.

Aim and Objectives

The aim of this lab is to provide students with a strong foundation in using **NumPy** for numerical computations and **Matplotlib** for data visualization, focusing on the computational ease and speed these tools offer in Python programming.
After successful completion of this lab, students should be able to:

- Understand and effectively use **NumPy arrays**, perform vectorized operations, and apply key NumPy functions to simplify and accelerate numerical computations.
- Utilize **Matplotlib** to create a variety of plots and visualizations, demonstrating the ability to clearly represent data graphically.
- Recognize the advantages of **computational efficiency** and **speed** provided by NumPy when handling large datasets and performing complex mathematical operations.
- Develop Python programs that integrate **NumPy** for data manipulation and **Matplotlib** for interactive visualizations.

1 NumPy

1.1 Introduction

1. **NumPy** (**Num**erical **Py**thon) is the most basic and a powerful package for working with multi-dimensional data in Python. It is widely used in data science, machine learning, and scientific computing for performing fast array operations and complex numerical calculations.
2. You can create and manipulate multi-dimensional objects using **NumPy**. It also provides functionalities for vector element-wise operations and operations related to high dimensional vectors (arrays, matrices, tensors etc.)
3. Lets start exploring the functionalities of **NumPy** library using the Python terminal. Since this is not an inbuilt module in Python, you have to start by **installing** it.

   Open your command-line interface (CLI) or terminal and type the following command:

   ```
   pip install numpy
   ```

4. As any usual module, you have to **import** the module (library) using the import command.

   ```
   >>> import numpy as np
   ```

5. Since we have already used Python **lists** to create matrices and lists, let's identify key differences between Python **lists** and **NumPy arrays** or ***ndarray*** for short.

   - **NumPy arrays** can handle ***vectorized operations***, which means they can do math on entire lists of numbers all at once, without needing to go through each number one by one in a loop. To create a NumPy array, we use the **array()** function.

     ```
     >>> python_list = [1,2,3,4]
     >>> python_list + 1
     TypeError: can only concatenate list (not "int") to list
     >>> numpy_array = np.array(python_list)
     >>> type(numpy_array)
     <class 'numpy.ndarray'>
     >>> numpy_array + 2
     array([3, 4, 5, 6])
     >>> 1/numpy_array
     array([1., 0.5, 0.33333333, 0.25])
     ```

- Unlike Python **lists**, items inside the **NumPy array** should be of the ***same type***. If not, the library itself will convert the items to a unified data type. This conversion might vary depending on the data types in the given list. [In the below example, *'<U11'* refers to a data type representing a Unicode string with a maximum length of 11 characters.]

```
>>> python_list =  [1,2,True,1]
>>> numpy_array = np.array(python_list)
>>> numpy_array
array([1, 2, 1, 1])

>>> python_list =  [1,2,"Hello",1]
>>> numpy_array = np.array(python_list)
>>> numpy_array
array([ '1' , '2' , 'Hello' , '1' ], dtype= '<U11' )
```

- Once a **NumPy array** is created, you ***can't add additional items*** to the array. However, in Python **lists**, we can use the *append()* function to add new items to the existing list.

  Adding a new element to a **NumPy array**, can be done in **3 steps**. First, you have to convert it to a Python list, then add an item to the list using the *append()* function, and lastly convert it again to a **NumPy array**.

```
>>> numpy_array = np.array([1,2,3,4])
>>> python_list = numpy_array.tolist()
>>> python_list.append(9)
>>> python_list
[1,2,3,4,9]
>>> numpy_array = np.array(python_list)
>>> numpy_array
array([1,2,3,4,9])
```

6. NumPy has special tools to help you easily create vectors and matrices of numbers. To tell NumPy how big you want your matrix or array to be, you give the size as a pair of numbers (rows and columns), called a **tuple**. For example, a 3×2 matrix would have 3 rows and 2 columns.

```
>>> np.ones((3,2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])

>>>  np.zeros((3,3))
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

7. Furthermore, we can use the **np.arange()** method to create arrays/lists, which works the same way as the built-in function **range()**. The only difference is that **arange()** will return a **NumPy array** or ***(ndarray)*** object instead of a Python **list** object.

```
>>> np.arange(0,20,1)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

>>> type(np.arange(1,5,1))
<class 'numpy.ndarray'>
```

8. After creating arrays, we can use the **np.reshape()** method to reshape it as we want. To do this, we tell NumPy the new shape we want by using a pair of numbers (called a tuple) as the second argument.

