Atma Ram Sanatan Dharma College, University of Delhi

Name: Arvind Kumar Patel

College-Roll-No: 22/26089

University-Roll-No: 22003567041

Course: B.Sc Hons Physics

Paper: Mathematical Physics-3

Date: __ - __ - 2023
(Arvind Kumar Patel)

Practical_01

Aim: Determine the nth roots of a complex number and represent it In Cartesian and polar form.

Algorithm:

1. We import the cmath module to work with complex numbers and mathematical operations.

2. The function takes a complex number z and an integer n as inputs.
It calculates the nth roots of z.

3. Inside the find nth_roots funption, we first calculate the magnitude z- and argument theta of the complex number z using the abs and cmath.phase Functions.

4. We then calculate the nth roots in polar form using the formula mentioned earlier. We
vary the value of k from O to n-1 to find all the nth roots. These roots are stored in the
roots list.

5. The display_roots function takes the list of roots and displays them in both Cartesian and polar forms. It iterates through the list of roots, converts each root from polar to Cartesian form, and prints both forms.

6. Finally, we input a complex number z and a value for n and call the functions to find
and display the nth roots.

7. You can change the values of z and n to find the nth roots of different complex numbers and for different values of n

Code:

```
import cmath
a = float(input("Enter the real part(a):"))
b = float(input("Enter the imaginary part(b):"))
n = int(input("Enter the value of n:"))
z = complex(a,b)
r = abs(z)
theta = cmath.phase(z)

roots = []
```

```
for k in range(n):
        root_polar = cmath.rect(r ** (1/n), (theta + 2 * cmath.pi * k) / n)
        roots.append(root_polar)

for k, root_polar in enumerate(roots):
        root_cartesian = complex(root_polar.real, root_polar.imag)
        print(f"Root {k+1} (Cartesian): {root_cartesian}")
        print(f"Root {k+1} (Polar): {abs(root_polar):.2f} *
(cos({cmath.phase(root_polar):.2f}) + i *
sin({cmath.phase(root_polar):.2f}))")
```

Output:

```
PS P:\Mp_Project_sem_3> & "C:/Program Files/Python311/python.exe"
p:/Mp_Project_sem_3/nth_root_of_complex_no.py
Enter the real part(a):2
Enter the imaginary part(b):3
Enter the value of n:4
Root 1 (Cartesian): (1.3365960777571289+0.33517136966065714j)
Root 1 (Polar): 1.38 * (cos(0.25) + i * sin(0.25))
Root 2 (Cartesian): (-0.33517136966065697+1.3365960777571289j)
Root 2 (Polar): 1.38 * (cos(1.82) + i * sin(1.82))
Root 3 (Cartesian): (-1.3365960777571289-0.33517136966065714j)
Root 3 (Polar): 1.38 * (cos(-2.90) + i * sin(-2.90))
Root 4 (Cartesian): (0.3351713696606571-1.3365960777571289j)
Root 4 (Polar): 1.38 * (cos(-1.33) + i * sin(-1.33))
```

Project_02

Aim: Transformation of complex numbers as 2-D vectors e.g. translation, scaling, rotation, reflection.

Algorithm:

You import the math module to use mathematical functions and constants.

You define an original complex number z as complex(2, 2) (which is equivalent to 2 + 2i). For each transformation:

- Translation: You define a complex number z_2 as complex(2.1) and add it to z to translate the original complex number.

- Scaling: You define a scaling factor as scaling_factor and multiply z by this factor to scale it.

- Rotation: You specify an angle in degrees (theta) and convert it to radians using math.radians(). Then, you create a complex number rotation_factor using the cosine and sine of the angle and rotate z by multiplying it by this factor.

- Reflection: You define a complex reflection factor as complex(1, -1) and reflect z by multiplying it with this factor.

You print the original complex number z and the results of the transformations, showing the translated, scaled, rotated, and reflected complex numbers.

