MACHINE PROBLEM NO. 2

ES 204 1st Sem 2023-2024

As a student of the University of the Philippines, I pledge to act ethically and uphold the values of honor and excellence.

I understand that suspected misconduct on this Assignment will be reported to the appropriate office and if established, will result in disciplinary action in accordance with University rules, policies and procedures. I may work with others only to the extent allowed by the Instructor.

Name: Salas, Jeryl
Student Number: 202321128

GENERAL INSTRUCTIONS: Solve all the problems using appropriate numerical and programming techniques, independently and completely. Cite all references or any assistance that you received during the development of your solution. Submit a brief but comprehensive EXECUTIVE SUMMARY of your solution to the problems. Add the complete solution(s) and source codes in the APPENDIX.

PROBLEM 1 [30 points]: Given the hypothetical data below:

x	1	2	3	4
y	1	2	0	3

Determine and plot the interpolating polynomial using the Lagrange and cubic spline methods. Comment on the estimate at $x = 2.75$.

1.1 Method(s) of solution

We were tasked to plot the interpolating polynomial using the Lagrange method. Lagrange method is just one of the methods to find a polynomial that passes through the set of points.

We can construct a Lagrange Polynomial of degree n using n+1 data points

$$I\,(x) = \sum_{i=0}^{n} y_i \cdot\ L_i(x) \qquad (1)$$

The Lagrange coefficient $L_i(x)$ can be seen as,

$$L_i = \prod_{j=0(j\neq i)}^{n} \left(\frac{x\ -\ x_j}{x_i - x_j}\right) \quad for\ \ 0 \leq\ i\ \leq n \qquad (2)$$

The steps used to find the interpolating polynomial are as follows:

1. Import numpy (for matrix operations), sympy (for representing the created polynomial with variable x) and matplotlib (used for visualizing the polynomial curve)
2. Initialize x and y data given from the problem in array type
3. Initialize variable x using sympy which will help represent the created polynomial later
4. Call on the function “Lagrange interpolation method”. It uses formulas 1 and 2 to help find the Lagrange coefficients and the interpolating polynomial

```
function LAGRANGE-METHOD-INTERPOLATION(x variable, x, fx)
returns Interpolating Polynomial
      coefficients = GET-LAGRANGE-COEFFICIENT(x variable, x)
returns Lagrange coefficient
      Initialize interpolation polynomial as zero
      loop do
         interpolation polynomial += fx + x
         end
         return interpolation polynomial
```

Figure 1: Lagrange Interpolation Algorithm

5. Input the x = 2.75 to the created interpolating polynomial and get the output
6. Plot the interpolating polynomial on a chart as well as the given [x, f(x)].

We were also tasked to plot interpolating polynomial using the cubic spline method. The following cubic equation for each interval is as follows:

$$f_i(x) = \frac{f''_i(x_{i-1})}{6(x_i - x_{i-1})}(x_i - x)^3 + \frac{f''_i(x_i)}{6(x_i - x_{i-1})}(x - x_{i-1})^3 + \left[\frac{f(x_{i-1})}{x_i - x_{i-1}} - \frac{f''_i(x_{i-1})(x_i - x_{i-1})}{6}\right](x_i - x) + \left[\frac{f(x_i)}{x_i - x_{i-1}} - \frac{f''_i(x_i)(x_i - x_{i-1})}{6}\right](x - x_{i-1}) \quad (3)$$

To find the second derivatives at the end of each interval. We use the following formula:

$$(x - x_{i-1})f''(x_i - 1) + 2(x_{i+1} - x_{i+1})f''(x_i) + (x_{i+1} - x_{i+1})f''(x_{i+1}) = \frac{6}{x_{i+1} - x_i}[f(x_{i+1}) - f(x_i)] + \frac{6}{x_{i+1} - x_i}[f(x_{i-1}) - f(x_i)] \quad (4)$$

The steps used to find the interpolating polynomial are as follows:

1. Import numpy (for matrix operations), sympy (for representing the cubic polynomials with variable x) and matplotlib (used for visualizing interval cubic equations)
2. Initialize x and y data given from the problem in array type
3. Initialize variable x and f''(x) using sympy which will help in solving the cubic equations.
4. We call the cubic spline interpolation function with inputs of x, f''(x), x and y data
5. We initialize x0, x1, x2, y0, y1, y2 which will be used in equations 1 and 2
6. We create two equations so that we can solve simultaneously the second derivatives at the end of each interval
7. We use those values as substitutes for equation 1 where we will output the cubic equation
8. Repeat steps 5 to 7 for the other two cubic equations
9. Combine these three cubic equations in their respective intervals to create a polynomial curve
10. Input the x = 2.75 to the created interpolating polynomial and get the output

```
function EQUATION-1
        return cubic equation

function EQUATION-2
        return equation

function CUBIC-SPLINE-INTERPOLATION(x, f''(x), x_data, y_data)
        equations = empty array
      polynomials = empty array
        do until i = 1
                initialize x0, x1, x2, y0, y1, y2
                equation = EQUATION-2(f''(x), x0, x1, x2, y0, y1, y2)
```

```
        equations.append(equation)
    a, b = SOLVE(equations[0], equations[1])
    do until i = 2
        cubic polynomial = EQUATION-1(x, a, b, x0, x1, x2, y0, y1, y2)
        polynomials.append(cubic polynomial)

    PLOT_CUBIC_EQUATIONS(polynomials)
```

Figure 2: Cubic Spline Method Pseudocode

1.2 Results

```
Enter the number of data points: 4
Enter x1: 1
Enter y1: 1
Enter x2: 2
Enter y2: 2
Enter x3: 3
Enter y3: 0
Enter x4: 4
Enter y4: 3

Interpolating polynomial:
f(x) =  1.33333333333333*x**3 - 9.5*x**2 + 20.1666666666667*x - 11.0

Please enter value for x: 2.75
f(2.75) =  0.343750000000000
```

