Archisman Chakraborti

Semester 6, UG, PHSA, 3rd Year
Roll no. 167
Date 22/02/2023

Exercise 1: Recap

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy.optimize import curve_fit
        plt.style.use(["science", "notebook", "grid"])
```

Question 1 - Array Manipulation

Form the 2-D array (without typing it in explicitly):

```
[ [ 1 , 6 , 1 1 ] ,
[ 2 , 7 , 1 2 ] ,
[ 3 , 8 , 1 3 ] ,
[ 4 , 9 , 1 4 ] ,
[ 5 , 10 , 1 5 ] ]
```

and generate a new array containing its 2nd and 4th rows.

```
In [ ]: my_arr : np.ndarray = np.arange(1,16).reshape(3, 5).T
        print(my_arr)

[[ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]
 [ 5 10 15]]
```

```
In [ ]: ## Second row
        print(f"Second row = {my_arr[1,:]}")

        # 4th row
        print(f"Fourth row = {my_arr[3,:]}")

Second row = [ 2  7 12]
Fourth row = [ 4  9 14]
```

Question 2 -Generating Numbers from the normal dstribution

Write a NumPy program to generate five random numbers from the normal distribution.

```
In [ ]: randn_nos : np.ndarray = np.random.randn(5)
        print(f"Random numbers from normal distribution = {randn_nos}")

Random numbers from normal distribution = [-0.89546656  0.3869025  -0.51080514 -1.18063218 -0.02818223]
```

Question 3 - Sorting complex numbers

Write a NumPy program to sort a given complex array using the real part first, then the imaginary part (use help on the function `sort_complex` to solve this).

```
In [ ]: ## Generating random complex numbers from a uniform distribution
        N_NUMBERS : int = 10
        LOWER_BOUND : int = -5
        UPPER_BOUND : int = 5

        rand_complex : np.ndarray = np.random.uniform(LOWER_BOUND, UPPER_BOUND, N_NUMBERS) +\
                    1j * np.random.uniform(LOWER_BOUND, UPPER_BOUND, N_NUMBERS)
        print(f"Random complex numbers = \n {rand_complex}")

Random complex numbers =
 [ 4.76761088-4.35852504j  1.0484552 +1.92472119j  2.39263579+0.66601454j
 -4.60812208-2.34610509j -2.17193037+0.23248053j -3.79803439-4.06059489j
 -2.03859802+0.75946496j -3.81272281+4.29296198j -1.82016821-1.81431048j
 -0.85737005+1.6741038j ]
```

```
In [ ]: # Sort the random complex numbers
        sorted_rand_nos = np.sort_complex(rand_complex)
        print(f"Sorted random complex numbers = \n {sorted_rand_nos}")