Code:

```python
import math
z = complex(2,2)

#Translation
z_2 = complex(2.1)
translated_z = z+z_2

#Scaling
scaling_factor = 2
scaled_z = z*scaling_factor

#Rotation
theta = 45
angle_radian = math.radians(theta)
rotation_factor = complex(math.cos(angle_radian),math.sin(angle_radian))
rotated_z = z*rotation_factor

#Reflaction(multiplying by an other complex number)
reflection_factor = complex(1,-1)
reflected_z = z*reflection_factor

#Printing
print(f"Original Complex no: {z}\nTransleted comple no:{translated_z}\n"
[redacted]
f"Scaled Complex no:{scaled_z}\nRotated complex no:{rotated_z}\n"
      f"Reflected Complex no:{reflected_z}")
```

Output:

```
PS P:\Mp_Project_sem_3> & "C:/Program Files/Python311/python.exe"
p:/Mp_Project_sem_3/transformation_of_complex_number.py
Original Complex no: (2+2j)
Transleted comple no:(4.1+2j)
Scaled Complex no:(4+4j)
Rotated complex no:2.8284271247461903j
Reflected Complex no:(4+0j)
```

Project_03

Aim: To Solve an Value Problem (IVP) for Order Ordinary Differential Equation.

Algorithm:

1. Import necessary libraries:
    a. numpy is imported as np for numerical operations.
    b. matplotlib.pyplot is imported as plt for creating plots.
    c. odeint is imported from scipy.integrate for solving the ODE.

2. Define function This function represents the first-order ODE dN/dt = -kN, where N is the quantity of the radioactive substance, t is time, and k is the decay constant.

3. Then set initial condition
    N0 is the initial quantity of the radioactive substance.
    k is the decay constant.
    t is an array of time points where you want to evaluate the solution. In this case, it spans from 0 to 1000 with 10 time points.

4. Using odeint to solve ODE odeint is used to solve the ODE defined by the radioactive_decay function with initial conditions N0, over the specified time points t, and passing the decay constant k as an argument.

5. Now Plot This code creates a plot of time (t) on the x-axis and the quantity of the radioactive substance (solution) on the y-axis.It sets labels, a title, and a legend for the plot.

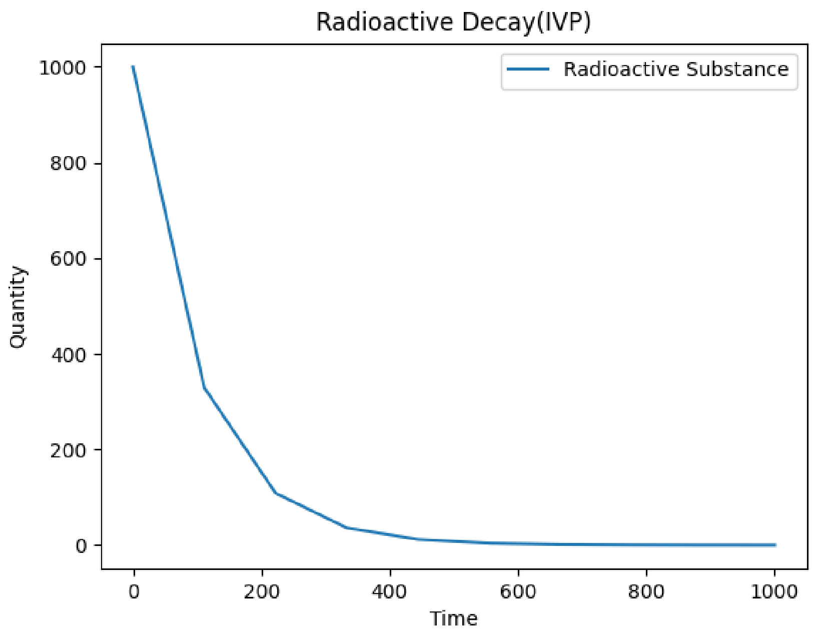
Finally, it displays the plot using plt.show().