Figure 3: Results of the Lagrange Method

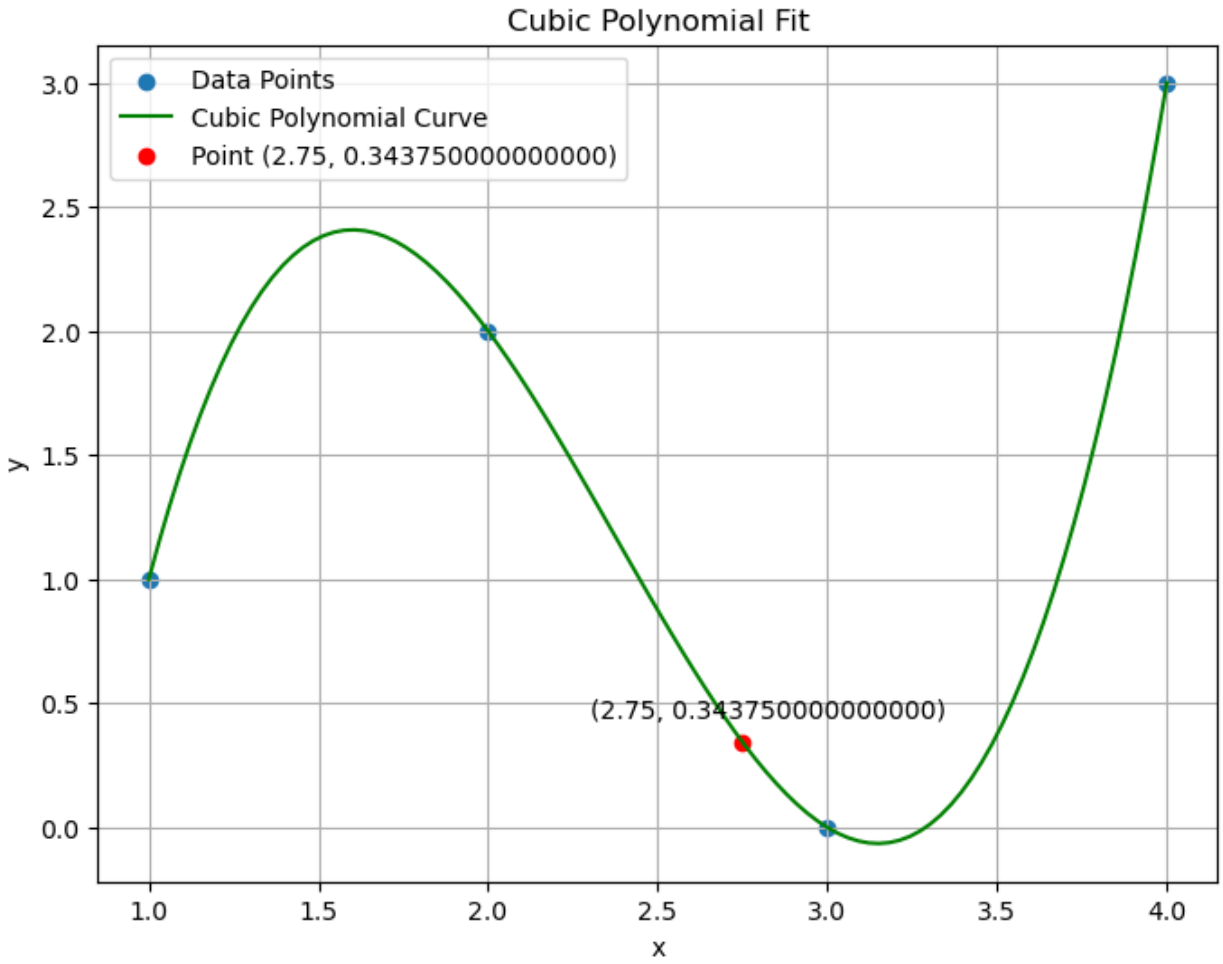

Figure 4: Interpolating Polynomial Curve as well as the input x = 2.75

From figure 3, we can see the results of the program that we created. We first enter manually the data points of both x and y, then the program outputs the created polynomial. The interpolating polynomial formed was $f(x) = 1.33x^3 - 9.5x^2 + 20.17x - 11$. We are then asked what x value should we enter to check the output, f(x). We enter the value of 2.75 and got the output of $f(2{,}75) = 0.34375$. From figure 4, we can see the cubic polynomial curve as well as f(2.75) to see visually where the data points are in the curve.

```
equation 1: 4*f''(2) + f''(3) + 18.0
equation 2: f''(2) + 4*f''(3) - 30.0
First interval cubic:  2.13333333333333*x - 1.13333333333333*(x - 1)**3 -
1.13333333333333
Second interval cubic:  -4.66666666666667*x - 1.13333333333333*(3 - x)**3
+ 1.53333333333333*(x - 2)**3 + 12.4666666666667
Third interval cubic:  4.53333333333333*x + 1.53333333333333*(4 - x)**3 -
15.1333333333333

Inputting x = 2.75
f(2.75) =  0.262500000000001
```

Figure 5: Results from the Cubic Spline Method

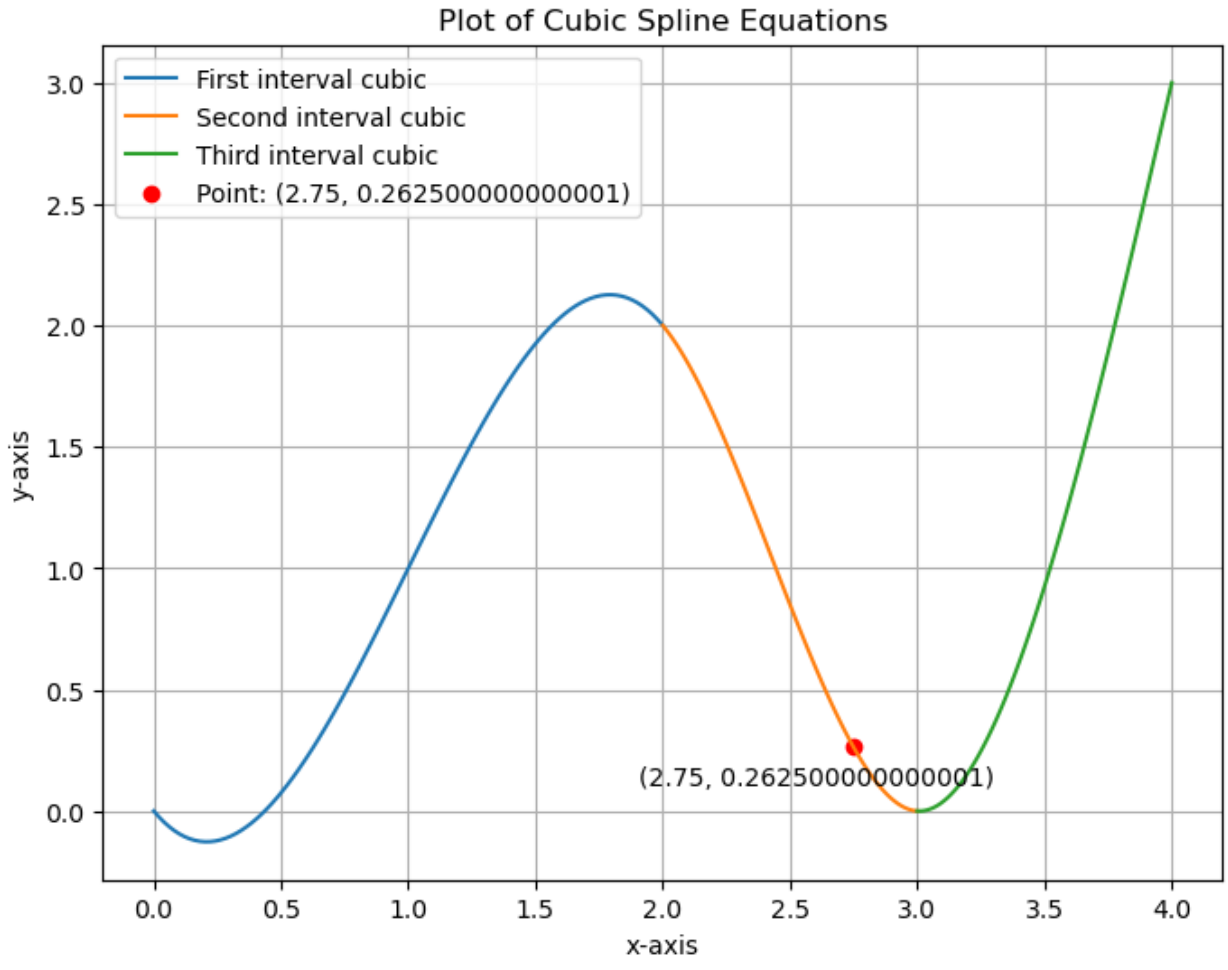

Figure 6: Plot of the Cubic Spline Method

As shown from figure 5, we can see that the result of f(2.75) differs from the Lagrange method. The output from f(2.75) is 0.2625. Figure 6 shows the three cubic equations that are plotted on the graph as well as the input x = 2.75.

1.3 Problems encountered

Out of the four problems in this MP, this Lagrange is the least challenging. The Lagrange method is fairly easy to understand and easy to implement in code. The example on the PDF was easy to follow so I was able to do the examples on the PDF before proceeding with the actual problem. The problem I encountered was how to plot the curve with an additional data point since I'm not that familiar in using Matplotlib. I had to get the coefficients of the polynomial as an array and make a polynomial curve with a degree of 3. We also need to determine whether the additional input and output of f(2.75) is beyond the range of the given x and y data points so It took me a while to output a working chart.

As for the cubic spline method, it is challenging to get the outputs right since I have to rely on sympy to create the cubic equations for me. The formulas are long and you can easily create mistakes with the proper use of x and y values. We also had to make sure that the three cubic equations have proper intervals so as to make sure that the overall curve is accurate and well connected.

