PROJECT3(MATH214)

NAME : MUHAMMED SAID MODALI

NO : 210102002071

Everything done in this code is original, except for what is indicated with references.

```
import numpy as np                          #for linespace
from matplotlib import pyplot as plt    #for plot, xlabel, y label, grid, show
import math
```

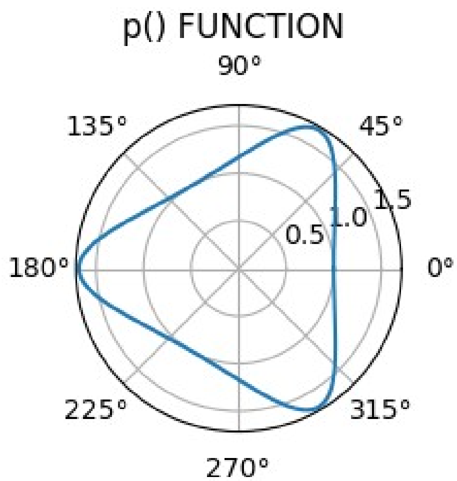
Section a-)

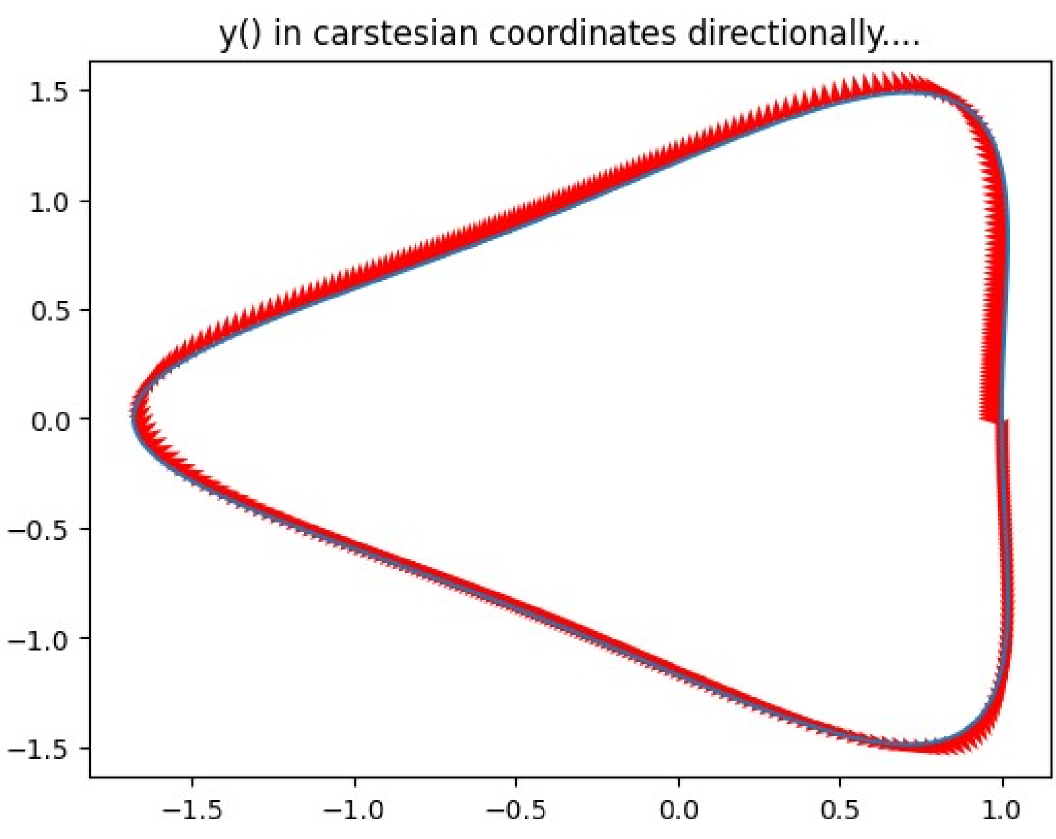
In section A, the necessary functions are provided, and it is requested that these functions be plotted and displayed in both polar and Cartesian coordinates in a computer environment. Additionally, in the Cartesian representation, the inclusion of orthogonal vectors is also requested. Taking all these into consideration, the detailed explanation is written below.

```
#for polar graph....

def p(x):   # x is radian. (p functions in the report)
    y1 = abs(math.cos((3 * x) / 4))**8
    y2 = abs(math.sin((3 * x) / 4))**8
    rval = (y1 + y2)**(-(1/4))
    return rval


x_values =  np.linspace(0, 2 * np.pi, 360) # it is x values betweeen 0 and 2pi.

# for y values
y_values = []


for i in range(0, len(x_values)): #adding y values according to p function...
    y_values.append(p(x_values[i]))

plt.subplot(2, 1, 1, projection='polar')
```

```
plt.plot(x_values, y_values, label= 'p()')
plt.title('p() FUNCTION ')
plt.show()
#*********************************************************************
***************
#for cartesian graph....

def v_vector(x): # t_vector produces vertical vectors
    def dx_dx(x):
        y = -(3/4) * ((np.abs(np.cos(3*x/4)))**7) *
np.sign(np.cos(3*x/4))
        return y
    def dy_dx(x):
        y = (3/4) * ((np.abs(np.sin(3*x/4)))**7) *
np.sign(np.sin(3*x/4))
        return y
    dx_dx1 = dx_dx(x)
    dy_dx1 = dy_dx(x)
    l = np.sqrt(dx_dx1**2 + dy_dx1**2)
    ax = dx_dx1 / l
    ay = dy_dx1 / l
    return ax, ay


def calculate_coordinates(x_values, y_values): # the function in the
report
    x_values_c = y_values * np.cos(x_values)
    y_values_c = y_values * np.sin(x_values)
    return x_values_c, y_values_c


t_vectors = []

for i in range(0,len(x_values)): # adding vertical vectors..
    t_vectors.append(v_vector(x_values[i]))

t_vector_x = []
t_vector_y = []

for i in range(0, len(t_vectors)):
    t_vector_x.append(t_vectors[i][0])
    t_vector_y.append(t_vectors[i][1])

plt.subplot(1, 1, 1) #visualizing for the graph
x_values_c, y_values_c = calculate_coordinates(x_values, y_values)
plt.plot(x_values_c, y_values_c, label='y()')

for i in range(len(x_values_c)):
    plt.quiver(x_values_c[i], y_values_c[i], t_vector_x[i],
t_vector_y[i], color='red', scale=40)
```

```
plt.title('y() in carstesian coordinates directionally....')
plt.show()
```





Section b-)

In section B, the task is to find the derivatives dx/dp, dx/dx and the function L(x) using the provided functions p(x) , x, y. Additionally, plotting is required. To accomplish this, Forward Difference, Backward Difference, Three-Point Endpoint, and Three-Point Midpoint formulas are separately employed. The completed version of these requirements is presented below, with explanations articulated using comment lines.ions are expressed with comment lines.

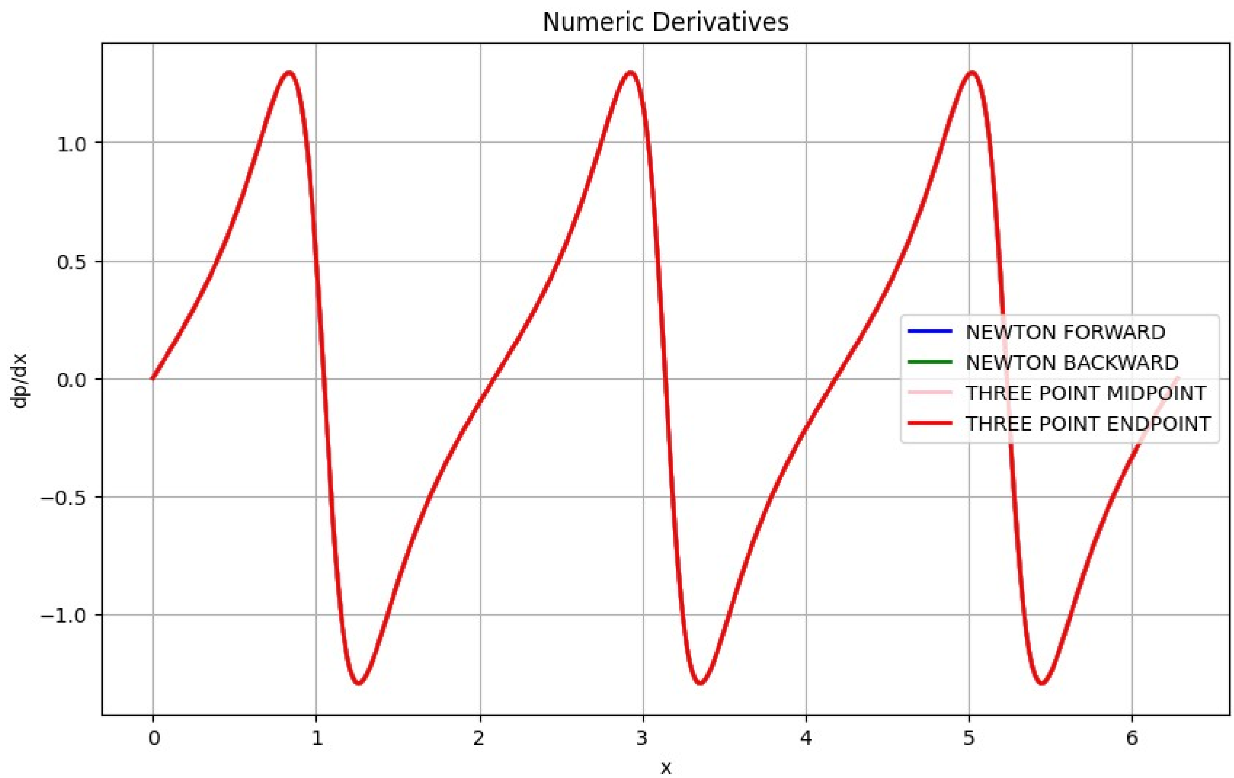
```
def newton_forward(f, x, h): #newton forward formula
    der_val = (f(x + h) - f(x)) / h
    return der_val

def newton_backward(f, x, h): #newton backward formula
    der_val = (f(x) - f(x - h)) / h
    return der_val

def three_point_midpoint(f, x, h): #three point midpoint formula
    der_val = (f(x + h) - f(x - h)) / (2 * h)
    return der_val

def three_point_endpoint(f, x, h): #three point endpoint formula
    der_val = (-3 * f(x) + 4 * f(x + h) - f(x + 2 * h)) / (2 * h)
    return der_val

h= 0.0000000001  #smal h value (to approximate)

# fiind_numeric_and_show function finds df/dx according to newton
forward, backward... and function which is send, after that shows the
graph .
#if true ==> return that y_values, if not only shows the graph.The
reason for using x is for seciton d
def find_numeric_and_show(function, y_label, label = False):

    df_dx_values1 = []
    df_dx_values2 = []
    df_dx_values3 = []
    df_dx_values4 = []
    for i in range(0, len(x_values)): #applying methods according to
the sent function...
        df_dx_values1.append(newton_forward(function, x_values[i], h))
        df_dx_values2.append(newton_backward(function, x_values[i],
h))
        df_dx_values3.append(three_point_midpoint(function,
x_values[i], h))
        df_dx_values4.append(three_point_endpoint(function,
x_values[i], h))

    if (label):
        return df_dx_values3
```

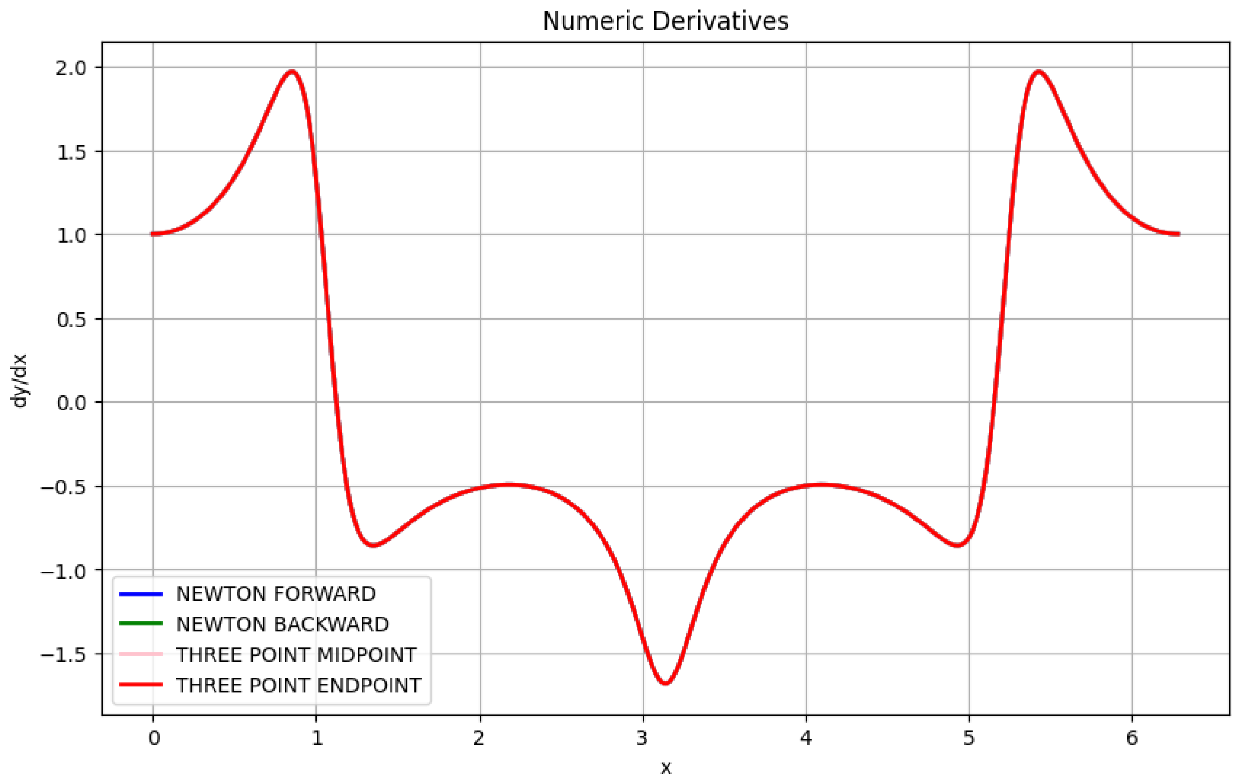
```
    #create the graph
    plt.figure(figsize=(10, 6))

    plt.plot(x_values, df_dx_values1, color="blue", linewidth=2,
label="NEWTON FORWARD" )
    plt.plot(x_values, df_dx_values2, color="green", linewidth=2,
label="NEWTON BACKWARD" )
    plt.plot(x_values, df_dx_values3, color="pink", linewidth=2,
label="THREE POINT MIDPOINT" )
    plt.plot(x_values, df_dx_values4, color="red", linewidth=2,
label="THREE POINT ENDPOINT" )

    plt.title('Numeric Derivatives')
    plt.xlabel('x')
    plt.ylabel(y_label)
    plt.legend()
    plt.grid(True)
    plt.show()
```