Code:

```
#   Lets_take_example_1st_order_ODE_Radioactive_decay

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

#Function defining the ODE dN/dt = -kN
def radioactive_decay(N,t,k):
    return -k*N

N0 = 1000
k = 0.01
t = np.linspace(0,1000,10)
```

```
#Solvi the ODE
solution = odeint(radioactive_decay,N0,t,args=(k,))

#Plot the solution
plt.plot(t,solution,label="Radioactive Substance")
plt.xlabel("Time")
plt.ylabel("Quantity")
plt.title("Radioactive Decay(IVP)")
plt.lagend()
plt.show()
```

Output:



Project_04

Aim: The equilibrium temperature of a bar of length L with insulated horizontal sides and the ends maintained at fixed temperatures.

Algorithm:

1.Import necessary libraries:

- numpy is imported as np for numerical operations.
- matplotlib.pyplot is imported as plt for creating plots.

- odeint is imported from scipy.integrate for solving ODEs.

2.Define the given parameters:
- L is the length of the rod in meters.
- T1 is the temperature at one end of the rod in degrees Celsius.
- T2 is the temperature at the other end of the rod in degrees Celsius.
- Ta is the surrounding air temperature in degrees Celsius.
- h is the heat transfer coefficient in m^-2.

3.Convert temperatures to Kelvin:
- The temperatures T1, T2, and Ta are converted to Kelvin by adding 273.15 to each value.

4.Define the differential equation:
- The diff_eq function represents the differential equation for heat transfer. It takes a state vector T (temperature and its derivative) and the spatial coordinate x. The equation models heat transfer from the rod to the surrounding air, and it returns the temperature gradient and heat transfer at that point.

5.Implement the shooting method to solve the ODE:
- The solve_shooting_method function takes an initial guess for the temperature at the rod's starting point.
- It sets up a spatial grid using np.linspace from 0 to L with 100 points.
- The initial conditions T_initial are defined based on the guessed temperature and a derivative of 0.
- The ODE is solved using odeint by providing the diff_eq function, initial conditions, and the spatial grid.
- The function returns the temperature profile along the rod.

6.Find the initial guess for the shooting method:
- The find_initial_guess function performs a binary search to find the correct initial guess for the shooting method.
- It starts with initial guesses based on the boundary conditions (T1_K and T2_K) and iteratively refines the guess until the final temperature matches T2_K.

7.Solve the problem:
- The code calls find_initial_guess to obtain the initial guess for the shooting method.
- It then uses the shooting method to solve the ODE with the obtained initial guess, resulting in the temperature_profile.

8.Plot the temperature distribution:
- It generates a spatial grid x and plots the temperature distribution along the rod.
- It adds a horizontal dashed line representing the surrounding air temperature (Ta).
- Labels, title, legend, and grid settings are applied to the plot.
- Finally, the plot is displayed using plt.show().

When you run this code, it will calculate and visualize the temperature distribution along the rod, taking into account the given boundary conditions and surrounding air temperature. The result is presented in degrees Celsius.