```
>>> np.reshape(np.arange(0,20,1),(4,5))
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

1.2 Matrix Operations

1. We can use the same **np.array()** function to construct matrices. To view the size, we use the **shape** attribute of the **NumPy array**.

```
>>> matrix = np.array([[1,2,3],[-1,-2,-3],[6,7,8]])
>>> matrix
array([[ 1,  2,  3],
       [-1, -2, -3],
       [ 6,  7,  8]])
>>> matrix.shape
(3, 3)
```

2. You can extract specific portions of an array using indexing, starting with 0, something similar to how you would do with Python lists **slicing**. For this, we use the **:** operator.

```
>>> matrix = np.array([[1,2,3],[-1,-2,-3],[6,7,8]])
# Get all rows of the second column
>>> matrix[:,2]
array([ 3, -3,  8])

# Get all columns of the second row
>>> matrix[2,:]
array([6, 7, 8])

# Get all rows of the first two columns
>>> matrix[:,:2]
array([[ 1,  2],
       [-1, -2],
       [ 6,  7]])
```

Pause and Reflect: What are the outputs of below lines of code?

```
>>> matrix = np.array([[1,2,3],[-1,-2,-3],[6,7,8]])
>>> matrix[:,0:2]
>>> matrix[:1,:]
>>> matrix[:2,:]
>>> matrix[2:,:]
>>> matrix[1:,:]
```

1.3 Mathematical Operations

1. We can use NumPy's in-built functions for element-wise operations on NumPy arrays. Let us use the **np.sin()** function to convert the elements inside a vector to its **sin()** values.

```
>>> vector = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
>>> np.sin(vector)
array([0.09983342, 0.19866933, 0.29552021, 0.38941834, 0.47942554,
       0.56464247])
```

2. We can use the **np.dot()** function to compute the dot product of two matrices.

```
>>> mat1 = np.array([[1, 2], [-1, -2]])
>>> mat2 = np.array([[4, 5], [-8, -9]])
>>> np.dot(mat1, mat2)
array([[-12, -13],
       [ 12,  13]])
```

3. To calculate the determinant of a matrix, we use the **det()** method provided in the Numpy linear algebra (**numpy.linalg**) module.

```
>>> np.linalg.det(mat2)
3.999999999999999
```

4. Often in linear algebra, we have to compute the inverse of a matrix. For this, the **numpy.linalg** module provides the **inv()** method.

```
>>> np.linalg.inv(mat2)
array([[-2.25, -1.25],
       [ 2.  ,  1.  ]])
```

5. Moving on, we can use the **solve()** function in the **numpy.linalg** module to solve a system of linear equations provided in the form of $Ax = b$.

   Given the system of equations:

$$x + y + z = 6$$
$$2y + 5z = 4$$
$$2x + 5y - z = 27$$

   We can solve these equations using NumPy and find solutions to x,y,z:

```
>>> A = np.array([[1, 1, 1], [0, 2, 5], [2, 5, -1]])
>>> B = np.array([6, 4, 27])
>>> np.linalg.solve(A, B)
array([ 2.71428571,  4.14285714, -0.85714286])
```

6. We can use the same programming structure (with **for** loops) to iterate through a NumPy matrix.

```
# CO1010 Lab 07
# NumPy with for loop

# Import the module
import numpy as np

# Create a matrix of size (3,3)
mat1 = np.reshape(np.arange(0, 9, 1), (3, 3))

# Get the shape of the matrix
(r, c) = mat1.shape

# Print all the values
for x in range(r):
    for y in range(c):
        print("Item @ (%d,%d): %d" % (x, y, mat1[x, y]))
```

7. NumPy also provides methods for statistical calculations. The following code segment describes the use of the **numpy.mean()** method. For this function, we have to provide an extra argument called **axis**. This argument specifies whether we are considering the row-wise mean or the column-wise mean in the matrix. By default, it will compute the mean of the whole matrix.

```
# CO1010 Lab 07
# NumPy with statistical methods

# Import the module
import numpy as np

# Create a matrix of size (3,3)
mat1 = np.reshape(np.arange(0, 9, 1), (3, 3))

print(mat1)
print("Mean value of all items: %.2f" % np.mean(mat1))
print("Mean value of columns:", np.mean(mat1, axis=0))
print("Mean value of rows:", np.mean(mat1, axis=1))
```

8. NumPy's **polynomial** and **roots** modules provide a set of useful functions for solving higher-order polynomials.

- To find the roots of a polynomial, we use the **numpy.roots()** function. The **coefficients** for the polynomial should be passed as a **Python list**.