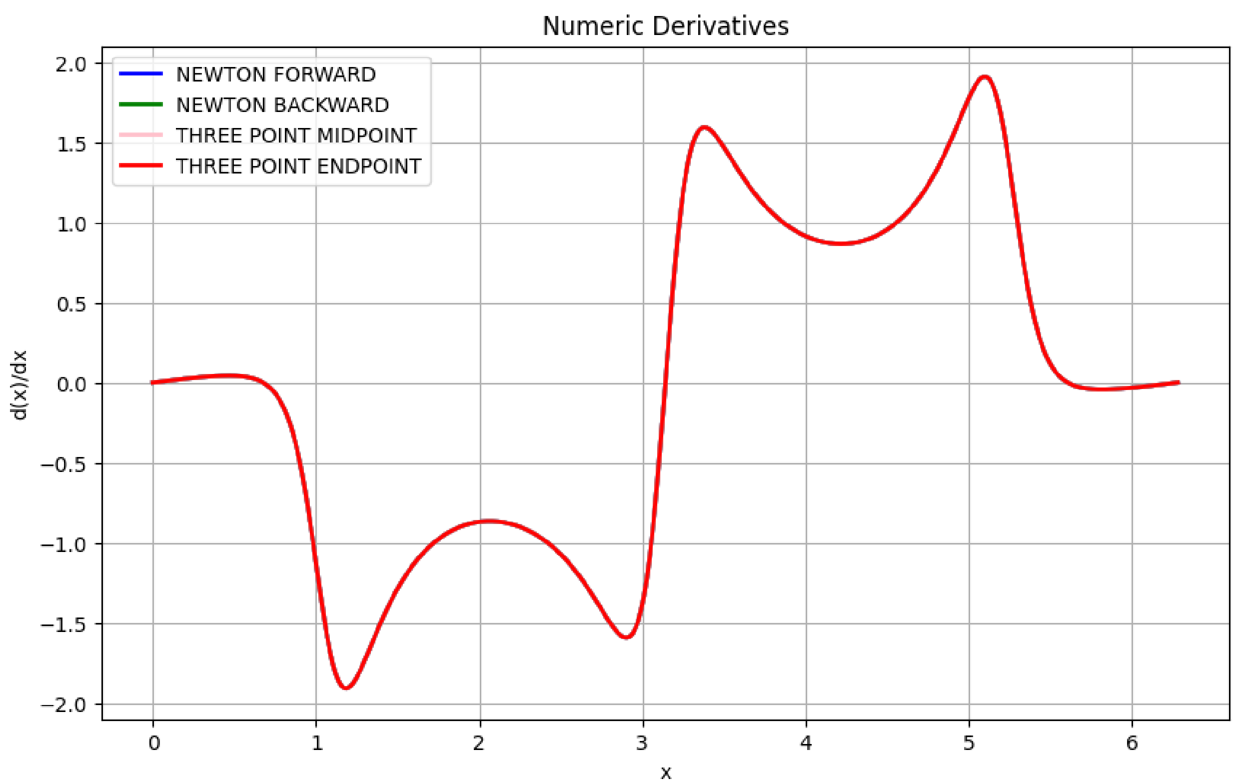
The functions dp/dx, dy/dx, and L(x) obtained using the forward difference, backward difference, three-point endpoint, three-point midpoint, and three-point central difference formulas are overlaid in the graph below.

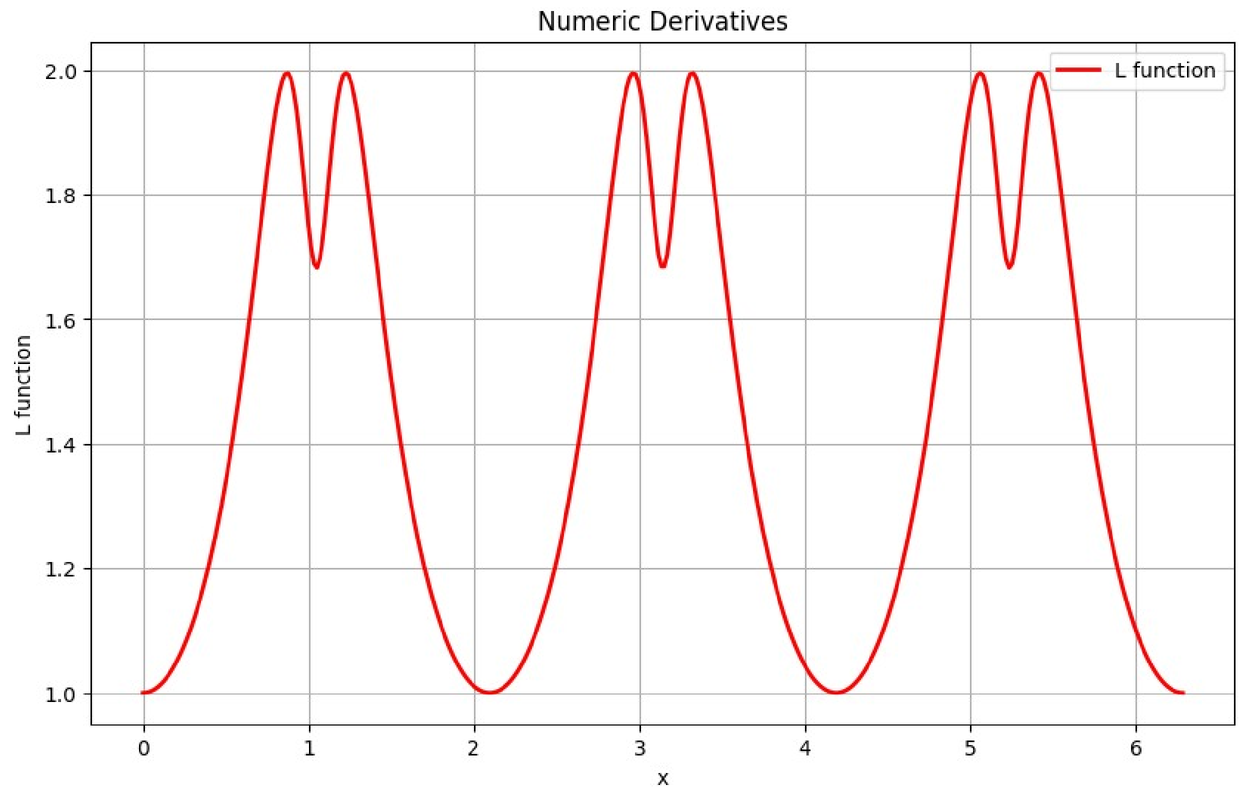
```
#creating desired functions ...
#for dp/dx:
find_numeric_and_show(p, "dp/dx")
dp_dx_list = find_numeric_and_show(p, "empty", True) #for using d
section...
#for dy/dx:
def y_function(x):
    y = p(x) * np.sin(x)
    return y
find_numeric_and_show(y_function, "dy/dx")
dy_dx_list = find_numeric_and_show(y_function, "empty", True) #for
using (d) section...
#for d(x)/dx
def x_function(x):
    x = p(x) * np.cos(x)
    return x
find_numeric_and_show(x_function, "d(x)/dx")
dx_dx_list = find_numeric_and_show(x_function, "empty", True)
#for L(x):
dx_x = find_numeric_and_show(x_function, "empty", True)
dy_dx = find_numeric_and_show(y_function, "empty", True)
sum_list = []
for i in range(0, len(x_values)):
    sum_list.append(np.sqrt(dx_x[i]**2 + dy_dx[i]**2))

#show l(x) function.
```

```
plt.figure(figsize=(10, 6))
plt.plot(x_values, sum_list, color="red", linewidth=2, label="L function" )
plt.title('Numeric Derivatives')
plt.xlabel('x')
plt.ylabel("L function")
plt.legend()
plt.grid(True)
plt.show()
#*****************************************************************************************************
```




Numeric Derivatives
dy/dx
x
NEWTON FORWARD
NEWTON BACKWARD
THREE POINT MIDPOINT
THREE POINT ENDPOINT


Numeric Derivatives
d(x)/dx
x
NEWTON FORWARD
NEWTON BACKWARD
THREE POINT MIDPOINT
THREE POINT ENDPOINT



As seen, the functions obtained with the four methods overlap, indicating that there is not much difference between the four methods.

Section c-)

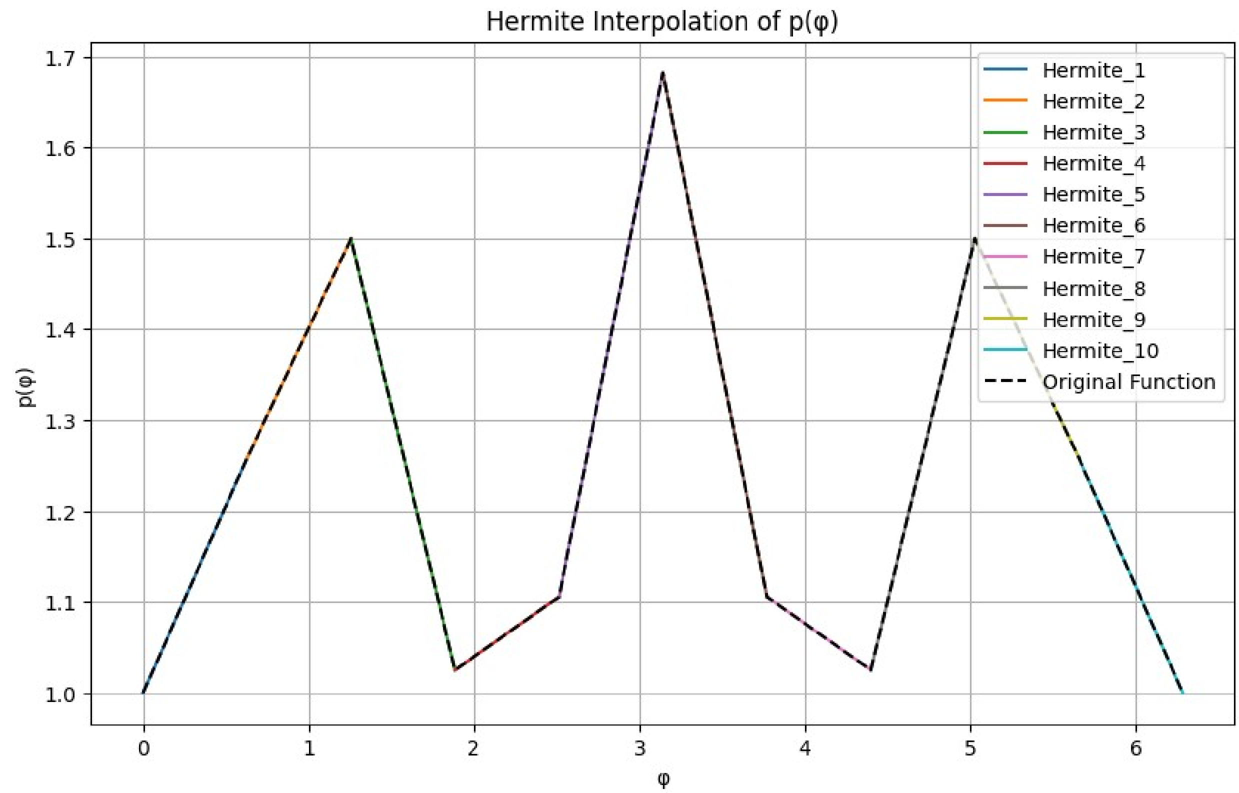
In Section C, it is mentioned that the Hermite function is found, but only the y values of Hermite have been determined. What is requested, however, is the p(x) function. To solve this problem, while finding the coefficients of Hermite (piecewise), they have also been printed. Later, with the help of AI (ChatGPT), these printed piecewise functions were converted into an array and written in function form. So, now we have the piecewise Hermite(x) function that approximates the value of p(x).

(Note: The reason for using ChatGPT is that the functions are too long to be written by an individual.)

```
try_list = []

#Calculating 3rd-degree Hermite interpolation
def hermite_interpolation(x_start, x_end, y_start, y_end, slope_start, 
slope_end, x_value):
    t = (x_value - x_start) / (x_end - x_start)
    h00 = (1 + 2 * t) * (1 - t)**2
    h10 = t * (1 - t)**2
    h01 = t**2 * (3 - 2 * t)
```

```
    h11 = t**2 * (t - 1)

    if y_start not in try_list: #for observing section d , to collect 
form of formula
        try_list.append(y_start)
        print("x = {}, {} *(1 + 2 * t) * (1 - t)**2, {}* t * (1 - 
t)**2, {}* t**2 * (3 - 2 * t), {}* t**2 * (t - 1)".format(x,y_start,
(x_end - x_start)* slope_start,y_end,(x_end - x_start) * slope_end   )
)
    return h00 * y_start + h10 * (x_end - x_start) * slope_start + h01 
* y_end + h11 * (x_end - x_start) * slope_end


N = 10 # subintervals N
x_division = np.linspace(0, 2 * np.pi, N + 1)

plt.figure(figsize=(10, 6))

# Calculating the required values for Hermite interpolation while 
iterating over subintervals....
for i in range(N):
    x0, x1 = x_division[i], x_division[i + 1]
    p0, p1 = p(x0), p(x1)
    dp0, dp1 = (p(x1) - p(x0)) / (x1 - x0), (p(x1) - p(x0)) / (x1 - 
x0)

    x_vals = np.linspace(x0, x1, 100)
    # Calculating Hermite interpolation at each point...
    p_vals = []
    for x in x_vals:
        p_vals.append(hermite_interpolation(x0, x1, p0, p1, dp0, dp1, 
x))

    # shows each  Hermite interpolation....
    plt.plot(x_vals, p_vals, label='Hermite_{}'.format(i + 1))

real_values = []
for x in x_division:
    real_values.append(p(x))

plt.plot(x_division, real_values, label='Original Function', 
linestyle='--', color='black')

# for graph
plt.title('Hermite Interpolation of p(φ)')
plt.xlabel('φ')
plt.ylabel('p(φ)')
plt.legend()
plt.grid(True)
plt.show()
```

```
x = 0.0, 1.0 *(1 + 2 * t) * (1 - t)**2, 0.2581896742413201* t * (1 -
t)**2, 1.25818967424132* t**2 * (3 - 2 * t), 0.2581896742413201* t**2
* (t - 1)
x = 0.6283185307179586, 1.25818967424132 *(1 + 2 * t) * (1 - t)**2,
0.24137856876817596* t * (1 - t)**2, 1.499568243009496* t**2 * (3 - 2
* t), 0.24137856876817596* t**2 * (t - 1)
x = 1.2566370614359172, 1.499568243009496 *(1 + 2 * t) * (1 - t)**2, -
0.4744827135571837* t * (1 - t)**2, 1.0250855294523125* t**2 * (3 - 2
* t), -0.4744827135571837* t**2 * (t - 1)
x = 1.8849555921538759, 1.0250855294523125 *(1 + 2 * t) * (1 - t)**2,
0.08045294732286301* t * (1 - t)**2, 1.1055384767751755* t**2 * (3 - 2
* t), 0.08045294732286301* t**2 * (t - 1)
x = 2.5132741228718345, 1.1055384767751755 *(1 + 2 * t) * (1 - t)**2,
0.5762543537322538* t * (1 - t)**2, 1.6817928305074292* t**2 * (3 - 2
* t), 0.5762543537322538* t**2 * (t - 1)
x = 3.141592653589793, 1.6817928305074292 *(1 + 2 * t) * (1 - t)**2, -
0.5762543537322535* t * (1 - t)**2, 1.1055384767751757* t**2 * (3 - 2
* t), -0.5762543537322535* t**2 * (t - 1)
x = 3.7699111843077517, 1.1055384767751757 *(1 + 2 * t) * (1 - t)**2,
-0.08045294732286323* t * (1 - t)**2, 1.0250855294523125* t**2 * (3 -
2 * t), -0.08045294732286323* t**2 * (t - 1)
x = 5.026548245743669, 1.4995682430094963 *(1 + 2 * t) * (1 - t)**2, -
0.24137856876817576* t * (1 - t)**2, 1.2581896742413206* t**2 * (3 - 2
* t), -0.24137856876817576* t**2 * (t - 1)
x = 5.654866776461628, 1.2581896742413206 *(1 + 2 * t) * (1 - t)**2, -
0.25818967424132055* t * (1 - t)**2, 1.0* t**2 * (3 - 2 * t), -
0.25818967424132055* t**2 * (t - 1)
```



As seen, the piecewise Hermite function and the actual p(x) values overlap, indicating that there is no error. When testing different values for n, it has been concluded that as n decreases, the function truly diverges to that extent.