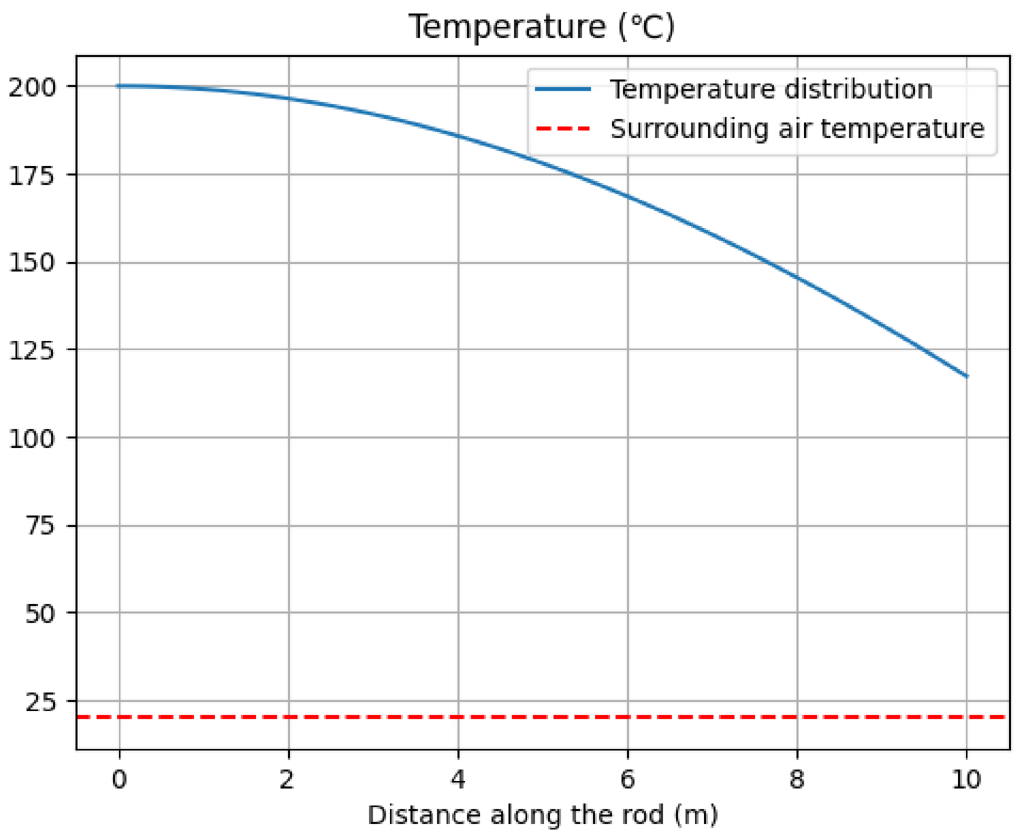
Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

#Given Parameter
L = 10   #length of the rod in meter
T1 = 40  #Temp at one end in degree celsius
T2 = 200 #Temp at other end in degree celsius
Ta = 20  #Surrounding air temp in degree calsius
h = 0.01  #Heat transfer Coefficient in m^-2

#Converting temp in Kelvin
T1_K = T1 + 273.15
T2_K = T2 + 273.15
Ta_K = Ta + 273.15

#Function for the differential equation
def diff_eq(T,x):
    return [T[1],h*(Ta_K - T[0])]

#Function to solve the differential equ using the shooting method
def solve_shooting_method(T_guess):
    x = np.linspace(0,L,100)
    T_initial = [T_guess, 0]
    T_solution = odeint(diff_eq, T_initial, x)
    return T_solution[:,0]

#Perform a binary search to finnd the correct intial gues
def find_initial_guess():
    lower_bound = T1_K
    upper_bound =T2_K

    while abs(upper_bound - lower_bound) > 1e-5:
        guess = (lower_bound + upper_bound)/2
        T_end = solve_shooting_method(guess)[-1]

        if T_end > T2_K:
            upper_bound = guess
        else:
            lower_bound = guess
    return guess

#Solve the problem
initial_guess = find_initial_guess()
```

```python
temperature_profile = solve_shooting_method(initial_guess)

#Plot the temp distribution
x = np.linspace(0,L,100)
plt.plot(x, temperature_profile - 273.15, label="Temperature distribution")
plt.axhline(y=Ta, color="r",linestyle="--", label="Surrounding air
temperature")
```