  Let's find the roots of $x^3 + 3x + 1$:

```
>>> import numpy as np
>>> coeff = [1, 0, 3, 1]
>>> np.roots(coeff)
array([ 0.16109268+1.75438096j,  0.16109268-1.75438096j, -0.32218535+0.j ])
```

- To evaluate a polynomial at a given point, we use the **polyval()** function. As arguments, you have to provide the evaluation point and the coefficient vector. For instance, if we want to evaluate $x^2 + 4x + 9$ at 2 and 1, then the calculation looks like this:

```
>>> from numpy.polynomial.polynomial import polyval
>>> polyval(2, [9, 4, 1])
21.0
>>> polyval(1, [9, 4, 1])
14.0
```

9. A complete documentation of the NumPy library can be found at the official link: https://numpy.org/ or simply, you can use the **help()** function.

2 Matplotlib: Plotting 2D and 3D Graphs

1. **Matplotlib** is a powerful library in Python that allows users to create a wide range of visualizations, including 2D and 3D graphs.
2. Since this is not an in-built module in Python, you have to install it by typing the following command in the command-line interface (CLI) or terminal:

```
pip install matplotlib
```

3. A complete documentation on matplotlib can be found in the official website, https://matplotlib.org/3.1.1/

2.1 Linear 2D Graphs

1. To plot a linear graph, we are using several helper functions provided in **matplotlib**.
   - **plot(x, y, color)** : Function used to plot a graph. (**x** and **y** can be NumPy arrays or Python lists)
   - **title(str, fontsize)** : Title of the graph.
   - **xlabel(str, fontsize)** : X-axis label of the graph.
   - **ylabel(str, fontsize)** : Y-axis label of the graph.
   - **grid()** : Add a grid to the graph (optional).
2. Let's plot $\sin(x)$ where x is between $[0, 3\pi]$ as shown in Figure 1. For this step, first you have to import some libraries, and then you have to use the functions mentioned above. We use the **linspace(start, end, n)** function from NumPy to generate n number of values between the start and end.

```
# CO1010 Lab 07
# Linear Plots
# Import libraries

import matplotlib.pyplot as plt
import numpy as np

# Create data arrays
x = np.linspace(0, 3*np.pi, 1000)
```

```
y = np.sin(x)

# plot the graph
plt.plot(x,y,"g")
plt.title("sin(x) x in range [0,3pi]")
plt.xlabel("angle (radians)")
plt.ylabel("sin(x)")
plt.grid()
plt.show()
```

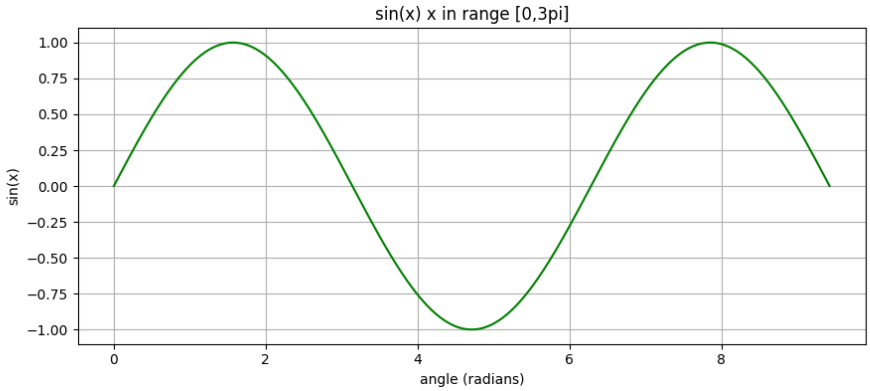

Figure 1: Linear Plot of $\sin(x)$

3. Figure 1 shows the rendered plot. You can save the plot using the **save** button at the bottom of the plot window.

2.2 Linear Plots with Multiple Series

1. We can use the same set of functions to render multiple graphs in the same figure.
2. Figure 2 shows a figure containing multiple plots ($\sin(x)$ and $\cos(x)$). To achieve this task, we can call the same **plot()** function again.
3. Since we are plotting two series, we need to add a **legend** to the graph. As shown in the following code segment, we use **legend()** function to add a legend to the graph. You have to make sure that the list of labels given in the legend() function is in the **same order** as the plotting order.