Section d-)

In Section D, the task is to rediscover dp/dx, dy/dx, dx/dx, and L(x) functions using the obtained Hermite function. These functions, along with the actual p(x) functions, are overlaid, and the relative error is determined

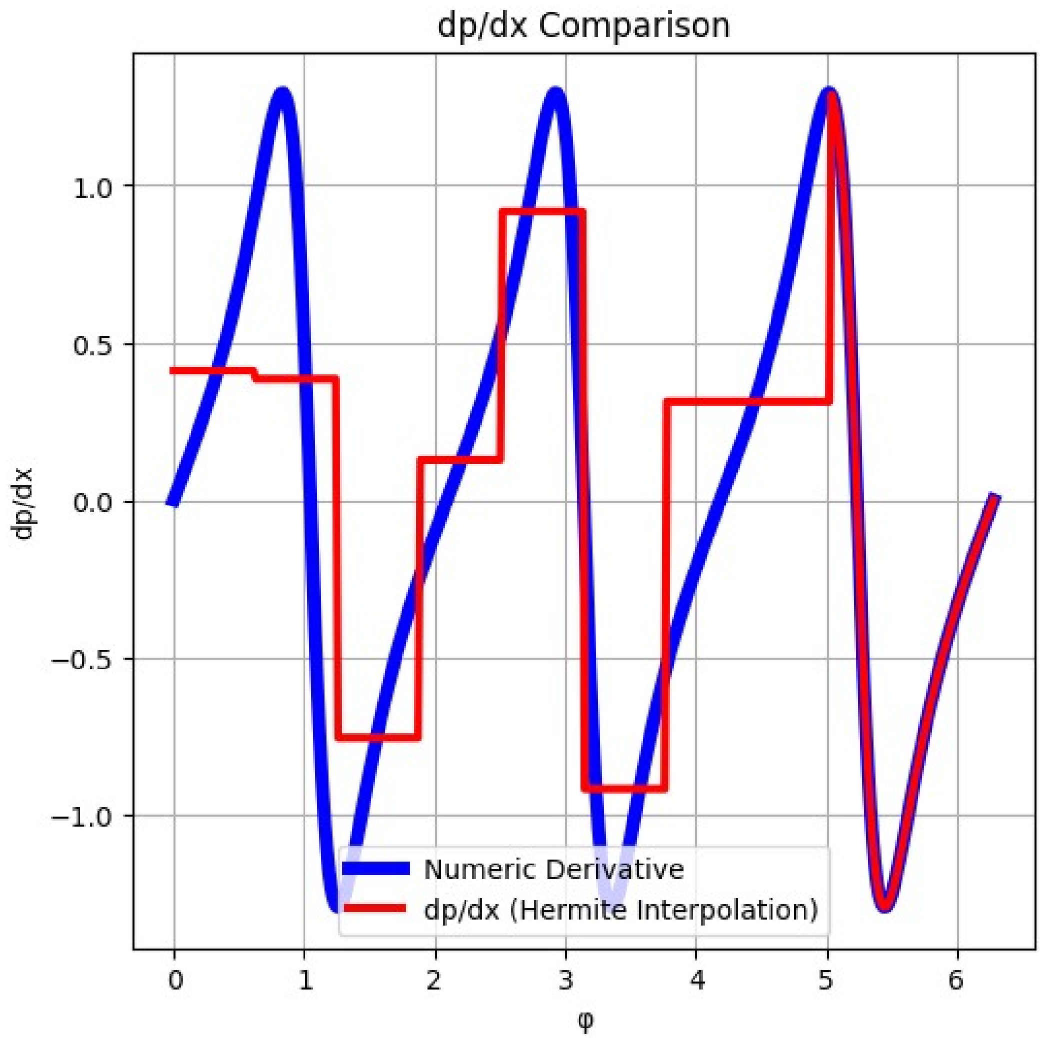
```
# the printed outputs were copied and transformed into an array in AI.
Later, the piecewise_hermit_f function was obtained below.

def piecewise_hermit_f(x):
    x_division = [0, 0.6283185307179586, 1.2566370614359172,
1.8849555921538759, 2.5132741228718345, 3.141592653589793,
3.7699111843077517, 5.026548245743669]

    for i in range(len(x_division) - 1):
        x0, x1 = x_division[i], x_division[i + 1]
        if x0 <= x <= x1:
            p0, p1 = p(x0), p(x1)
            dp0, dp1 = (p(x1) - p(x0)) / (x1 - x0), (p(x1) - p(x0)) /
(x1 - x0)
```

```
            t = (x - x0) / (x1 - x0)
            h00 = (1 + 2 * t) * (1 - t)**2
            h10 = t * (1 - t)**2
            h01 = t**2 * (3 - 2 * t)
            h11 = t**2 * (t - 1)
            return h00 * p0 + h10 * (x1 - x0) * dp0 + h01 * p1 + h11 *
(x1 - x0) * dp1

    return p(x)  # Default: return p(x) if x is outside the specified
range

def x_function_H(x): #it is created to obtain x function of hermit
    return piecewise_hermit_f(x) * np.cos(x)
def y_function_H(x): #it is created to obtain y function of hermit
    return piecewise_hermit_f(x) * np.sin(x)
def calculate_derivative(f, x): # calculating a derivative of any
function with name of the function....
    h = 1e-10
    return (f(x + h) - f(x)) / h
def L_function_H(x): #it is created to obtain l(x) function....
    return (calculate_derivative(x_function_H, (x))**2 +
calculate_derivative(y_function_H, (x))**2)**0.5

#obtaining the functions by using hermit formula
#for dp/dx:
y_val_dp_dx = []
for i in range(0, len(x_values)):
    y_val_dp_dx.append(calculate_derivative(piecewise_hermit_f,
x_values[i]))
#for dx_dx:
y_val_dx_dx = []
for i in range(0, len(x_values)):
    y_val_dx_dx.append(calculate_derivative(x_function_H,
x_values[i]))
#for dy_dx:
y_val_dy_dx = []
for i in range(0, len(x_values)):
    y_val_dy_dx.append(calculate_derivative(y_function_H,
x_values[i]))
#for L function :
y_val_L =[]
for i in range(0, len(x_values)):
    y_val_L.append(L_function_H(x_values[i]))

#shows graphs :

plt.figure(figsize=(6, 6))
plt.plot(x_values, dp_dx_list, label='Numeric Derivative',
color="blue", linewidth=5)
plt.title('Numeric Derivative of p(φ)')
```

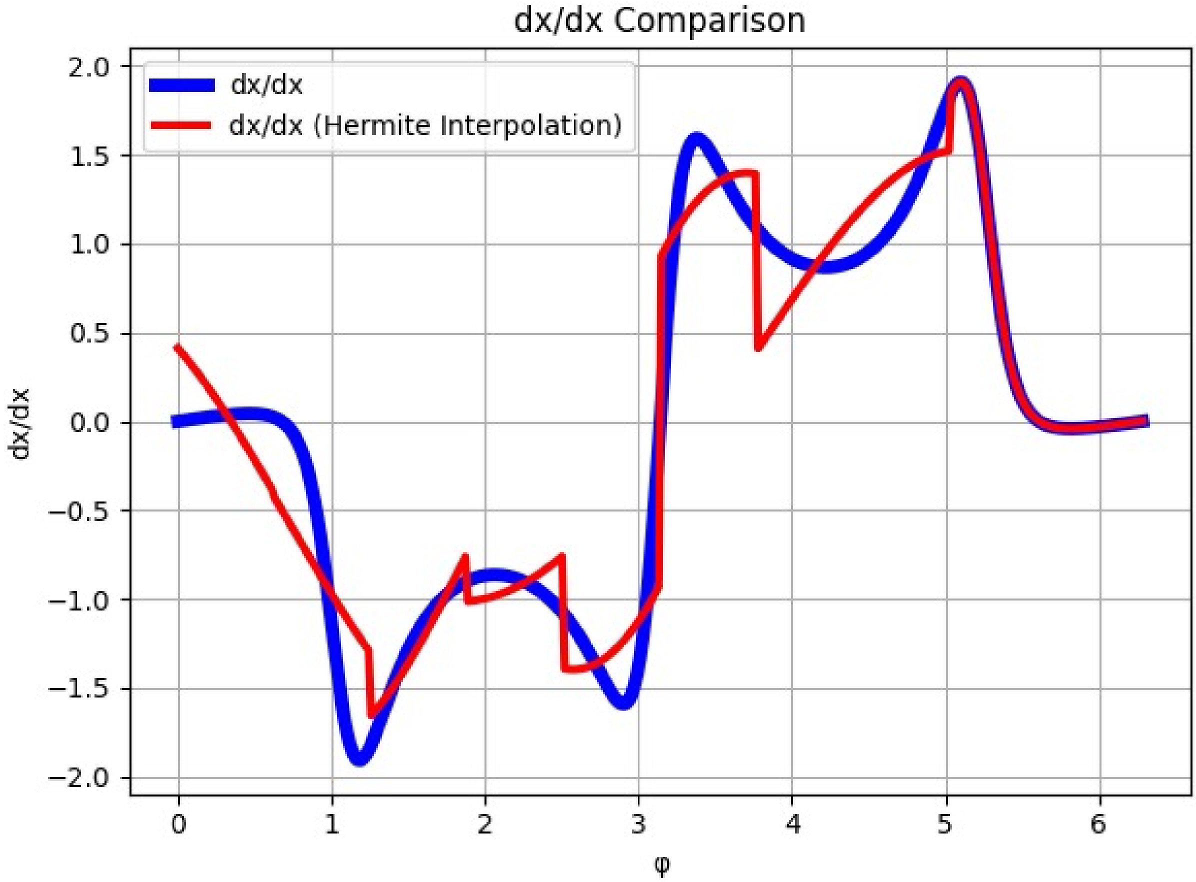
```
plt.xlabel('φ')
plt.ylabel('dp/dx')
plt.legend()
plt.grid(True)
plt.plot(x_values, y_val_dp_dx, label='dp/dx (Hermite Interpolation)',
color="red", linewidth=3)
plt.title('dp/dx Comparison')
plt.xlabel('φ')
plt.ylabel('dp/dx')
plt.legend()
plt.grid(True)
plt.show()

#****************************************

plt.plot(x_values, dx_dx_list, label='dx/dx',  color="blue",
linewidth=5)
plt.title('dx/dx')
plt.xlabel('φ')
plt.ylabel('dx/dx')
plt.legend()
plt.grid(True)
plt.plot(x_values, y_val_dx_dx, label='dx/dx (Hermite Interpolation)',
color="red", linewidth=3)
plt.title('dx/dx Comparison')
plt.xlabel('φ')
plt.ylabel('dx/dx')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
#****************************************

plt.plot(x_values, dy_dx_list, label='dy/dx',  color="blue",
linewidth=5)
plt.title('dy/dx')
plt.xlabel('φ')
plt.ylabel('dy/dx')
plt.legend()
plt.grid(True)
plt.plot(x_values, y_val_dy_dx, label='dy/dx (Hermite Interpolation)',
color="red", linewidth=3)
plt.title('dy/dx Comparison')
plt.xlabel('φ')
plt.ylabel('dy/dx')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```python
#*****************************************

plt.plot(x_values, sum_list, label='L Function',  color="blue",
linewidth=5)
plt.title('L(x)')
plt.xlabel('φ')
plt.ylabel('L(x)')
plt.legend()
plt.grid(True)
plt.plot(x_values, y_val_L, label='L(x) (Hermite Interpolation)',
color="red", linewidth=3)
plt.title('L(x) Comparison')
plt.xlabel('φ')
plt.ylabel('L(x)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

#*****************************************

# calculating relative errros....

r_e_dp_dx = np.abs(np.array(dp_dx_list) - np.array(y_val_dp_dx))
r_e_dx_dx = np.abs(np.array(dx_dx_list) - np.array(y_val_dx_dx))
r_e_dy_dx = np.abs(np.array(dy_dx_list) - np.array(y_val_dy_dx))
r_e_L = np.abs(np.array(sum_list) - np.array(y_val_L))

# showing graphs(relative errors)...

plt.figure(figsize=(15, 10))

# for dp/dx

plt.subplot(2, 2, 1)
plt.plot(x_values, r_e_dp_dx, label='Relative Error (dp/dx)')
plt.title('Relative Error (dp/dx)')
plt.xlabel('φ')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)

# for dx/dx
plt.subplot(2, 2, 2)
plt.plot(x_values, r_e_dx_dx, label='Relative Error (dx/dx)')
plt.title('Relative Error (dx/dx)')
plt.xlabel('φ')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)
```

