ENEE 467: Robotics Project Laboratory

Introduction to Linux Operating Systems and Python Programming

Lab 0

Learning Objectives

The purpose of this lab is to provide an introduction to the Linux workflow and Python programming. After completing this lab students should be able to:

- Navigate a Linux-based PC using terminal commands

- Create, move, copy, edit, and rename files

- Run docker containers to launch virtual Python environments

- Feel comfortable programming in Python and using scientific libraries.

Being well versed in Linux-based operating systems and Python programming is crucial for working in the field of robotics. A majority of the open-source robotics community relies on Robot Operating System (ROS) for the execution, development, and distribution of their projects. ROS runs natively on Ubuntu, and heavy emphasis is placed on working with ROS in later labs. Additionally, many open-source robotics tools are packaged with a Python API, allowing for general and accessible use. Therefore this lab is designed to give you the necessary knowledge required to comfortably work with Python and Linux-based operating systems.

Related Reading

1. Introduction to Scientific Programming with Python

2. The Linux Cookbook

3. Python code collections and textbook for robotics algorithms

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab Questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab Exercises at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

Throughout this document, code blocks with a light gray background represent inputs to a terminal and code blocks with a light yellow background represent terminal outputs. Code blocks with text enclosed by angle brackets indicates a placeholder. For example, the command `command <filename>` indicates that you should replace `<filename>` with an actual filename. The final command would then be written as `command file.txt`.

List of Questions

List of Exercises

1. Lab Procedure

1.1 Prequisites

This lab assumes that you have access to a Debian Linux-based PC. The lab computers are running the latest Long-Term-Support (LTS) version of Ubuntu (Ubuntu 24.04). You will also need to have a recent version of Python 3 ($\geq$ 3.8). The OS-virtualization base image that we will be using (continuumio/miniconda3) is based on Debian linux, and includes a system-wide Python installation as well as a preconfigured virtual environment.

For Python dependency management, we recommend using Python virtual environments. This allows you to isolate specific Python installations and its packages from one another. Doing so prevents mismatches that can often occur when certain packages are build on conflicting versions of another package. It is a good habit that whenever you need to install many third-party packages for a project, you do so in a virtual environment to prevent issues with your system-wide Python installation. In this lab you will be introduced virtualization with Docker containers, to the environment management system Conda, and Python's package manager pip. Since the lab computers are shared systems, all software will be distributed in the form of Docker containers. This prevents dependency issues and makes portability and reusability much simpler.

1.2 Linux Command Line

The Linux command line, also referred to as the *terminal*, is a powerful text-based interface used to control and interact with the operating system. Unlike most modern day operating systems, many programs installed on a Linux system do not have a GUI and users must navigate and interact with them using the accompanying command line interface (cli).

The terminal in Ubuntu 24.04 can be launched from the applications tab, usually in the lower left or right corner. Open the applications tab and in the search bar type the first few letters of "terminal," "command," "prompt," or "shell." Look for the following icon and click on it to open a terminal: Many keyboard shortcuts exist in the Ubuntu workflow. A quick way to open a terminal is to
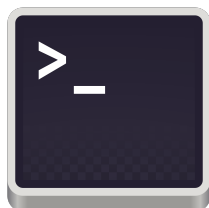

Figure 1: The Ubuntu terminal icon.

use the default keyboard shortcut: `Ctrl+Alt+T`. Either way you launch the terminal, you will be greeted with a purple window running a Bash shell. Commands are entered into the terminal by typing them. **Note:** commands are case sensitive, most commands are in all lowercase. Be extra careful with case when typing into the command line as sometimes the wrong case can result in a command appearing to run, but not doing what you expected.

1.3 Docker

Docker is an open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux.

Through out this course, software will be distributed to you in the form of a Docker container. This allows for better portability as well as ensuring all dependencies are met prior to the start of the lab. In most cases the Docker image for the week's lab we be prebuilt on the lab computers. In the event that you have to build it yourself, instructions will be provided in the lab's repository.

Ensure that the lab-0 image is build on your computer by running the following command in a terminal to list all built images on the host computer:

```
docker image ls
```

Make sure that you see the following line included in the command's output:

```
lab-0-image    latest    6f7e528c49b5    2 days ago    2.1GB
```

If the lab-0-image is not seen, ask an instructor to help you with the build process.

Now that we have the lab-0 image, clone the lab-0 repository. This repo contains the Dockerfile for this lab as well as a Docker compose file for launching the image. Run the following commands in a terminal on the host machine to clone the repo:

```
cd
git clone git@github.com:ENEE467-F2025/lab-0.git
cd lab-0
```

You should have now downloaded the necessary code for the remainder of the lab. Launch the docker image with the following commands in a terminal on the host machine:

```
cd docker
docker compose -f lab-0-compose.yml run --rm lab-0-docker
```

This command loads the image with the configuration specified in lab-0-compose.yml. You should be greeted with the following prompt indicating that you are now inside the Docker container:

```
(lab-0) robot@desktop:~$
```

where desktop may be replaced with the name of the host machine you are currently using.