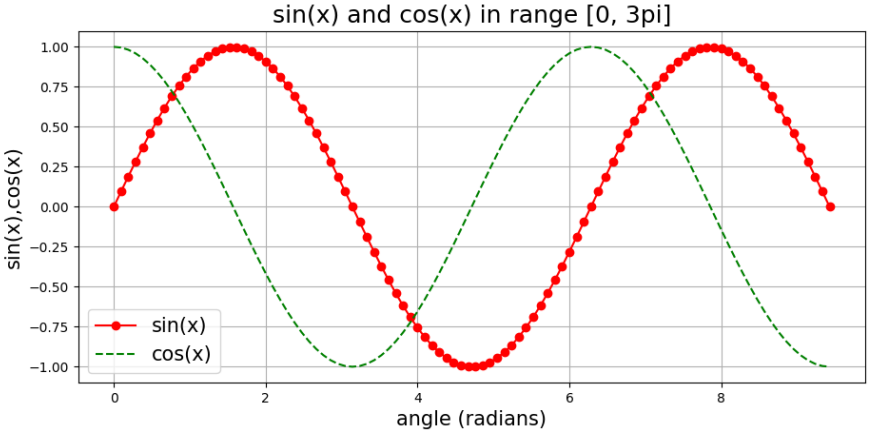

Figure 2: Multiple plots of $\sin(x)$ and $\cos(x)$

```
# CO1010 Lab 07
# Multiple Linear Plots
# Import libraries
import matplotlib.pyplot as plt
import numpy as np

# Create data arrays
x = np.linspace(0, 3*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Plot the graphs
plt.plot(x, y1, '-or')
plt.plot(x, y2, '--g')

# Format the figure
plt.title("sin(x) and cos(x) in range [0, 3pi]", fontsize=18)
plt.xlabel("angle (radians)", fontsize=15)
plt.ylabel("sin(x),cos(x)",fontsize=15)
plt.legend(["sin(x)","cos(x)"],fontsize=15)
plt.grid()
plt.show()
```

4. Additionally, **matplotlib** provides options to format the plotting line. Here, we can use the option **'-or'** to specify a continuous **red line segment with o pattern (scatter)**. Further, we can use the option `'--'` to specify a **dashed line** plot.

2.3 Bar Charts

1. To plot a bar chart **matplotlib** provides a function called **bar()**. This function has several mandatory arguments, and takes different arguments to modify the bar chart (alignment, width etc).

2. Additionally, we use the **xticks()** function to specify the x-labels of the graph. Similar to the **plot()** function, we need to provide the x-locations of the bars. The second argument is a tuple describing the independent variable (i.e., x).

3. The following code segment can be used to render a bar chart shown in Figure 3.

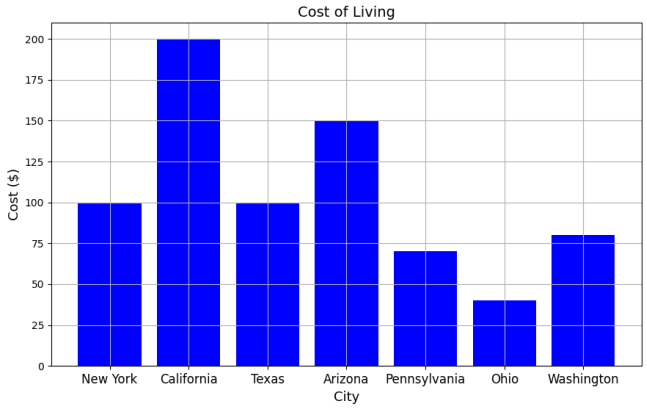

Figure 3: Bar Chart of Cost of Living

```
# CO1010 Lab 07
# Bar Plots
# Import libraries

from matplotlib.ticker import FuncFormatter
import matplotlib.pyplot as plt
```

```
import numpy as np

# x locations of bars
x = np.arange(7)
money = [100, 200, 100, 150, 70, 40, 80]
plt.bar(x, money, color=["b"] * 7)
plt.xticks(x, ("New York", "California", "Texas", "Arizona", "Pennsylvania", "Ohio", "
    Washington"), fontsize=12)
plt.grid()
plt.ylabel("Cost ($)", fontsize=13)
plt.xlabel("City", fontsize=13)
plt.title("Cost of Living", fontsize=14)
plt.show()
```

4. Edit the same code providing the width argument (enter the argument as **width=0.8**).

2.4 Pie Charts

1. **Matplotlib** provides **pie()** method to plot pie charts.

