MESA Course Notes

Author: Joshua Aiken

2025

Contents

1 Linux

1.1 Absolute Basics

The command line, also known as the terminal or shell, is a powerful way to interact with a Linux system. It allows users to execute commands, manipulate files, and run programs with greater control than a graphical user interface.

Opening the Terminal

On a *native* Linux system, you can open the terminal via the applications menu, or by pressing `Ctrl + Alt + T`. We will not be running Linux naively, however. Instead, we `ssh` onto a server that runs Linux. To do this, we will use Putty (Windows) or use an `ssh` command from some other command line native to the OS that we are using (Command Prompt or WSL (Windows Subsystem for Linux) for Windows, Terminal for MacOS, or even just the regular terminal from Linux).

Basic Command Structure

A typical command has the following structure:

```
command [options] [arguments]
```

For example:

```
ls -l /home/user
```

This lists the contents of the `/home/user` directory in long format.

Essential Commands

Here are a few commands that are essential for getting started:

- `pwd` – Print the current working directory.
- `ls` – List files and directories in the current directory.
  - `-l` (long listing), `-a` (include hidden files)
- `cd [directory]` – Change the current directory.
  - `cd ..` moves up one directory.
  - `cd` with no arguments returns to the home directory.
- `mkdir [name]` – Create a new directory.
- `touch [file]` – Create a new empty file.
- `cp [source] [destination]` – Copy files or directories.
- `mv [source] [destination]` – Move or rename files or directories.
- `rm [file]` – Remove (delete) a file.

  - Use `rm -rf [directory]` to remove a directory and its contents.
  - **Be careful**: deleted files are not moved to a trash bin.
- `man [command]` – Show the manual page for a command (e.g., `man ls`).

Tab Completion and Command History

- Press `Tab` to auto-complete file and directory names.
- Use the `Up/Down` arrow keys to scroll through your command history.

File Paths

- Absolute paths start from the root directory, e.g., `/home/user/docs`.
- Relative paths are based on your current directory, e.g., `../` refers to the parent directory.
- The tilde `~` refers to the home directory.

Getting Help

- `man [command]` – Displays the manual for a command.
- `[command] --help` – Shows (typically) usage instructions for many commands.

Examples and Environment Setup

1. Copy the text editor program `micro` into your `~/bin/` (binaries - or executable programs) directory:

   ```
   cp ~/projects/def-lnelson/jaiken_public/micro/micro ~/bin/micro
   ```

   This will give us a nicer command line text editor to use (you can still use `nano`, `vim`, or `emacs` if you would prefer). Placing since this program is placed in the binaries directory, we can invoke it without needing to specify it's path (note that this will *create* the file `filename.txt` if it doesn't exist):

   ```
   micro filename.txt
   ```

   To save a file, you can press `ctrl + s` and to quit `micro`, press `ctrl + q`

2. Copy the full MESA tutorials directory in the projects directory into your home directory:

   ```
   cp -r ~/projects/def-lnelson/jaiken_public/MESA_tutorial ~/.
   ```

   This will be where we run all of our examples and practice.

3. Change your directory to the `MESA_tutorial` directory and verify the contents:

```
cd ~/MESA_tutorial
ls
```

There should be four directories: `bash/`, `python/`, `MESA/`, and `SLURM/`.

4. Now enter the `bash/` directory:

   ```
   cd bash
   ```

   This directory contains two directories: `practice/` and `examples/`. The `practice/` directory is where you will write your own scripts and the `examples/` directory holds the example scripts that I have written.

5. Finally, enter the `examples/` directory and run the following command:

   ```
   ./hello_world.sh
   ```

   If you see the following printed to the screen `Hello World!` then you have successfully ran a bash script.

Practice Questions

1. - Use the command line to create a directory called `linux_practice` in your bash practice directory.
   - Enter that directory and create a new directory called `inside`.
   - Enter that directory and enter `pwd` into the terminal. The output should be

     `/home/username/MESA_tutorial/bash/practice/linux_practice/inside/`

2. - Inside the directory created in the previous question (`linux_practice`), create an empty file called `test1`.
   - Write `hello world!` in this file with the `echo` command.
   - Show the file's contents with the `cat` command.

3. - Create a new directory called `nest1`, then create a second directory `nest2` within it (without entering the directory).
   - Copy the `test1` file into the `nest1` directory (keep the same name), and create a new, empty file `test2` in `nest2`.
   - Make a copy of the directory `test1` (and all of its contents) called `test1_copy`.

4. - Create a new empty file called `hello_world.py`.
   - Using `micro`, type the following Python script into the previous file

     ```python
     print("Hello World!")
     ```

     Listing 1: `hello_world.py`

   - Save and exit the file (`ctrl+s` to save and `ctrl+q` to quit).
   - Run the file by typing `python3 hello_world.py` into the command line.

1.2 Bash

This section introduces foundational concepts needed to write basic Bash scripts. Bash is a Unix shell commonly used for automating tasks, managing files, and processing data on Linux systems. These topics will help students understand how to create and run simple yet powerful scripts.

What is Bash?

Bash (Bourne Again SHell) is a command processor that typically runs in a terminal window. It allows users to interact with the system by typing commands. Bash scripts are plain text files that contain a sequence of commands to be executed automatically. They are widely used for automation and task management on Unix-like systems.

Shebang Line

A Bash script should begin with a special line called the *shebang* to indicate which interpreter should execute the file:

```
#!/bin/bash
```

This ensures that the system uses the correct shell to interpret the script, regardless of the user's default shell.

Script Permissions

Most likely, after writing and saving a Bash script, you will get an error when you try to run it. This is because it is not *executable* yet. Before a Bash script can be executed, it must have the appropriate permissions. Use the following command to make a script executable:

```
chmod +x script.sh
```

Then, run the script using:

```
./script.sh
```

Variables

Variables store values that can be reused throughout a script. To define a variable, use the syntax:

```
name="Alice"
```

To access the variable's value, prefix it with a dollar sign:

```
echo $name
```

Note that there should be no spaces around the equals sign when assigning a value.

Quoting

Quoting controls how the shell interprets text. Single quotes (`'...'`) preserve the literal value of characters, including variables, while double quotes (`"..."`) allow variable and command substitution. For example:

```
echo '$name'   # prints: $name
echo "$name"   # prints: Alice
```

Comments

Comments begin with a hash symbol (`#`) and are ignored by the shell. They are useful for explaining code:

```
# This is a comment
```

Try it out:

1. Enter the following commands into the terminal:

   ```
   hello='Hello'
   world='world!'
   echo "$hello $world"
   ```

   The output should be

   ```
   Hello world!
   ```

2. Write a Bash script `variables.sh` that performs these three commands and verify that the output is the same.
3. Comment out the `world='world!'` line. What do you expect the script will output? Run the script and verify.

Conditionals

Bash provides `if` statements to execute code based on conditions. The general syntax is:

```
if [ condition ]; then
  commands
elif [ condition ]; then
  commands
else
  commands
fi
```

Conditions can test file properties, string comparisons, or numeric relationships.

Loops

Loops allow repetition of commands. A *for loop* iterates over a list of values:

```
for var in list; do
  commands
done
```

A *while loop* continues executing as long as a condition is true:

```
while [ condition ]; do
  commands
done
```

Command Substitution

Command substitution allows the output of a command to be used as part of another command or stored in a variable:

```
today=$(date)
echo "Today is $today"
```

This is particularly useful for capturing dynamic values within a script.

Arithmetic Operations

Bash supports simple integer arithmetic using double parentheses:

```
count=5
((count = count + 1))
echo $count
```

You can also use the `let` command or the `expr` utility for arithmetic, though `((...))` is more commonly used.