1.4 References

- ES_204_L4_Interpolation_and_Data_Fitting.pdf

PROBLEM 2 [30 points]: Given the following experimental data,

x	0.1	0.2	0.4	0.6	0.9	1.3	1.5	1.7	1.8
y	0.75	1.25	1.45	1.25	0.85	0.55	0.35	0.28	0.18

Fit the data into the equation using non-linear least squares. **DO NOT LINEARIZE** the equation when solving this problem.

$$y = \alpha x e^{\beta x}$$

2.1 Method of solution

We will use non-linear least squares to find the best fit into the equation with minimal residuals. The residual, r_i can be outputted by:

$$r_i = \alpha x_i e^{\beta x_i} - y_i \qquad (5)$$

We can also get the squared residual norm by:

$$||r||^2 = \sqrt{\sum_{i=1}^{n} |r_i|^2} \qquad (6)$$

From there, we can form the Jacobian matrix:

$$J(x) = \left(\frac{\partial r_i}{\partial x_j}\right) = \begin{pmatrix} \nabla r_1(x)^T \\ \vdots \\ \nabla r_n(x)^T \end{pmatrix} \qquad (7)$$

With the Jacobian matrix, we will perform repeated operations with k iterations:

$$p^k = ((J^T)^K \cdot J^K)^{-1}(-(J^T)^K \cdot r^k \qquad (8)$$

From vector p, we can finally update α and β

$$\alpha = \alpha + p_1 \quad (9)$$

$$\beta = \beta + p_2 \quad (10)$$

The steps used to fit the data in the equation using non-linear least squares are as follows:

1. Import numpy (for matrix operations since Jacobian and residuals are matrices), sympy (for model representation), and matplotlib (for visualizing the curve fitting)
2. Initialize x and y data points. Since there are many data points in this problem. We decided to put the values in the program instead of asking the user's input
3. Initialize variable x using sympy as this will be used for model representation later.
4. Initialize parameters, α and β.
5. Use the non-linear least squares function with the x and y data points, α, β, max iterations, x variable, and tolerance as inputs. This function calls on two other functions named Jacobian and Residuals. For k iterations which is equal to max iterations, we use equation 5 to solve for the residuals and equation 7 to solve for the Jacobian matrix
6. We will also use equation 6 to get the squared residual norm
7. We will use equation 8 to get vector p which will then be used to update the parameters of α and β.
8. Repeat steps 5 to 7 until the difference of the previous squared residual norm, $prev\ ||r||^2$ and the current $||r||^2$ is less than the tolerance.
9. Using x variable that was created using sympy, we construct the model equation and plot it in on a curve along with the given data points

```
function RESIDUALS (x values, y values, a, b) outputs RESIDUAL vector
      return r, r_2

function JACOBIAN (x values, a, b) returns the JACOBIAN matrix
      return J

function NON-LINEAR-LEAST-SQUARES (x variable, x values, y values, a, b, N-
iterations, tolerance)
      previous r_2 = infinity
      r_2 values = []
      k = 0
      loop do
            r, r_2 = RESIDUALS(x values, y values, a, b)
                J = JACOBIAN(x values, a, b)
                Jr = J.T · r
                Jt = J.t · J
                p = GET-P-VECTOR(Jt^-1, -Jr.T)
                a, b = UPDATE-PARAM(p)

            difference = |previous r_2 - r_2|
            if difference < tolerance
                  PRINT-MODEL(a, b)
            if k == N-iterations
                  return "The model was not able to converge"

            append r_2 to r_2 values
            previous r_2 = r_2
            k += 1
      end
      PLOT_FITNESS_CURVE(x values, y values, a, b)
      PLOT_ERROR_CURVE(k, r_2 values)
```

Figure 7: Non-linear Least Squares Pseudocode

2.2 Results

```
At k = 225, r_2 = 0.01847131336803916

Model obtained after fitting:
a = 9.983588735581057, b = -2.5440459644052758

Model equation:
y = 9.983588735581057*exp(-2.5440459644052758*x)

Symbolic representation:
y = 9.98358873558106*exp(-2.54404596440528*x)
```

Figure 8: Non-linear Least Squares Results

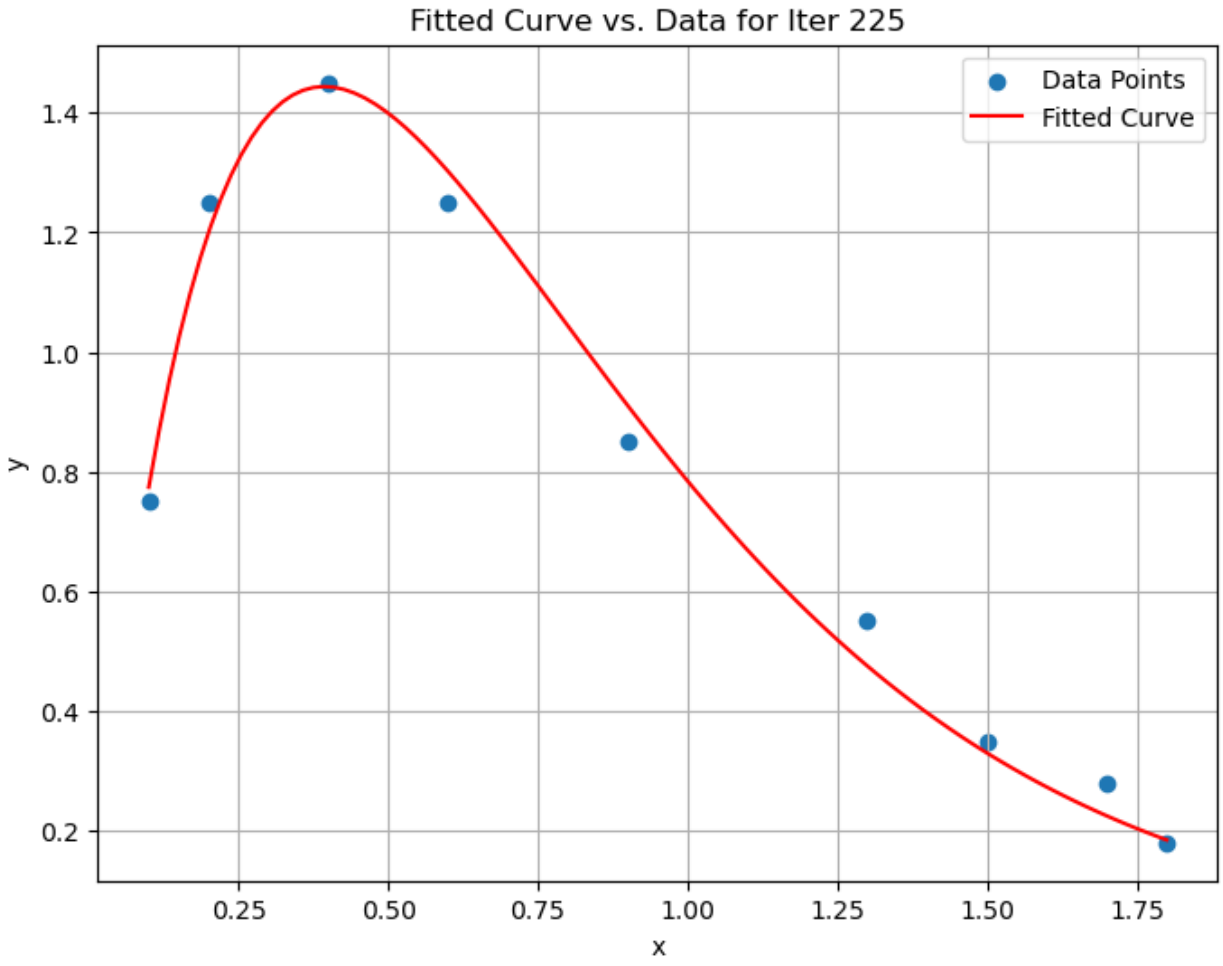

Figure 9: Fitness Curve after 225 iterations

Figure 8 shows the results after 225 iterations, we were able to get a $||r||^2$ of 0.01847. The value for α is 9.98359 while the value for β is -2.54405. And therefore, the model obtained after fitting is:

$$f(x) = 9.998359xe^{-2.54405x}$$

From figure 9, we can see the fitted curve versus the data points after 255 iterations. In the next section, we'll be discussing the challenges that were encountered in this problem

2.3 Problems encountered

One of the biggest challenges in this problem are the data points itself. The points are placed all over the chart and therefore takes a while for the model to converge. What's more challenging is after around 20 iterations, the change in $||r||^2$ seems to approach zero. We can visualize it with a convergence plot.