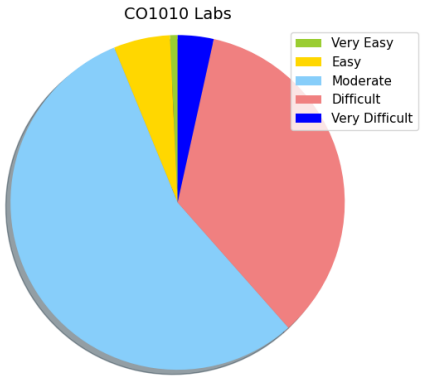

Figure 4: Pie chart for CO1010 lab difficulty level

2. Following code segment describes how to plot a pie chart using matplotlib as shown in Figure 4. Here, the **pie()** method multiple arguments as shown in the code segment, and it will return some objects which will be used as arguments to the **legend()** function.

```
# CO1010 Lab 07
# Pie chart
# Import libraries
import matplotlib.pyplot as plt

# Plot labels
labels = ["Very Easy", "Easy", "Moderate", "Difficult", "Very Difficult"]
# data (statistics)
sizes = [0.74 ,5.46, 55.33, 34.99,3.47]
# colours
colors = ["yellowgreen", "gold", "lightskyblue", "lightcoral", "b"]
# plot setup
patches, texts = plt.pie(sizes, colors=colors, shadow=True, startangle
=90)
plt.legend(patches, labels, loc="best")
plt.axis("equal")
plt.title("CO1010 Labs",fontsize=14)
plt.show()
```

3. See the difference after changing the **shadow** argument to **False**.

2.5 Sub-Plots

1. Often, you have to plot multiple graphs in the same figure. We can plot each graph on top of the first graph (as shown in the Multiple Series section 2.2) or we can plot them as separate graphs.

2. For this, we use the **subplot(r, c, loc)** method. The r argument specifies the number of rows in the subplot, c specifies the number of columns, and `loc` specifies the location of the graph/plot.

3. For instance, if you want to plot 3 graphs, you can specify the subplot pattern as **subplot(3, 1, loc)** or **subplot(1, 3, loc)**. As you can see, it is kind of a matrix of graphs where the **loc** parameter indicates the plotting location.

4. The following code segment describes how to plot multiple graphs in the same figure using **subplot()**:

```
# CO1010 Lab 07
# Sub-plots

# Import libraries
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 5*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# plot the graphs
plt.subplot(1, 2, 1)
plt.plot(x, y1, "-r")
plt.title("sin(x)", fontsize=18)
plt.xlabel("angle (radians)", fontsize=15)
plt.ylabel("sin(x)", fontsize=15)
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(x, y2, "g")
plt.title("cos(x)", fontsize=18)
plt.xlabel("angle (radians)", fontsize=15)
plt.ylabel("cos(x)", fontsize=15)
plt.grid()

plt.show()
```

5. The resulting figure looks like Figure 5.

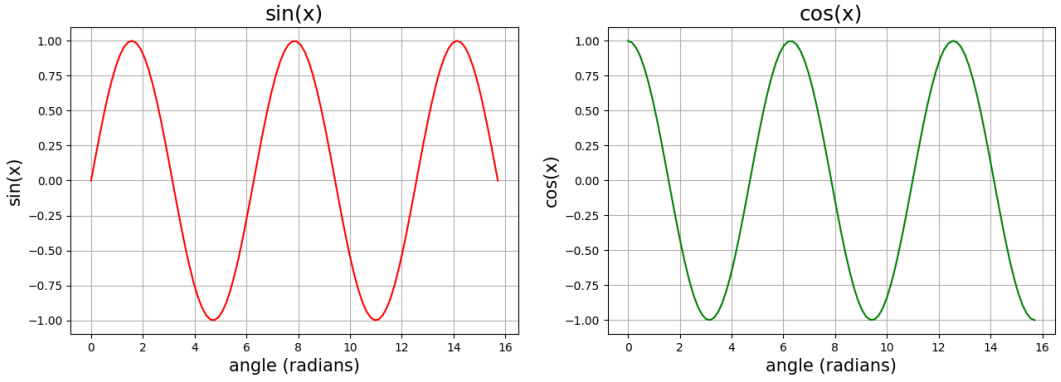

Figure 5: Sub-plots

2.6 3D Plots

Matplotlib provides the functionality to plot various types of 3D plots, including 3D Scatter, 3D Mesh, 3D Wireframe, and 3D Surface plots.

More information on how to create 3D plots using Matplotlib can be found at
https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html.

Let's plot a 3D surface plot of the function $z = \cos(x^2 + y^2)$.