Try it out:

1. Create a directory `some_files` and three files `file1.txt`, `file2.txt`, and `file3.txt` within the directory. Then, enter the following into a Bash script (in the same directory that `some_files/` can be found):

   ```
   #!/bin/bash

   for file in some_files/*.txt; do
       echo $file
   done
   ```

What should the output of this script be? Check it by running it (it needs to be ran in the same directory that `some_files` is in)

2. Write a script that counts the number of text files in the `some_files` directory created in the previous exercise.

Input and Output

Bash provides several mechanisms for handling input and output:

- Use `read` to get input from the user:

```
read name
```

- Redirect output to a file with > (overwrite) or >> (append).
- Redirect input from a file using <.
- Use a pipe (|) to send the output of one command into the input of another:

```
ls | grep ".txt"
```

Practice Questions

1. Write a bash script that outputs `Hello World!` to the screen (no need to use variables).
2. Write a bash script that creates ten directories with indexed names: `dir1`, `dir2`, ..., `dir10`.
3. What does the following bash script do?

```
#!/bin/bash

CURRENT_DIR=$(pwd)

DIR1=funny_dir
DIR2=funny_dir_2

mkdir $DIR1
cd $DIR1
mkdir $DIR2
cd $DIR2

pwd

cd $CURRENT_DIR

rm -d $DIR1/$DIR2
rm -d $DIR1

cd /dev
pwd
```

Listing 2: `cd_script.sh`

On line 20, the script changes the directory to the `/dev` directory. When the script ends, is that the directory that the terminal is now in? Check it with the `pwd` command.

4. Use Google to figure out what the following script does (look up any commands you haven't seen before):

```
1 #!/bin/bash
2
3 sleep_time=5
4
5 echo "I am tired, time for bed."
6 sleep $sleep_time
7 echo "What a great $sleep_time seconds rest!"
```

Listing 3: `sleep_script.sh`

Research how to use the `time` command and use it to check your understanding of the script.

```
1 #!/bin/bash
2
3 echo 'hello world!'
```

Listing 4: `hello_world.sh`

2 Python

2.1 What is Python?

Python is a high-level, general-purpose programming language that is widely used for scripting, automation, data analysis, and scientific computing. It is designed to be readable and easy to learn, making it an excellent choice for beginners as well as professionals.

Interpreter vs Compiler

Python is an *interpreted* language, meaning that code is executed line-by-line by an interpreter, rather than being fully compiled into machine code before execution (as is the case for languages like C or Fortran). This allows for interactive development and rapid prototyping, but often comes at the cost of slower execution compared to compiled programs.

The standard Python interpreter (commonly referred to as `CPython`) reads and executes Python source files directly. Other implementations exist, such as `PyPy` (a just-in-time compiling interpreter), `Jython` (for the Java Virtual Machine), and `IronPython` (for .NET), but `CPython` is by far the most commonly used. Note that the name "`CPython`" refers to the fact that that the interpreter is written in C - you do not need to know anything about C to use it.

Why Use Python?

Python is widely appreciated for its clear syntax, vast ecosystem of third-party libraries, and broad community support. It is especially well-suited for:

- Rapid prototyping and scripting
- Data analysis and scientific computing* (with libraries such as NumPy, SciPy, and pandas)
- Automation of repetitive tasks and workflows
- Teaching and learning programming concepts

Its versatility and ease of use make it a popular language in many fields, including physics, astronomy, biology, engineering, and finance.

When *Not* to Use Python

Despite its strengths, Python is not always the right tool for every job. You might want to choose another language if:

- Performance is critical and the application is CPU-bound or GPU-bound (e.g., in hardcore numerical computations)
- You need tight control over memory usage or hardware (e.g., embedded systems)
- You are deploying to a system where the Python interpreter is not available or not permitted

In such cases, compiled languages like (modern) Fortran, C, or C++ (maybe even Rust) may be more appropriate. However, Python can often be combined with these compiled languages using interfaces such as `F2PY` or `Cython`.

Ways to Run Python Code

Python code can be run in several different ways, depending on the use case and development environment:

- **Interactive mode:** By launching the Python interpreter from the terminal using the command `python` or `python3`. This allows execution of individual commands in a read-eval-print loop (REPL).
- **Script mode:** By writing code in a `.py` file and running it with `python filename.py`. This is the most common method for running reusable programs.
- **Notebooks:** Using tools like Jupyter Notebook or JupyterLab, which allow code to be run in cells with rich output, including plots and formatted text. This is widely used in data science and research.
- **Integrated Development Environments (IDEs):** Applications like VS Code, PyCharm, and Thonny provide graphical environments for writing, running, and debugging Python code.

These flexible execution options make Python highly adaptable for a variety of workflows and development styles.

2.2 The Basic Stuff

Before working with more complex programs, it is essential to understand some of the basic building blocks of Python code. These include how to define and use variables, how to write comments, how Python handles the end of statements, and how to display output using the `print()` function.

Variables

In Python, variables are created by simply assigning a value to a name using the = operator. There is no need to declare the type of the variable beforehand — Python figures it out automatically based on the value assigned (This is an additional aspect of Python that makes it slow).

```
x = 10
name = "Alice"
pi = 3.14159
```

Variable names can contain letters, numbers, and underscores, but must not begin with a number. They are also case-sensitive, so `score` and `Score` would refer to different variables.

Comments

Comments are used to add notes or explanations to your code. In Python, anything following a hash mark # on a line is ignored by the interpreter:

```
# This is a comment
x = 42  # This is also a comment
```

Comments are helpful for documenting what your code does, especially in more complex scripts.

Statement Termination

Unlike some programming languages (such as C, Java, and Rust) where each statement must end with a semicolon, Python uses newlines to mark the end of a statement. In most cases, each statement simply ends at the end of the line:

```
x = 3
y = 4
z = x + y
```

Semicolons are technically allowed to separate multiple statements on a single line, but their use is discouraged in idiomatic Python (i.e. multiple statements on a single line makes code harder to read and thus is discouraged):

```
x = 3; y = 4; print(x + y)  # This works, but it's not recommended
```

The `print()` Function

The built-in `print()` function is used to display output in the terminal. It can take one or more arguments and prints them with spaces in between:

```
print("Hello, world!")
print("The value of x is", x)
```

The `print()` function automatically adds a newline at the end of its output. You can suppress this by adding the `end` keyword argument:

```
print("Hello", end=" ")
print("world!")
# Output: Hello world!
```

You can also use formatted strings (f-strings) to embed variable values inside strings:

```
name = "Alice"
age = 30
print(f"{name} is {age} years old")
```

This makes it easy to combine variables and text in readable output.

Special Characters in Strings

When using the `print()` function, you can include special characters inside string literals by using escape sequences. These are preceded by a backslash (\) and represent characters that are not easy to type directly or have special meaning.

Common examples include:

```
print("Line 1\nLine 2")     # \n starts a new line
print("Tab\tseparated")     # \t inserts a tab
print("A backslash: \\")    # \\ prints a single backslash
print("He said: \"Hello\"") # \" allows quotes inside strings
```

These sequences help format output more clearly, especially when printing multi-line or structured text.

Calling Functions

In Python, a function is a reusable block of code that performs a specific task. To *call* a function means to execute it. Functions may take input arguments and may return output values.

A function call consists of the function's name followed by parentheses. If the function takes arguments, they are placed inside the parentheses and separated by commas.

```
print("Hello")           # Calling the built-in print() function
len("Python")            # Returns the length of the string
max(3, 7, 2)             # Returns the maximum of the given values
```

Python provides many built-in functions like `print()`, `len()`, `type()`, `max()`, and `abs()`. You can also use functions from imported modules (this will be discussed more later):

```
import math
math.sqrt(16)            # Calls the sqrt function from the math module
```

For a complete list of Python's built-in functions, see the official documentation at https://docs.python.org/3/library/functions.html.

Lists and Tuples

Lists and tuples are two of the most commonly used data structures in Python for storing collections of items.