> ⚠ The remainder of the lab is intended to be completed from within the Docker container to prevent any mishaps happening on the host machine. Whenever you are inside a terminal, ensure that the Docker container is active indicated by the above prompt. To reactivate the container should you close the terminal, follow the above instructions.

Containers are isolated development environments. To pass files between the host machine and the container, the `lab-0` directory you downloaded located at `/home/<user>/lab-0` on the host machine is mounted in the container at `/home/robot/lab-0`.

1.4   File Navigation

> ✎ The `Ctrl` key is a reserved key when interacting with a Linux terminal. To copy and paste text from or to the terminal, you must also include the `Shift` key. The copy and paste commands are therefore `Ctrl+Shift+C` and `Ctrl+Shift+V` respectively.

The first command we will run will be to print the current directory. Type the following command into the terminal and press `Enter` to run it:

```
pwd
```

You should see a directory path printed out of the form:

```
/home/robot
```

where `robot` is the account name of the currently logged user. This is the default, or `HOME` directory which is opened by default when launching a terminal window. The command `pwd` is an abbreviation of "**p**rint **w**orking **d**irectory." A working directory is the location in the systems file structure where commands typed into the terminal explicitly take place. For example, typing a command to create a new file will then place the file in the current working directory. It is important to be aware if the working directory the shell is currently in as mistakenly running commands in the wrong directory can have unintended consequences. If you are ever unaware of the current working directory, be sure to run the `pwd` command!

> ⚠ The command line is a powerful tool that must be approached with caution. **It will do whatever you tell it to do,** even if that is something you didn't intend for it to do. Sometimes there are warnings when attempting to work with protected files, but rarely is this the case for files owned by the user.

To change the current working directory, use the `cd` (**c**hange **d**irectory) command followed by the directory you wish to navigate to. To navigate to the root directory, try the following commands:

```
cd /
pwd
```

Your working directory is now `/`. Linux-based systems has a single unified file system meaning everything in the entire system branches from the ***root*** directory (`/`). A shortcut to navigate back to the `HOME` directory is typing `cd` on its own:

```
cd
pwd
```

To navigate back one directory to the parent of the current working directory, typing `cd ..` with two dots will take you there:

```
cd ..
pwd
```

which should output that you are in the directory `/home`:

```
/home
```

One final useful shortcut is to use the tilde ˜ character. The tilde character is short for your default or `HOME` directory. To navigate to the `HOME` directory or any directory that branches from it, try using the tilde character:

```
cd ~
pwd

cd ~/Desktop
pwd
```

Question 1. Assume you're a user with the username `robot` and your current working directory is `/home`. You wish to navigate to a directory on your Desktop named `important_files`. Provide three unique `cd` commands that will take you from your current working directory to the desired directory.

1.5 Creating Folders and Files

The command to create a new directory is `mkdir` which is short for `make dir`ectory. The argument following the command should be the path to the new directory.

> ✎ If you are ever unsure about the arguments to a command, try running the command followed by the help flag (`--help`). For example, to see the help text for the `mkdir` command, type `mkdir --help` into the terminal and press `Enter`.

Make a new directory in the `/tmp` directory name `tutorial`. The `/tmp` directory is the standard location for storing temporary files. These files will be deleted periodically or after a system shutdown, so don't store anything important here!

```
mkdir /tmp/tutorial
cd /tmp/tutorial
```

The `mkdir` command accepts multiple arguements whenever you wish to create multiple directories simultaneously. Try creating a few new subdirectories:

```
mkdir dir1 dir2 dir3
```

To list the contents of the working directory, use the command `ls` (list):

```
ls
```

You should see the following output listing the newly created directories:

```
dir1   dir2   dir3
```

The `mkdir` command has a helpful flag for creating nested directories in one go. The parent flag `-p` (or `--parent`) allows you to create nested directories as well as the parent directories as needed. Try creating a nested directory:

```
mkdir -p dir4/dir5/dir6
```

Empty files can be created using the command `touch`. Try creating an empty text file:

```
touch file1.txt
```

To output the contents of a file, we can use the command `cat` (concatenate). Read the contents of `file1.txt` using `cat`:

```
cat file1.txt
```

Of course nothing is printed as `file1.txt` is an empty file.

> ✎ Typing out Bash commands and filenames can be tedious. Pressing the `Tab` key when typing a command will automatically complete commands, filenames, and arguments. Try it out!

We can use the command `echo` along with a redirect to add some text to our file. Try running the following commands:

```
echo "this adds text to an existing file" > file1.txt
echo "this creates a new file with text" > file2.txt
```

Now `cat` the two text files to see their contents:

```
cat file1.txt
cat file2.txt
```

Or to print the contents of both files after *concatenating* them, try:

```
cat file1.txt file2.txt
```

Two things were introduced without explanation. The command `echo` simply prints out its arguments. The redirect "`>`" is for overwriting. It either overwrites the contents of the file it is directed towards, or creates a new one if the target file does not exist. If we want to append text to the bottom of a file instead of overwriting its contents, we can use the append redirect "`>>`". Try appending some text to the bottom of `file1.txt`:

```
echo "some text to be appended" >> file1.txt
cat file1.txt
```

The append redirect will also create a new file if the target filename does not exist.