Sorted random complex numbers =
 [-4.60812208-2.34610509j -3.81272281+4.29296198j -3.79803439-4.06059489j
 -2.17193037+0.23248053j -2.03859802+0.75946496j -1.82016821-1.81431048j
 -0.85737005+1.6741038j   1.0484552 +1.92472119j  2.39263579+0.66601454j
  4.76761088-4.35852504j]
```

Question 4: Plotting the wave function

The wave-function (ψn(x)) corresponding to different energy eigen-states (charac- terised by En) of a particle in a box of length L in one dimension (confined within

an infinite potential well) in quantum mechanics is given by

$$\psi_n = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right)$$

where n is an integer. Write a python code, using numpy and matplotlib, to plot the wave-functions for n = 1 ,2 ,3 in a single figure (use subplot). Take the length of the box, L = 1. The plot should have a global title, individual titles indicating the value of n and x and y-axes labels.

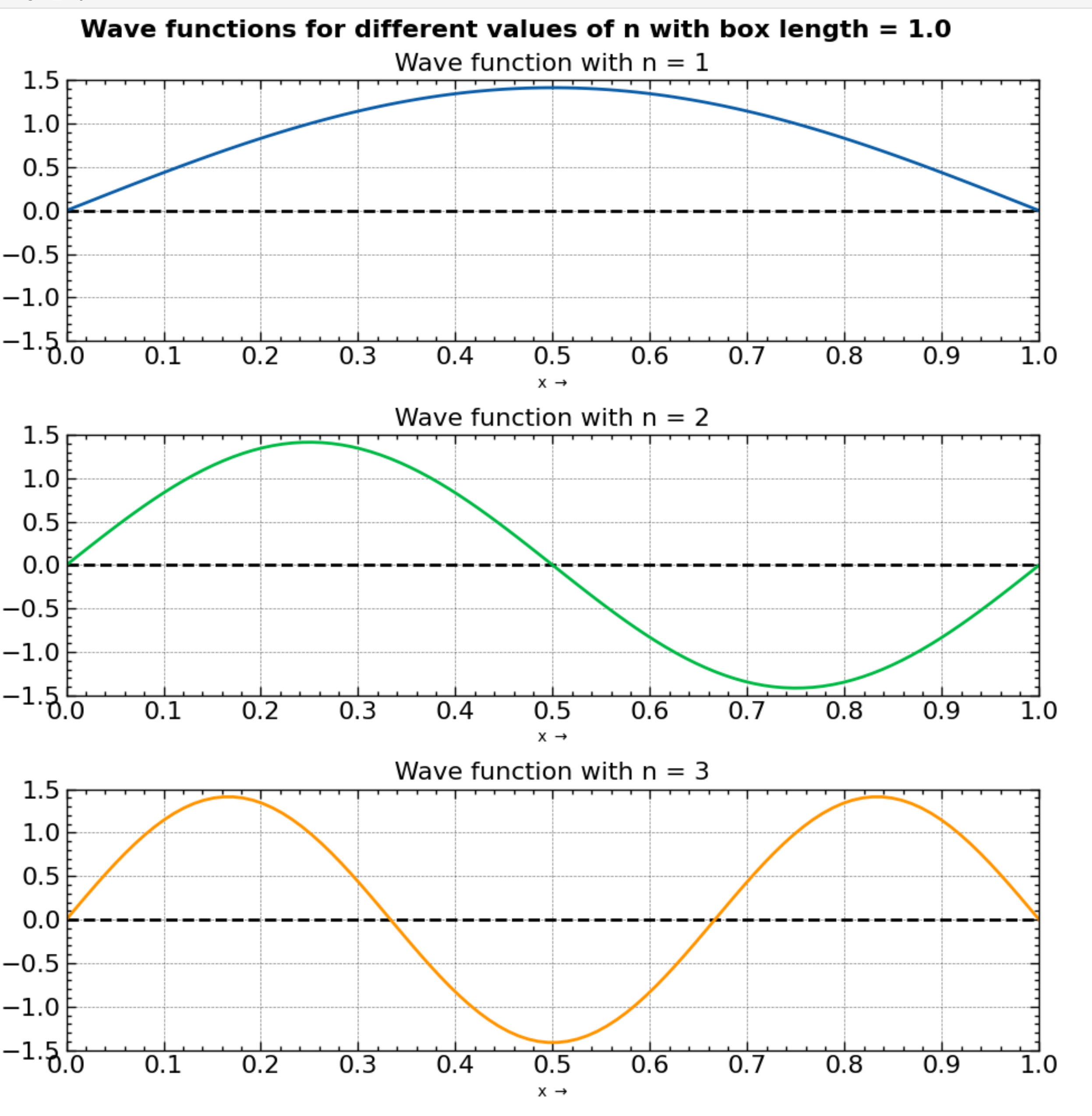
```
In [ ]: def wave_func(x:np.ndarray, L:float, n:int):
            """Function to calculate the wave function for a given x, L and n
            Args:
                x (np.ndarray): x values
                L (float): Length of the box
                n (int): n value
            Returns:
                np.ndarray: Wave function values"""
            return np.sqrt(2/L) * np.sin(n*np.pi*x/L)

        # Constants
        L : float = 1.0
        n : list = [1, 2, 3]
        x : np.ndarray = np.linspace(0, L, 100)
```

```
In [ ]: fig, ax = plt.subplots(3, 1, figsize=(10, 10))

        # Plot the wave functions
        for n_idx, n_val in enumerate(n):
            y = wave_func(x, L, n_val) # Calculate the wave function values
            ax[n_idx].axhline(0, color="k", linestyle="--")
            ax[n_idx].plot(x, y, color = f"C{n_idx}",
                           linestyle = "-", linewidth = 2)
            ax[n_idx].set_xlabel(r"x $\rightarrow$")
            ax[n_idx].set_ylabel(rf"$\psi_{n_val}(x) \; \rightarrow$")
            ax[n_idx].set_title(f"Wave function with n = {n_val}")
            ax[n_idx].set_xlim(0, L)
            ax[n_idx].set_ylim(-1.5, 1.5)
            ax[n_idx].locator_params(axis = "both", nbins = 10)