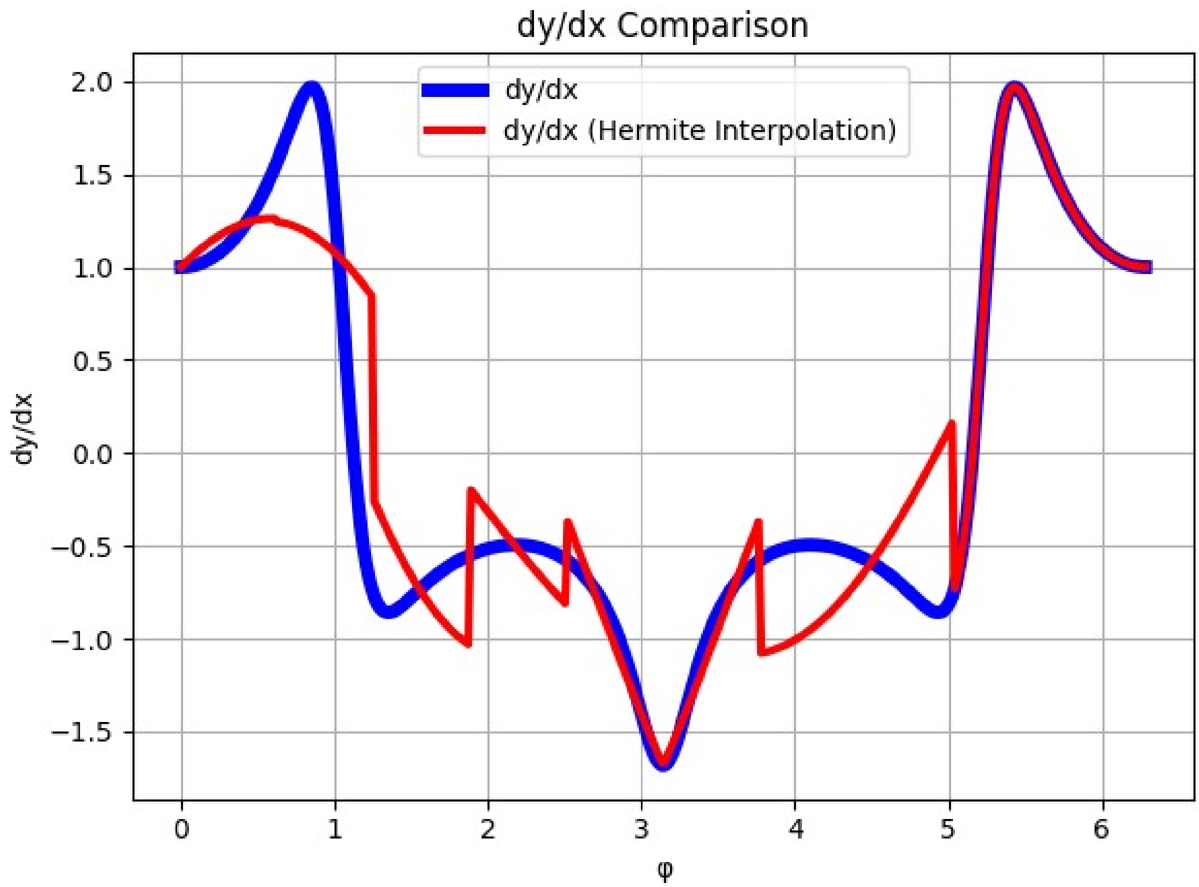
```
# for dy/dx
plt.subplot(2, 2, 3)
plt.plot(x_values, r_e_dy_dx, label='Relative Error (dy/dx)')
plt.title('Relative Error (dy/dx)')
plt.xlabel('φ')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)

# for L(x)
plt.subplot(2, 2, 4)
plt.plot(x_values, r_e_L, label='Relative Error (L(x))')
plt.title('Relative Error (L(x))')
plt.xlabel('φ')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

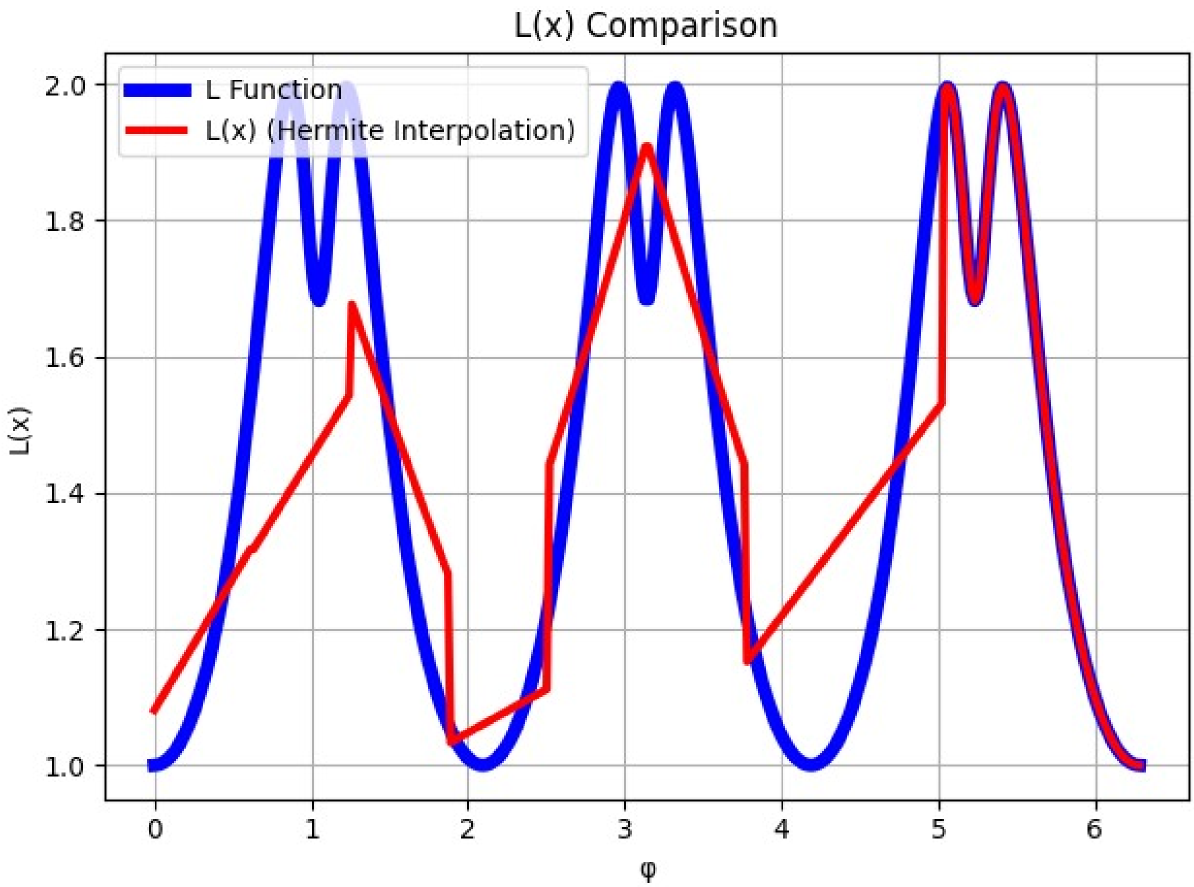


dp/dx Comparison
dp/dx
φ
Numeric Derivative
dp/dx (Hermite Interpolation)

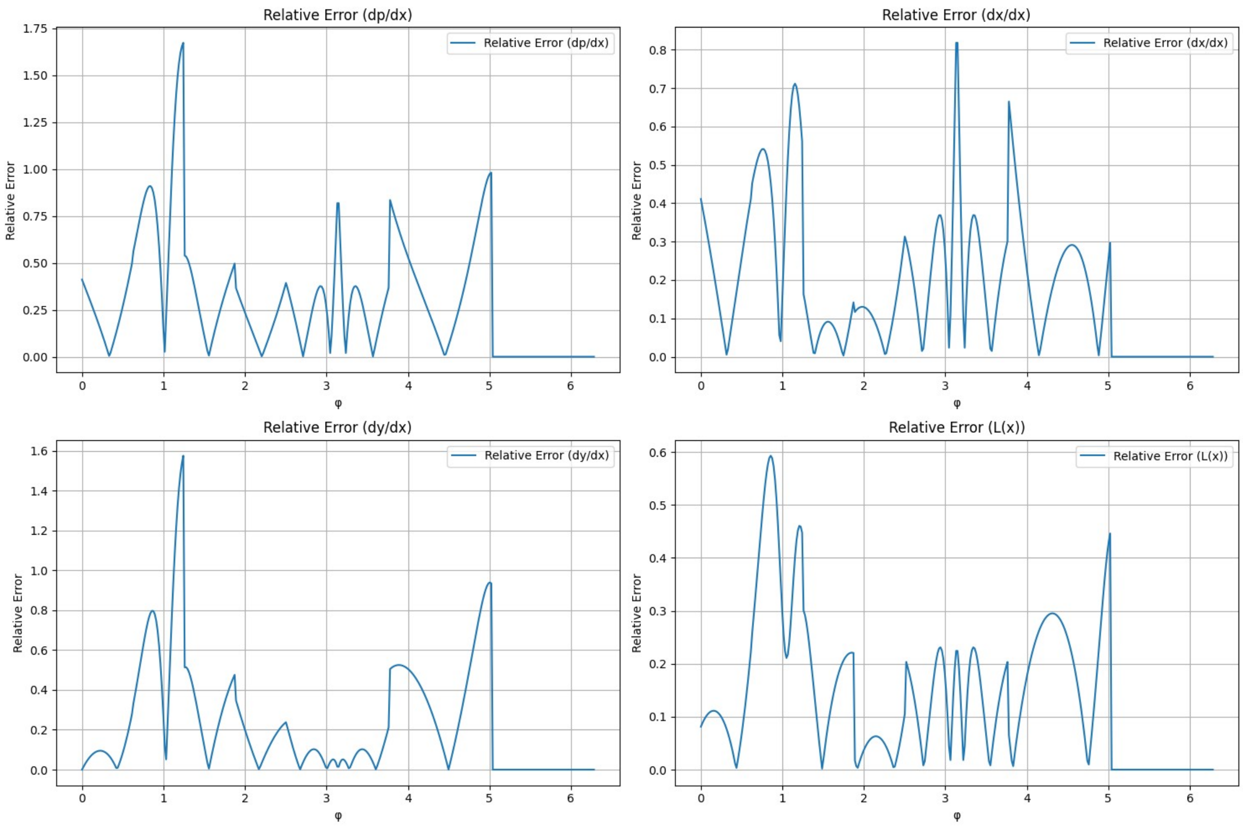


dx/dx Comparison
dx/dx
dx/dx (Hermite Interpolation)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
1
2
3
4
5
6
φ
dx/dx


dy/dx Comparison
dy/dx
dy/dx (Hermite Interpolation)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5
6
φ
dy/dx



L(x) Comparison
L Function
L(x) (Hermite Interpolation)
2.0
1.8
1.6
1.4
1.2
1.0
L(x)
0
1
2
3
4
5
6
φ



As seen, the method tested operates with an average error of approximately 0.5, demonstrating accurate performance.

Section e and f-)

In Section e and f, it is requested to find all functions, previously obtained using the Hermite method in Section D, with the cubic spline method instead. However, this is a more challenging problem than Hermite because the boundaries of piecewise functions have extremely large slope values for both the function and its derivative. It has been determined through trial and error that dividing the interval [0, 2π] into 85 segments, rather than 360, is the most optimized approach. Additionally, various online research has been conducted, and it has been identified that the tridiagonal matrix algorithm is one of the best methods for solving this problem. The necessary references have been added to the end of the report. This algorithm has been adapted and edited with ChatGPT, resulting in the creation of the derivative_cubic(x) and cubic_formula functions. Subsequently, graphs along with error rates have been presented.

```
# Section e-)
# Cubic spline interpolation function
def cubic_spline_interpolation(x, y, x_interp):
    # Get the number of data points
    n = len(x)

    # Calculate the distances h between data points
```

```
    h = np.diff(x)

    # Calculate the differences in y values between data points
    delta_y = np.diff(y)

    # Calculate the derivative alpha for each h distance
    alpha = delta_y / h

    # Allocate space for the lower triangle of the tridiagonal matrix
    l = np.zeros(n)
    mu = np.zeros(n)
    z = np.zeros(n)

    # Manually set the first element
    l[1] = 3 * (alpha[1] - alpha[0]) / (h[1] + h[0])

    # Calculate the remaining elements with a for loop
    for i in range(2, n-1):
        l[i] = 3 * (alpha[i] - alpha[i-1]) / (h[i] + h[i-1])
        mu[i] = h[i-1] / (h[i-1] + h[i] - h[i-1] * mu[i-1])
        z[i] = (l[i] - h[i-1] * z[i-1]) / (h[i-1] + h[i] - h[i-1] *
mu[i-1])

    # Calculate spline coefficients
    l_c_e = np.zeros(n)
    c_c_e = np.zeros(n)
    d_c_e = np.zeros(n)

    for j in range(n-2, 0, -1):
        c_c_e[j] = z[j] - mu[j] * c_c_e[j+1]
        l_c_e[j] = alpha[j] - (alpha[j-1] - l_c_e[j-1]) * mu[j]
        d_c_e[j] = (c_c_e[j+1] - c_c_e[j]) / (3 * h[j])

    # Calculate the b coefficients
    b_c_e = alpha - h * (c_c_e[1:] + 2 * c_c_e[:-1]) / 3

    interpolated_values = np.zeros_like(x_interp)

    # Perform interpolation for each x_interp value
    for i in range(len(x_interp)):
        # Determine the interval in which x_interp lies
        index = np.searchsorted(x, x_interp[i])
        index = np.clip(index - 1, 0, n-2).astype(int)

        # Calculate the first, second, third, and fourth-degree terms
based on the x value in the respective interval
        dx = x_interp[i] - x[index]
        interpolated_values[i] = y[index] + b_c_e[index] * dx +
c_c_e[index] * dx**2 + d_c_e[index] * dx**3
```

```
    # Print the cubic spline interpolation function
    derivative_values = np.zeros_like(x_interpolation_points)

    for i in range(len(x_interpolation_points)):
        # Determine the interval in which x_interp lies
        index = np.searchsorted(x_values, x_interpolation_points[i])
        index = np.clip(index - 1, 0, len(x_values)-2).astype(int)

        # Calculate the derivative using the coefficients of the cubic
spline in the respective interval
        dx = x_interpolation_points[i] - x_values[index]
        derivative_values[i] = b_c_e[index] + 2 * c_c_e[index] * dx +
3 * d_c_e[index] * dx**2

