Test1 ~/testresults
```

`rm` – **rem**oves files or directories ***permanently*** from the system.

```
$ rm file1
```

- **Note:** Linux does not typically have a "trash bin" or equivalent as on other OS, so when you issue `rm` to remove a file, it is difficult to impossible to recover removed files without resorting to backup restore.
- With the `-r` or `-R` option, it will also remove/delete entire directories recursively and permanently.

```
$ rm -r Junk
```

- Avoid using wildcards like `*`. For instance, the following will remove all of the files and subdirectories within your current directory, so ***use with caution***.

```
rm -r *
```

- To remove an empty directory, use `rmdir`

`touch` – changes a file's modification timestamp without editing the contents of the file. It is also useful for creating an empty file when the filename given does not exist.

Wildcard operator *
A way of saying ‘select everything subject to certain constraints’
So, if I want to only modify .txt files, I can specify ‘*.txt’
…many other uses of this feature

# Manipulating files & directories exercise1

1. Navigate to your ‘lab-manual’ repository and then to ‘training/command_line’ directory via the cd command
2. There is already a ‘data’ directory; create a new directory called ‘output’ using mkdir command
3. Copy the ‘copy.me.txt’ files from the ‘data’ directory to the ‘output’ directory using cp command; try using both **absolute** and **relative** path methods
4. Now move the ‘move.me.txt’ files between the directories; note that the mv command deletes the original file
5. Remove one of your copied files with the rm command
6. Make an unused directory ‘junk’ and then remove it with rm and the ‘-r’ recursive flag

# Manipulating files & directories exercise2

1. Remove all the .sh files (junk1.sh, junk2.sh) from the data directory using rm and the wildcard operator
2. Try rename command (may only work in actual LINUX systems)

   rename item-to-change change-to filex

   $ rename _ . *.txt
3. Any other fun tricks from the group?

# Shell scripting – Text editors

So, we've played around with some direct shell commands that are useful. However, to use command line effectively (e.g. for remote HPC computing), we need to be able to build and run shell scripts (bash scripts are a form of shell script). In concept, this is no different than writing a script in R or Python, except that in the case of remote computing, we have no nice IDE (like Rstudio) to manipulate the scripts.

We have to interact with the script files in either .txt or .sh format via a text editor. Most HPC will have both **vim** and **emacs**, we will focus on some very basic **vim** operations today

# Shell scripting – Vim familiarization


Source: vim.org

**Basic Functions**

- **Open** an existing file by entering `vim` in the shell followed by the name of the file.
- **Create** a new file in the same way as opening a file by specifying the new filename. The new file will not be saved unless specified.
- **Save** a file that is currently open by entering the `:w` command.
- **Quit** an open file by entering the `:q` command. If you have made any edits without saving, you will see an error message. If you wish to *quit without saving* the edits, use `:q!`.
- **Save and Quit** at the same time by combining the commands: `:wq`.
- **Edit** the file by entering insert mode to add and remove text. Entering into normal mode will allow you to easily copy, paste, and delete (as well as other functionality).
- **Cancel** a command before completely entering it by hitting `Esc` twice.

**Insert Mode**

When you first open a document, you will always start in normal mode and have to enter insert mode. To enter insert mode where the cursor is currently located in the file, press the letter `i` or the `Insert` key. Additionally, you can press the letter `a` (for append) if you would like to enter insert mode at the character after the cursor. To exit insert mode, press the `Esc` key. When in insert mode, `-- INSERT --` will be visible at the bottom of the shell. Navigation in insert mode is done with the standard arrow keys.

**Normal (Command) Mode**

Vim starts in normal mode, and returns to normal mode whenever you exit another mode. When in normal mode, there is no text at the bottom of the shell, except the commands you are entering.

***Navigation***

Navigation in normal mode has a large number of shortcuts and extra features, which we will only cover some of here. Basic movement can be done using the arrow keys or using the letter keys in the following table:

| Move | Key |
|---|---|
| ← | h |
| ↓ | j |
| ↑ | k |
| → | l |

The benefits of using the alternate keys is that you do not have to move your hand back-and-forth to the arrow keys while in this mode, and can more effectively enter Vim commands (once you are practiced). Some other examples of navigation shortcuts include:

- Move to the **beginning of the line**: `0`
- Move to the **end of the line**: `$`
- Move to the **beginning of the next word**: `w` This can also be used with a number to move multiple words at once (i.e. `5w` moves 5 words forward).
- Move to the **end of the current word**: `e` This can be used with a number in the same way that `w` can to move multiple words at once.

…and lots more shortcuts, tricks, fun-facts, etc

See a Vim tutorial if you want to get super fancy, my experience is that the working knowledge contained on this page is sufficient for non software engineers

# Working with Vim1

1. First, navigate to your ‘output’ directory using cd
2. Vim can be used to edit an existing file or create a new file with the same command: vim myfile
3. Create a new .txt file called ‘myfirstvim.txt’ with Vim
4. Go over normal versus insert mode and basic Vim stuff.

***Note: You can’t edit the file in normal mode and you can’t do anything else in your command line until you exit Vim with the :q or :wq command from normal mode***

5. Write a one line script to print something fun to the terminal like ‘happy happy joy joy’ or ‘none shall pass’ using the echo command in the script
6. Save and quit this file with the :wq command and then run it using the source command

# Working with Vim2 – simple for loop script

Here, we will create a very simple for loop in command line to do a task. One simple construction of a for loop is shown below. Importantly, this construction uses the $ operator which is incredibly useful in more complex tasks. This operator inserts the thing after it into the command arguments. In the example below, this ‘thing’ is the index ‘i’ in the for loop.