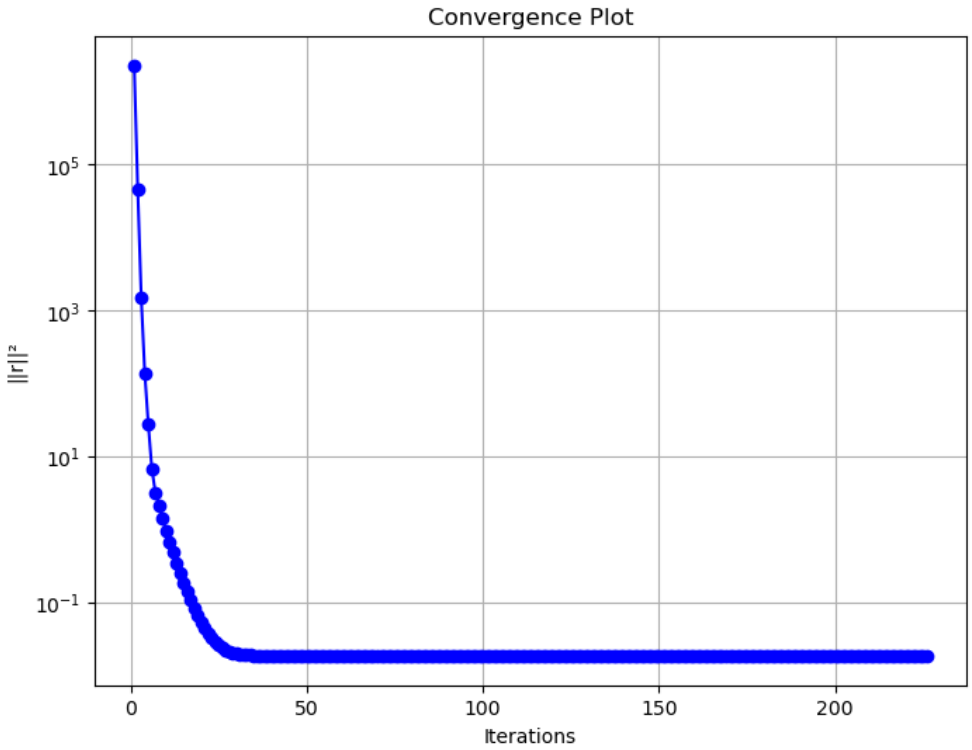

Figure 10: Convergence plot for 225 iterations

From figure 10, we can see the changes of $||r||^2$ per iteration. As we can see from the chart, there is a drastic change of $||r||^2$ on the first few iterations but ultimately reaches a “flat plateau” after 20 or so iterations.

The potential reasons for this are the limitations of the model or that we have hit the “local minima” and can’t get out. According to Mohit Mishra, a local minimum is a point in space where the loss function is the minimum in a local neighborhood. There are ways to avoid the local minima. We can initialize different values for α and β which I tried but still reaches the same $||r||^2$. I also tried to add a fixed learning rate, η, when updating the parameters but unfortunately the model still reaches the same $||r||^2$ which is 0.018471

2.4 References

- Mishra, M. (2023). *The Curse of Local Minima: How to Escape and Find the Global Minimum*. Retrieved from Medium: https://mohitmishra786687.medium.com/the-curse-of-local-minima-how-to-escape-and-find-the-global-minimum-fdabceb2cd6a

- Weisstein, Eric. *L^2-Norm*. (n.d.). Retrieved from Wolfram MathWorld: https://mathworld.wolfram.com/L2-Norm.html

PROBLEM 3 [30 points]: The error function erf(x) appears in many solutions of classical physical problems (for example, statistics, temperature distribution, diffusion, and viscous flow velocity distribution). The error function is given by

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt$$

Use three-point quadratures to estimate erf(0.5) using the trapezoidal rule, Simpson's rule and Gaussian quadrature. Compare your results using the three methods.

3.1 Method of solution

For this problem, we will be using three methods for obtaining the area under the curve. The first is the trapezoidal rule:

$$\int_{x_0=a}^{x_n=b} f(x)dx \approx \sum_{i=2}^{n} \frac{h}{2}[f(x_i) + f(x_{i+1})] \quad where\ h = \frac{b-a}{n} \qquad (11)$$

Then we have the Simpson's 3/8 rule:

$$\int_b^a f(x)dx \approx \frac{3h}{8}\left[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b)\right] \qquad (12)$$

Lastly, we have the Gaussian quadrature:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i \cdot f(x_i) \qquad (13)$$

The steps used to approximate the area under the curve of the error function are as follows:

1. Get the actual result of error function where x = 0.5, we can also denote this as err(0.5). Since error functions are non-elementary, this function cannot be integrable using normal integration methods. We will instead use a lookup table to get the output of err(0.5). We will be using the published lookup table from AGU publications. The associated output for err(0.5) is 0.520499877813
2. Initialize the limits of integration, a and b. In our case, a = 0 and b = 0.5.
3. Initialize n sub-intervals
4. We find the approximation using the Trapezoidal rule, Simpson's 3/8 rule, and Gaussian Quadrature.
5. We compare the approximated values with the exact value that we got from the lookup table and get the differences
6. We display the differences for these 3 methods in a bar graph for comparison

```
function GAUSSIAN_FUCNTION(t) outputs the result of e^2(-t^2)
        return e^2(-t^2)

function TRAPEZOIDAL_RULE(f, a, b, n)
        h = (b - a) / n
        integral_approx = 0
        initialize x and y values

        loop do
                integral_approx += h x (y_values[0] + 2*y_values[1] + y_values[2]) /
2

        PLOT_TRAPEZOIDAL_RULE
        return integral_approx
```

```
function SIMPSONS_3/8_RULE(f, a, b, n)
        h = (b - a) / n
        integral_approx = 0
        initialize x and y values
        loop do
                integral_approx += (h/8) x (y_values[0] + 3*y_values[1] +
3*y_values[2] + y_values[3])
        PLOT_SIMPSONS_RULE
        return integral_approx

function GAUSSIAN_3_POINT_QUADRATURE(f, a, b, n)
        h = (b - a) / n
        integral_approx = 0

        initialize weights and nodes
        loop do
                get x and y values for the specific sub interval
                integral_approx += 0.5 * h * sum(weights * y values)
        PLOT_GAUSSIAN_QUADRATURE
        return integral_approx

function error_function(a, b, n) outputs the results of the three methods
        result_t = TRAPEZOIDAL_RULE(f, a, b, n)
        result_s = SIMPSONS_3/8_RULE(f, a, b, n)
        result_g = GAUSSIAN_3_POINT_QUADRATURE(f, a, b, n)

print difference abs(exact value - result) # for all three values

PLOT_BAR_GRAPH_DIFFERENCES
```

Figure 11: Pseudocode for the three methods

3.2 Results

```
Absolute difference from exact value (Trapezoidal Rule): 0.520498962414096
7
Absolute difference from exact value (Simpsons's 3/8 Rule): 4.702904732312
163e-13
Absolute difference from exact value (Gaussian Quadrature): 4.651834473179
406e-14
```