    # Prints the derivative values to collect values to obtain the
formula....
    for j in range(len(x_values)-1):
        print(f"S'{j}(x) = {derivative_values[j]}  for  {x_values[j]}
<= x <= {x_values[j+1]}")

    for j in range(n-1): # prints the y values to collect to obtain
the formula...
        print(f"S{j}(x) = {y[j]} + {b_c_e[j]} * (x - {x[j]}) +
{c_c_e[j]} * (x - {x[j]})^2 + {d_c_e[j]} * (x - {x[j]})^3  for  {x[j]}
<= x <= {x[j+1]}")

    return interpolated_values

# that derivative_cubic function is obtained with chatgpt to do array.
# how? => in 443.row, it is printed
def derivative_cubic(x):
    coefficients = [
        (0.0, 0.07479982508547127, 0.0421633375956607),
        (0.07479982508547127, 0.14959965017094254,
0.0421633375956607),
        (0.14959965017094254, 0.2243994752564138,
0.24941121029552277),
        (0.2243994752564138, 0.2991993003418851, 0.31110017929807104),
        (0.2991993003418851, 0.37399912542735636, 0.1738361874833575),
        (0.37399912542735636, 0.4487989505128276,
0.12391793718750627),
        (0.4487989505128276, 0.5235987755982989, 0.08869526553991368),
        (0.5235987755982989, 0.5983986006837702, 1.0597191905899273),
        (0.5983986006837702, 0.6731984257692414, 1.0646561834341783),
        (0.6731984257692414, 0.7479982508547127, 0.993584375518995),
        (0.7479982508547127, 0.822798075940184, 0.9135152731495572),
        (0.822798075940184, 0.8975979010256552, 0.884330314711378),
        (0.8975979010256552, 0.9723977261111265, 0.8994026575507549),
        (0.9723977261111265, 1.0471975511965979, 1.3872608153234127),
```

```
        (1.0471975511965979, 1.121997376282069, 0.8568832059636751),
        (1.121997376282069, 1.1967972013675403, 0.11780084609304631),
        (1.1967972013675403, 1.2715970264530116, -
0.02447394507634859),
        (1.2715970264530116, 1.3463968515384828,
0.009545701108192972),
        (1.3463968515384828, 1.421196676623954, -0.08642471655514229),
        (1.421196676623954, 1.4959965017094254, -0.86448141397149),
        (1.4959965017094254, 1.5707963267948966, -1.274717946126089),
        (1.5707963267948966, 1.645596151880368, -0.9425783571713636),
        (1.645596151880368, 1.7203959769658392, -0.91677913243866),
        (1.7203959769658392, 1.7951958020513104, -0.8260017950366566),
        (1.7951958020513104, 1.8699956271367817, -1.0887854523220406),
        (1.8699956271367817, 1.944795452222253, -1.0359617743696525),
        (1.944795452222253, 2.019595277307724, -0.9208954607541006),
        (2.019595277307724, 2.0943951023931957, 0.09017549003080116),
        (2.0943951023931957, 2.169194927478667, -0.11757119873398669),
        (2.169194927478667, 2.243994752564138, -0.31087210367024),
        (2.243994752564138, 2.3187945776496095, -0.12472060640197141),
        (2.3187945776496095, 2.3935944027350806, -
0.37697171414338226),
        (2.3935944027350806, 2.4683942278205517, -
0.39180604488009224),
        (2.4683942278205517, 2.5431940529060233, 0.07983651962219795),
        (2.5431940529060233, 2.6179938779914944, 0.41482513536209553),
        (2.6179938779914944, 2.6927937030769655, 0.37319898910001437),
        (2.6927937030769655, 2.767593528162437, 0.10323247680712755),
        (2.767593528162437, 2.842393353247908, 0.3472324247403847),
        (2.842393353247908, 2.9171931783333793, 0.09313050171393192),
        (2.9171931783333793, 2.991993003418851, -0.12084938494867242),
        (2.991993003418851, 3.066792828504322, 0.8228592868012312),
        (3.066792828504322, 3.141592653589793, 1.0527476669538136),
        (3.141592653589793, 3.2163924786752647, 1.138487003980263),
        (3.2163924786752647, 3.291192303760736, 0.7409787886247263),
        (3.291192303760736, 3.365992128846207, 0.9895544517533754),
        (3.365992128846207, 3.4407919539316785, 0.9460230191672777),
        (3.4407919539316785, 3.5155917790171496, 1.1146778562432378),
        (3.5155917790171496, 3.5903916041026207, 0.7595559160260281),
        (3.5903916041026207, 3.6651914291880923, 0.05279825421218379),
        (3.6651914291880923, 3.7399912542735634, 0.11068940570883012),
        (3.7399912542735634, 3.814791079359035, -0.1140233380377007),
        (3.814791079359035, 3.889590904444506, -0.04579100722166839),
        (3.889590904444506, 3.964390729529977, -0.7612528276902472),
        (3.964390729529977, 4.039190554615448, -1.0895435999442125),
        (4.039190554615448, 4.11399037970092, -0.9556654701539165),
        (4.11399037970092, 4.188790204786391, -0.9968012627436127),
        (4.188790204786391, 4.263590029871862, -0.7214343589850729),
        (4.263590029871862, 4.338389854957334, -1.1597483089504645),
        (4.338389854957334, 4.413189680042805, -1.0463393299631163),
```

```
        (4.413189680042805, 4.487989505128276, -0.7918557125111225),
        (4.487989505128276, 4.562789330213747, 0.16079668189208496),
        (4.562789330213747, 4.637589155299219, -0.0917130813824023),
        (4.637589155299219, 4.71238898038469, -0.3778367256971559),
        (4.71238898038469, 4.787188805470161, -0.061608231866168484),
        (4.787188805470161, 4.861988630555633, -0.4016871413927982),
        (4.861988630555633, 4.9367884556411035, -0.4296182123412435),
        (4.9367884556411035, 5.011588280726575, -
0.009017830864457732),
        (5.011588280726575, 5.086388105812047, 0.4234085291683167),
        (5.086388105812047, 5.161187930897517, 0.40270488044014097),
        (5.161187930897517, 5.235987755982989, 0.06740491948363336),
        (5.235987755982989, 5.31078758106846, 0.36802808125538705),
        (5.31078758106846, 5.385587406153931, 0.09830626042819768),
        (5.385587406153931, 5.460387231239403, -0.15252202094195313),
        (5.460387231239403, 5.535187056324874, 0.8183021779006625),
        (5.535187056324874, 5.609986881410345, 1.0418111286945562),
        (5.609986881410345, 5.684786706495816, 1.1463407032398598),
        (5.684786706495816, 5.759586531581288, 0.7443703621533055),
        (5.759586531581288, 5.834386356666759, 0.977169224018968),
        (5.834386356666759, 5.90918618175223, 0.954736230153974),
        (5.90918618175223, 5.983986006837702, 1.132716385329106),
        (5.983986006837702, 6.058785831923172, 0.7895577730785437),
        (6.058785831923172, 6.133585657008644, 0.05486215960022809),
        (6.133585657008644, 6.208385482094116, 0.08158843709487829),
        (6.208385482094116, 6.283185307179586, -0.07332805778385199),
    ]

    for start, end, value in coefficients:
        if start <= x <= end:
            return value

def cubic_formula(x):# the same logic with derivative_cubic
    coefficients = [
        [1.0, 0.0421633375956607, 0.0, 0.0],
        [1.003153810277175, -0.13890165065311255, 0.0,
47.624624345020734],
        [1.0126951848095227, -0.3764176330917197, 10.686940712306482,
-36.95200835048849],
        [1.0288681535333022, -0.05325042327184648, 2.394929428786245,
32.994628651544225],
        [1.0520931463771646, -0.21251265736174524, 9.798906784473006,
-19.202478652723546],
        [1.0829859021764197, 0.011441056621822487, 5.489880651179353,
18.685256468071852],
        [1.12237754238504, -0.06225796468446909, 9.682842397646187, -
1.3789267277433406],
        [1.171319205485879, 0.07770664407227579, 9.373411963523537,
3.4618419817427633],
```

```
        [1.2310248203015035, 0.1276003539927386, 10.150247487647237,
12.5919764873163],
        [1.3026298979784323, 0.2807897875592916, 12.975880403842114, -
23.33606162913964],
        [1.3864668722100848, 0.6904750589161444, 7.73928041971184, -
2.8480823123308436],
        [1.4802237271514045, 1.3567777574120985, 7.100172243337726, -
110.5745433351254],
        [1.5751599119723292, 2.858284478080943, -17.71269725778199, -
91.61333089438563],
        [1.6515155710097078, 4.080950850673459, -38.27068063697435, -
145.40330272602762],
        [1.6817928305074292, 4.052481236962798, -70.89910546924446,
151.2028983475595],
        [1.6515155710097078, 1.3292623961406997, -36.969254422803296,
74.21454402978603],
        [1.575159911972329, -0.530513924889316, -20.315549686125276,
139.57252144280835],
        [1.4802237271514045, -1.8653611579308293, 11.004450885855508,
-37.749087038446895],
        [1.3864668722100852, -1.7329337405967218, 2.533575563019351,
75.53242222300155],
        [1.3026298979784323, -1.987225946089488, 19.483011474706803, -
76.38752832424304],
        [1.2310248203015035, -1.375828867411834, 2.3416902026122024,
71.9329010982375],
        [1.171319205485879, -1.557839434678787, 18.483395462728176, -
85.61500759528502],
        [1.12237754238504, -0.9192163626536118, -0.728567315727684,
79.9078690980073],
        [1.0829859021764199, -1.1906765104118022, 17.202716578723376,
-90.98983815641958],
        [1.0520931463771646, -0.5668070330558219, -3.215355357243153,
88.79687940067652],
        [1.0288681535333022, -0.7548449616943804, 16.309013369128735,
-88.27683163600172],
        [1.0126951848095227, -0.4911079836069612, -2.6884550065693583,
86.2136288975572],
        [1.003153810277175, -0.6314925128961323, 16.14101364891655, -
84.55011043878449],
        [1.0, -0.41821003472906365, -2.094710056518205,
82.78222416213256]
    ]

    result = 0.0

    for i, coeff in enumerate(coefficients):
        result = coeff[0] * x**3 + coeff[1] * x**2 + coeff[2] * x +
coeff[3]
```

```
    return result


#!!! In the cubic spline method, the array size becomes excessively large for x value 360.
#!!! To eliminate this issue, the interval has been necessarily reduced to 85.

x_interpolation_points = np.linspace(0, 2 * np.pi, 85)
x_values = []
x_values = np.linspace(0, 2 * np.pi, 85)
y_values = []

for i in range(0, len(x_values)):
    y_values.append(p(x_values[i]))

# Since the intervals have been reduced, new y values for the initial p(x) function have been calculated for the new (85) x values.
#for dp/dx:
dp_dx_list_2 = []
for i in range(0, len(x_values)):
    dp_dx_list_2.append(newton_forward(p, x_values[i], h))

#for dy/dx:
def y_function(x):
    y = p(x) * np.sin(x)
    return y

dy_dx_list_2 = []
for i in range(0, len(x_values)):
    dy_dx_list_2.append(newton_forward(y_function, x_values[i], h))

#for d(x)/dx
def x_function(x):
    x = p(x) * np.cos(x)
    return x
dx_dx_list_2 = []

for i in range(0, len(x_values)):
    dx_dx_list_2.append(newton_forward(x_function, x_values[i], h))
#for L(x):

sum_list_2 = []
for i in range(0, len(x_values)):
    sum_list_2.append((dx_dx_list_2[i]**2 + dy_dx_list_2[i]**2)**0.5)


# Perform cubic spline interpolation
y_interp_cubic = cubic_spline_interpolation(x_values, y_values,
```

```
x_interpolation_points)

# Visualization between p(x) and cubic spline function and
comparing...

plt.figure(figsize=(12, 8))
plt.plot(x_values, y_interp_cubic, 'o', label='Data Points', linewidth
= 2, color = "black")
plt.plot(x_interpolation_points, y_interp_cubic, label='Cubic Spline
Interpolation',  color="blue", linewidth=5)
plt.title('Cubic Spline Interpolation')
plt.legend()
plt.plot(x_values, y_values, label = 'Real P(x) Function',
color="red", linewidth=3)
plt.title('real P(x) Funciton')
plt.legend()
plt.tight_layout()
plt.show()

#**********************************************************************
**************************
# for the other functions, calculating the functions(dp/dx, dx/dx,
dy/dx, l(x)) with cubic_formula and derivative_cubic

def x_function_C(x):
    return cubic_formula(x) * np.cos(x)
def y_function_C(x):
    return cubic_formula(x) * np.sin(x)
def derivative_x_C(x):
    return derivative_cubic(x) * np.cos(x) + cubic_formula(x) * -
np.sin(x)
def derivative_y_C(x):
    return derivative_cubic(x) * np.sin(x) + cubic_formula(x) *
np.cos(x)

def L_function_C(x):
    return ((derivative_x_C(x))**2 + (derivative_y_C(x))**2)**0.5
#for dp/dx
y_val_dp_dx_C = []
for i in range(0, len(x_values)):
    y_val_dp_dx_C.append(derivative_cubic(x_values[i]))


#for dx_dx:
y_val_dx_dx_C = []
for i in range(0, len(x_values)):
    y_val_dx_dx_C.append(derivative_x_C(x))
#for dy_dx:
y_val_dy_dx_C = []
for i in range(0, len(x_values)):
```

```
    y_val_dy_dx_C.append(derivative_y_C(x))
#for L function :
y_val_L_C =[]
for i in range(0, len(x_values)):
    y_val_L_C.append(L_function_C(x_values[i]))


# shows the graph has numeric functions and cubic spline
functions.....
plt.figure(figsize=(6, 6))
plt.plot(x_values, dp_dx_list_2, label='Numeric Derivative',
color="blue", linewidth=5)
plt.title('Numeric Derivative of p(φ)')
plt.xlabel('φ')
plt.ylabel('dp/dx')
plt.legend()
plt.grid(True)
plt.plot(x_values, y_val_dp_dx_C, label='dp/dx (Cubic Spline)',
color="red", linewidth=3)
plt.title('dp/dx Comparison')
plt.xlabel('φ')
plt.ylabel('dp/dx')
plt.legend()
plt.grid(True)
plt.show()

#*****************************************
# finding relative errors and showing graph...
relative_errors_dp_dx_2 = np.abs(np.array(dp_dx_list_2) -
np.array(y_val_dp_dx_C))
relative_errors_p_x_2 = np.abs(np.array(np.array(y_values)) -
np.array(y_interp_cubic))
plt.subplot(2, 2, 1)
plt.plot(x_values, relative_errors_dp_dx_2, label='Relative Error
(dp/dx)')
plt.title('(NUMERİC AND CUBIC SPLINE)')
plt.xlabel('φ')
plt.ylabel('Relative Error')
plt.legend()
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(x_values, relative_errors_p_x_2, label='Relative Error p(x)')
plt.title('(NUMERİC AND CUBIC SPLINE)')
plt.xlabel('φ')
plt.ylabel('Relative Error ')
plt.legend()
plt.grid(True)
plt.show()
```