        # Global parameters
        plt.suptitle(f"Wave functions for different values of n with box length = {L}",
                     fontsize=16, fontweight="bold")
        plt.tight_layout()
```



Question 5: Matrix solving

5. Create a 100 × 100 random matrix, A and a 100×1 random matrix b using the

available functions in Numpy. Let these matrices define the system of equations

$$Ax = b$$

Solve for x, using solver from `linalg` sub-package from Scipy and also by using the inv function to take the inverse of a matrix. Which process is faster?

```
In [ ]: A =  np.random.randint(low=-10, high=10, size=(100, 100))
        b = np.random.randint(low=-10, high=10, size=(100, 1))
```

Using the matrix A and the vector b solve the system of linear equations $Ax = b$.

- Using the `solve` function from the `numpy.linalg` module.

```
In [ ]: from scipy.linalg import solve
        X_linalg : np.ndarray = (solve(A,b).T).squeeze()
        X_linalg

Out[ ]: array([-0.70752285,  0.40506242,  2.25555686, -0.07697214,  0.7877112 ,
                1.21859326,  0.91642236, -2.90239689, -2.61689427, -1.34939303,
               -0.84491074,  1.28257139,  2.04933731,  0.25296794,  0.20778487,
                0.02809647,  0.48750993,  3.38195461, -0.36646886, -0.18983898,
                0.38176373, -3.10759505,  0.05598491, -1.61427358, -0.16694052,
                1.06963472,  0.42637416,  1.32766572, -0.59137531,  0.9065506 ,
                0.60725862, -1.57749351, -1.13083732, -0.12291138, -2.23397516,
                0.06060809,  1.24312793, -1.26630829, -1.42208281, -0.59256336,
               -4.01615765, -1.25552134, -1.99040014, -0.2655614 , -0.54635392,
                0.06811491,  0.0867268 ,  0.95634844,  1.2507637 , -0.75140922,
               -1.22308092,  1.15263705, -0.00483627,  0.12406654,  0.26881737,
               -0.78833465, -0.44189365,  1.99473789,  1.26583162, -2.99702688,
                2.49406131,  0.23427915, -0.74316691, -0.34187308,  0.31593435,
                2.3217607 , -1.59331989, -1.00364137,  0.03230556, -0.72295345,
                1.83596611, -0.82846391,  0.75341457,  0.9890183 ,  0.27880939,
                0.72225099, -1.87694641,  0.01955515,  1.23060276,  0.03308443,
               -1.46832371, -2.16953542,  0.30946677, -0.13707271,  0.04795578,
               -0.5649576 , -0.49973304, -0.11333093,  0.10362992,  0.62989526,
                2.28232179, -1.59106421,  0.23414482, -0.02399699,  1.29777144,
                0.29946044, -0.39905107, -2.04420039, -0.43405492, -1.97878157])
```

- Using the `inv` function from the `numpy.linalg` module.

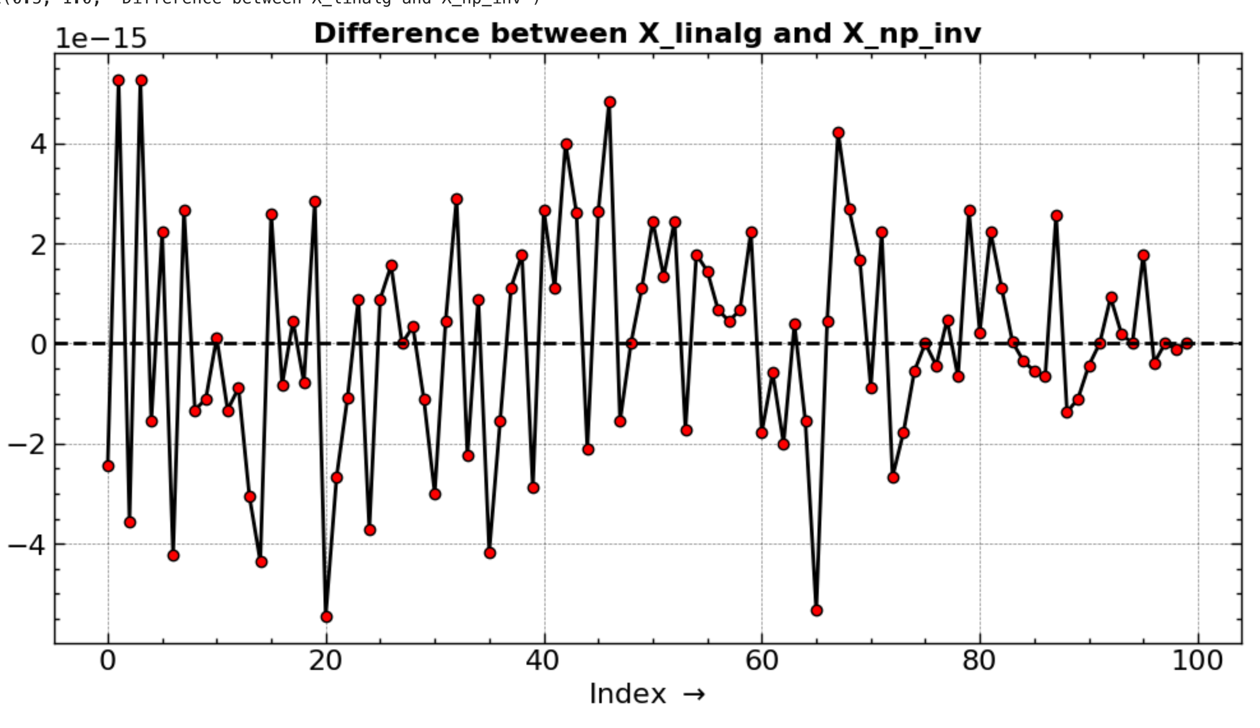
```
In [ ]: X_np_inv : np.ndarray  = ((np.linalg.inv(A) @ b).T).squeeze()
        X_np_inv