Figure 12: Absolute differences from the three methods

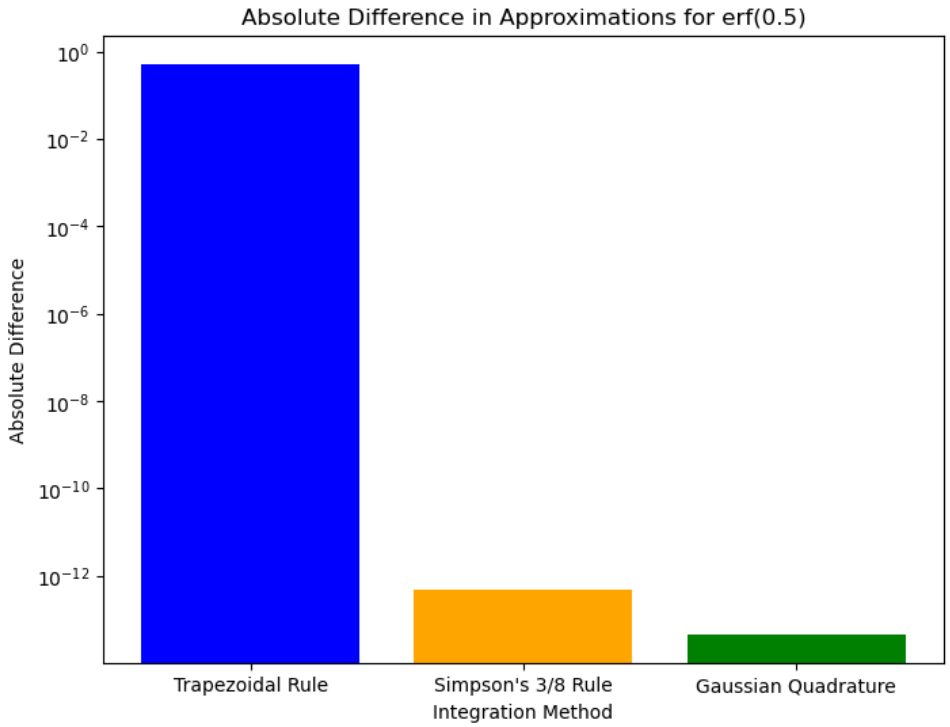

Figure 13: Absolute differences from the three methods

From figure 12, we can see the absolute difference from the three methods. We can see that Trapezoidal rule has the highest error with 0.5205. Simpson's 3/8 rule has significantly lower difference with 4.7029×10^{-13} and Gaussian Quadrature with the lowest difference of 4.6518×10^{-14}

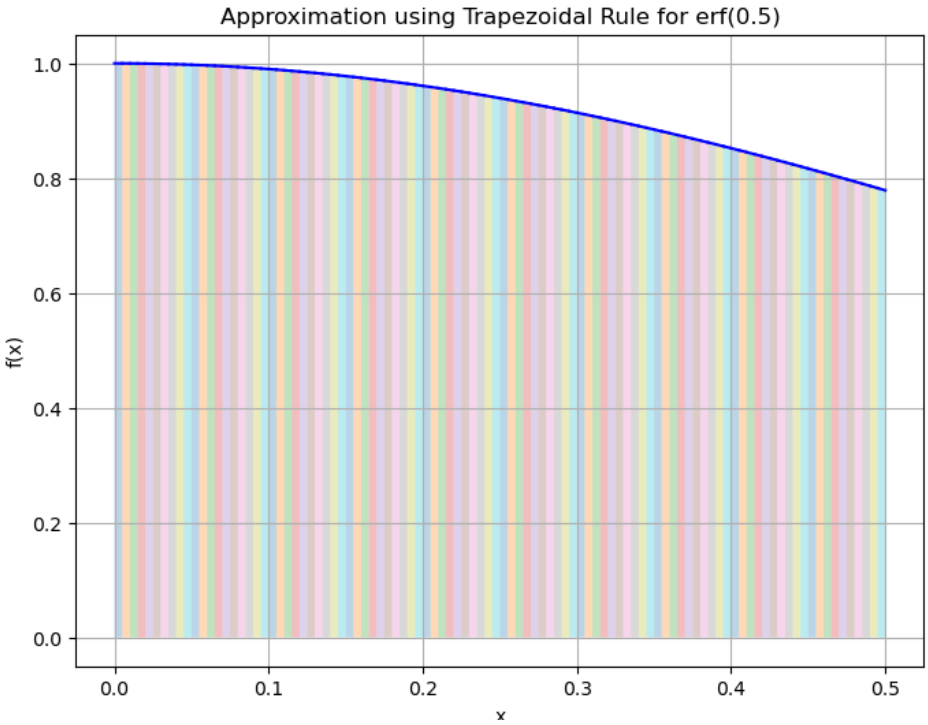

Figure 14: Approximation using Trapezoidal Rule

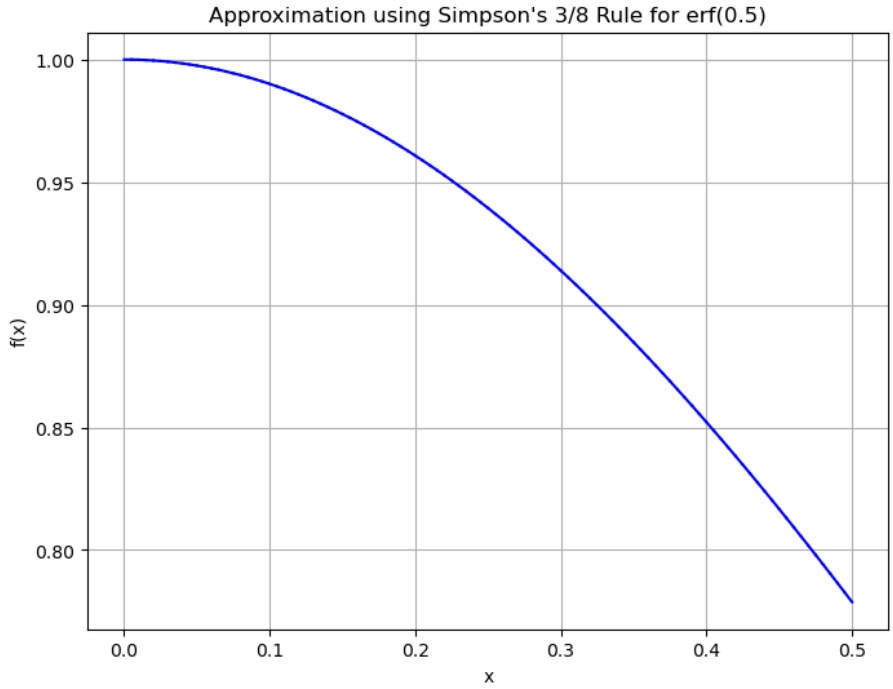

Figure 15: Approximation using Simpson's 3/8 Rule

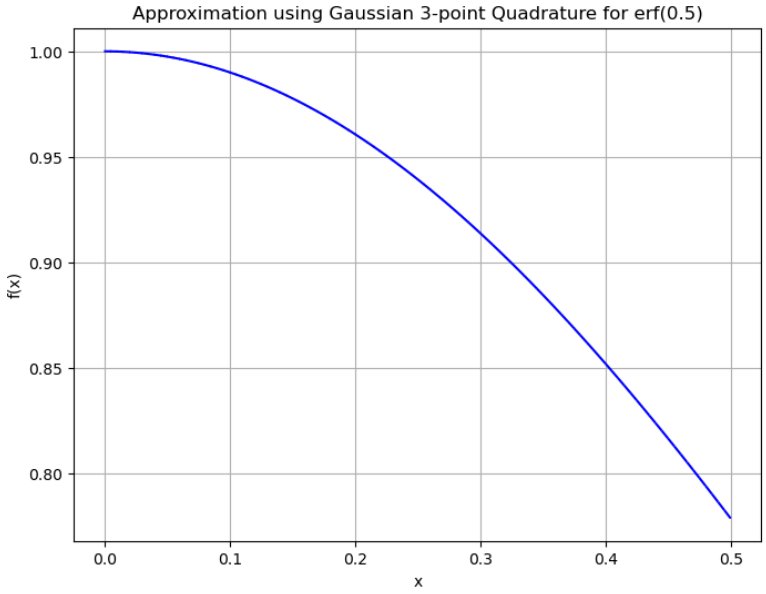

Figure 16: Approximation using Gaussian Quadrature

3.3 Problems encountered

The primary problem encountered was lowering the difference for the Trapezoidal rule. I have used the composite Trapezoidal rule from the given lecture slides. Perhaps I could've used the Trapezoidal rule with end correction to get better results since I can get the derivatives and integrands of the endpoints which is 0 and 0.5.

3.4 References

- Javandel, I., Doughty, C., & Tsang, C. (1984). *Groundwater Transport: Handbook of Mathematical Models. Water Resources Monograph*. doi:10.1029/wm010. Retrieved from: Tables of Error Function - 1984 - Water Resources Monograph - Wiley Online Library

PROBLEM 4 [30 points]: Evaluate

$$\int_0^3 \frac{\sin x}{\sqrt{1+x^2}} dx$$

Use an **adaptive version** of Newton-Cotes. Set tolerance to 1×10^{-5}.