Question 2.   Suppose you have three nonempty text files in a working directory: `a.txt`, `b.txt`, and `c.txt`. In one line, write the command(s) that will take the text from `a.txt`, `b.txt`, and `c.txt` and place them in a new file `combined.txt` in that order.

1.6   Moving and Manipulating Files

Now that we have created a few files and directories, let's go over how we can move, copy, and rename them from the command line.

To move a file, we can use the `mv` command (`move`). Try moving one of the text files to a new directory:

```
mv file1.txt dir1
ls dir1
```

You should now see that `file1.txt` has been taken from the working directory and placed inside the subdirectory `dir1`. The `mv` command has two arguments: 1) the source file(s) and 2) the destination. The source(s) can include as many files as you wish. All of the source files will then be moved to the final argument taken to be the destination. To move a file to the current working directory, refer to the destination with a dot (`.`). A dot points to the current working directory. To move `file1.txt` back to the current working directory, try the following command:

```
mv dir1/file1.txt .
```

To rename a file, the traditional Linux handles renaming as *moving* the file from one name to another. In this case, the destination becomes the new file name. To rename one of the text files, try the following command:

```
mv file1.txt my_file.txt
ls
```

You should now see that `file1.txt` has been renamed to `my_file.txt`. The same can be done for directory names:

```
mv dir1 my_dir
```

To copy a file, use the command `cp` (`cop`y). Try copying one of the text files into a one of the directories:

```
cp my_file.txt my_dir/my_file_copy.txt
```

To copy a directory and all of its contents, you must use the recursive flag `-r` (or `--recursive`) to copy all files and subdirectories. Try copying `my_dir` without the recursive flag:

```
cp my_dir my_dir_copy
```

which fails and returns the following output:

```
cp: -r not specified; omitting directory 'my_dir/'
```

Try again with the recursive flag:

```
cp -r my_dir my_dir_copy
ls
```

and you should see that a copy was made along with all of the contents.

Question 3.   Suppose you have a file `my_file.txt` in your working directory that you would like to copy onto a flash drive located at `/media/USERNAME/lexar`. In one line, write a command to copy the file to the flash drive and rename it to `important_file.txt`.

1.7   Removing Files

To complement the creation of files and directories, the Bash shell also provides commands for deleting them. The command `rm` (`rem`ove) is used for deleting files. Similarly, the command `rmdir` is used for deleting empty directories. Use these commands and their flags sparingly. File deletion from the command line cannot be reversed. When removing a large number of files, it is a good idea to use the interactive flag `-i` which will prompt you before removing each file in order to prevent mistakes.

⚠ The `rm` command **does not** move files to a trash folder or similar. Instead it deletes them irrevocably. You must be careful when using the `rm` command and make sure you are only deleting the file you intend to.

For an interesting story on the perils of `rm`, see: UNIX Recovery Legend.

1.8   Text Editing

Most files we will be working with are plain text files in some form. In the days of old many command line text editors existed due to the lack of computer graphics. A few that are still popular today are the nano text editor and VIM. To edit a text file with nano, first create a file and then open it with the nano editor:

```
touch file.txt
nano file.txt
```

You will be greeted with a simple cli where you can type to add text to the file. To save changes to the file press `Ctrl+O` which prompts the "Write Out" command. It will then ask you to write the filename. To keep the same filename simply press `Enter` or change the filename if you wish then press `Enter`. To then exit the editor, press `Ctrl+X`. nano and VIM both have capabilities which far exceed the scope of this lab.

A perhaps more inviting text editor is VS Code. VS Code is installed on the host machines. Two options exist for editing files inside of a container. 1) VS Code provides an interface for connecting to containers remotely. This is useful for developing applications inside of a fixed environment. 2) The lab directory `lab-0` is a volume that is mounted on both the host machine and the container, therefore changes made to files in this directory will be reflected on both machines.

To open the lab directory with VS Code, run the following command in a terminal on the **host machine:**

```
cd ~/lab-0
code .
```

Though you are free to use any text editor you wish, we highly recommend VS Code due to the many available plugins for Python and the robotics tools we will use in later labs.

1.9   superuser

The superuser is a Linux user with administrative or super powers. Certain operations and commands such as software updating or installation require elevated privileges over those available to normal users. An example of a command that requires administrative privileges is when you try to access a file considered sensitive. Try running the command inside the container:

```
cat /etc/shadow
```

Without elevated privileges this action is not allowed and you will receive the following output:

```
cat: /etc/shadow: Permission denied
```

Luckily, for operations requiring administrative privileges, Ubuntu provides a command that allows a user to run a single command as a superuser. This command is the `sudo` command (super**user**

do). Only certain accounts on an Ubuntu system are allowed to use the `sudo` command, and only administrators have the ability to add users to the sudo group. **For now, class accounts have sudo disabled.**

> ⚠ Always make sure you understand what a command does before running it with `sudo`. Running a command with `sudo` gives the command the same powers as a superuser. `sudo` may only run one command at a time, but that command itself could run many others, all of which now have superuser privileges.