```
plt.xlabel("Distance along the rod (m)")
plt.title("Temperature (℃)")
plt.legend()
plt.grid()
plt.show()
```

Output:



Project_05

Aim: Solving a definite integral by gauss legendre quadrature method. Application representation of a function as linear combination legendre polynomial.

Algorithm:

1. It plots the `cos(x)` function and the first 5 Legendre polynomials.

2. It calculates the integral of the `cos(x)` function over the interval [-1, 1] using Gauss-Legendre quadrature.

1. Import necessary libraries:

   - The code begins by importing the required libraries: `numpy` for numerical operations, `matplotlib.pyplot` for creating plots, and `scipy.special.legendre` for generating Legendre polynomials.

2. Define the interval for plotting:

   - `x_min` and `x_max` set the interval for which the `cos(x)` function and Legendre polynomials will be plotted, which is from -1 to 1.

3. Create an array of `x` values:

   - `np.linspace(x_min, x_max, 1000)` generates an array of `x` values within the specified interval. This array is used for plotting.

4. Plot the `cos(x)` function:

   - The code plots the `cos(x)` function using `plt.plot(x, np.cos(x), label='cos(x)', linewidth=2)`. This line creates the plot of the `cos(x)` function and adds it to the legend with a label "cos(x)".

5. Choose the degree of the Gauss-Legendre quadrature:

   - `degree = 4` sets the degree of the Gauss-Legendre quadrature. The degree determines the number of nodes and weights used in the quadrature. A higher degree generally results in more accurate integration.

6. Calculate the nodes (x) and weights (w) for Gauss-Legendre quadrature:

   - `x_gauss, w = np.polynomial.legendre.leggauss(degree)` calculates the nodes (`x_gauss`) and weights (`w`) for the Gauss-Legendre quadrature of the specified degree. These nodes and weights are used for the numerical integration.

7. Define the function to be integrated (`cos(x)`):

- `def function_to_integrate(x):` defines a Python function that represents the function to be integrated, which is `cos(x)`.

8. Calculate the integral using Gauss-Legendre quadrature:

- The code calculates the integral of the `cos(x)` function over the interval [-1, 1] using Gauss-Legendre quadrature with the following line: `integral = sum(w * function_to_integrate(x_gauss))`. It computes the weighted sum of function values at the Gauss-Legendre nodes (`x_gauss`) to approximate the integral.

9. Print the result:

- `print(f"Approximate integral of cos(x) over [-1, 1]: {integral}")` prints the approximate value of the integral of `cos(x)` over the interval [-1, 1] obtained using Gauss-Legendre quadrature.

10. Plot the first 5 Legendre polynomials:

- A `for` loop iterates through the first 5 Legendre polynomials. For each polynomial, it evaluates the Legendre polynomial function and adds it to the existing plot.

11. Set the plot title, labels, and legend:

- `plt.title('Legendre Polynomials vs. cos(x)')` sets the title for the plot.

- `plt.xlabel('x')` sets the x-axis label.

- `plt.legend()` adds a legend to the plot, distinguishing the `cos(x)` function from the Legendre polynomials.

12. Display the grid and show the plot:

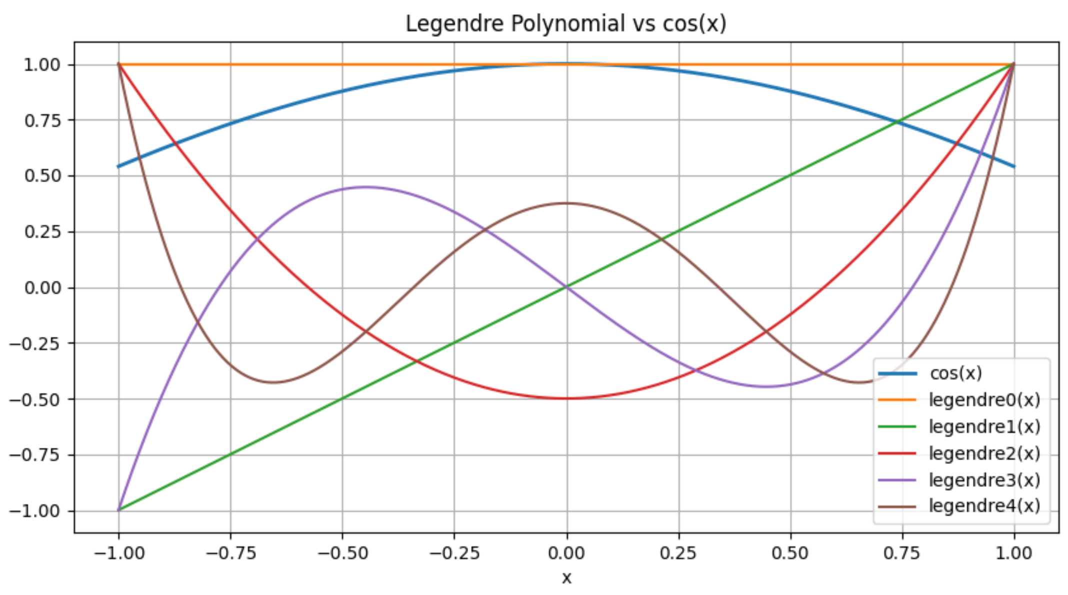
- `plt.grid(True)` displays a grid on the plot.

- `plt.show()` displays the plot on your screen.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import legendre

#define the intervel for which you want to plot the legendre polynomial

x_min = -1.0
x_max = 1.0

#create an array of x-value with in interval

x = np.linspace(x_min,x_max,1000)

#plot the cos(x)
plt.figure(figsize=(10,5))
plt.plot(x,np.cos(x),label="cos(x)",linewidth=2)

#choose the degree of gauss-legendre quadrature
degree = 4

#calculate the node(x) and weights(w)using numpy for the intervel [-1.1]
x_guass,w = np.polynomial.legendre.leggauss(degree)

#define the function to be integrate(cos(x))
def function_to_integrate(x):
    return np.cos(x)

#calculate the integral using guass legendre quadrature

integral = sum(w*function_to_integrate(x_guass))

print(f"Approximate integral of cos(x) over [-1,1]:{integral}")

#plot the legendre polynomials
num_polynomial = 5

for i in range(num_polynomial):
    legendre_poly = legendre(i)
    plt.plot(x,legendre_poly(x),label=f"legendre{i}(x)")

plt.title("Legendre Polynomial vs cos(x)")
plt.xlabel("x")
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Project_06

Aim: Computation of Discrete Fourier Transform (DFT) using complex numbers.

Algorithm:

1. Define a function to compute the DFT of an input signal x. Accept the input signal x.

2. Calculate N, the length of x. Create an array X of complex numbers of length N for storing DFT values.

3. DFT Calculation: For each frequency component k (0 to N-1), calculate its DFT value and store it in X. The DFT value for each k is calculated by summing up contributions from all elements in x.

4. Return the array X containing DFT coefficients. • Define a sample input signal x.

5. Call the functon with the input signal to calculate the DFT and store it input signal x.

6.Print the DFT result, i.e., the array X.

Code:

```
import numpy as np

def compute_dft(x):
    N = len(x)
    X = np.zeros(N, dtype = complex)
    for k in range(N):
        X[k] = sum(x[n]*np.exp(-2j * np.pi *k * n /N)
                   for n in range(N))
        return X
x = np.array([11, 2, 3, 4]) #Replace this with your input signal
X = compute_dft(x)
print("DFT Result:", X)
```

Output:

```
DFT Result: [20.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

Project_07

Aim: Fast Fourier Transform of given function in tabulated or mathematical form e.g. function exp (-x^2).

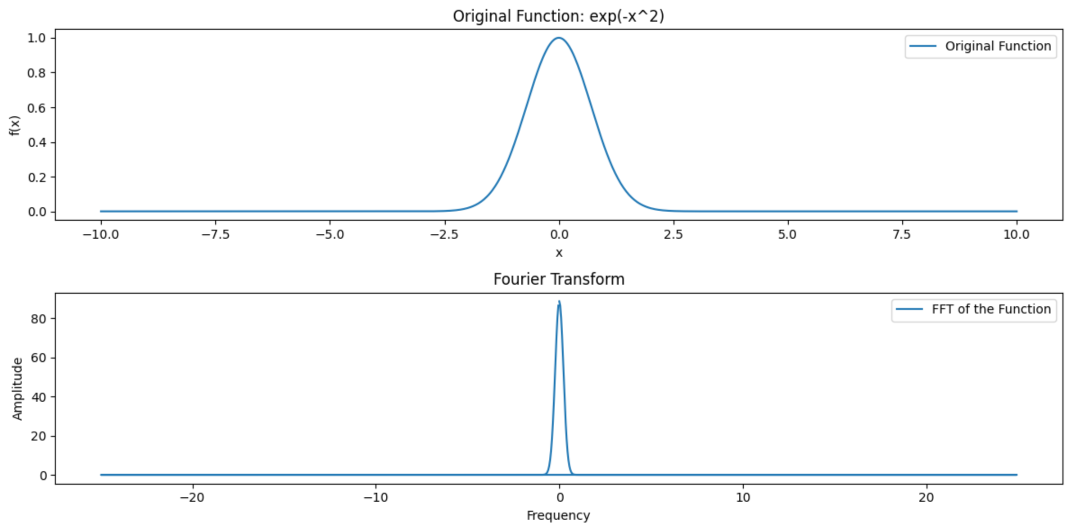
Algorithm:

1. Import numpy and matplotlib.pyplot.
2. Define the original function which return the value of function.
3. Set Parameters and Tabulate Data: Generate num_points (e.g., 1000) equally spaced x_values in the range from - 10 to 10.
4. Calculate y_values by applying original_function to each x value.
5. Compute the Fast Fourier Transform (FFT) of y_values using np.fft.fft, storing the result in fft_result.
6. Calculate the corresponding frequency values using np.fft.fftfreq and store them in freq.
7. Plot the original function and the Fourier transform.

Code:

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

#defin the function e^(-x^2).
def original_fun(x):
    return np.exp(-x**2)

#Number of points for the tabulated function.
num_points = 1000
x_values = np.linspace(-10, 10, num_points)
y_values = original_fun(x_values)

#perform the fast fourier transform(FFT).
fft_result = np.fft.fft(y_values)
freq = np.fft.fftfreq(num_points, x_values[1] - x_values[0])

#Plot the original function.
plt.figure(figsize=(12,6))
plt.subplot(2,1,1)
plt.plot(x_values,y_values,label="Original Function")
plt.title("Original Function: exp(-x^2)")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()

#Plot the fourier Transform
plt.subplot(2,1,2)
plt.plot(freq, np.abs(fft_result),label="FFT of the Function")
plt.title("Fourier Transform")
plt.xlabel("Frequency")
plt.ylabel("Amplitude")
plt.legend()
plt.tight_layout()
plt.show()
```

Output:



Project_08

Aim: Representation of a function as a linear combination of Legendre polynomial.

Algorithm:
Import Libraries: Import the necessary libraries. You'll need numpy for numerical computations and scipy.special for Legendre polynomials.

Define the Function: Define the function you want to represent. For this example, let's say we have a function f(x).

Define the Domain: Define the domain over which you want to approximate the function. For Legendre polynomials, it's typically [-1, 1].

Compute Legendre Coefficients: For each Legendre polynomial up to the chosen degree, compute the integral of their product with the function f(x) over the domain.

Code:

```
import numpy as np
from scipy.special import legendre

def legendre_poly(n, x):
    if n == 0:
        return np.ones_like(x)
    elif n == 1:
        return x
    else:
        Pnm2 = np.ones_like(x)
        Pnm1 = x
        for k in range(2, n + 1):
            Pn = ((2 * k - 1) * x *Pnm1 - (k - 1) * Pnm2) / k
            Pnm2, Pnm1 = Pnm1, Pn
        return Pn

def gauss_legendre_quadrature(f, a, b, degree):
    x, w = np.polynomial.legendre.leggauss(degree)

    x_scaled = 0.5 * (b - a) * 0.5 * (b + a)
    w_scaled = 0.5 * (b - a) * w

    integral = np.sum(w_scaled * f(x_scaled))
    return integral
```

```python
def function_to_represent(x):
    return np.sin(x)

a = -1
b = 1
degree = 5

coefficients = [gauss_legendre_quadrature(lambda x: function_to_represent(x)
* legendre_poly(n, x), a, b, degree)
                for n in range(degree + 1)]

def integrate_with_legendre_poly(x):
    result = 0
    for n, coefficient in enumerate(coefficients):
        legendre_poly1 = legendre_poly(n, x)
        result += coefficient * legendre_poly1
    return result

result = gauss_legendre_quadrature(integrate_with_legendre_poly, a, b,
degree)

print(f"Approximate integral of sin(x) using Legendre Polynomials:", result)
```

Output:

```
Approximate integral of sin (x) using Legendre polynomials :
2.7755575615628914e-17
```

ThankYou