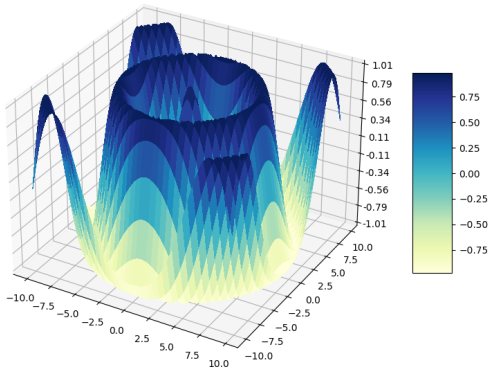

Figure 6: 3D Surface Plot

```python
# CO1010 Lab 07
# 3D Plots

# Import libraries
import matplotlib.pyplot as plt
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np

# Create a figure object
fig = plt.figure()
ax = fig.gca(projection="3d")

# Create the data X, Y
X = np.arange(-10, 10, 0.25)
Y = np.arange(-10, 10, 0.25)
X, Y = np.meshgrid(X, Y)
Z = np.cos(np.sqrt(X**2 + Y**2))

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap="YlGnBu", linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

2.7 Polynomial Curve Fitting

Often in engineering measurement tasks, we fit the parameter variation to some kind of polynomial to understand its behavior. Usually, we fit a set of data points (x, y) to a first-order polynomial (i.e., a line) to see whether it is increasing or not.

However, since most of the engineering problems we deal with comprise non-linear behavior, we sometimes fit the dataset to a second-order polynomial.

The following code segment shows how to fit a dataset (x,y readings) to first and second-order polynomials using NumPy's curve fitting functions. More information on these numerical computing methods can be found via the link: https://en.wikipedia.org/wiki/Curve_fitting.

```
# CO1010 Lab 07
# Polynomial curve fitting

# Import libraries
import numpy as np
from numpy.polynomial.polynomial import polyval, polyfit
import matplotlib.pyplot as plt

# Generate x, y using the polynomial 10 + x - 2x^2 + x^3
x = np.linspace(1, 10, 30)
y = [polyval(x_, [10, 1, -2, 1]) for x_ in x]