```
S'0(x) = 0.0421633375956607  for  0.0 <= x <= 0.07479982508547127
S'1(x) = 0.0421633375956607  for  0.07479982508547127 <= x <=
0.14959965017094254
S'2(x) = 0.6604796453262142  for  0.14959965017094254 <= x <=
0.2243994752564138
S'3(x) = 0.6021039652528655  for  0.2243994752564138 <= x <=
0.2991993003418851
S'4(x) = 0.8588463926008312  for  0.2991993003418851 <= x <=
0.37399912542735636
S'5(x) = 0.9310859685940023  for  0.37399912542735636 <= x <=
0.4487989505128276
S'6(x) = 1.1463580867442706  for  0.4487989505128276 <= x <=
0.5235987755982989
S'7(x) = 1.3631465283152764  for  0.5235987755982989 <= x <=
0.5983986006837702
S'8(x) = 1.538072956048157  for  0.5983986006837702 <= x <=
0.6731984257692414
S'9(x) = 1.8574306751803795  for  0.6731984257692414 <= x <=
0.7479982508547127
S'10(x) = 1.830280193778536  for  0.7479982508547127 <= x <=
0.822798075940184
S'11(x) = 1.800463522476382  for  0.822798075940184 <= x <=
0.8975979010256552
S'12(x) = 0.5629627426164796  for  0.8975979010256552 <= x <=
0.9723977261111265
S'13(x) = -1.3292623961406735  for  0.9723977261111265 <= x <=
1.0471975511965979
S'14(x) = -4.084930054690027  for  1.0471975511965979 <= x <=
1.121997376282069
S'15(x) = -4.016053215219023  for  1.121997376282069 <= x <=
1.1967972013675403
S'16(x) = -2.9556309312626476  for  1.1967972013675403 <= x <=
1.2715970264530116
S'17(x) = -1.2269824865030365  for  1.2715970264530116 <= x <=
1.3463968515384828
S'18(x) = -0.8527191475525082  for  1.3463968515384828 <= x <=
1.421196676623954
S'19(x) = -0.0860968811957501  for  1.421196676623954 <= x <=
1.4959965017094254
S'20(x) = -0.3547420780834336  for  1.4959965017094254 <= x <=
1.5707963267948966
S'21(x) = 0.18188389774557812  for  1.5707963267948966 <= x <=
1.645596151880368
S'22(x) = -0.22978139486054694  for  1.645596151880368 <= x <=
1.7203959769658392
S'23(x) = 0.3130471206504122  for  1.7203959769658392 <= x <=
1.7951958020513104
S'24(x) = -0.1444243376377803  for  1.7951958020513104 <= x <=
1.8699956271367817
S'25(x) = 0.4426362359986864  for  1.8699956271367817 <= x <=
```

```
1.944795452222253
S'26(x) = -0.045416997362374145  for  1.944795452222253 <= x <=
2.019595277307724
S'27(x) = 0.593185098834468  for  2.019595277307724 <= x <=
2.0943951023931957
S'28(x) = 0.07983651962219795  for  2.0943951023931957 <= x <=
2.169194927478667
S'29(x) = 0.7756108406029134  for  2.169194927478667 <= x <=
2.243994752564138
S'30(x) = 0.2439992072508783  for  2.243994752564138 <= x <=
2.3187945776496095
S'31(x) = 1.0059833999595154  for  2.3187945776496095 <= x <=
2.3935944027350806
S'32(x) = 0.4675679612628847  for  2.3935944027350806 <= x <=
2.4683942278205517
S'33(x) = 1.3097633965632651  for  2.4683942278205517 <= x <=
2.5431940529060233
S'34(x) = 0.7802816621436872  for  2.5431940529060233 <= x <=
2.6179938779914944
S'35(x) = 1.7166219495471422  for  2.6179938779914944 <= x <=
2.6927937030769655
S'36(x) = 1.1971985381850265  for  2.6927937030769655 <= x <=
2.767593528162437
S'37(x) = 2.185704089674891  for  2.767593528162437 <= x <=
2.842393353247908
S'38(x) = 1.514607782652483  for  2.842393353247908 <= x <=
2.9171931783333793
S'39(x) = 2.1035349302339243  for  2.9171931783333793 <= x <=
2.991993003418851
S'40(x) = 0.27249233822742314  for  2.991993003418851 <= x <=
3.066792828504322
S'41(x) = -1.0513929951200065  for  3.066792828504322 <= x <=
3.141592653589793
S'42(x) = -4.350198452342207  for  3.141592653589793 <= x <=
3.2163924786752647
S'43(x) = -3.763385820935423  for  3.2163924786752647 <= x <=
3.291192303760736
S'44(x) = -3.1956973221775233  for  3.291192303760736 <= x <=
3.365992128846207
S'45(x) = -0.9995170989570457  for  3.365992128846207 <= x <=
3.4407919539316785
S'46(x) = -1.0675835317295221  for  3.4407919539316785 <= x <=
3.5155917790171496
S'47(x) = 0.11616649961223846  for  3.5155917790171496 <= x <=
3.5903916041026207
S'48(x) = -0.5444044555224243  for  3.5903916041026207 <= x <=
3.6651914291880923
S'49(x) = 0.35894527181558233  for  3.6651914291880923 <= x <=
3.7399912542735634
S'50(x) = -0.3942417655615347  for  3.7399912542735634 <= x <=
```

```
3.814791079359035
S'51(x) = 0.464906487982379  for  3.814791079359035 <= x <=
3.889590904444506
S'52(x) = -0.2836827016006769  for  3.889590904444506 <= x <=
3.964390729529977
S'53(x) = 0.5692935965924775  for  3.964390729529977 <= x <=
4.039190554615448
S'54(x) = -0.1594733545871141  for  4.039190554615448 <= x <=
4.11399037970092
S'55(x) = 0.6946404526901688  for  4.11399037970092 <= x <=
4.188790204786391
S'56(x) = -0.009017830864457732  for  4.188790204786391 <= x <=
4.263590029871862
S'57(x) = 0.8518641877205186  for  4.263590029871862 <= x <=
4.338389854957334
S'58(x) = 0.18034686350229556  for  4.338389854957334 <= x <=
4.413189680042805
S'59(x) = 1.0570347403391023  for  4.413189680042805 <= x <=
4.487989505128276
S'60(x) = 0.42911762425227407  for  4.487989505128276 <= x <=
4.562789330213747
S'61(x) = 1.33561273020492  for  4.562789330213747 <= x <=
4.637589155299219
S'62(x) = 0.7670333318709601  for  4.637589155299219 <= x <=
4.71238898038469
S'63(x) = 1.717269276450966  for  4.71238898038469 <= x <=
4.787188805470161
S'64(x) = 1.2091522146500813  for  4.787188805470161 <= x <=
4.861988630555633
S'65(x) = 2.161149409840853  for  4.861988630555633 <= x <=
4.9367884556411035
S'66(x) = 1.551763465855728  for  4.9367884556411035 <= x <=
5.011588280726575
S'67(x) = 2.0537782436615304  for  5.011588280726575 <= x <=
5.086388105812047
S'68(x) = 0.3348500281688911  for  5.086388105812047 <= x <=
5.161187930897517
S'69(x) = -1.1263516884306555  for  5.161187930897517 <= x <=
5.235987755982989
S'70(x) = -4.26263875566247  for  5.235987755982989 <= x <=
5.31078758106846
S'71(x) = -3.863546520984135  for  5.31078758106846 <= x <=
5.385587406153931
S'72(x) = -3.0829356187599677  for  5.385587406153931 <= x <=
5.460387231239403
S'73(x) = -1.1248798057433942  for  5.460387231239403 <= x <=
5.535187056324874
S'74(x) = -0.9296198215744509  for  5.535187056324874 <= x <=
5.609986881410345
S'75(x) = -0.0343982139114003  for  5.609986881410345 <= x <=
```

```
5.684786706495816
S'76(x) = -0.3812387386300855  for  5.684786706495816 <= x <=
5.759586531581288
S'77(x) = 0.18317855155443596  for  5.759586531581288 <= x <=
5.834386356666759
S'78(x) = -0.2058740419316456  for  5.834386356666759 <= x <=
5.90918618175223
S'79(x) = 0.26393776098372057  for  5.90918618175223 <= x <=
5.983986006837702
S'80(x) = -0.07011297123320093  for  5.983986006837702 <= x <=
6.058785831923172
S'81(x) = 0.3431228628561913  for  6.058785831923172 <= x <=
6.133585657008644
S'82(x) = 0.07929838251793031  for  6.133585657008644 <= x <=
6.208385482094116
S'83(x) = 0.4432677122163522  for  6.208385482094116 <= x <=
6.283185307179586
S0(x) = 1.0 + 0.0421633375956607 * (x - 0.0) + 0.0 * (x - 0.0)^2 + 0.0
* (x - 0.0)^3  for  0.0 <= x <= 0.07479982508547127
S1(x) = 1.003153810277175 + -0.13890165065311255 * (x -
0.07479982508547127) + 0.0 * (x - 0.07479982508547127)^2 +
47.624624345020734 * (x - 0.07479982508547127)^3  for
0.07479982508547127 <= x <= 0.14959965017094254
S2(x) = 1.0126951848095227 + -0.3764176330917197 * (x -
0.14959965017094254) + 10.686940712306482 * (x -
0.14959965017094254)^2 + -36.95200835048849 * (x -
0.14959965017094254)^3  for  0.14959965017094254 <= x <=
0.2243994752564138
S3(x) = 1.0288681535333022 + -0.05325042327184648 * (x -
0.2243994752564138) + 2.394929428786245 * (x - 0.2243994752564138)^2 +
32.994628651544225 * (x - 0.2243994752564138)^3  for
0.2243994752564138 <= x <= 0.2991993003418851
S4(x) = 1.0520931463771646 + -0.21251265736174524 * (x -
0.2991993003418851) + 9.798906784473006 * (x - 0.2991993003418851)^2 +
-19.202478652723546 * (x - 0.2991993003418851)^3  for
0.2991993003418851 <= x <= 0.37399912542735636
S5(x) = 1.0829859021764197 + 0.011441056621822487 * (x -
0.37399912542735636) + 5.489880651179353 * (x - 0.37399912542735636)^2
+ 18.685256468071852 * (x - 0.37399912542735636)^3  for
0.37399912542735636 <= x <= 0.4487989505128276
S6(x) = 1.12237754238504 + -0.06225796468446909 * (x -
0.4487989505128276) + 9.682842397646187 * (x - 0.4487989505128276)^2 +
-1.3789267277433406 * (x - 0.4487989505128276)^3  for
0.4487989505128276 <= x <= 0.5235987755982989
S7(x) = 1.171319205485879 + 0.07770664407227579 * (x -
0.5235987755982989) + 9.373411963523537 * (x - 0.5235987755982989)^2 +
3.4618419817427633 * (x - 0.5235987755982989)^3  for
0.5235987755982989 <= x <= 0.5983986006837702
S8(x) = 1.2310248203015035 + 0.1276003539927386 * (x -
0.5983986006837702) + 10.150247487647237 * (x - 0.5983986006837702)^2
```

```
+ 12.5919764873163 * (x - 0.5983986006837702)^3  for
0.5983986006837702 <= x <= 0.6731984257692414
S9(x) = 1.3026298979784323 + 0.2807897875592916 * (x -
0.6731984257692414) + 12.975880403842114 * (x - 0.6731984257692414)^2
+ -23.33606162913964 * (x - 0.6731984257692414)^3  for
0.6731984257692414 <= x <= 0.7479982508547127
S10(x) = 1.3864668722100848 + 0.6904750589161444 * (x -
0.7479982508547127) + 7.73928041971184 * (x - 0.7479982508547127)^2 +
-2.8480823123308436 * (x - 0.7479982508547127)^3  for
0.7479982508547127 <= x <= 0.822798075940184
S11(x) = 1.4802237271514045 + 1.3567777574120985 * (x -
0.822798075940184) + 7.100172243337726 * (x - 0.822798075940184)^2 + -
110.5745433351254 * (x - 0.822798075940184)^3  for  0.822798075940184
<= x <= 0.8975979010256552
S12(x) = 1.5751599119723292 + 2.858284478080943 * (x -
0.8975979010256552) + -17.71269725778199 * (x - 0.8975979010256552)^2
+ -91.61333089438563 * (x - 0.8975979010256552)^3  for
0.8975979010256552 <= x <= 0.9723977261111265
S13(x) = 1.6515155710097078 + 4.080950850673459 * (x -
0.9723977261111265) + -38.27068063697435 * (x - 0.9723977261111265)^2
+ -145.40330272602762 * (x - 0.9723977261111265)^3  for
0.9723977261111265 <= x <= 1.0471975511965979
S14(x) = 1.6817928305074292 + 4.052481236962798 * (x -
1.0471975511965979) + -70.89910546924446 * (x - 1.0471975511965979)^2
+ 151.2028983475595 * (x - 1.0471975511965979)^3  for
1.0471975511965979 <= x <= 1.121997376282069
S15(x) = 1.6515155710097078 + 1.3292623961406997 * (x -
1.121997376282069) + -36.969254422803296 * (x - 1.121997376282069)^2 +
74.21454402978603 * (x - 1.121997376282069)^3  for  1.121997376282069
<= x <= 1.1967972013675403
S16(x) = 1.575159911972329 + -0.530513924889316 * (x -
1.1967972013675403) + -20.315549686125276 * (x - 1.1967972013675403)^2
+ 139.57252144280835 * (x - 1.1967972013675403)^3  for
1.1967972013675403 <= x <= 1.2715970264530116
S17(x) = 1.4802237271514045 + -1.8653611579308293 * (x -
1.2715970264530116) + 11.004450885855508 * (x - 1.2715970264530116)^2
+ -37.749087038446895 * (x - 1.2715970264530116)^3  for
1.2715970264530116 <= x <= 1.3463968515384828
S18(x) = 1.3864668722100852 + -1.7329337405967218 * (x -
1.3463968515384828) + 2.533575563019351 * (x - 1.3463968515384828)^2 +
75.53242222300155 * (x - 1.3463968515384828)^3  for
1.3463968515384828 <= x <= 1.421196676623954
S19(x) = 1.3026298979784323 + -1.987225946089488 * (x -
1.421196676623954) + 19.483011474706803 * (x - 1.421196676623954)^2 +
-76.38752832424304 * (x - 1.421196676623954)^3  for  1.421196676623954
<= x <= 1.4959965017094254
S20(x) = 1.2310248203015035 + -1.375828867411834 * (x -
1.4959965017094254) + 2.3416902026122024 * (x - 1.4959965017094254)^2
+ 71.9329010982375 * (x - 1.4959965017094254)^3  for
1.4959965017094254 <= x <= 1.5707963267948966
```