2. Python

Python programming can be approached in two ways: 1) executing Python scripts or 2) the interactive Python shell. All python scripts end in the `.py` extension. To run a Python script using the system wide Python installation, enter the following command:

```
python3 my_file.py
```

To launch a Python shell, simply type:

```
python3
```

which will greet you with the following prompt:

```
Python 3.10.18 (main, Jun  5 2025, 13:14:17) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

in which you can now type Python code line by line. To exit the shell, type the quit function `quit()` or enter the keyboard shortcut `Ctrl+D`. The interactive shell is useful for quick debugging or simple calculations. More sophisticated programming should be done using a text editor for better organization and to avoid losing progress.

2.1 Conda and Python Virtual Environments

The system wide Python installation contains important packages as well as their dependencies and is used by various OS applications. Toying with the system wide installation is not recommended and attempting to install new packages via pip will throw the following error:

```
error: externally-managed-environment
```

Instead, if we require particular packages for a project, we should create a separate Python virtual environment. Virtual environments contain their own Python interpreter and an isolated space for installing packages. Largely they prevent dependency conflicts between various versions of packages. To create a new virtual environment with Conda, enter the following command:

```
conda create --name <environment_name> python=<python_version>
```

This command creates a new virtual environment with the name `<environment_name>`. Conda also allows us to specify the Python version which is extremely useful when installing third-party projects written in an older version of Python.

The `lab-0` Docker container automatically activates a Conda virtual environment. This is indicated by the prefix on the terminal prompt `(lab-0)`. To deactivate the environment, try the following command:

```
conda deactivate
```

Notice that the prefix disappears, indicating that we are now in the system-wide environment:

```
robot@desktop:~$
```

To see the currently installed Conda environments, try the following command:

```
conda env list
```

Which should have the following output:

```
# conda environments:
#
lab-0                    /home/robot/.conda/envs/lab-0
base                     /opt/conda
```

Notice that we have two environments available: `lab-0` and `base`. The `lab-0` environment has been configured for this lab. The `base` environment is automatically installed when Conda is installed. The `base` environment provides a layer of isolation from the system wide environment. It is useful for quick development and experimentation without the risk of breaking the system wide installation.

To reactivate the `lab-0` environment, enter the following command:

```
conda activate lab-0
```

Ensure that you have the `lab-0` environment active for the remainder of the lab, i.e. make sure your prompt has the `(lab-0)` prefix:

```
(lab-0) robot@desktop:~$
```

> **Question 4.** Explain why it is a good idea to use virtual environments for new projects with a large number of dependencies. Why is it generally inadvisable to install the dependencies system wide?

2.2 Python Packages and pip

`pip` is the **p**ackage **i**nstaller for **P**ython. Packages listed on the Python Package Index can be installed directly from the command line. Each Conda environment includes a matching version of pip automatically. Inside the Docker container shell, ensure the environment `lab-0` is active and list the currently installed Python packages:

```
pip list installed
```

This command will print out all of the currently installed Python packages accessible by your Python installation. To install a new package, use the command `pip install <package>`. To see information about an installed package, use the `pip show` command. For example, to see the information about the NumPy installation, run the following command:

```
pip show numpy
```

Which will print out information on the version, location, and licensing for the NumPy package. To remove a package, use the command `pip uninstall <package>`. (**Note:** Don't uninstall any packages from the `lab-0` environment as you will need them for the exercises!)

2.3 IPython

IPython, or Interactive Python, is an interactive command-line shell for Python that offers an enhanced Read-Eval-Print Loop (REPL) environment. IPython allows you to run a subset of Bash commands while in the Python shell. Inside the Docker shell, ensure you have the `lab-0` environment active then show IPython with pip:

```
pip show ipython
```

To launch an IPython shell, use the following terminal command:

```
ipython
```

which will greet you with the following shell:

```
Python 3.10.18 (main, Jun  5 2025, 13:14:17) [GCC 11.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.37.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

13

From here on out, whenever referring to the Python shell, we mean an IPython shell.

2.4 Python Data Types

Python is an implicitly typed language meaning that data types are not declared when creating a variable, rather they are inferred from the value they are assigned. Python has the following data types built-in by default, in these categories:

Text:	`str`
Number:	`int`, `float`, `complex`
Sequence:	`list`, `tuple`, `range`
Mapping:	`dict`
Boolean:	`bool`
Binary:	`bytes`, `bytearray`
None:	`NoneType`

Make sure you have the `lab-0` environment active and launch a Python interactive shell. Try typing the following lines in the shell and make note of the type of each variable.

```python
x = 1
type(x)
y = 2.0
type(y)
type(x + y)
```

> **Question 5.**   In the previous example variable x has the type `int` and variable y has type `float`. Why does their sum have type `float`?