# Fit the data to 1st and 2nd order polynomials
f1 = polyfit(x, y, deg=1)
f2 = polyfit(x, y, deg=2)

# Evaluate them to estimate y
y_es_1 = [polyval(x_, f1) for x_ in x]
y_es_2 = [polyval(x_, f2) for x_ in x]

# Plot
plt.plot(x, y, "-og"), plt.plot(x, y_es_1, "b"), plt.plot(x, y_es_2, "r")
plt.legend(["Data Points", "1st Order", "2nd Order"])
plt.xlabel("X"), plt.ylabel("Y")
plt.grid()
plt.show()
```

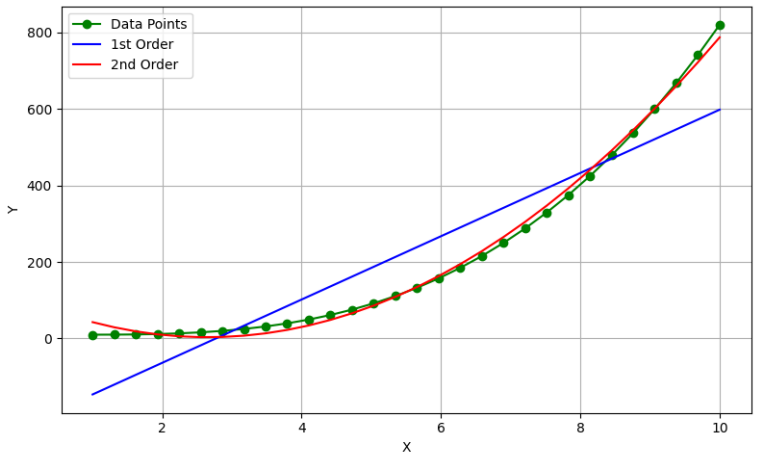

Figure 7: Polynomial Curve Fitting

3 Practice Exercises

1. Create a pie chart and a bar chart to represent the below information:

   - **Categories:** ["Apples", "Bananas", "Cherries", "Dates"]
   - **Values:** [30, 20, 25, 25]

   Plot the two charts side by side using **subplot(1, 2, loc)**. The expected output is given in Figure 8

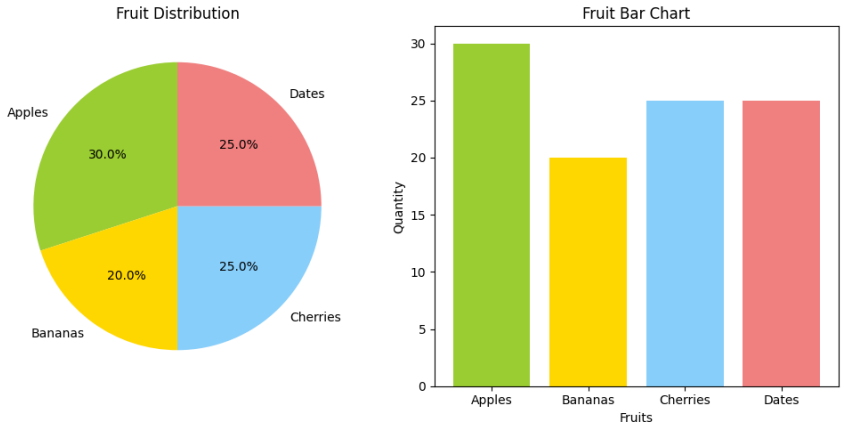

Figure 8: Expected output of practice exercise 1

2. Generate vector y whose elements are the values of the function:

$$y = \frac{\cos(x)}{1 + \sin(x)} \quad (2)$$

   for values of x from -2π to $+2\pi$, with increments $\frac{\pi}{24}$.

   Plot a graph for y versus x. Give meaningful labels to the x-axis and y-axis of the graph and give a suitable title to the plot. The expected output is given in Figure 8

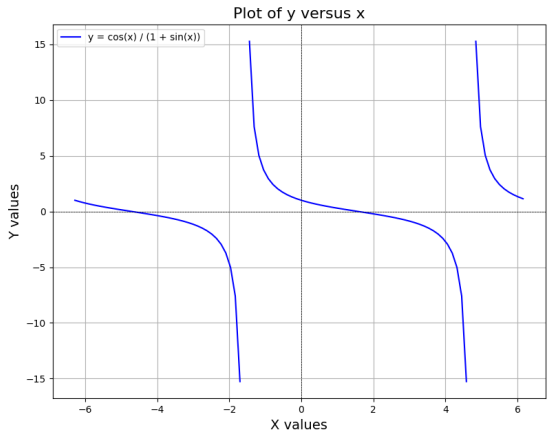

Figure 9: Expected output of practice exercise 2

Lab Task

Complete the exercise below, show it to an instructor, and get it marked within the lab hours.
Make sure to include meaningful comments in your Python scripts.

While peer-learning is highly encouraged in our labs, copying someone else's codes will earn you zero marks for all lab exercises.

1. Solve the polynomial equation below for its roots using NumPy.

$$y(x) = x^5 - 5x^3 + 4x = 0$$

2. Calculate the first derivative and integral of the polynomial.

3. Plot the **polynomial function**, its **first derivative**, and its **integral** over the **range of the roots** in the same plot. **Mark the roots** on the same plot.

   The expected output is shown in Figure 10

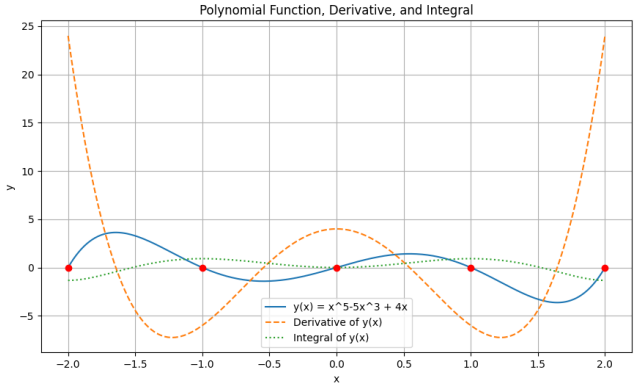

Figure 10: Expected output of the lab task

Bonus Task: Drawing the Mandelbrot Set

*While this task completion is **optional**, if you complete the task and get it marked by an instructor within the lab hours, you will earn **extra marks**.*

The **Mandelbrot set** (Figure 11) is a famous pattern in mathematics. Even though it has complex rules behind it, we can understand it visually and appreciate the beauty it creates.

Imagine you start with a number (called a **seed**). You apply a certain process to this number repeatedly. Sometimes the number stays small, sometimes it grows very big. The Mandelbrot set contains all the "seeds" where the number stays small no matter how many times we repeat the process. When you color these seeds based on how fast the number grows, you get an interesting and intricate image.

Curious to learn more? Check out this video on Mandelbrot sets: https://youtu.be/u9GAnW8xFJY

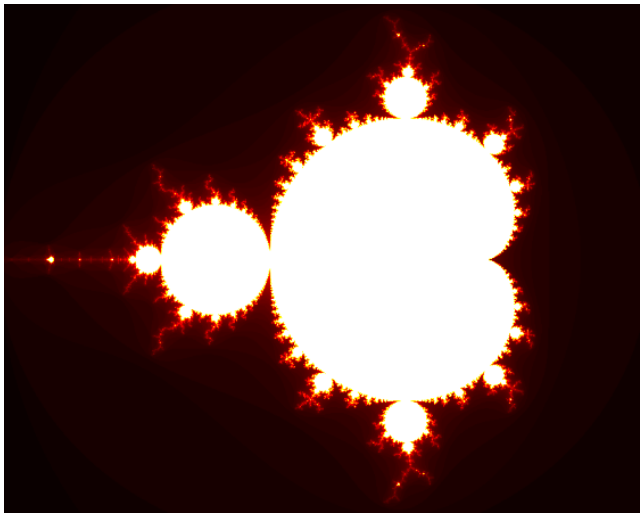

Figure 11: Expected output of the bonus task

Step-by-Step Guide to Draw the Mandelbrot Set

We'll use **NumPy** to perform calculations and **matplotlib** to draw the image.

1. Import Libraries

First, import the necessary libraries to work with numbers and to create the image.