4.1 Method of solution

For this problem, we'll be using the Adaptive quadrature. Consider the Simpson's rule with two sub-intervals:

$$S(a,b) = \int_a^b f(x)dx = \frac{h}{3}[f(a) + 4f(c) + f(b) \quad where\ h = \frac{b-a}{2} \tag{14}$$

Applying the error estimate, we will get:

$$\int_a^b f(x)dx = S(a,b) - \frac{16}{15}(S(a_1,b_1) + S(a_2,b_2) - S(a,b)) \quad where\ c = \frac{a+b}{2} \tag{15}$$

The steps used to approximate the area under the curve of the given function are as follows:

1. Define the function to be integrated
2. We initialize parameters, a and b which are the limits of integration for our function
3. We define the tolerance limit
4. Using the adaptive quadrature function, we perform the adaptive quadrature on the provided function within the interval [a, b] with a specified tolerance, tol.
5. We Implements Simpson's rule for numerical integration of the function within the given interval [a, b]

6. Using the plot curve function, we plot the function, dividing into sub-intervals. until the error estimate meets the specified tolerance. It fills the area under the curve using the Simpson's rule for subintervals.
7. We performs the adaptive integration using recursive subdivision of intervals until the error estimate is below the specified tolerance.
8. We print out the approximate integral value obtained using the said method

```
function ADAPTIVE-QUADRATURE(func, a, b, tol)
    function PLOT_CURVE(f, a, b, total_area)
        intervals = [(a, b)]
        while intervals:
            approx_full = SIMPSONS-RULE(func, left, right)
            approx_split = SIMPSONS-RULE(func, left, mid) + SIMPSONS-
RULE(func, mid, right)
            error_estimate = 16 / 15 * abs(approx_full - approx_split)

            if error_estimate > tol:
                intervals.extend([(left, mid), (mid, right)])
            else:
                generate x and y values within interval [left, right]
                plot function curve with x and y values
                shade area under the curve for the interval [left, right]

        display legend, labels, title, and grid for the plot

    function SIMPSONS-RULE(f, a, b)
        return (b - a) / 6 * (f(a) + 4 * f((a + b) / 2) + f(b))

    function ADAPTIVE-INTEGRATION(f, a, b, tol)
        c = (a + b) / 2
```

```
        left = simpsons_rule(f, a, c)
        right = simpsons_rule(f, c, b)
        approx = left + right

        if abs(approx - SIMPSONS-RULE(f, a, b)) <= 15 * tol:
            return approx + (approx - SIMPSONS-RULE(f, a, b)) / 15

        return ADAPTIVE-INTEGRATION(f, a, c, tol / 2) + ADAPTIVE-
INTEGRATION(f, c, b, tol / 2)

    PLOT-CURVE(func, a, b, ADAPTIVE-INTEGRATION(func, a, b, tol))
    return ADAPTIVE-INTEGRATION(func, a, b, tol)

# Define the function, interval, and tolerance
function FUNC(x)
    return function

Initialize parameters and tolerance

Result = ADAPTIVE-QUADRATURE(func, a, b, tol)
PRINT_RESULT
```

Figure 17: Adaptive Quadrature Pseudocode

4.2 Results

```
Approximated integral using adaptive quadrature: 1.1422181166497736
```

Figure 18: Adaptive Quadrature Result

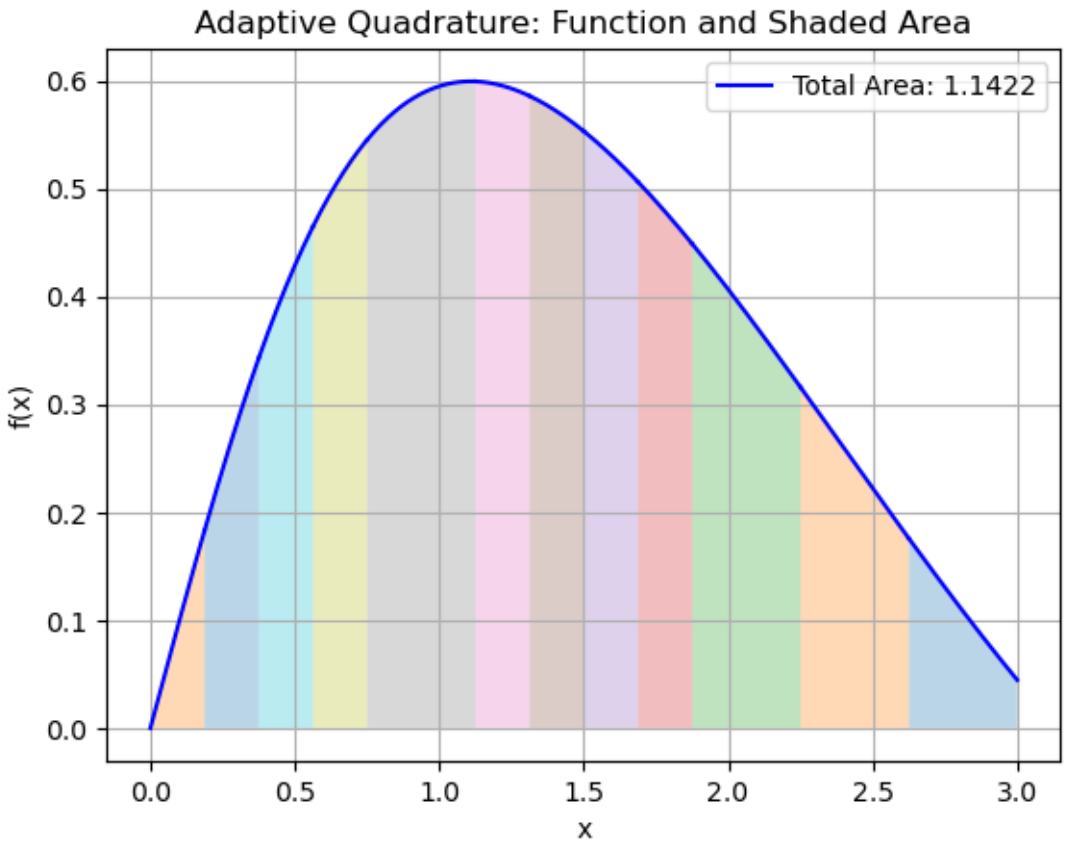

Figure 19: Adaptive Quadrature Approximation under the Curve

As we can see from figure 18, the approximated result is 1.1422. When we use wolfram to solve for the given function, it also outputs the same value. Figure 19 shows the visual representation of the function as well as the shaded area under the curve with limits of integration [0, 3].

4.3 Problems encountered

I have encountered an issue with getting the wrong approximation of the values. At first, I always get a value of 1.2422. I double checked every function like the Simpson's rule, Adaptive integration functions to make sure they're correct. Turns out the error was on the function itself. I got the formula wrong for np.sin(x) / np.sqrt(1 + x**2). The exponent 2 was placed on the outside.