2.5 Lists and Dictionaries

Python provides utilities for storing *vectors* of data. A Python `list` is a collection of objects. Objects in a list can be any built-in or user-defined types. To create a list, try the following:

```python
my_list = [1, True, "hello", None]
print(my_list)
```

To access elements of a list, use the indexing operator `[ ]`. Note that like most programming languages, Python uses zero-based indexing meaning the first element in a list is at position " 0."

```python
print( my_list[0] )
```

Python indexing also has a useful feature for accessing elements at the end of a list. To access the *last* element of a list, use the index `-1`:

```
print( my_list[-1] )
```

You can use negative indexing to access elements of a list in reverse order. For example, to access the second-to-last element of a list, use the index -2, the third-to-last -3, so on and so forth. Lists have a couple important built-in methods. To add an element to a list, use the append method:

```
my_list.append("a new string")
print(my_list)
```

To get the index of a specific element in a list, use the index method:

```
idx = my_list.index("hello")
print(idx)
```

Dictionaries are a useful structure for storing data with key-value pairs of the form "key": value. To create a dictionary in Python, try the following:

```
my_dictionary = {"an_int": 5, "a_float": 3.14, "a_bool": True}
```

You access elements of a dictionary using the key:

```
print( my_dictionary["an_int"] )
print( my_dictionary["a_float"] )
```

To add new elements to a dictionary, you can simply use a new key and assign it a value:

```
my_dictionary["a_new_key"] = "a_new_value"
print( my_dictionary["a_new_key"] )
```

3. Exercises

Exercise 1. Matrix Arithmetic

Linear algebra provides the basis for many important engineering applications including aspects of robotics and machine learning. Therefore having tools for the efficient computation of arithmetic operations with matrices is important. Your task is to write a Python class for matrices and define methods for adding, subtracting, scalar multiplication, and matrix multiplication. For a real-valued matrix A, let a_{ij} be the element at row i and column j. Recall the rules for matrix arithmetic are as follows:

Addition: Let $A, B \in \mathbb{R}^{n \times m}$. $C = A + B$ implies $c_{ij} = a_{ij} + b_{ij}$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$.

Subtraction: Let $A, B \in \mathbb{R}^{n \times m}$. $C = A - B$ implies $c_{ij} = a_{ij} - b_{ij}$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$.

Scalar Multiplication: Let $A \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}$. $C = bA = Ab$ implies $c_{ij} = ba_{ij}$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$.

Matrix Multiplication: Let $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times l}$. $C = AB \in \mathbb{R}^{n \times l}$ implies $c_{ik} = \sum_{j=1}^{m} a_{ij} b_{jk}$ for $i = 1, \ldots, n$, $k = 1, \ldots, l$

Matrix Transpose: Let $A \in \mathbb{R}^{n \times m}$. $B = A^T \in \mathbb{R}^{m \times n}$ implies $b_{ij} = a_{ji}$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$.

Note that matrix addition/subtraction is only defined for matrices of the same shape. Similarly, matrix multiplication is only defined for matrices that share a common dimension.

To get started, open VS Code on the host machine and navigate to the `lab-0` directory. Inside this directory are two starter scripts for the exercises. Open `matrix.py`. Inside this file you will find a partially completed class for matrices. Go through the remainder of the exercise carefully. Some methods have been populated for you and some you will have to complete yourself.

The matrix class loads data in the form of a nested list and creates a matrix object. Inspect the initialization (`__init__`) method. We include a routine to verify that the list being passed is properly formatted:

```python
class Matrix:

    def __init__(self, data):

        # ensure data is properly formatted
        n = len(data) # number of rows
        m = len(data[0]) # number of columns
        for row in data:
            if len(row) != m:
                raise Exception("All matrix rows must be of the same length!")

        self.data = data
        self.shape = (n, m)
        self.size = n * m
```

It may be helpful to include a few class variables for things like the size (number of elements) of a matrix and the shape (number of rows, number of columns) for quick lookup when doing a computation.

Next we include methods to allow indexing of the form `A[i,j]`. To properly do this, we overwrite two Python dunder methods, `__getitem__` and `__setitem__`:

```python
def __getitem__(self, key):
    # access matrix elements by A[i, j]
    i, j = key
    return self.data[i][j]

def __setitem__(self, key, value):
    # set matrix element by A[i, j] = x
    i, j = key
    self.data[i][j] = value
```

It might be useful to have a method for converting a matrix to a string for easy printing. To do so, we overwrite the `__str__` method which returns a string when we try to print a matrix object:

```python
def __str__(self):
    # to string method, useful for printing matrices
    s = ""
    for row in self.data:
        s += str(row) + "\n"
    return s
```