```
import numpy as np
import matplotlib.pyplot as plt
```

2. Create a Grid of Points

Let's think of the image as a grid where each pixel has a coordinate. Each point in this grid corresponds to a complex number. Fill in the blanks in the code snippet below to create the grid:

```
# Define the image size
width, height = 800, 800

# Define the range of values on the complex plane
# TODO: use np.linspace() to create a set of 'real' values in the range (-2,1). The total
    number of 'real' values should equal to the 'width' of the image.

real = #TODO

# TODO: use np.linspace() to create a set of 'imaginary' values in the range (-1.5, 1.5).The
    total number of 'real' values should equal to the 'height' of the image.

imaginary = #TODO

# Create a 2D grid of complex numbers using np.meshgrid()
x, y = np.meshgrid(real, imaginary)
c = x + 1j * y
```

Try to create the complex 2D grid **c** using **for** loops instead of using **np.meshgrid()**. Which method takes longer to create the 2D grid as the number of points in the image increase?

3. Mandelbrot Function

Now, we define the **Mandelbrot** function, which will repeatedly apply the process to each point on the grid. At this stage, you don't need to worry about the underlying mechanics of Mandelbrot set generation. Simply copy the function below into your code.

```
def mandelbrot(c, max_iter):
    # Initialize z. z represents the complex number z_n in the iteration z_n+1 = z_n^2 + c.
    z = np.zeros_like(c)

    # Initialize the output array to store the iteration count for each point.
    # The output will be an integer array, where each value represents the number of
    iterations before the point 'escapes' (i.e., |z| > 2).
    output = np.zeros(c.shape, dtype=int)

    for i in range(max_iter):
        mask = np.abs(z) <= 2
        z[mask] = z[mask]**2 + c[mask]
        output[mask] = i

    return output
```

4. Generate the Mandelbrot Set

Next, use the function to generate the Mandelbrot set for the points on the grid.

```
max_iterations = 100
# TODO: Call the mandelbrot() function with the two parameters
mandelbrot_set = #TODO
```

5. Plot the Mandelbrot Set

Now, let's use **matplotlib** to display the Mandelbrot set.

- Use the **plt.imshow()** function to plot the **mandelbrot_set**.
- Add a suitable title to the plot.
- Add a color scale (color bar) next to a plot to represent the mapping between the data values and the colors used in the plot.

What happens if you use **plt.plot()** instead of **plt.imshow()**?

6. Change the color scheme

Change the color scheme of the plot by changing the **cmap** parameter in the **plt.imshow()** function. For more information on **cmap**, refer to this link.

Submission

Please follow these steps for submission:
1. Complete the tasks and save all your Python scripts.
2. Compress all scripts into a single ZIP folder.
3. Rename the ZIP folder to `E22xxx.ZIP`, where `xxx` represents the last three digits of your student registration number.
4. Submit the ZIP folder to the link provided on the FEeLS course page.

Steps for Creating a ZIP File

If you are unfamiliar with how to create a ZIP file, please follow the steps below:
1. Select the all Python scripts.
2. Right-click on one of the selected files.
3. Choose **Send to → Compressed (zipped) folder**.
4. A new ZIP folder will appear. Rename this folder to `E22xxx.ZIP`, replacing `xxx` with the last three digits of your registration number.