Out[ ]: array([-0.70752285,  0.40506242,  2.25555686, -0.07697214,  0.7877112 ,
                1.21859326,  0.91642236, -2.90239689, -2.61689427, -1.34939303,
               -0.84491074,  1.28257139,  2.04933731,  0.25296794,  0.20778487,
                0.02809647,  0.48750993,  3.38195461, -0.36646886, -0.18983898,
                0.38176373, -3.10759505,  0.05598491, -1.61427358, -0.16694052,
                1.06963472,  0.42637416,  1.32766572, -0.59137531,  0.9065506 ,
                0.60725862, -1.57749351, -1.13083732, -0.12291138, -2.23397516,
                0.06060809,  1.24312793, -1.26630829, -1.42208281, -0.59256336,
               -4.01615765, -1.25552134, -1.99040014, -0.2655614 , -0.54635392,
                0.06811491,  0.0867268 ,  0.95634844,  1.2507637 , -0.75140922,
               -1.22308092,  1.15263705, -0.00483627,  0.12406654,  0.26881737,
               -0.78833465, -0.44189365,  1.99473789,  1.26583162, -2.99702688,
                2.49406131,  0.23427915, -0.74316691, -0.34187308,  0.31593435,
                2.3217607 , -1.59331989, -1.00364137,  0.03230556, -0.72295345,
                1.83596611, -0.82846391,  0.75341457,  0.9890183 ,  0.27880939,
                0.72225099, -1.87694641,  0.01955515,  1.23060276,  0.03308443,
               -1.46832371, -2.16953542,  0.30946677, -0.13707271,  0.04795578,
               -0.5649576 , -0.49973304, -0.11333093,  0.10362992,  0.62989526,
                2.28232179, -1.59106421,  0.23414482, -0.02399699,  1.29777144,
                0.29946044, -0.39905107, -2.04420039, -0.43405492, -1.97878157])
```

An interesting this to note is that the solutionsof both these functions are not exactly the same, they are slightly varying from each other. We can examine this by making a simply comparison plot among these 2 solutions.