Next we include a method for adding matrices using the `+` operator. This is done by overwriting the `__add__` dunder method:

```python
def __add__(self, B):
    # be sure that the matrices self and B have the same shape
    assert self.shape == B.shape

    # create a new, empty matrix
    data = [[0 for _ in range(self.shape[1])] for __ in range(self.shape[0])]
    X = Matrix(data) # zero matrix with shape self.shape

    for i in range(self.shape[0]):
        for j in range(self.shape[1]):
            # use our custom indexing
            X[i, j] = self[i, j] + B[i, j]

    return X
```

Finally, we implement a method for transposing a matrix as a property. Properties in Python are a way of handling attributes (class variables) without having to expose getter or setter methods. They are commonly used for accessing attributes more complex than the built in types which have the

assignment (=) operator defined. To create a transpose property, we use the `@property` decorator above the following method:

```python
@property
def T(self):
    # return self transposed

    # create a new, empty matrix with shape = (self.shape[1], self.shape[0])
    data = [[0 for _ in range(self.shape[0])] for __ in range(self.shape[1])]
    X = Matrix(data)

    for i in range(self.shape[0]):
        for j in range(self.shape[1]):
            X[j, i] = self[i, j]

    return X
```

Now, given a matrix object `A`, we can get the transpose of `A` with the following syntax:

```python
A_transpose = A.T
```

To learn more about properties and where they are appropriate, see: Python Property Decorator.

By now we have a matrix class that can load data from an array, index elements, set elements by index, and add two matrix objects with the same shape (this is the extent of the included `matrix.py` file you downloaded from the repo). To complete the class, you must implement the methods for subtracting two matrices, multiplying two matrices, and multiplying a matrix with a scalar value. Note that matrix multiplication depends on the order of the matrices. To account for both left and right multiplication, you will have to use two methods, `__mul__` for left multiplication and `__rmul__` for right multiplication. To do so, overwrite the following dunder methods:

- Matrix subtraction:

```python
def __sub__(self, B):
    # implement the code to calculate X = self - B
```

- Left matrix multiplication:

```python
def __mul__(self, B):
    # implement the code to calculate X = self * B

    if isinstance(B, Matrix):
        # matrix multiplication
        assert self.shape[1] == B.shape[0]
    else:
        # B is a number, scalar multiplication
```

- Right matrix multiplication:

```python
    def __rmul__(self, B):
        # implement the code to calculate X = B * self

        if isinstance(B, Matrix):
            # matrix multiplication
            assert B.shape[1] == self.shape[0]
        else:
            # B is a number, scalar multiplication
```

Note that both multiplication methods should be able to handle scalar multiplication as well as matrix multiplication. To distinguish between the two, you can use the builtin method `isinstance(<object>, <type>)` to see if the object B is a matrix or a number.

Deliverables: The templates for the above methods are already included in `matrix.py`. Make all of your changes to the file `matrix.py`. To complete the exercise, make sure you have done the following:

a. Implement matrix subtraction and multiplication by completing the following class methods: `__sub__`, `__mul__`, and `__rmul__`

b. Demonstrate each method working properly. Create a few matrix objects using the following syntax:

```python
A = Matrix([[1., 2.], [3., 4.]]) # 2x2 matrix
B = Matrix([[1., 2., 3.], [4., 5., 6.]]) # 2x3 matrix
```

And demonstrate that each method works using the following syntax:

```python
C = 2.*A # scalar multiplication
D = A + C # addition
E = A * B # multiplication
# so on and so forth
```

Include your matrix class file `matrix.py` along with the submission of your lab report as well as your demonstrations of all matrix arithmetic operators.

Note: The Python linear algebra library NumPy implements all matrix arithmetic operators and provides convenience functions for matrix and vector manipulation. Though many of the algorithms are written in C++ under the hood, the same basic principles you implemented above are present. In many scientific and engineering contexts, it is important to understand the tools and algorithms you are using at a low level.

Exercise 2. Simulating an ODE

Consider a simple pendulum as shown in Figure 2. The equation of motion for a simple pendulum is described by the following ordinary differential equation:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin\theta = 0$$

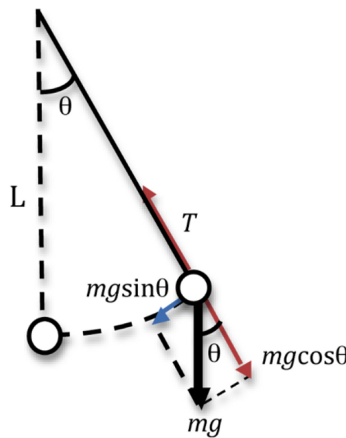We wish to write a Python script to simulate the motion of the pendulum over time given an initial

Figure 2: A simple pendulum.

starting configuration. First, let's convert the second-order ODE into a coupled first-order ODE of the form $\dot{x} = f(x)$ by introducing the variables $x_1 = \theta$ and $x_2 = \dot{\theta}$:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -g/L\sin x_1 \end{bmatrix}$$

We can then approximate a numerical solution using the coupled ODE using the Euler method:

$$x(k+1) = x(k) + \delta t f(x(k))$$

with the initial condition $x(0)$ given. To simulate the pendulum's motion, open the Python file `pendulum.py` from the VS Code window and inspect the included code schematic. We have populated the method `simulate_pendulum` for you. Here's the process in which it was derived:

The function `simulate_pendulum` has the following arguments:

- `x_0` : A length 2 `list` containing the initial pendulum state
- `L` : The length of the pendulum string
- `N` : The total number of time steps to simulate
- `dt` : The discretized time resolution.