Lists are ordered, mutable (changeable) sequences of elements. They are defined using square brackets:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
```

You can modify the contents of a list after it is created:

```
fruits[0] = "orange"  # Replace "apple" with "orange"
```

Tuples are similar to lists, but they are immutable — once created, their contents cannot be changed. Tuples are defined using parentheses:

```
coordinates = (3, 4)
colors = ("red", "green", "blue")
```

Tuples are often used when you want to ensure that a collection of values should not be altered, such as fixed coordinates or configuration settings.

Indexing

Both lists are *indexed*, meaning you can access individual elements by their position in the sequence. **Python uses zero-based indexing**, so the first element is at index `0`, the second at `1`, and so on:

```
print(fruits[0])   # Outputs: orange
print(colors[2])   # Outputs: blue
```

You can also use negative indices to count from the end of the sequence:

```
print(fruits[-1])  # Outputs the last element: cherry
print(fruits[-2])  # Outputs the second-to-last element: banana
```

You can also use index *slices* to retrieve *parts* of a list:

```
x = [1,2,3,4,5]
print(x[0:3])   # Outputs [1,2,3]. Note the slice is from 0 to 3, excluding 3
```

Attempting to access an index that is out of range will result in an `IndexError`.

Control Flow: `if` Statements and Loops

Control flow allows your program to make decisions and repeat actions. This is essential for writing dynamic and responsive code.

Conditional Statements (`if`)

The `if` statement allows you to execute code only when a certain condition is true. It can be extended using `elif` (else if) and `else` for alternative paths:

```
x = 10

if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

Note that the condition is followed by a colon, and the indented block underneath it defines the code that will be executed if the condition is true. Python uses indentation (usually 4 spaces) to mark blocks of code, rather than braces or keywords.

`for` Loops

A `for` loop is used to iterate over a sequence such as a list, tuple, or string. It repeats a block of code once for each item in the sequence:

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

You can also use `range(n)` to loop over a sequence of numbers:

```
for i in range(5):
    print(i)  # Prints numbers 0 to 4
```

`while` Loops

A `while` loop keeps executing as long as a specified condition remains true:

```
x = 0
while x < 5:
    print(x)
    x += 1  # Same as x = x + 1
```

This means that `while` can lead to *infinite* loops:

```
x = 0
while x < 5:
    print(x)
    x -= 1  # x will always be less than 5 !
```

On the other hand, `for` loops should *never* lead to infinite loops.

Breaking Out of Loops

You can use the `break` statement to exit a loop early, and `continue` to skip the rest of the current iteration and move to the next one:

```
for i in range(10):
    if i == 5:
        break    # Exit the loop when i is 5
    print(i)
```

```
for i in range(5):
    if i == 2:
        continue  # Skip printing 2
    print(i)
```

These control flow tools are the foundation for making decisions and writing logic in Python programs.

Defining Functions

Functions allow you to group reusable blocks of code under a single name. You can define your own functions in Python using the `def` keyword, followed by the function name, parentheses (which may include parameters), and a colon. The function body is indented beneath the definition.

```
def greet():
    print("Hello!")
```

This function can be called later using its name followed by parentheses:

```
greet()
```

Functions can also take arguments and return values:

```
def square(x):
    return x * x

result = square(4)  # result is now 16
```

Using Type Hints

Python is a dynamically typed language, which means that types do not need to be declared. However, it is considered good practice—especially in scientific or collaborative code—to use *type hints* to indicate the expected types of function arguments and return values. This makes code easier to read, maintain, and debug.

As an example, take this code:

```
def add(a,b):
    return a + b
```

This function is obviously meant to be used with numeric data, but what happens when you call this function with `add("hello","world")`? The code doesn't crash, but it returns the value `"helloworld"` (The '+' operator in Python is addition for numeric types and concatenation for strings).

Here is the same `square` function with type hints:

```
def square(x: float) -> float:
    return x * x
```

Note that Python **does not enforce** these types at runtime—they are primarily for the benefit of developers and tools such as linters and editors.

Importing Modules and Functions

Python comes with a large standard library of modules that provide useful functions and constants. Python also has a large array of external libraries that will also be useful and so it is important to know how to access the code in these libraries (installation of external libraries will not be covered as it is complicated to do so on the Digital Research Alliance of Canada clusters without causing problems). To access code from another module, you use the `import` statement.

One of the most commonly used standard modules is `math`, which includes mathematical functions and constants. We will use this module as the primary example (the documentation for this module can be found at https://docs.python.org/3/library/math.html.

Importing the Whole Module You can import the entire module using the `import` keyword. Then, you access its contents using dot notation:

```
import math

print(math.sqrt(16))        # Square root of 16
print(math.pi)              # The value of pi
print(math.sin(math.pi/2))  # Sine of 90 degrees (in radians)
```

Importing Specific Functions Alternatively, you can import specific functions or constants directly using the `from ... import ...` syntax:

```
from math import sqrt, pi

print(sqrt(25))    # No need to prefix with math.
print(pi)
```

This can make your code shorter, but it may reduce clarity if many names are imported from different modules.

Using Aliases You can also assign an alias to a module or function using the `as` keyword. This is useful to shorten long module names or avoid name conflicts:

```
import math as m

print(m.cos(m.pi))  # Cosine of pi
```

Why Use Modules? Modules help organize code into reusable components and give you access to tools without having to write them from scratch. Python includes many built-in modules (like `math`, `os`, and `random`), and you can also install third-party modules using tools like `pip`.

Understanding how to import and use modules is essential for leveraging the full power of Python's ecosystem.

Reading and Writing to Files

Reading from and writing to files is a common task in many Python programs. The built-in `open()` function is used to work with files. It takes the filename and the mode as arguments. Common modes include:

- `'r'` – read (default mode)
- `'w'` – write (creates a new file or overwrites an existing one)
- `'a'` – append (writes to the end of the file if it exists)

Writing to a File To write to a file, open it in `'w'` or `'a'` mode. Use the `write()` method to write text to the file.

```
with open("example.txt", "w") as file:
    file.write("This is a line of text.\n")
    file.write("Here is another line.")
```

The `with` statement ensures the file is properly closed after writing, even if an error occurs.

Reading from a File To read the contents of a file, open it in `'r'` mode and use methods such as `read()`, `readline()`, or `readlines()`:

```
with open("example.txt", "r") as file:
    contents = file.read()
    print(contents)
```

This reads the entire file as a single string. You can also read the file line by line:

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # .strip() removes the newline character
```

2.3 Some Useful Python Modules

To make the most of Python, we often rely on additional modules that provide specialized functionality for tasks such as numerical computing, data analysis, and plotting.

On the cluster, we do not have permission to install Python packages system-wide using `pip` (we can install packages for just our user using pip, but this will likely lead to problems). Instead,

we must either use the modules that are already provided in the system environment, or set up isolated installations using virtual environments.

For this course, we will make use of a pre-configured environment module called the **SciPy stack**, which includes a wide range of commonly used scientific Python packages. These include `numpy` for numerical computing, `scipy` for scientific computing, `matplotlib` for plotting, `pandas` for data manipulation, and others.

To access these modules on the cluster, you must load the SciPy stack environment with the following command:

```
module load scipy-stack
```

Once the module is loaded, you can run Python as usual, and these additional libraries will be available for import in your scripts.

If you need to use a Python package that is not included in the SciPy stack, you **cannot** simply run `pip install` as you would on your personal machine (even on your personal machine this will lead to dependency issues - the Python module versioning and dependency requirements are a complete nightmare). Instead, you must create a virtual environment in your user space and install the package there. While we will not go into detail about virtual environments in this course, it's important to be aware that this is the proper method for working with additional Python modules on the cluster (and is good practice in general).

For our purposes, the SciPy stack provides everything we need.

Working with the `os` Module

The `os` module is part of Python's standard library and provides functions for interacting with the operating system in a way that works across different platforms (Linux, macOS, Windows). It is particularly useful when working with file paths, directories, and environment variables—especially when automating tasks or integrating Python with the command line.

Accessing Environment Variables

You can use `os.environ` to access environment variables:

```
import os

home_dir = os.environ["HOME"]
print("My home directory is:", home_dir)
```

If you are unsure whether a variable exists, it's safer to use the `get` method:

```
username = os.environ.get("USER", "unknown")
```

Working with Directories

You can get the current working directory using:

```
cwd = os.getcwd()
print("Current directory:", cwd)
```

You can change directories using:

```
os.chdir("/path/to/another/directory")
```

You can also list the contents of a directory:

```
files = os.listdir(".")
print("Files in current directory:", files)
```

Creating and Removing Directories

To create or remove directories:

```
os.mkdir("new_folder")       # Create a single folder
os.makedirs("a/b/c")         # Create nested directories
os.rmdir("new_folder")       # Remove an empty folder
```

For removing non-empty directories, use the `shutil` module instead.

Working with File Paths

The `os.path` submodule contains tools for safely constructing and manipulating file paths:

```
path = os.path.join("data", "file.txt")
print("Constructed path:", path)

exists = os.path.exists(path)
is_file = os.path.isfile(path)
is_dir = os.path.isdir(path)
```

Using `os.path.join` is better than manually combining paths with slashes, because it works correctly on all operating systems.

The `os` module is an essential tool for writing portable and flexible Python scripts that interact with the file system or depend on the runtime environment.

Working with the `pandas` Module

The `pandas` module is a powerful library for data manipulation and analysis. It is widely used in scientific computing, data science, and machine learning. The core data structures in `pandas` are the `Series` (1D) and `DataFrame` (2D), which allow you to store, manipulate, and analyze structured data efficiently.

To use `pandas`, you must first import it (commonly using the alias `pd`):

```
import pandas as pd
```

Creating DataFrames

You can create a `DataFrame` from a dictionary or a list of lists:

```
data = {
    "name": ["Einstein", "Eddington", "Kepler"],
    "age": [146, 142, 453]
}

df = pd.DataFrame(data)
print(df)
```

This will produce a table-like structure with columns labeled `name` and `age`.

Reading and Writing Data Files

A major strength of `pandas` is its ability to read and write various data formats, especially CSV (comma-separated values) files:

```
df.to_csv("output.csv", index=False) # Write DataFrame to CSV
df_new = pd.read_csv("output.csv")         # Read CSV into a DataFrame
```

The `read_csv` function automatically parses headers and infers data types.

Accessing and Modifying Data

You can inspect the top of the dataset using:

```
print(df.head())   # First 5 rows
```

Accessing a column is straightforward:

```
print(df["name"])
```

You can also filter rows based on conditions:

```
pre_Newton = df[df["age"] > 382]
```

New columns can be created or modified as follows:

```
df["age_plus_one"] = df["age"] + 1
```

Summary Statistics

pandas makes it easy to get a quick statistical summary of your data:

```
print(df.describe())
```

You can also compute specific statistics:

```
mean_age = df["age"].mean()
```

The pandas library is an essential tool for any kind of structured data analysis in Python. It enables you to work with tabular data efficiently and is particularly well-suited to tasks involving reading, filtering, transforming, and analyzing data from external sources. There is much more to the pandas library than I have described here, but this should be a good starting point for anyone who is interested.

Working with the matplotlib Library

matplotlib is Python's most widely used library for creating static, animated, and interactive visualizations. It is highly customizable and integrates well with other libraries like numpy and pandas. For basic plotting, we use the pyplot module, which is typically imported using the alias plt:

```
import matplotlib.pyplot as plt
```

Plotting Data

The most basic type of plot is a line plot, which can be created from lists or numpy arrays:

```
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

plt.plot(x, y)
plt.title("Example Plot")
plt.xlabel("x values")
plt.ylabel("y values")
plt.show()
```

The plt.show() function opens a window or displays the plot inline (e.g., in a Jupyter notebook).

Scatter Plots and Histograms

You can also easily create other types of plots:

```
# Scatter plot
plt.scatter(x, y)
plt.show()

# Histogram
import numpy as np
data = np.random.randn(1000)
plt.hist(data, bins=30)
plt.show()
```

Saving Figures

You can save plots to image files (PNG, PDF, etc.) using:

```
# ... create plot
plt.savefig("myplot.png")
```

This is useful when running scripts on a remote cluster or when generating plots for a report.

Multiple Plots and Subplots

You can overlay multiple plots or create subplots:

```
# Overlaying multiple lines
plt.plot(x, y, label="squares")
plt.plot(x, [i**3 for i in x], label="cubes")
plt.legend()
plt.show()

# Creating subplots
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(x, y)
ax2.plot(x, [i**3 for i in x])
plt.show()
```

`matplotlib` is an essential tool for visualizing data. Whether you're exploring datasets, debugging computations, or preparing publication-quality figures, `matplotlib` provides the flexibility and control to produce a wide range of visualizations.

Other Scientific Modules

In addition to `pandas`, `matplotlib`, and `os`, there are several other Python modules that are widely used in scientific computing. While we will not cover these in detail during this course, it is helpful to be aware of them and their typical uses:

- `numpy` – The foundational library for numerical computing in Python. It provides efficient array and matrix operations, mathematical functions, and tools for working with large datasets in a vectorized way. Many other scientific libraries depend on `numpy`.
- `scipy` – Builds on `numpy` to provide advanced numerical routines for tasks such as integration, optimization, root-finding, signal processing, and linear algebra.
- `sympy` – A symbolic mathematics library useful for algebraic manipulation, calculus, equation solving, and working with mathematical expressions analytically.
- `astropy` – A library developed for astronomy and astrophysics. It includes tools for handling units and constants, celestial coordinates, time conversions, and common data formats used in astronomy.
- `seaborn` – A statistical data visualization library built on top of `matplotlib`. It offers high-level interfaces for creating attractive and informative plots, especially with `pandas` DataFrames.
- `numba` – A just-in-time (JIT) compiler for Python that speeds up numerical code by compiling annotated functions to machine code. It is especially useful for accelerating loops and array operations.
- `cython` – A superset of Python that allows writing C extensions for Python. It can significantly speed up Python code by compiling it to C, especially in performance-critical applications.
- `F2PY` – A tool included with `numpy` that allows you to call Fortran code directly from Python. It can be used to incorporate modern Fortran routines into Python programs, enabling significant speed improvements for numerically intensive computations.

These modules, along with those we have covered more fully, form the backbone of the Python scientific computing ecosystem. Note that not all of them are included in the SciPy stack module available on the cluster.

Practice Questions

1. Write a program that prints out the first 20 Fibonacci numbers (assume $n_0 = 0$ and $n_1 = 1$).
2. Write a function that accepts an integer and returns `True` if the integer is even and `False` otherwise.
3. Write a program that uses the Rydberg formula to output the wavelengths of the first five hydrogen emission lines for the first three line series (Lyman, Balmer, and Paschen series).
4. Write a python program using `matplotlib` that generates the plot of $\sin(x)$ and $\cos(x)$ on the interval $x \in [-2\pi, 2\pi]$. Use the `linspace` function from `numpy` (look up how to use it) to generate `x`, the list of 500 equally spaced numbers on the given interval. You can then simply call `cos` and `sin` from the `numpy` library on this list to generate the y values for each.

3 MESA

3.1 Introduction to MESA

MESA (Modules for Experiments in Stellar Astrophysics) is a state-of-the-art, open-source software instrument for simulating the structure and evolution of stars. It is designed to be flexible, highly configurable, and extensible, making it suitable for a wide range of research problems in stellar astrophysics.

At its core, MESA solves the coupled differential equations that describe stellar structure and evolution, incorporating detailed physical models for nuclear burning, opacities, mass loss, convection, rotation, diffusion, and more. It supports stellar models from pre-main-sequence through advanced nuclear burning stages, including white dwarfs, supernova progenitors, and planetary evolution under certain conditions.

Why Use MESA?

MESA is widely used by researchers in the astrophysics community because it strikes a balance between usability and physical fidelity:

- It allows users to define the physical assumptions and numerical methods used in a model through simple text-based input files.
- It is actively developed and maintained by a community of stellar astrophysicists and software engineers.
- It is open source - allowing users to make any modifications to the source code that they desire and offer their suggestions to the developers.
- It is modular, meaning that different parts of the code (e.g., nuclear networks, opacity tables, or mixing prescriptions) can be modified or replaced independently.

Who is MESA for?

MESA is intended for researchers, advanced students, and educators who need to model stellar interiors or evolutionary pathways. A typical use case might include studying the evolution of stars of various masses and compositions, modelling the internal structure of a white dwarf, or computing the effect of mass loss on stellar lifetimes. I personally use MESA to model Cataclysmic Variables (binary systems consisting of a main sequence star losing mass to a white dwarf).

While MESA is a powerful tool, it is also complex and computationally intensive. A basic understanding of stellar structure and programming practices is recommended before diving into full-scale simulations. In this course, we will focus on learning how to run and modify simple MESA simulations to explore key physical processes in stellar evolution.

What You Will Learn

In the following sections, we will learn how to:

- Set up and compile a MESA working directory.
- Run stellar evolution simulations using predefined inlists.

- Read and interpret MESA output files.
- Modify input parameters to explore the effects of physical assumptions.

By the end of this part of the course, you should be comfortable using MESA to perform basic stellar evolution experiments and have a foundation for learning more advanced features on your own.

3.2 Installing and Setting up MESA

To install MESA on a local machine, one must simply follow the instructions on the documentation: https://docs.mesastar.org/en/stable/installation.html. For the purposes of this tutorial (and for the purposes of running many concurrent MESA runs), we will be using one of the Digital Research Alliance of Canada clusters (Narval). This presents some difficulties, however, as some of the software packages that MESA depends on needs to be configured a certain way on the cluster and this is not a simple task. I have previously been able to install MESA (with the help of one of the DRAC technicians) and since we will all be using it on the same compute nodes, we can just use copies of my installation.

There are two major components that we need: MESA itself, and the MESA SDK (software development kit). The SDK is a somewhat complex piece of software but makes the installation and compilation of MESA significantly easier. The installation of MESA can be found in the `projects` file system as `projects/def-lnelson/jaiken_public/mesa_files/mesa_24_08_1/` and the SDK as `projects/def-lnelson/jaiken_public/mesa_files/mesasdk/`. You should copy both of these to your home directory by running the following commands:

```
cp -r ~/projects/def-lnelson/jaiken_public/mesa_files/mesa_24_08_1 ~/.
cp -r ~/projects/def-lnelson/jaiken_public/mesa_files/mesasdk ~/.
```

You should also copy a Bash script that initializes the necessary environment variables:

```
cp ~/projects/def-lnelson/jaiken_public/mesa_files/mesa_init.sh ~/.
```

You can store this script anywhere you like, but it will need to be modified if you don't keep MESA and the MESA SDK in your home directory (`~/`) with their original names.

For a quick test, let's copy out one of the example simulations and run it:

```
cp -r ~/mesa_24_08_1/binary/test_suite/jdot_gr_check ~/.
cd ~/jdot_gr_check
source ../mesa_init.sh
./mk
./rn
```

Once you see some output from the run (specifically lines saying '`error in separation`'), then you know that MESA is running properly. You should quickly kill the process (`ctrl + c`) as only quick tests of intensive programs are permitted on login nodes.

You can now delete the test working directory that we made (make sure to double check that your commands are correct):

```
cd ~/
rm -rf ~/jdot_gr_check
```

3.3 Setting Up and Compiling a MESA Working Directory

To run stellar evolution simulations with MESA, you work inside a **working directory**—a local folder that contains all the files necessary to compile and execute a particular simulation. Each working directory is often (but not always) based on one of the example cases ('test cases') provided in the main MESA installation, and it can be customized to suit your specific scientific goals. These test cases can generally be found in the `test_suite/` directory of a specific module. The most common test cases one will generally use are found in either `star/test_suite` and `binary/test_suite` as these are test cases specific to stellar or binary evolutions respectively.

What Is a MESA Working Directory?

A MESA working directory is a self-contained environment for compiling and running one specific stellar evolution model. It typically includes the following key components:

- `make/` directory - This directory contains the `makefile` file which can be thought of as the recipe for how the MESA program files (found in `src/`) and MESA libraries should be compiled and linked to generate the MESA executable. This directory is also where the compiled object files will be saved before being linked into the final executable file. This directory and its contents will rarely (if ever) need to be modified by the user.

- `src/` directory - This directory contains the user-defined Fortran program and module source code files. These are the files that need to be compiled and linked with the MESA libraries (the MESA libraries are pre-compiled at the time of installation). These files can be modified to change how what the program is actually doing or to allow user-defined procedures to replace their MESA counterparts for additional control on how the physics in implemented. Since the modification of these files requires being familiar with Fortran, this will not be discussed.

- `inlist` files – These are plain text configuration files that specify the physical parameters, control options, and runtime settings for the simulation. Generally, there will be a 'top-level' inlist file `inlist` that points to the other inlist files. When running a binary simulation (modelling a binary system), there will typically be `inlist1` and `inlist2` files which hold the configurations for how star 1 and star 2 are modelled, respectively, (i.e. contains controls related to things like nuclear reactions, convection, stellar winds, etc.) and a `inlist_project` file that holds the configuration of the binary system itself (i.e. contains controls related to mass transfer, angular momentum losses, etc.).

- `history_columns.list` and `profile_columns.list` optional files – These files control what data gets written to output files. They are not necessary to include as there is a default list of data columns that will be written, but this may not suit all projects.

- `mk` – A Bash script that is used to call `make` to compile the project according to the makefile found in the `make/` directory. This script needs to be executed by the user before MESA can run **and** if any of the source code files found in `src/` are modified.

- `clean` – A Bash script that is used to clean up the compiled executable and compiled object files generated from compiling the project. This script should only be run when modifying the code files in `src/` to ensure that previous versions of compiled object files don't interfere with the new ones.

- `rn` - A Bash script that is used to run MESA. It isn't *essential* to use this script as all it does is print the time to the screen and call the generated executable (either `binary` or `star`), but it is considered good practice.
- `re` - A Bash script that will restart a MESA run from a specified *photo* (a 'save' file generated by MESA in a non-human readable format meant for restarts).

When you call the `mk` script to compile the project, it will generate several files in the `make/` directory and a binary executable file called either `star` or `binary`, dependent on if you are simulating a single star or a binary system. This executable file *is* the MESA program. If you were to run this executable, by either calling the `rn` script or by calling the executable directly, it will generate some additional directories and files:

- `LOGS/` or `LOGS1/` (and possibly `LOGS2/`) directories - These directories (whichever ones are generated) will contain the *human readable* output that MESA writes to files. These files are of two types: `history.data`, history files that contain summary data of a star at various points of time and `profileXX.data`, profile files that contains the data throughout the star at a single point in time. Additionally, a `profiles.index` file is generated to as an identification list of the profile files (they are not indexed chronologically in time).
- `photos/` directory - This directory contains the photos discussed earlier.
- `.mesa_temp_chache/` hidden directory - This directory contains cached data relating to the equation of state, nuclear network, and opacity. This data is simply here to potentially speed up reruns of the same simulation.
- `log` or `log1` (and possibly `log2`) - These files save what would normally be output to the screen from MESA. These files *can* be useful to look through to help understand why certain simulations are struggling (additional information is periodically printed to the screen) but there is a better method that will be discussed in the section about SLURM.
- `binary_history.data` - This file will be generated when simulating a binary system. It is simply the history file containing just the binary-specific data. All of this data is also written in the regular history files as well (furthest columns to the right).

This structure allows you to manage and reproduce simulations in an organized way. You can create multiple working directories to compare models with different physical assumptions or initial conditions.

Setting Up a Working Directory

To create a new working directory, start by copying one of the example cases provided with the MESA distribution. For example, let's make a working directory from the `evolve_both_stars` binary test:

```
cd ~/MESA_tutorial/MESA
cp ~/mesa_init.sh .
source mesa_init.sh      # if not already done before
cp -r $MESA_DIR/binary/test_suite/evolve_both_stars practice/.
cd practice/evolve_both_stars
```

You can rename `evolve_both_stars/` to anything you like. This new directory is now your personal workspace for evolving a two stars in a binary system starting from the main-sequence.

Compiling the Code

Before running MESA, you must compile the working directory. This links your custom (in our case, unmodified) source files with MESA's main libraries:

```
./mk
```

This command runs the Makefile and builds an executable called `binary`, which is used to run the simulation.

If this is the first time you've used MESA in your shell session, make sure to initialize the MESA environment first using the `mesa_init.sh` script I provided (or by following the steps in the install MESA page on the MESA documentation website).

If the build is successful, you're ready to run your simulation:

```
./rn
```

This will begin the stellar evolution run as specified by your input files. **Note:** if you make any changes to the *inlist* files, you do **not** need to recompile as they are parse at *runtime*. If you modify the source code (any file ending in `.f90`), however, you will need to recompile (run `./mk`).

In the next section, we will explore the contents of the `inlist` files and explain how to control the physical and numerical setup of your simulations.

3.4 Understanding the Inlist Files

MESA simulations are controlled almost entirely through plain-text configuration files called **inlists**. These files allow you to choose the physical assumptions, numerical methods, and runtime behaviour of the stellar evolution calculation.

An inlist is written using Fortran syntax as inlists are simply collections of a Fortran structure called a `namelist`, where each line specifies a variable name and its value. MESA reads these values at runtime and uses them to control how the simulation proceeds.

Structure of the Inlist Files

A typical working directory will contain a primary file called `inlist`, which acts as a master configuration file. This file usually includes references to other, more specific inlists. This more specific inlists will typically then contain multiple namelists that MESA will read in. Here is an example for how a inlist for a single star may be set up:

```
&star_job
   ! controls related to job setup and data output
/

&controls
```

```
   ! main physical and numerical parameters
/

&pgstar
   ! settings for real-time plotting (optional)
/
```

Each of these blocks is a **namelist**, and they must begin with an ampersand (e.g., `&controls`) and end with a forward slash. This is the syntax dictated by Fortran.

`star_job` Namelist

The `star_job` namelist controls aspects of the simulation setup, including what files to read or write, how to start the model (e.g., from a pre-main-sequence model or a saved checkpoint), and whether to save output at various points in the run. Some common options include:

```
create_pre_main_sequence_model = .true.
save_model_when_terminate = .true.
saved_model_name = 'final.mod'
```

`controls` Namelist

The `controls` namelist is where most of the physical and numerical settings are specified. This includes parameters such as:

- Initial mass and composition
- Stopping conditions (e.g., stop at a given central temperature or age)
- Mixing parameters and convection models
- Nuclear reaction networks
- Time-stepping and mesh refinement controls

Example settings might include:

```
initial_mass = 1.0
initial_z = 0.02
max_model_number = 1000
max_years_for_timestep = 1d6
```

`pgstar` Namelist

The `pgstar` namelist controls the live graphical output during the simulation (if enabled). This includes which plots to show, how frequently to update them, and how to configure the plot appearance. Since we will be using DRAC clusters which typically have no graphical user interface, we will not use `pgstar`. To disable `pgstar`, one can include the following line in the `star_job` namelist:

```
pgstar_flag = .false.
```

Other Namelists

Similar to their single star counterparts, there are specific namelists for simulating binary systems. These namelists are `binary_job`, `binary_controls`, and `pgbinary`. These namelists are typically contained in a file called `inlist_project` for binary simulations and are completely unneeded for single star simulations.

There is also namelists specific to the equation of state (the `eos` namelist) and the opacity (the `kap` namelist). It is quite common to leave these namelists empty (there are defaults that are used instead) but they still **must** be included with the other single star namelists.

Documentation

For all namelists, the MESA documentation website has pages that list all the possible options and controls and provide brief descriptions for some of the controls. The documentation page to visit is https://docs.mesastar.org/en/stable/reference.html#. On this page, one can click into any of the possible namelists to see what options are available.

This page also provides links to the namelist options available for the asteroseismology modules (ADIPLS or GYRE) and additional information on the other ways to customize MESA ('hooks' which allow users to use their own Fortran code instead of certain procedures).

How MESA Reads the Inlists

When you run `./rn`, MESA first reads the default inlists (located within the MESA installation, `$MESA_DIR`) to load all the default options. MESA then reads the top-level `inlist` file and follows through to read the additional defined inlist files and replaces the loaded defaults with any options that are present. All settings are parsed and checked for consistency before the evolution begins. If you make changes to the inlist files, you do **not** need to recompile—just re-run the executable.

Commenting and Good Practices

As with any Fortran code, lines beginning with `!` are treated as comments. You should use comments liberally to document your choices and make your inlists easier to understand and reuse.

MESA will throw an error for any parameters that are not valid for the current version or are misspelled. In order to not waste time (waiting for MESA jobs to be started by the scheduler), it is then important to ensure that the inlists are written correctly. This can be tested before submitting MESA as a job.

In the next section, we will examine the output that MESA produces during a simulation and how to interpret the contents of the `LOGS/` directory.

3.5 MESA Output Files

After running a simulation, MESA produces a variety of output files that are saved in a directory called `LOGS/` inside your working directory. These files contain detailed information about the star's structure, evolution, and internal state at each model step. Understanding how to read and interpret these files is essential for analyzing your simulations.

The `LOGS/` Directory

The `LOGS/` directory is automatically created when the simulation begins. It contains the following important files and subdirectories:

- `history.data` – A table where each row corresponds to a model (a time step in the evolution) and each column represents a physical quantity, such as age, luminosity, central temperature, or surface gravity.
- `profiles.index` – A reference table that maps model numbers to corresponding profile files. Each entry tells you which profile file corresponds to a given stage in the evolution.
- `profileNNNN.data` – A set of detailed profiles of the star's internal structure, where `NNNN` is the model number. These files include data at different radial points within the star (mass coordinate, temperature, density, etc.).

Reading `history.data`

The `history.data` file is usually the first output file to examine. You can open it in any text editor or load it using `pandas` in Python for more advanced analysis. A few key columns you might encounter:

- `model_number` – A sequential counter for the number of time steps.
- `star_age` – The age of the star in years.
- `log_Teff` – The logarithm of the effective temperature.
- `log_L` – The logarithm of the total luminosity in solar units.
- `center_h1` – The central hydrogen abundance.
- `log_R` – The logarithm of the stellar radius.

These data are ideal for plotting HR diagrams, radius or luminosity evolution, and identifying when key transitions (e.g., hydrogen exhaustion) occur.

Reading Profile Files

The `profileNNNN.data` files provide snapshots of the internal structure of the star at a particular point in time. They contain many columns, each corresponding to a different physical variable, such as:

- `mass` – Enclosed mass coordinate.
- `radius` – Radial coordinate.
- `logRho` – Logarithm of the local density.
- `logT` – Logarithm of the local temperature.
- `h1` – Hydrogen mass fraction.

You can use these files to study the star's internal structure, such as the location of burning shells, convective zones, or chemical composition gradients.

Saving Stellar Models

If configured in the `inlist`, MESA can save model snapshots that can then be loaded into future MESA simulations. These saved stellar models could allow one to 'build' special stars for use in other simulations. For example, I have used MESA to artificially strip the hydrogen envelope off the hot helium core of a star, and then transferred this model into a binary simulation to as the 'start' of the evolution of an AM CVn star. These stellar models are saved in separate files and directories outside `LOGS/`, and can be enabled with parameters like:

```
save_model_when_terminate = .true.
saved_model_name = 'final.mod'
```

You can later start a new run using this saved model by adjusting `load_saved_model` in the `star_job` namelist. In fact, when you begin a simulation from the ZAMS (zero age main sequence), MESA is actually using a set of pre-saved models to generate your desired star. These pre-save models are saved in the exact same format as the models that we can save from our runs.

A Little Practice

1. Given the `history.data` in `MESA/examples/CV/`, plot the mass-period diagram for this system. This plot should have the donor mass, $M_{\rm d}$ (known as `m1` to MESA), in units of $M_\odot$ as the x-axis and the orbital period, $P_{\rm orb}$, in units of hours as the y-axis. The columns that will be useful are `star_1_mass` and `period_days`.

2. Copy the `history.data` file from `MESA/examples/1Msun/` and plot the $\log(L/L_\odot)$ and $\log(T_{\rm eff}/{\rm K})$ columns such that the luminosity is on the y-axis and the effective temperature is on the x-axis. The x-axis should also be *reversed* so that the temperatures are decreasing from left to right. This plot is the *HR Diagram* for the 1 $M_\odot$ evolution.

3.6 MESA Documentation

Probably the most important tool for using MESA is the documentation. The main documentation page can be found at https://docs.mesastar.org/en/stable/index.html. Within this page, there is plenty of information to be found for using MESA. This includes things like installing MESA, troubleshooting (if/when the installation fails), how to use the different aspects of MESA, and how to get MESA to use your own implementations of certain procedures. Additionally, the MESA documentation provides a reference for all of the inlist controls, and even gives a simple guide on how to contribute to the MESA source code.

The MESA source code can be found here https://github.com/MESAHub/mesa. Most likely, you will only ever need to download the source code from the MESA releases (linked on the MESA documentation website), however, if you ever find a bug with MESA (and can fix it) or want to contribute your own code to the MESA project, this repository is where you will need to do it.

4 SLURM

What is SLURM?

SLURM (Simple Linux Utility for Resource Management) is an open-source workload manager used to schedule and manage jobs on high-performance computing (HPC) clusters. It allows multiple users to share computing resources efficiently by queuing and running jobs in a controlled environment.

When working on a shared computing cluster, users do not typically run programs interactively or directly on compute nodes. Instead, they submit jobs to the SLURM scheduler, which then allocates the appropriate resources (such as CPUs, memory, and time) and runs the jobs when those resources become available.

SLURM handles all of the following:

- Scheduling jobs based on priority, resource availability, and user limits.
- Allocating compute resources (CPUs, memory, time, etc.) for each job.
- Monitoring and logging job output and runtime statistics.
- Allowing for batch job execution, which is essential for long-running or resource-intensive computations.

Using SLURM allows you to run jobs efficiently and reproducibly, and ensures that the cluster remains fair and responsive for all users. In the following sections, we will learn how to write and submit SLURM job scripts, monitor job progress, and handle common issues.

Additionally, when using the DRAC (Digital Research Alliance of Canada) clusters (e.g. Narval), you can only directly interact with what are called *login* nodes. These nodes are not very powerful, there are not many of them (typically only 3), and are shared by all users logged in. Because of this (and because it is DRAC policy), these nodes are **not** where we want to run our scientific programs. Instead, we want our programs to be run on *compute* node, which are nodes that are designed specifically for running compute-heavy programs (there are also many of them). The compute nodes do **not** have access to the internet and so the **only** way to interact with them is through the SLURM scheduler.

Why Use SLURM?

Using SLURM is essential for working productively and responsibly on a shared computing cluster. Here are some of the main reasons why we use SLURM:

- **Efficient use of resources:** SLURM schedules jobs to make the best use of available compute nodes, minimizing idle time and balancing load across users.
- **Reproducibility:** A job script serves as a complete and repeatable record of what code was run, with what resources, and with what settings.
- **Batch processing:** You can run jobs in the background, freeing your terminal and allowing long-running tasks to complete without supervision.
- **Fairness:** SLURM enforces user limits and job priorities so that everyone has a fair chance at accessing compute resources.

- **Monitoring and control:** SLURM provides tools to track, cancel, or modify jobs during execution.

Even for small or short jobs, it is good practice to use SLURM to ensure that all users share the system fairly and that your computations are consistent and documented.

4.1 Writing a Simple SLURM Job Script

To run a program on the cluster using SLURM, you must write a **job script**—a small shell script that tells SLURM how to run your code, what resources to request, and what environment to set up. In this section, we will create a simple job script that runs a Python program called `job.py`.

Basic Structure of a SLURM Job Script

A SLURM job script is just a Bash script with special comment lines that begin with `#SBATCH`. These lines tell SLURM how to schedule the job. Below is an example:

```
#!/bin/bash
#SBATCH --job-name=testjob
#SBATCH --output=slurm-%j.out
#SBATCH --time=00:10:00
#SBATCH --ntasks=1
#SBATCH --mem=1G

# Not always needed
module load scipy-stack

python3 job.py
```

Explanation of the Script

- `#!/bin/bash` This tells the system that the script should be interpreted with Bash.
- `#SBATCH --job-name=testjob` Sets the job name as it will appear in the queue.
- `#SBATCH --output=slurm-%j.out` Redirects standard output to a file named `slurm-[jobid].out`. The variable `%j` is replaced with the job ID.
- `#SBATCH --time=00:10:00` Requests a maximum of 10 minutes of runtime. SLURM will kill the job if it exceeds this time **regardless of whether or not the program is finished**.
- `#SBATCH --ntasks=1` Requests one task (suitable for simple jobs that only run one program).
- `#SBATCH --mem=1G` Requests 1 GB of memory. You must estimate the memory your job needs and adjust this accordingly.
- `module load scipy-stack` Loads the Python environment available on the cluster that includes common scientific packages like `numpy`, `scipy`, and `matplotlib`. This line should only be included if the packages are actually needed.
- `python job.py` Runs your Python script.

Submitting the Job

To submit the job, save the script to a file (e.g., `run_job.sh`) and submit it using the `sbatch` command:

```
sbatch run_job.sh
```

SLURM will respond with a job ID, and your script will be added to the job queue. You can check on it using the `squeue` command. The DRAC clusters have a special alias `sq` that calls `squeue` to show just your jobs.

4.2 Running Parallel Programs with OpenMP

For jobs that make use of parallelism — such as Fortran programs compiled with OpenMP (e.g. MESA) — you will need to modify your SLURM job script to properly request multiple CPUs and configure the OpenMP environment. In this example, we assume you have already compiled your Fortran program into an executable called `job`.

SLURM Job Script for OpenMP Programs

Here is an example SLURM job script to run a parallel OpenMP program:

```
#!/bin/bash
#SBATCH --job-name=openmp_job
#SBATCH --output=slurm-%j.out
#SBATCH --time=00:10:00
#SBATCH --cpus-per-task=4
#SBATCH --mem=4G

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

./job
```

Explanation of the Script

- `#SBATCH --cpus-per-task=4` Requests 4 CPUs for a single task. This is appropriate for programs that use multithreading (like OpenMP) rather than multiple distributed processes.
- `export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK` Sets the number of OpenMP threads to match the number of CPUs allocated by SLURM. This is crucial — without it, your program might default to using only one thread.
- `./job` Runs your compiled Fortran program. Make sure the executable has execute permissions (you can set this with `chmod u+x job` if needed).

4.3 Submitting a Batch Array of Jobs

SLURM supports job arrays, which allow you to run many similar jobs in parallel, each with a different set of inputs or parameters. This is especially useful for parameter studies, convergence tests, or running models with varied initial conditions.

In this section, we will set up a job array that runs a new Fortran program, `white_dwarf`, multiple times using different initial conditions. Each job in the array will receive its input via a command-line argument.

Example: Varying Initial Conditions

Suppose your program is structured to accept an initial central temperature as a command-line argument:

```
./white_dwarf 1.0e7
```

To run several simulations with different initial temperatures, we can use a SLURM array job.

SLURM Job Script for a Job Array

Here's a sample SLURM job script that runs a job array:

```
#!/bin/bash
#SBATCH --job-name=wd_array
#SBATCH --output=slurm-%A_%a.out
#SBATCH --array=0-2
#SBATCH --time=00:10:00
#SBATCH --ntasks=1
#SBATCH --mem=1G

temps=(1.0e7 1.2e7 1.5e7)
initial_temp=${temps[$SLURM_ARRAY_TASK_ID]}

./white_dwarf $initial_temp
```

Explanation of the Script

- `#SBATCH --array=0-2` This tells SLURM to submit 3 jobs with array indices 0, 1, and 2.
- `temps=(1.0e7 1.2e7 1.5e7)` Defines a Bash array of initial temperatures corresponding to each job index.
- `initial_temp=${temps[$SLURM_ARRAY_TASK_ID]}` Extracts the appropriate temperature for the current job using SLURM's built-in `$SLURM_ARRAY_TASK_ID` environment variable.
- `./white_dwarf $initial_temp` Runs the compiled Fortran program with the selected initial temperature as a command-line argument.
- `#SBATCH --output=slurm-%A_%a.out` Ensures that each array task logs its output to a unique file. `%A` is the overall job ID, and `%a` is the array index.

Running the Array Job

Save the script (e.g., as `run_array.sh`), then submit it with:

```
sbatch run_array.sh
```

This will queue three jobs, each running `white_dwarf` with a different initial condition.

4.4 Job Arrays with Dynamically Created Working Directories

When running large parameter studies or multiple related simulations, it is often necessary for each job to have its own working directory. This ensures that input files, intermediate results, and output logs are kept separate and organized. SLURM job arrays can be paired with a script that generates unique working directories for each job based on a shared template and a set of parameters.

Below is a SLURM job script that accomplishes this setup (Note that this script, as well as additional required scripts, can be found in the following github repository: https://github.com/jaiken17/slurm-script-example. Additionally, this repository provides more details about how this script works.):

```
#!/bin/bash
#SBATCH --time=00:05:00
##SBATCH --account=#{groupName}
#SBATCH --job-name=example_job
#SBATCH --output=output-%x_%j.out
#SBATCH --error=error-%x_%j.out
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=1G
#SBATCH --array=1-3
##SBATCH --mail-user=<#{email}>
##SBATCH --mail-type=END
##SBATCH --mail-type=FAIL

# two '#' means SLURM skips command

# Set location of template working directory and parameters file
TEMPLATE_DIR=template_wd/
PARAMS_FILE=params.txt

# Call python script that copies template dir into a new run dir and sets initial parameters
# Python script prints the name of new dir to stdout which is saved to $DIR var here
DIR=$(python3 make_wds.py $PARAMS_FILE $TEMPLATE_DIR $SLURM_ARRAY_TASK_ID)

cd $DIR

# Copy the script that runs the program, and run script with srun (SLURM)
cp ../prog_run.sh .
srun prog_run.sh
```

Explanation of the Script

- `#SBATCH --array=1-3` Submits a job array of 3 tasks, each with a unique `SLURM_ARRAY_TASK_ID` from 1 to 3. This ID is used to select parameters and generate a working directory for each job.
- `#SBATCH --output=output-%x_%j.out` and `--error=error-%x_%j.out` Specify output and error filenames that include the job name (`%x`) and job ID (`%j`), ensuring unique log files per job.
- `##SBATCH --account` and `##SBATCH --mail-*` These lines are commented out (with double `#`) and ignored by SLURM. You can uncomment and customize them if you need to specify a project account or receive email notifications.
- `TEMPLATE_DIR` and `PARAMS_FILE` Set the location of a template working directory and a file containing job-specific parameters (e.g., initial conditions, model settings, etc.).
- `DIR=$(python3 make_wds.py ...)` Calls a Python script named `make_wds.py` that:
  - Copies the template working directory.
  - Modifies or inserts job-specific parameters based on the task ID.
  - Prints the name of the new directory, which is captured in the `DIR` variable.
- `cd $DIR` Enters the newly created working directory.
- `cp ../prog_run.sh .` and `srun prog_run.sh` Copies a run script (`prog_run.sh`) into the directory and executes it using `srun`. The run script itself should execute the compiled program (e.g., `./white_dwarf`) and handle any output redirection or post-processing.

Advantages of This Approach

- Each job runs in an isolated directory, avoiding file conflicts and making it easy to track individual results.
- Parameter selection and directory setup are automated via a Python script, making the job array highly scalable.
- Output and error files are named automatically with job-specific identifiers.

Submitting the Job

To launch the full array, save the script (e.g., as `run_dynamic_array.sh`), and run:

```
sbatch run_dynamic_array.sh
```

Each job will create its own directory, initialize inputs, and run the program independently.

4.5 Other Useful SLURM Script Options

In addition to the basic options used to specify runtime, resources, and output, SLURM provides many optional directives and environment variables that can enhance your workflow. Below are some commonly useful features you may want to include in your job scripts.

Email Notifications

You can receive email updates when a job starts, ends, or fails. Just specify your email address and the types of events to be notified about:

```
#SBATCH --mail-user=your_email@domain.com
#SBATCH --mail-type=BEGIN
#SBATCH --mail-type=END
#SBATCH --mail-type=FAIL
```

This is especially helpful for long jobs or batch arrays.

Job Arrays with Custom Indices

You can run non-sequential or irregular job arrays by specifying a custom index list:

```
#SBATCH --array=1,3,5,7-9
```

This allows you to skip failed or unnecessary configurations in reruns.

Runtime and Memory Limits

It's good practice to specify conservative resource limits to avoid long queue times or job failures:

```
#SBATCH --time=01:00:00    # Maximum wall time
#SBATCH --mem=4G           # Total memory per task
#SBATCH --cpus-per-task=2  # Number of CPUs per task
```

Note that exceeding the time or memory limit will cause the job to be killed.

It is also important to **always** specify these controls as otherwise an unknown default value is used.

Restricting to Specific Nodes or Partitions

In some cases, you may want to run on a specific partition or type of node (e.g., high-memory, GPU-enabled, or specific CPU/GPU desired):

```
#SBATCH --partition=highmem
#SBATCH --nodelist=node123
```

Check your cluster's documentation for available partition names and hardware configurations.

SLURM Environment Variables

Within a job script, SLURM provides a set of environment variables that are useful for scripting and logging:

- `$SLURM_JOB_ID` – The unique ID of the job.
- `$SLURM_ARRAY_TASK_ID` – The array index (if part of a job array).
- `$SLURM_JOB_NAME` – The name assigned to the job.
- `$SLURM_CPUS_PER_TASK` – The number of CPUs allocated per task.

You can use these to customize file names, select parameters, or log job metadata automatically.

Automatic Output File Naming

Use SLURM formatting codes to name output and error files in a structured way:

```
#SBATCH --output=output-%x_%A_%a.out
#SBATCH --error=error-%x_%A_%a.err
```

`%x` inserts the job name, `%A` the main job ID, and `%a` the array task ID (if applicable).

These options might help make your SLURM jobs more robust, flexible, and easier to monitor or reproduce. You can refer to the official SLURM documentation (https://slurm.schedmd.com/documentation.html) for additional details an other options not discussed.

4.6 DRAC Documentation

This section has been all about how to write scripts to submit to the SLURM scheduler on clusters. For jobs that are simple enough, it isn't necessary to know much about the cluster that you are actually submitting jobs on. However, it can be *very* useful to know more about the clusters you are using to ensure for somewhat more complex jobs and to ensure that your use of the cluster is as efficient as possible. This information can easily be found on the Digital Research Alliance of Canada wiki: https://docs.alliancecan.ca/wiki/Technical_documentation.

Additionally, there is a website where you can find up to date information on the operational status of the clusters: https://status.alliancecan.ca/.

Practice Questions

1. In it's own directory, write a Bash script that submits a job to the SLURM schedule with a maximum job time of five minutes, using one CPU, and 1GB of RAM. Also, direct SLURM to save the screen's output to a file called `output.txt`. Within this script, have it write the job name (you can set the job name to whatever you want), the job ID, and the number of CPUs used to the screen. This output will *only* appear in the `output.txt` file. Use this as the final line of the script:

   ```
   sleep 60
   ```

2. While the job in the previous question is running or in queue, use the `sq` command to verify the job ID and the job name.

3. Copy the `jdot_gr_check` test from the `$MESA_DIR/binary/test_suite/` directory into your `MESA_tutorial/SLURM/practice/` directory. Enter this copied MESA working directory and copy the script from question 1 here. Modify this script so that it will submit a job with 2 CPU cores, 8GB of RAM, and 10 minutes of possible job time. This job should *run* the MESA test (you should compile it beforehand) so that the final line of your Bash script that submits the job is:

   ```
   srun ./rn
   ```

   Note: MESA is a program that uses OpenMP (parallelized) and so we **must** specify how many CPU cores it should expect (this is different than requesting CPUs from SLURM):

   ```
   export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
   ```

4. In a new directory, write a modified version of question 1's script to submit an array of 3 jobs. For this script, have it write the job ID and the job array ID to the screen and save the output to a file named like `output_{ARRAY_ID}.txt`

5. Clone the example scripts for the 'complex' array of jobs from the Github repository:

   ```
   git clone github.com/jaiken17/slurm-script-example complex_job_example
   ```

   Read through, and modify the scripts so that it can submit its three jobs. Look through the new directories that are created (once the jobs begin running) to verify that you understand what it is doing. The job that is run is only a simple projectile motion simulation written in python.