```
S21(x) = 1.171319205485879 + -1.557839434678787 * (x -
1.5707963267948966) + 18.483395462728176 * (x - 1.5707963267948966)^2
+ -85.61500759528502 * (x - 1.5707963267948966)^3  for
1.5707963267948966 <= x <= 1.645596151880368
S22(x) = 1.12237754238504 + -0.9192163626536118 * (x -
1.645596151880368) + -0.728567315727684 * (x - 1.645596151880368)^2 +
79.9078690980073 * (x - 1.645596151880368)^3  for  1.645596151880368
<= x <= 1.7203959769658392
S23(x) = 1.0829859021764199 + -1.1906765104118022 * (x -
1.7203959769658392) + 17.202716578723376 * (x - 1.7203959769658392)^2
+ -90.98983815641958 * (x - 1.7203959769658392)^3  for
1.7203959769658392 <= x <= 1.7951958020513104
S24(x) = 1.0520931463771646 + -0.5668070330558219 * (x -
1.7951958020513104) + -3.215355357243153 * (x - 1.7951958020513104)^2
+ 88.79687940067652 * (x - 1.7951958020513104)^3  for
1.7951958020513104 <= x <= 1.8699956271367817
S25(x) = 1.0288681535333022 + -0.9265921425251447 * (x -
1.8699956271367817) + 16.710617784675733 * (x - 1.8699956271367817)^2
+ -96.43869094481563 * (x - 1.8699956271367817)^3  for
1.8699956271367817 <= x <= 1.944795452222253
S26(x) = 1.0126951848095227 + -0.3035426503266591 * (x -
1.944795452222253) + -4.93017385775638 * (x - 1.944795452222253)^2 +
97.36526619336648 * (x - 1.944795452222253)^3  for  1.944795452222253
<= x <= 2.019595277307724
S27(x) = 1.003153810277175 + -0.7359152118857268 * (x -
2.019595277307724) + 16.91854078423606 * (x - 2.019595277307724)^2 + -
102.18956273506218 * (x - 2.019595277307724)^3  for  2.019595277307724
<= x <= 2.0943951023931957
S28(x) = 1.0 + -0.09968433398079207 * (x - 2.0943951023931957) + -
6.01274347019436 * (x - 2.0943951023931957)^2 + 105.73697386784309 *
(x - 2.0943951023931957)^3  for  2.0943951023931957 <= x <=
2.169194927478667
S29(x) = 1.003153810277175 + -0.5931850988344705 * (x -
2.169194927478667) + 17.714577980950722 * (x - 2.169194927478667)^2 +
-108.00749959170697 * (x - 2.169194927478667)^3  for
2.169194927478667 <= x <= 2.243994752564138
S30(x) = 1.0126951848095227 + 0.06526481172096987 * (x -
2.243994752564138) + -6.522248251185603 * (x - 2.243994752564138)^2 +
114.17574660871513 * (x - 2.243994752564138)^3  for  2.243994752564138
<= x <= 2.3187945776496095
S31(x) = 1.0288681535333022 + -0.4823318647158646 * (x -
2.3187945776496095) + 19.09872937481944 * (x - 2.3187945776496095)^2 +
-113.62875733013368 * (x - 2.3187945776496095)^3  for
2.3187945776496095 <= x <= 2.3935944027350806
S32(x) = 1.0520931463771646 + 0.20396778071358093 * (x -
2.3935944027350806) + -6.399504144100886 * (x - 2.3935944027350806)^2
+ 122.91653835142691 * (x - 2.3935944027350806)^3  for
2.3935944027350806 <= x <= 2.4683942278205517
S33(x) = 1.0829859021764197 + -0.3924383780848143 * (x -
2.4683942278205517) + 21.18290256229412 * (x - 2.4683942278205517)^2 +
```

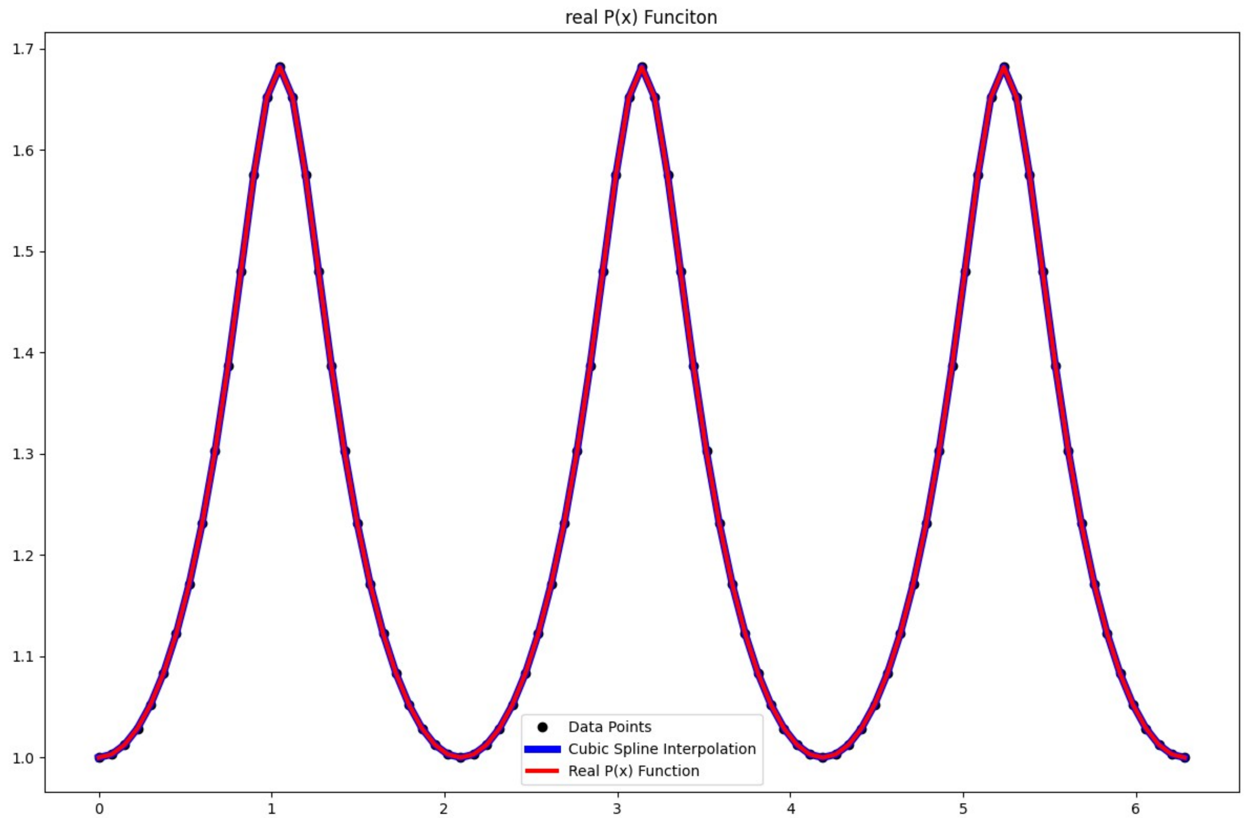
```
-118.929391558552 * (x - 2.4683942278205517)^3  for
2.4683942278205517 <= x <= 2.5431940529060233
S34(x) = 1.1223775423850402 + 0.32902046665349877 * (x -
2.5431940529060233) + -5.504790496009623 * (x - 2.5431940529060233)^2
+ 131.73135232135309 * (x - 2.5431940529060233)^3  for
2.5431940529060233 <= x <= 2.6179938779914944
S35(x) = 1.171319205485879 + -0.30097078389700893 * (x -
2.6179938779914944) + 24.055655839719716 * (x - 2.6179938779914944)^2
+ -125.144068089819 * (x - 2.6179938779914944)^3  for
2.6179938779914944 <= x <= 2.6927937030769655
S36(x) = 1.2310248203015035 + 0.4936767785932689 * (x -
2.6927937030769655) + -4.026607371088531 * (x - 2.6927937030769655)^2
+ 136.69351758134692 * (x - 2.6927937030769655)^3  for
2.6927937030769655 <= x <= 2.767593528162437
S37(x) = 1.3026298979784332 + -0.07268563367245373 * (x -
2.767593528162437) + 26.647346245119216 * (x - 2.767593528162437)^2 +
-142.93323374566816 * (x - 2.767593528162437)^3  for
2.767593528162437 <= x <= 2.842393353247908
S38(x) = 1.3864668722100848 + 1.0313494767792795 * (x -
2.842393353247908) + -5.426796404110998 * (x - 2.842393353247908)^2 +
112.24472082672061 * (x - 2.842393353247908)^3  for  2.842393353247908
<= x <= 2.9171931783333793
S39(x) = 1.480223727151404 + 1.0285043429174983 * (x -
2.9171931783333793) + 19.760860049707716 * (x - 2.9171931783333793)^2
+ -221.16297749670764 * (x - 2.9171931783333793)^3  for
2.9171931783333793 <= x <= 2.991993003418851
S40(x) = 1.5751599119723287 + 3.173956889206956 * (x -
2.991993003418851) + -29.86799604669973 * (x - 2.991993003418851)^2 +
14.470734289743309 * (x - 2.991993003418851)^3  for  2.991993003418851
<= x <= 3.066792828504322
S41(x) = 1.6515155710097074 + 3.777879442915891 * (x -
3.066792828504322) + -26.620770865506344 * (x - 3.066792828504322)^2 +
-246.98299893271394 * (x - 3.066792828504322)^3  for
3.066792828504322 <= x <= 3.141592653589793
S42(x) = 1.6817928305074292 + 4.342951641351994 * (x -
3.141592653589793) + -82.04362622326265 * (x - 3.141592653589793)^2 +
248.27822557678925 * (x - 3.141592653589793)^3  for  3.141592653589793
<= x <= 3.2163924786752647
S43(x) = 1.6515155710097074 + 1.0513929951199397 * (x -
3.2163924786752647) + -26.330122686237388 * (x - 3.2163924786752647)^2
+ -18.35641422196005 * (x - 3.2163924786752647)^3  for
3.2163924786752647 <= x <= 3.291192303760736
S44(x) = 1.5751599119723294 + -0.26524552723705663 * (x -
3.291192303760736) + -30.449292405234587 * (x - 3.291192303760736)^2 +
227.63911071702717 * (x - 3.291192303760736)^3  for  3.291192303760736
<= x <= 3.365992128846207
S45(x) = 1.4802237271514045 + -2.1180285522144517 * (x -
3.365992128846207) + 20.63280458750289 * (x - 3.365992128846207)^2 + -
121.31130733509316 * (x - 3.365992128846207)^3  for  3.365992128846207
<= x <= 3.4407919539316785
```

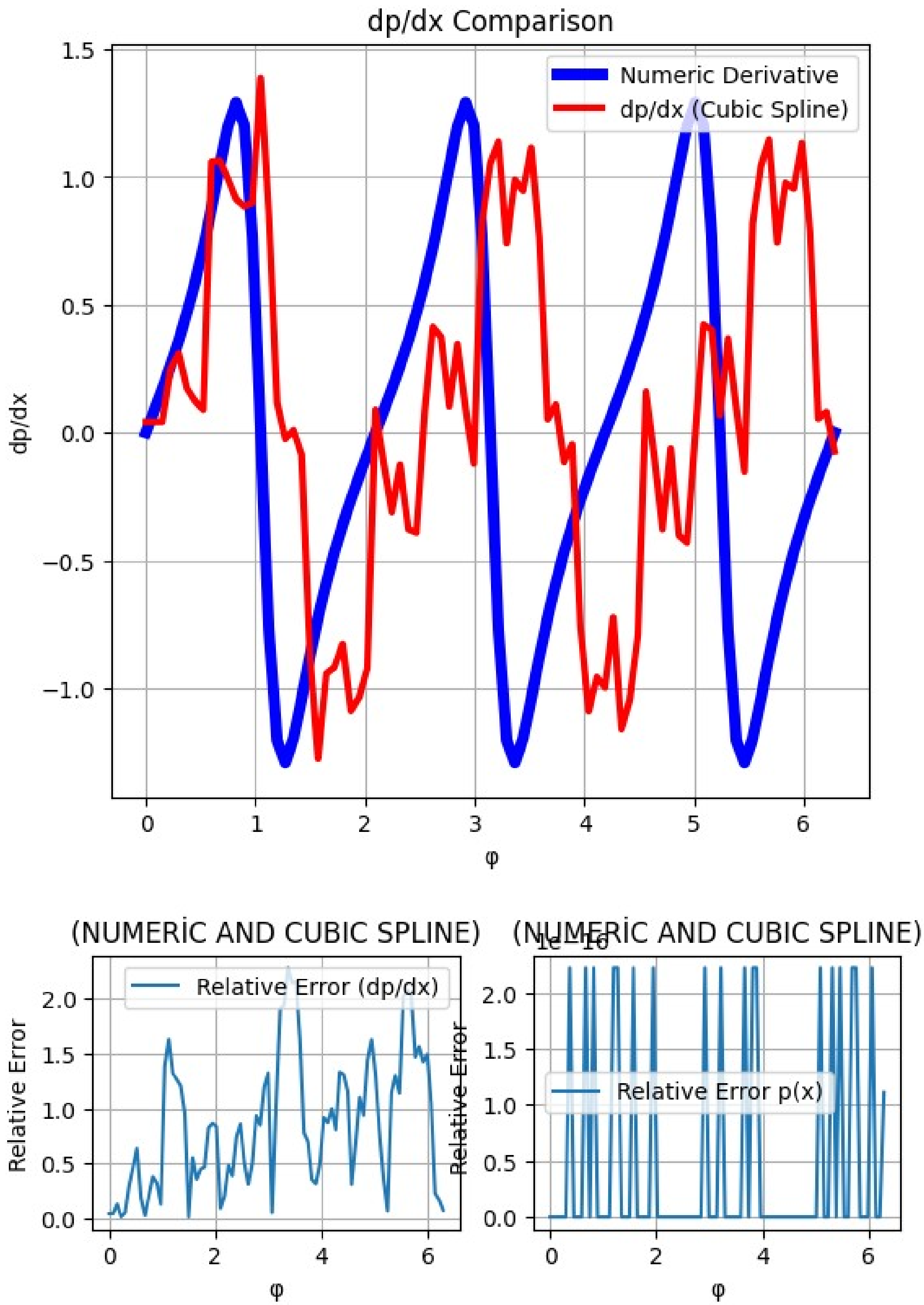
```
S46(x) = 1.3864668722100852 + -1.4928673496818858 * (x -
3.4407919539316785) + -6.589389121161666 * (x - 3.4407919539316785)^2
+ 154.59027354204326 * (x - 3.4407919539316785)^3  for
3.4407919539316785 <= x <= 3.5155917790171496
S47(x) = 1.3026298979784323 + -2.214691333635404 * (x -
3.5155917790171496) + 28.100587141418245 * (x - 3.5155917790171496)^2
+ -150.94101066565972 * (x - 3.5155917790171496)^3  for
3.5155917790171496 <= x <= 3.5903916041026207
S48(x) = 1.2310248203015037 + -1.160964483234888 * (x -
3.5903916041026207) + -5.770496446628481 * (x - 3.5903916041026207)^2
+ 141.98201446202154 * (x - 3.5903916041026207)^3  for
3.5903916041026207 <= x <= 3.6651914291880923
S49(x) = 1.1713192054858788 + -1.7601028154867628 * (x -
3.6651914291880923) + 26.090193094497835 * (x - 3.6651914291880923)^2
+ -151.159751981435 * (x - 3.6651914291880923)^3  for
3.6651914291880923 <= x <= 3.7399912542735634
S50(x) = 1.12237754238504 + -0.7295539852146185 * (x -
3.7399912542735634) + -7.829975930025764 * (x - 3.7399912542735634)^2
+ 140.9482445065165 * (x - 3.7399912542735634)^3  for
3.7399912542735634 <= x <= 3.814791079359035
S51(x) = 1.0829859021764197 + -1.3677378844818118 * (x -
3.814791079359035) + 23.798736175549372 * (x - 3.814791079359035)^2 +
-147.5258445872823 * (x - 3.814791079359035)^3  for  3.814791079359035
<= x <= 3.889590904444506
S52(x) = 1.0520931463771646 + -0.40234666235482214 * (x -
3.889590904444506) + -9.305985936595969 * (x - 3.889590904444506)^2 +
140.8285168538831 * (x - 3.889590904444506)^3  for  3.889590904444506
<= x <= 3.964390729529977
S53(x) = 1.0288681535333022 + -1.0784515098570862 * (x -
3.964390729529977) + 22.29585934655436 * (x - 3.964390729529977)^2 + -
143.9659594203603 * (x - 3.964390729529977)^3  for  3.964390729529977
<= x <= 4.039190554615448
S54(x) = 1.0126951848095227 + -0.16428428636379616 * (x -
4.039190554615448) + -10.010026402160591 * (x - 4.039190554615448)^2 +
140.38816569124933 * (x - 4.039190554615448)^3  for  4.039190554615448
<= x <= 4.11399037970092
S55(x) = 1.003153810277175 + -0.8625725724795267 * (x -
4.11399037970092) + 21.49300431116636 * (x - 4.11399037970092)^2 + -
140.70809325528856 * (x - 4.11399037970092)^3  for  4.11399037970092
<= x <= 4.188790204786391
S56(x) = 1.0 + 0.0143720232439584 * (x - 4.188790204786391) + -
10.081817979651058 * (x - 4.188790204786391)^2 + 139.75113541041506 *
(x - 4.188790204786391)^3  for  4.188790204786391 <= x <=
4.263590029871862
S57(x) = 1.003153810277175 + -0.6946404526901699 * (x -
4.263590029871862) + 21.278263472933865 * (x - 4.263590029871862)^2 +
-137.51729215662377 * (x - 4.263590029871862)^3  for
4.263590029871862 <= x <= 4.338389854957334
S58(x) = 1.0126951848095227 + 0.1541916220761124 * (x -
4.338389854957334) + -9.580544725695578 * (x - 4.338389854957334)^2 +
```