We want the function to return three lists:

`T` : A list containing the time steps

`X_1` : A list containing the trajectory of x_1

`X_2` : A list containing the trajectory of x_2.

To begin, we create some placeholder variables for `T`, `X_1`, and `X_2`:

```python
# import sine function from the math library
from math import sin

def simulate_pendulum(x_0, L, N, dt):
    T = [0.] # first time step is zero
    X_1 = [x_0[0]] # first state is given to us as x_0
    X_2 = [x_0[1]]
```

Next we will need a `for` loop to iterate through the time steps:

```python
def simulate_pendulum(x_0, L, N, dt):
    T = [0.] # first time step is zero
    X_1 = [x_0[0]] # first state is given to us as x_0
    X_2 = [x_0[1]]

    for k in range(N-1):
        # loop
```

Note that we only need to loop $N - 1$ times since we all ready have the first state (x_0)! To perform the integration step for x_1, we add the following line:

```python
def simulate_pendulum(x_0, L, N, dt):
    T = [0.] # first time step is zero
    X_1 = [x_0[0]] # first state is given to us as x_0
    X_2 = [x_0[1]]

    for k in range(N-1):
        X_1.append( X_1[k] + dt * X_2[k] )
```

To perform the integration step for x_2, first we need to calculate $\dot{x}_2$. We add the following line to the for loop:

```python
def simulate_pendulum(x_0, L, N, dt):
    T = [0.] # first time step is zero
    X_1 = [x_0[0]] # first state is given to us as x_0
    X_2 = [x_0[1]]

    for k in range(N-1):
        X_1.append( X_1[k] + dt * X_2[k] )
        x_2_dot = -9.81/L*sin(X_1[k])
```

then perform the integration step using the calculated derivative:

```python
def simulate_pendulum(x_0, L, N, dt):
    T = [0.] # first time step is zero
    X_1 = [x_0[0]] # first state is given to us as x_0
    X_2 = [x_0[1]]

    for k in range(N-1):
        X_1.append( X_1[k] + dt * X_2[k] )
        x_2_dot = -9.81/L*sin(X_1[k])
        X_2.append( X_2[k] + dt * x_2_dot)
```

Finally, we keep track of the time step by updating the list T and return the lists once the for loop terminates:

```python
def simulate_pendulum(x_0, L, N, dt):
    T = [0.] # first time step is zero
    X_1 = [x_0[0]] # first state is given to us as x_0
    X_2 = [x_0[1]]

    for k in range(N-1):
        X_1.append( X_1[k] + dt * X_2[k] )
        x_2_dot = -9.81/L*sin(X_1[k])
        X_2.append( X_2[k] + dt * x_2_dot)
        T.append( T[k] + dt )

    return T, X_1, X_2
```

We want to plot the trajectories and in order to do so we need some sort of plotting utility package. An excellent choice is Matplotlib. Verify that Matplotlib is installed in the docker container using pip:

```
pip show matplotlib
```

Then we add the following import statement to the file pendulum.py:

```python
# import plotting utilities:
import matplotlib.pyplot as plt
```

We add the following code following the simulate_pendulum method to run the simulation and visualize it:

```python
# import sine function
from math import sin
# import plotting utilities:
import matplotlib.pyplot as plt

def simulate_pendulum(x_0, L, N, dt):
    T = [0.] # first time step is zero
    X_1 = [x_0[0]] # first state is given to us as x_0
    X_2 = [x_0[1]]
```

22

```
    for k in range(N-1):
        X_1.append( X_1[k] + dt * X_2[k] )
        x_2_dot = -9.81/L*sin(X_1[k])
        X_2.append( X_2[k] + dt * x_2_dot)
        T.append( T[k] + dt )

    return T, X_1, X_2

def main():
    T, X_1, X_2 = simulate_pendulum(x_0=[1., 0.], L=1., N=500, dt=0.01)
    plt.plot(T, X_1)
    plt.plot(T, X_2)
    plt.show()

if __name__ == "__main__":
    main()
```

Thus completing the simulation code.

Ensure the `lab-0` environment is active and run the script with the following command from within the container:

```
python pendulum.py
```

Play around with different length `L`, initial states, and time resolutions `dt`. What happens if you increase the time resolution too much?