```
In [ ]: # Plot the difference between X_linalg and X_np_inv
        plt.figure(figsize=(12, 6))
        X_diff = X_linalg - X_np_inv
        plt.plot(X_diff, color="k", linestyle="-", linewidth=2,
                 marker="o", markersize=6, markerfacecolor="r")
        plt.xlabel(r"Index $\rightarrow$", fontsize=16)
        plt.ylabel(r"$X_{linalg} - X_{np\_inv} \; \rightarrow$", fontsize=16)
        plt.axhline(0, color="k", linestyle="--")
        plt.title("Difference between X_linalg and X_np_inv", fontsize=16, fontweight="bold")

Out[ ]: Text(0.5, 1.0, 'Difference between X_linalg and X_np_inv')
```



Timing the 2 methods

We can also time the 2 methods to see which one is faster. We will use the `timeit` function from the `timeit` module to time the 2 methods.

```
In [ ]: print(f"""===================\nTime for scipy linalg\n===================""")
        %timeit solve(A,b)

        print(f"""\n===================\nTime for numpy Inverse\n===================""")
        %timeit np.linalg.inv(A) @ b

        print(f"""\n===================\nTime for numpy Linalg\n===================""")
        %timeit np.linalg.solve(A,b)

===================
Time for scipy linalg
===================
271 µs ± 43.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

===================
Time for numpy Inverse
===================
314 µs ± 47.6 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

===================
Time for numpy Linalg
===================
178 µs ± 9.71 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Question 6 - Curve Fitting

In this exercise, we will go through the use of the random module in the Numpy package. We will also learn to use the curve fitting tool in Scipy package for data fitting. Please go through the documentation to learn about the usage of these modules, since they would be extremely handy in the following lab sessions. (a) Generate a set of 50 equally spaced datapoints (x) between −5 and +5 and plot

the function

$$y = 2.9sin(1.5x) + \text{a random component}$$

, drawn from a normal distribution, by initializing the random number generator to 0 (use the seed method

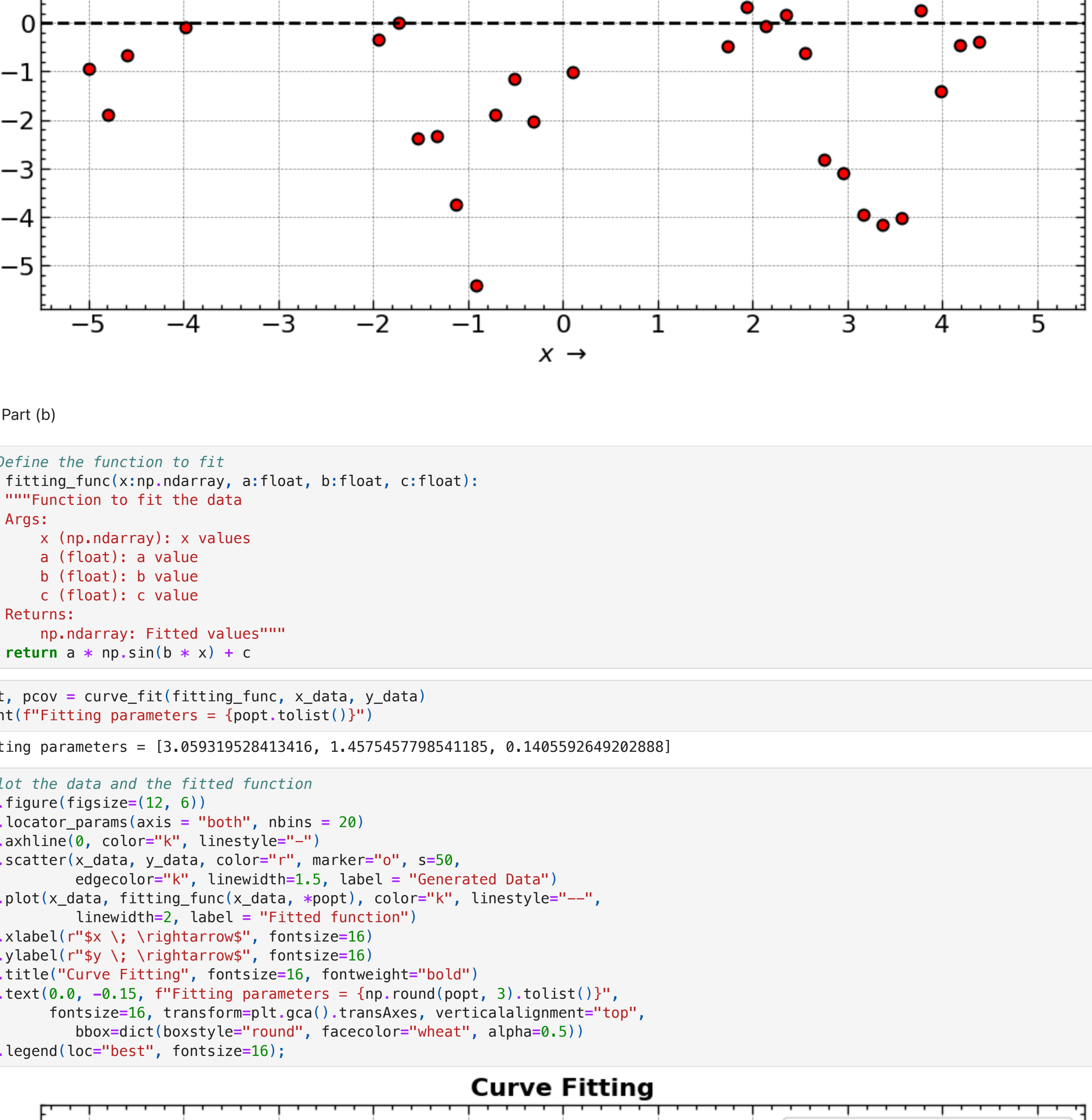
in the random module in Numpy). Use the scatter function in matplotlib to plot this data and label it suitably. (b) Use the curve fitting tool in the optimize sub-module in Scipy to fit this data. Print the best fit parameters on screen as output. Plot the data along with the fitted function, label the axes and the traces suitably.

```
In [ ]: # Set random seed
        np.random.seed(0)

        # Generate data
        N_POINTS : int = 50
        x_data : np.ndarray = np.linspace(-5, 5, N_POINTS)
        y_data : np.ndarray = 2.9 * np.sin(1.5 * x_data)  + np.random.randn(N_POINTS)
```