4.4 References

- N.A. (n.d.). *Wolfram Alpha*. Retrieved from: Integrate[Divide[Sin[x],Sqrt[1+Power[x,2]]],{x,0,3}] - Wolfram|Alpha (wolframalpha.com)

APPENDIX

PROBLEM 1

SOURCE CODE:

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

def lagrange_method_coefficient(x_var, x):
    # Method for obtaining the coefficients
    N = len(x)
    polynomials = []

    for i in range(N):
        n, d = 1, 1
        xi = x[i]
        for j in range(N):
            if i != j:
                xj = x[j]
                n *= (x_var - xj)
                d *= (xi - xj)
        polynomials.append(n/d)

    return polynomials

def lagrange_method_interpolation(x_var, x, fx):
    # Function for getting the interpolating polynomial
    coeff = lagrange_method_coefficient(x_var, x)
    interp_poly = 0
    for i in range(len(fx)):
        interp_poly += fx[i] * coeff[i]
    return interp_poly

def main():

    # Input for x, y dataset
    num_points = int(input("Enter the number of data points: "))
    x_data = []
    y_data = []
    for i in range(num_points):
        x = float(input(f"Enter x{i+1}: "))
        y = float(input(f"Enter y{i+1}: "))
```

```
    x_data.append(x)
    y_data.append(y)

x_values = x_data
y_values = y_data

# Symbolic variable
x_var = sp.symbols('x')

poly = lagrange_method_interpolation(x_var, x_data, y_data)

poly = sp.simplify(poly)
poly = sp.expand(poly)
coeffs = poly.as_poly().coeffs()
coeffs = np.array(coeffs)

print("\nInterpolating polynomial: ")
print("f(x) = ", poly)

x_val = float(input("\nPlease enter value for x: "))
output = poly.subs(x_var, x_val)
print(f"f({x_val}) = ", output)

# Fit a cubic polynomial to the given data
degree = 3  # Since this is cubic
coeffs_cubic = np.polyfit(x_values, y_values, degree)

# Create the polynomial curve using the coefficients of the cubic polynomial
poly_curve_cubic = np.poly1d(coeffs_cubic)
plt.figure(figsize=(8, 6))
plt.scatter(x_values, y_values, label='Data Points')

# Determine the range for plotting the curve, including the added point
if x_val > max(x_values):
    max_x = x_val
else:
    max_x = max(x_values)

if x_val < min(x_values):
    min_x = x_val
else:
    min_x = min(x_values)

# Plotting the curve for the cubic polynomial
x_curve = np.linspace(min_x, max_x, 100)  # X values for the curve
y_curve = poly_curve_cubic(x_curve)  # Y values for the curve
```

```
    plt.plot(x_curve, y_curve, label='Cubic Polynomial Curve', color='green')

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Cubic Polynomial Fit')
    plt.scatter(x_val, output, color='red', label=f'Point ({x_val}, {output})')
    plt.annotate(f'({x_val}, {output})', (x_val, output), textcoords="offset points", xytext=(10,10),
ha='center')

    plt.legend()
    plt.grid(True)
    plt.show()

    return None

if __name__ == "__main__":
    main()



def Equation_2_right(x0, x1, x2, y0, y1, y2):
    return (6/(x2 - x1)*(y2 - y1)) + (6/(x1 - x0))*(y0 - y1)

def Equation_2_left(i, x, f_2, x0, x1, x2):
    if i == 0:
        f_2_1 = f_2.subs(x, x1)
    else:
        f_2_1 = f_2.subs(x, x0)

    if i == 0:
        f_2_2 = f_2.subs(x, x2)
    else:
        f_2_2 = f_2.subs(x, x1)

    if i == 0:
        return x1, x2, f_2_1, f_2_2, 2*(x2 - x0)*f_2_1 + (x2 - x1)*f_2_2
    else:
        return x0, x1, f_2_1, f_2_2, (x1 - x0)*f_2_1 + 2*(x2 - x0)*f_2_2

def Equation_1(i, x, x0, x1, y0, y1, sol_a, sol_b):
        a = ((sol_a/(6*(x1 - x0)))*((x1 - x)**3))
        b = ((sol_b/(6*(x1 - x0)))*((x - x0)**3))
        c = ((y0/(x1 - x0))-((sol_a*(x1 - x0))/6))
        c = c*(x1 - x)
        d = ((y1/(x1 - x0))-((sol_b*(x1 - x0))/6))
        d = d*(x - x0)
```

```
        return a + b + c + d

def cubic_spline_interpolation(x, f_1, f_2, f_3, x_data, y_data):
    equations = []
    for i in range(2):
        x0, x1, x2 = x_data[i], x_data[i+1], x_data[i+2]
        y0, y1, y2 = y_data[i], y_data[i+1], y_data[i+2]
        a, b, f_2_1, f_2_2, Left = Equation_2_left(i, x, f_2, x0, x1, x2)
        equation = Left - Equation_2_right(x0, x1, x2, y0, y1, y2)
        equations.append(equation)
        print(f"equation {i+1}:", equation)

    solution = sp.solve(equations, (f_2_1, f_2_2))
    sol_a = solution.get(sp.Function("f''")(a))
    sol_b = solution.get(sp.Function("f''")(b))

    poly = []
    for i in range(3):
        x0, x1 = x_data[i], x_data[i+1]
        y0, y1 = y_data[i], y_data[i+1]
        if i == 0:
            a, b = 0, sol_a
        elif i == 1:
            a, b = sol_a, sol_b
        elif i == 2:
            a, b = sol_b, 0
        cubic_poly = Equation_1(i, x, x0, x1, y0, y1, a, b)
        poly.append(cubic_poly)

    print("First interval cubic: ", poly[0])
    print("Second interval cubic: ", poly[1])
    print("Third interval cubic: ", poly[2])

    print("\nInputting x = 2.75")
    x_val = 2.75
    output = poly[1].subs(x, x_val)
    print(f"f({x_val}) = ", output)

    # Convert SymPy expressions to lambdified functions
    equation1_func = sp.lambdify(x, poly[0], modules=['numpy'])
    equation2_func = sp.lambdify(x, poly[1], modules=['numpy'])
    equation3_func = sp.lambdify(x, poly[2], modules=['numpy'])


    x_values1 = np.linspace(0, 2, 100)
    x_values2 = np.linspace(2, 3, 100)
```

```
    x_values3 = np.linspace(3, 4, 100)


    y_values1 = equation1_func(x_values1)
    y_values2 = equation2_func(x_values2)
    y_values3 = equation3_func(x_values3)

    # Plotting
    plt.figure(figsize=(8, 6))

    plt.plot(x_values1, y_values1, label='First interval cubic')
    plt.plot(x_values2, y_values2, label='Second interval cubic')
    plt.plot(x_values3, y_values3, label='Third interval cubic')

    # Adding point and label
    plt.scatter(x_val, output, color='red', label=f'Point: ({x_val}, {output})')
    plt.annotate(f'({x_val}, {output})', (x_val, output), textcoords="offset points", xytext=(-15,-15),
ha='center')

    plt.xlabel('x-axis')
    plt.ylabel('y-axis')
    plt.title('Plot of Cubic Spline Equations')
    plt.legend()
    plt.grid(True)
    plt.show()

def main():

    x_data = [1, 2, 3, 4]
    y_data = [1, 2, 0, 3]

    # Symbolic variable
    x = sp.symbols('x')
    f_1 = sp.Function('f')(x)
    f_2 = sp.Function("f''")(x)
    f_3 = sp.Function("f'''")(x)

    poly = cubic_spline_interpolation(x, f_1, f_2, f_3, x_data, y_data)

    return None

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

SOLUTION:

$$I(x) = \sum_{i=0}^{n} y_i \cdot L_i(x) \tag{1}$$

$$L_i = \prod_{j=0(j\neq i)}^{n} \left(\frac{x - x_j}{x_i - x_j}\right) \quad for \;\; 0 \leq i \leq n \tag{2}$$

$$f_i(x) = \frac{f''_i(x_{i-1})}{6(x_i - x_{i-1})}(x_i - x)^3 + \frac{f''_i(x_i)}{6(x_i - x_{i-1})}(x - x_{i-1})^3 + \left[\frac{f(x_{i-1})}{x_i - x_{i-1}} - \frac{f''_i(x_{i-1})(x_i - x_{i-1})}{6}\right](x_i - x) + \left[\frac{f(x_i)}{x_i - x_{i-1}} - \frac{f''_i(x_i)(x_i - x_{i-1})}{6}\right](x - x_{i-1}) \tag{3}$$

$$(x - x_{i-1})f''(x_i - 1) + 2(x_{i+1} - x_{i+1})f''(x_i) + (x_{i+1} - x_{i+1})f''(x_{i+1}) = \frac{6}{x_{i+1} - x_i}[f(x_{i+1}) - f(x_i)] + \frac{6}{x_{i+1} - x_i}[f(x_{i-1}) - f(x_i)] \tag{4}$$