```
start=1
end=10

for ((i=start; i<=end; i++))
do
        echo $i spot in for loop
done
```

# Working with Vim2 – simple for loop script

1. Create a new .sh (shell script) file in Vim called ‘for.loop.sh’
2. Write the for loop script below, being mindful that command line is very sensitive to syntax

```
start=1
end=10

for ((i=start; i<=end; i++))
do
        echo $i spot in for loop
done
```

3. Save and quit vim and run your script with the source command

# Make a script to run a simple R script

One of the nice features of shell scripts is that they can be used to run any other program available in the OS. In this simple example, we will run a pre-constructed R script in the data directory using the Rscript command and a simple shell script to execute that command.

1. Create a new shell script in vim called ‘run.R.sh’; in that script, enter the Rscript command to execute the ‘my.Rscript.R’ script in the data folder
2. Run the script, you should see the script printout the ‘print’ block in the R script
3. Try modifying the R script with vim to save some output from this script to the output directory

# Review some more complex bash scripts

The purpose of this training is of course to just familiarize with shell script. There are a couple of scripts in the data folder (interactive_example with some more advanced examples of bash scripts used for processing CMIP6 data on the Cheyenne supercomputer. These scripts allow us to pull only desired elements of data from an absolutely massive dataset and reconfigure them to a size suitable for smaller computing systems.

This is one of many reasons to learn some basics of command line! We will cover examples (albeit simpler) in the spatial data training module.

# High level to command line operations

High level languages like R will generally have implementations to perform command line type operations

R example

R code below uses ‘dir.exists’, ‘unlink’, and ‘dir.create’ commands to identify if a named directory exists, if so remove it and its contents (unlink, recursive = T), then create a new directory to store generated files

```
if(dir.exists(paste('~/syn_forecast_test/output/',outfile_hefs,'/feather',sep=''))==T){
  unlink(paste('~/syn_forecast_test/output/',outfile_hefs,'/feather',sep=''),recursive = T)}
dir.create(paste('~/syn_forecast_test/output/',outfile_hefs,'/feather',sep=''))
```

That’s the extent will go into it for now; you may find some utility in this kind of functionality in the future

# Resources

- Learning Linux (linkedin.com)
- A very quick intro to Linux (linkedin.com)
- Cornell Virtual Workshop: Overview

# Objectives - closeout

- Familiarize with basic LINUX terminology and derivations
- Familiarize with basic LINUX shell command structure and commands
- Introduce some LINUX shell operations useful for later training (HPC, git, spatial data)
- Introduce some useful resources, motivate the utility of LINUX proficiency, pave the way for self study

# Additions?

- More complex scripting examples?
- Some basic input/output functionality examples…ie use command line to specify an input to an R script for example?