Now we want to edit the ODE to include friction. The new ODE has the following form:

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \frac{g}{L}\sin\theta = 0$$

with a friction term $q \geq 0$. The state space equations now take the form:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -qx_2 - g/L\sin x_1 \end{bmatrix}$$

Deliverables: Make a new function named `simulate_pendulum_with_friction` with the following arguments:

```
def simulate_pendulum_with_friction(x_0, L, q, N, dt):
    # code
```

Write a new simulation that includes the friction term. You can reuse most of the code from the first simulation method, just be sure to edit the `x_2_dot` assignment to include the friction term. Plot the new simulated trajectories with friction. Play around with different values of q starting with $q = 1.0$ and increasing/decreasing its value. What do you notice about the amplitude of the trajectories? Include a screenshot of the trajectories as well as a qualitative assessment of the differences in the simulation with and without friction in your lab report.

Simulation is paramount in the field of robotics. Researchers seek simulators that provide a good approximation of real world dynamics and sensing. Simulation enables safe prototyping and rapid data generation for machine learning that otherwise would be too costly to obtain with real world experimentation. In a nutshell, rigid body simulators calculate the sum of forces and torques acting on bodies due to gravity, contacts, impulses, actuation, and friction then perform an integration step at each time interval to compute the next state. The state of a rigid body in simulation consists of a body's position, orientation, and linear and angular velocities.

As you might be able to infer, the Euler method is not the most accurate integration scheme when solving ODEs. Instead, the Python scientific package SciPy provides routines for solving ODEs. Verify that SciPy is installed in your conda environment with the following command:

```
pip show scipy
```

The de facto ODE solver within SciPy is `odeint` which takes at least three arguments:

`func` : A function handle that computes the derivative of `y` at `t` of the form:

$$\dot{y} = \mathsf{func}(y, t)$$

Note that SciPy uses `y` as a state variable instead of `x`.

`y0` : An initial condition on `y`

`t` : A sequence of time points for which to solve for `y`. The initial value point should be the first element of this sequence.

To simulate the pendulum with `odeint`, add the following method to your Python script:

```python
def simulate_pendulum_odeint():
    from scipy.integrate import odeint
    from numpy import linspace

    dt = 0.01
    N = 500
    L = 1.
    x_0 = [1., 0.]
    T = linspace(0, N*dt, N)

    def x_dot(x, t):
        # Calculate the derivative of the pendulum in state space form.

        x1 = x[0]
        x2 = x[1]
        dx1 = x2
        dx2 = -9.81/L*sin(x1)
        return [dx1, dx2]

    X = odeint(x_dot, x_0, T)
    plt.plot(T, X[:, 0])
    plt.plot(T, X[:, 1])
    plt.show()
```
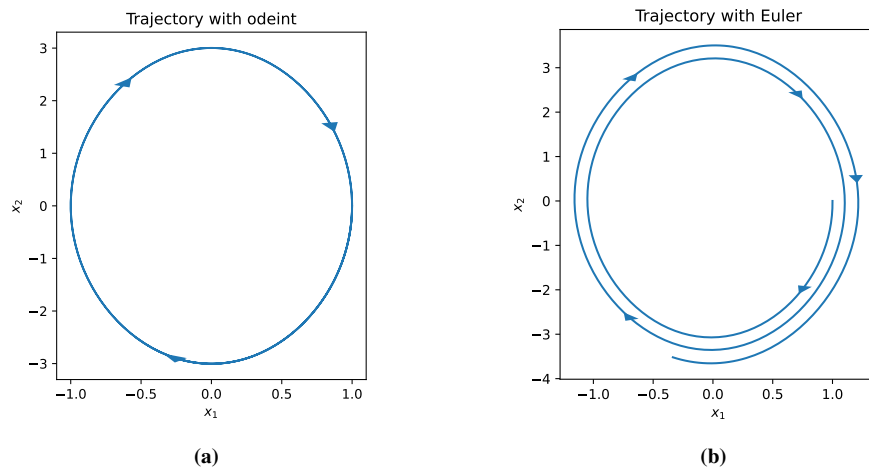
24

Figure 3: Two different trajectories using `odeint` vs the Euler method. Notice that the Euler method appears to gain energy over time (widening of the trajectory). Physically this is not possible as the pendulum is a passive system. Instead, this highlights the numerical errors that accumulate over time with the Euler method.

4. Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Your team is only responsible for submitting one lab report, make sure to include all members' names. Include your solutions to all of the question boxes in the lab report. Refer to the List of Questions to ensure you don't miss any. Include your full solution to the exercises in your lab report. Each exercise includes a box breaking down the required deliverables you must complete. If you wrote any Python programs during the exercises, include these files as a separate attachment. Submit your lab report as a `.pdf` along with any code to ELMS under the assignment for that weeks lab. You have one week to complete the lab report and submit it on the day of your next lab session. See the files section on ELMS for a lab report template. Below is a rubric breaking down the grading for this week's lab assignment:

Component	Points
Lab Report and Formatting	15
Question 1.	5
Question 2.	5
Question 3.	5
Question 4.	5
Question 5.	5
Exercise 1.	30
Exercise 2.	30
Total	100