- Part (a)

```
In [ ]: ## Plot the data
        plt.figure(figsize=(12, 6))
        plt.locator_params(axis="both", nbins=20)
        plt.axhline(0, color="k", linestyle="--")
        plt.scatter(x_data, y_data, color="r", marker="o", s=50,
                    edgecolor="k", linewidth=1.5)
        plt.xlabel(r"$x \; \rightarrow$", fontsize=16)
        plt.ylabel(r"$y \; \rightarrow$", fontsize=16)
        plt.title("Generated Data", fontsize=16, fontweight="bold");
```



- Part (b)

```
In [ ]: ## Define the function to fit
        def fitting_func(x:np.ndarray, a:float, b:float, c:float):
            """Function to fit the data
            Args:
                x (np.ndarray): x values
                a (float): a value
                b (float): b value
                c (float): c value
            Returns:
                np.ndarray: Fitted values"""
            return a * np.sin(b * x) + c
```

```
In [ ]: popt, pcov = curve_fit(fitting_func, x_data, y_data)
        print(f"Fitting parameters = {popt.tolist()}")

Fitting parameters = [3.059319528413416, 1.4575457798541185, 0.1405592649202888]
```

```
In [ ]: # Plot the data and the fitted function
        plt.figure(figsize=(12, 6))
        plt.locator_params(axis = "both", nbins = 20)
        plt.axhline(0, color="k", linestyle="-")
        plt.scatter(x_data, y_data, color="r", marker="o", s=50,
                    edgecolor="k", linewidth=1.5, label = "Generated Data")
        plt.plot(x_data, fitting_func(x_data, *popt), color="k", linestyle="--",
                    linewidth=2, label = "Fitted function")
        plt.xlabel(r"$x \; \rightarrow$", fontsize=16)
        plt.ylabel(r"$y \; \rightarrow$", fontsize=16)
        plt.title("Curve Fitting", fontsize=16, fontweight="bold")
        plt.text(0.0, -0.15, f"Fitting parameters = {np.round(popt, 3).tolist()}",
                 fontsize=16, transform=plt.gca().transAxes, verticalalignment="top",
                    bbox=dict(boxstyle="round", facecolor="wheat", alpha=0.5))
        plt.legend(loc="best", fontsize=16);
```