```
139.18117019598412 * (x - 4.338389854957334)^3  for  4.338389854957334
<= x <= 4.413189680042805
S59(x) = 1.0288681535333024 + -0.5585852118334583 * (x -
4.413189680042805) + 21.651636831857004 * (x - 4.413189680042805)^2 +
-134.12981193976248 * (x - 4.413189680042805)^3  for
4.413189680042805 <= x <= 4.487989505128276
S60(x) = 1.0520931463771646 + 0.2676201244621643 * (x -
4.487989505128276) + -8.447022583666923 * (x - 4.487989505128276)^2 +
138.9132239834201 * (x - 4.487989505128276)^3  for  4.487989505128276
<= x <= 4.562789330213747
S61(x) = 1.0829859021764199 + -0.4434897184643959 * (x -
4.562789330213747) + 22.725031984389346 * (x - 4.562789330213747)^2 +
-130.42170821291995 * (x - 4.562789330213747)^3  for
4.562789330213747 <= x <= 4.637589155299219
S62(x) = 1.1223775423850402 + 0.36747080366410056 * (x -
4.637589155299219) + -6.54153090063512 * (x - 4.637589155299219)^2 +
138.71929999810578 * (x - 4.637589155299219)^3  for  4.637589155299219
<= x <= 4.71238898038469
S63(x) = 1.1713192054858788 + -0.32682011753867324 * (x -
4.71238898038469) + 24.58700722687661 * (x - 4.71238898038469)^2 + -
127.62764678896256 * (x - 4.71238898038469)^3  for  4.71238898038469
<= x <= 4.787188805470161
S64(x) = 1.2310248203015035 + 0.5069251088660534 * (x -
4.787188805470161) + -4.052569740777628 * (x - 4.787188805470161)^2 +
134.6727273028806 * (x - 4.787188805470161)^3  for  4.787188805470161
<= x <= 4.861988630555633
S65(x) = 1.3026298979784328 + -0.07333296057641459 * (x -
4.861988630555633) + 26.167919597339008 * (x - 4.861988630555633)^2 +
-136.4080744895618 * (x - 4.861988630555633)^3  for  4.861988630555633
<= x <= 4.9367884556411035
S66(x) = 1.3864668722100848 + 1.0193958003142396 * (x -
4.9367884556411035) + -4.441980738856227 * (x - 4.9367884556411035)^2
+ 101.21519259293994 * (x - 4.9367884556411035)^3  for
4.9367884556411035 <= x <= 5.011588280726575
S67(x) = 1.480223727151405 + 1.0530590227516887 * (x -
5.011588280726575) + 18.270655366976445 * (x - 5.011588280726575)^2 +
-205.6290802852511 * (x - 5.011588280726575)^3  for  5.011588280726575
<= x <= 5.086388105812047
S68(x) = 1.5751599119723296 + 3.1368012060037014 * (x -
5.086388105812047) + -27.872402346493065 * (x - 5.086388105812047)^2 +
-5.567531899393487 * (x - 5.086388105812047)^3  for  5.086388105812047
<= x <= 5.161187930897517
S69(x) = 1.6515155710097074 + 3.8276361294882957 * (x -
5.161187930897517) + -29.121753583190298 * (x - 5.161187930897517)^2 +
-222.44036376588966 * (x - 5.161187930897517)^3  for
5.161187930897517 <= x <= 5.235987755982989
S70(x) = 1.681792830507429 + 4.28059395141039 * (x -
5.235987755982989) + -79.03725448810194 * (x - 5.235987755982989)^2 +
219.23122143229887 * (x - 5.235987755982989)^3  for  5.235987755982989
<= x <= 5.31078758106846
```

```
S71(x) = 1.6515155710097074 + 1.1263516884306586 * (x -
5.31078758106846) + -29.84188343887122 * (x - 5.31078758106846)^2 +
15.194958900164348 * (x - 5.31078758106846)^3  for  5.31078758106846
<= x <= 5.385587406153931
S72(x) = 1.5751599119723292 + -0.3528052239167968 * (x -
5.385587406153931) + -26.432142635131594 * (x - 5.385587406153931)^2 +
189.58336861729802 * (x - 5.385587406153931)^3  for  5.385587406153931
<= x <= 5.460387231239403
S73(x) = 1.4802237271514038 + -2.0178678521657765 * (x -
5.460387231239403) + 16.11026579993352 * (x - 5.460387231239403)^2 + -
78.75119625778645 * (x - 5.460387231239403)^3  for  5.460387231239403
<= x <= 5.535187056324874
S74(x) = 1.386466872210084 + -1.605629053099476 * (x -
5.535187056324874) + -1.561461316128689 * (x - 5.535187056324874)^2 +
107.52579348719156 * (x - 5.535187056324874)^3  for  5.535187056324874
<= x <= 5.609986881410345
S75(x) = 1.302629897978432 + -2.0893286268491242 * (x -
5.609986881410345) + 22.567270318926422 * (x - 5.609986881410345)^2 +
-99.37216163328209 * (x - 5.609986881410345)^3  for  5.609986881410345
<= x <= 5.684786706495816
S76(x) = 1.2310248203015037 + -1.2989281933897816 * (x -
5.684786706495816) + 0.268209393322298 * (x - 5.684786706495816)^2 +
85.90879645210751 * (x - 5.684786706495816)^3  for  5.684786706495816
<= x <= 5.759586531581288
S77(x) = 1.1713192054858788 + -1.6095381019631123 * (x -
5.759586531581288) + 19.546098237085364 * (x - 5.759586531581288)^2 +
-90.58216499397197 * (x - 5.759586531581288)^3  for  5.759586531581288
<= x <= 5.834386356666759
S78(x) = 1.1223775423850402 + -0.8927197021070103 * (x -
5.834386356666759) + -0.7804920551516812 * (x - 5.834386356666759)^2 +
75.86628854150332 * (x - 5.834386356666759)^3  for  5.834386356666759
<= x <= 5.90918618175223
S79(x) = 1.0829859021764197 + -1.1919711642207007 * (x -
5.90918618175223) + 16.243863283213408 * (x - 5.90918618175223)^2 + -
77.93951964471157 * (x - 5.90918618175223)^3  for  5.90918618175223 <=
x <= 5.983986006837702
S80(x) = 1.0520931463771646 + -0.5907143859847055 * (x -
5.983986006837702) + -1.2457240267968928 * (x - 5.983986006837702)^2 +
66.7378229337487 * (x - 5.983986006837702)^3  for  5.983986006837702
<= x <= 6.058785831923172
S81(x) = 1.0288681535333024 + -0.8774827828583871 * (x -
6.058785831923172) + 13.730208419291655 * (x - 6.058785831923172)^2 +
-65.3708965226618 * (x - 6.058785831923172)^3  for  6.058785831923172
<= x <= 6.133585657008644
S82(x) = 1.0126951848095227 + -0.3778540167312864 * (x -
6.133585657008644) + -0.9389864574350337 * (x - 6.133585657008644)^2 +
57.28873381599259 * (x - 6.133585657008644)^3  for  6.133585657008644
<= x <= 6.208385482094116
S83(x) = 1.003153810277175 + -0.6364018387432578 * (x -
6.208385482094116) + 11.916575348978123 * (x - 6.208385482094116)^2 +
```

```
-53.10429240247351 * (x - 6.208385482094116)^3  for  6.208385482094116
<= x <= 6.283185307179586
```




dp/dx Comparison
Numeric Derivative
dp/dx (Cubic Spline)
dp/dx
φ
1.5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6
(NUMERİC AND CUBIC SPLINE)
Relative Error (dp/dx)
Relative Error
φ
(NUMERİC AND CUBIC SPLINE)
1e−16
Relative Error p(x)
Relative Error
φ

As seen, the solution has been found with an average error of approximately 1.2. It is challenging to decide whether the Hermite function or the cubic spline method is more reasonable (because cubic spline requires a more powerful computer to match the interval with that of Hermite). However, for an engineer responsible for making the most efficient use of available resources, choosing the Hermite function is appropriate.

Section g-)

In Section G, a function is provided, and it is requested to calculate the integral of this function over the interval [0, 2π] using the Trapezoidal, Simpson, and Gauss quadrature methods separately. Various research has been conducted, and references have been added to the end of the report. Based on the findings of this research, algorithms have been developed using ChatGPT alongside discussions on the logic of these algorithms, primarily on YouTube. This collaborative approach was taken because the number of series and calculations becomes cumbersome at some point. The references for these investigations have also been included at the end of the report.

First, the given functions are written in code. Later, the integrals for each of the three methods are calculated with the described functions, and the results are printed with explanations

```
#section g-)

#creating f(x) function according to the report...
def l_function(x, x_function, y_function, h):
    return (newton_forward(x_function, x, h)**2 +
newton_forward(y_function, x, h)**2)**0.5

def f(x, p, x_function, y_function, h):
    return np.log(p(x)) * l_function(x, x_function, y_function, h)

# obtaining the gauss quadrature function....
def i_gauss(func, a, b, n, p, x_function, y_function, h):
    nod, mean = np.polynomial.legendre.leggauss(n)
```

```
    total = 0.0

    for i in range(n):
        x = 0.5 * (b - a) * nod[i] + 0.5 * (a + b)
        total += mean[i] * func(x, p, x_function, y_function, h)

    return 0.5 * (b - a) * total

# obtaining the simpsons function....
def i_simpson(func, a, b, n, p, x_function, y_function, h):
    h = (b - a) / n
    total = func(a, p, x_function, y_function, h) + func(b, p,
x_function, y_function, h)

    for i in range(1, n, 2):
        total += 4 * func(a + i * h, p, x_function, y_function, h)

    for i in range(2, n-1, 2):
        total += 2 * func(a + i * h, p, x_function, y_function, h)

    return (h / 3) * total

# obtaining trapezoidal function....
def i_trapezoidal(func, a, b, n, p, x_function, y_function, h):
    h = (b - a) / n
    total = 0.5 * (func(a, p, x_function, y_function, h) + func(b, p,
x_function, y_function, h))

    for i in range(1, n):
        total += func(a + i * h, p, x_function, y_function, h)

    return h * total


# limit of total....
(a, b) = (0, 2 * np.pi)

# creating subregions...
n = 1000


rval_trap = i_trapezoidal(f, a, b, n, p, x_function, y_function, h)
rval_simp = i_simpson(f, a, b, n, p, x_function, y_function, h)
rval_gauss = i_gauss(f, a, b, n, p, x_function, y_function, h)

# showing the result....
print("The Integral , according to Trapezoidal Rule:
{}".format(rval_trap))
print("The Integral , according to Simpson Rule :
{}".format(rval_simp))
print("The Integral , according to Gauss Quad. Rule:
```

```
{}".format(rval_gauss))
```

```
The Integral , according to Trapezoidal Rule: 2.215572011637046
The Integral , according to Simpson Rule :   2.21557201163704
The Integral , according to Gauss Quad. Rule: 2.215636111554518
```

As observed, the algorithms have been correctly implemented (without using libraries), and the integral has been calculated separately for each of the three methods.

The result is that as the value of n increases, the calculated values approach the actual values.

Section h-)

Finally,

In this project, it has been determined that issues and challenges arise when drawing functions, the extent to which data can approach accuracy, and the significance of the processor used are key factors. Regarding the internal structure of the given task, the graphing of the given function and its directional trends in space were initially visualized. Subsequently, the methods of using forward and backward difference techniques to derive derivative functions from this original function were learned. Later on, attempting to rediscover the desired values using Hermite and cubic spline methods contributed to a better understanding of the logic behind Hermite and cubic spline methods. Finally, the integral operation performed at the end has also been beneficial in this regard.

Refereces :

[13] : hermite_interpolation(x) function is from on chatgpt(only writing long array) [14] : using method of cubic_spline_interpolation() is from https://www.youtube.com/watch?v=ZhljXJG1Ylg(tridiagonal matrix) [14] : cubic formula(x) and derivative_cubic(x) is from on chatgpt (only writing long array) [16] : using method of simpson rule and trapezoidal method is from https://www.youtube.com/watch?v=4XdVlALIEkk