PROBLEM 2

SOURCE CODE:

```
def Residual(x_values, y_values, a, b):
    N = len(x_values)
    r_stack = []
    for i in range(N):
        r_i = (a*x_values[i]*np.exp(b*x_values[i])) - y_values[i]
        r_stack.append(r_i)

    r = (np.array(r_stack)).T
    r_n = np.linalg.norm(r)
    r_2 = np.linalg.norm(r)**2
    return r, r_2, r_n

def Jacobian(x_values, a, b):
    N = len(x_values)
    J = np.ones((N, 2))
    for i in range(N):
        J[i,0] = np.exp(x_values[i]*b)
        J[i,1] = (a*x_values[i]*np.exp(x_values[i]*b))*x_values[i]
    return J

def Non_linear_least_squares(x_var, x_values, y_values, a, b, N, tol, l_r = 0.99):
```

```
k = 0
iter = []
error = []
prev_r_n = 999999
for i in range(N):
    r, r_2, r_n = Residual(x_values, y_values, a, b)
    J = Jacobian(x_values, a, b)
    Jr = np.dot(J.T, r)
    Jt = np.dot(J.T, J)
    iter.append(i+1)
    error.append(r_2)

    p = np.matmul(np.linalg.inv(Jt),(-Jr.reshape(2, 1)))
    a += l_r * p[0,0]
    b += l_r * p[1,0]
    print(f"At k = {k}, r_2 = ", r_2)

    diff = abs(prev_r_n - r_n)

    if diff < tol:
        print("\nModel obtained after fitting:")
        print(f"a = {a}, b = {b}")

        # Construct the model equation
        model_equation = f"{a}*exp({b}*x)"
        print("\nModel equation:")
        print(f"y = {model_equation}")

        # Symbolic representation of the model equation
        symbolic_model = a * sp.exp(b * x_var)
        print("\nSymbolic representation:")
        print(f"y = {symbolic_model}")
        print(f"\nAfter {k+1} iterations:")
        print("r_2 = ", r_2)

        break
    if k == N - 1:
        print("\nThe model was not able to converge")
        print("r_2 = ", r_2)
    prev_r_n = r_n
    plot_title = f"Fitted Curve vs. Data for Iter {k+1}"
    plot_fitness_curve(x_values, y_values, a, b, plot_title)
    k += 1

plot_error_curve(iter, error)
```

```
    return r_2, k

def plot_fitness_curve(x_values, y_values, a, b, title):
    x_vals = np.linspace(min(x_values), max(x_values), 100)
    y_vals = a * x_vals* np.exp(b * x_vals)

    # Plotting
    plt.figure(figsize=(8, 6))
    plt.scatter(x_values, y_values, label='Data Points')
    plt.plot(x_vals, y_vals, label='Fitted Curve', color='red')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.title(title)
    plt.grid(True)
    plt.show()

    return None

def plot_error_curve(iterations, error_values):
    plt.figure(figsize=(8, 6))
    plt.plot(iterations, error_values, marker='o', linestyle='-', color='b')
    plt.xlabel('Iterations')
    plt.ylabel('||r||²')
    plt.title('Convergence Plot')
    plt.grid(True)
    plt.yscale('log')
    plt.show()

max_iter = 1000
tol = 1e-15
a = 2.99
b = 3

x_values = [0.1, 0.2, 0.4, 0.6, 0.9, 1.3, 1.5, 1.7, 1.8]
y_values = [0.75, 1.25, 1.45, 1.25, 0.85, 0.55, 0.35, 0.28, 0.18]

# Symbolic variable
x_var = sp.symbols('x')

r_2, k = Non_linear_least_squares(x_var, x_values, y_values, a, b, max_iter, tol)
```

SOLUTION:

$$r_i = \alpha x_i e^{\beta x_i} - y_i$$

$$||r||^2 = \sqrt{\sum_{i=1}^{n} |r_i|^2} \quad (6)$$

$$J(x) = \left(\frac{\partial r_i}{\partial x_j}\right) = \begin{pmatrix} \nabla r_1(x)^T \\ \vdots \\ \nabla r_n(x)^T \end{pmatrix} \quad (7)$$

$$p^k = ((J^T)^K \cdot J^K)^{-1}(-(J^T)^K \cdot r^k \quad (8)$$

$$\alpha = \alpha + p_1 \quad (9)$$

$$\beta = \beta + p_2 \quad (10)$$

PROBLEM 3

SOURCE CODE:

```
# Function to integrate (Gaussian function)
def gaussian_func(t):
    return np.exp(-t**2)  # Gaussian function e^(-t^2)

def trapezoidal_rule_3_point(f, a, b, n):
    delta_x = (b - a) / n  # Calculate the step size for subintervals
    integral_approx = 0

    plt.figure(figsize=(8, 6))
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title('Approximation using Trapezoidal Rule for erf(0.5)')
    plt.grid(True)

    # Iterate through subintervals
    for i in range(n):
```

```
        x_values = np.linspace(a + i * delta_x, a + (i+1) * delta_x, 3)  # 3 points within each
subinterval
        y_values = f(x_values)  # Calculate function values at the 3 points

        # Plot the function and trapezoidal approximation for this subinterval
        plt.plot(x_values, y_values, 'b', label='Function')
        plt.fill_between(x_values, y_values, alpha=0.3)

        integral_approx += delta_x * (y_values[0] + 2 * y_values[1] + y_values[2]) / 2  # Apply 3-point
Trapezoidal Rule formula

    plt.show()

    return integral_approx


def simpsons_3_point_rule(f, a, b, n):
    delta_x = (b - a) / n
    integral_approx = 0

    plt.figure(figsize=(8, 6))
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title("Approximation using Simpson's 3/8 Rule for erf(0.5)")
    plt.grid(True)

    # Iterate through subintervals
    for i in range(n):
        x_values = np.linspace(a + i * delta_x, a + (i+1) * delta_x, 4)  # 3 points within each
subinterval
        y_values = f(x_values)

        plt.plot(x_values, y_values, label='f(x) = erf(x)', color='blue')

        integral_approx += (delta_x / 8) * (y_values[0] + 3*y_values[1] + 3*y_values[2] + y_values[3])  #
Apply 3-point Simpson's 3/8 Rule formula

    plt.show()

    return integral_approx


def gaussian_3_point_quadruture(f, a, b, n):

    delta_x = (b - a) / n
    integral_approx = 0
```

```
    weights = [5/9, 8/9, 5/9]
    nodes = [-np.sqrt(15)/5, 0, np.sqrt(15)/5]

    plt.figure(figsize=(8, 6))
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title("Approximation using Gaussian 3-point Quadrature for erf(0.5)")
    plt.grid(True)

    # Iterate through subintervals
    for i in range(n):

        nodes = np.array([-np.sqrt(3/5), 0, np.sqrt(3/5)])
        weights = np.array([5/9, 8/9, 5/9])

        x_values = a + (i + 0.5) * delta_x + 0.5 * delta_x * nodes  # Calculate x_values for this
subinterval
        y_values = f(x_values)  # Calculate function values at the nodes

        plt.plot(x_values, y_values, label='f(x) = erf(x)', color='blue')

        integral_approx += 0.5 * delta_x * np.sum(weights * y_values)

    plt.show()

    return integral_approx

def error_functions(a, b, n):
    result_t = trapezoidal_rule_3_point(gaussian_func, a, b, n)
    result_s = simpsons_3_point_rule(gaussian_func, a, b, n)
    result_g = gaussian_3_point_quadruture(gaussian_func, a, b, n)
    return 2 / np.sqrt(np.pi) * result_t, 2 / np.sqrt(np.pi) * result_s, 2 / np.sqrt(np.pi) * result_g

exact_value = 0.520499877813

a, b = 0, 0.5
n = 100
result_t, result_s, result_g = error_functions(a, b, n)

t_diff = abs(exact_value - result_t)
s_diff = abs(exact_value - result_s)
g_diff = abs(exact_value - result_g)

# Output the differences
print(f"Absolute difference from exact value (Trapezoidal Rule): {t_diff}")
```

```
print(f"Absolute difference from exact value (Simpsons's 3/8 Rule): {s_diff}")
print(f"Absolute difference from exact value (Gaussian Quadrature): {g_diff}")

methods = ['Trapezoidal Rule', 'Simpson\'s 3/8 Rule', 'Gaussian Quadrature']
diffs = [t_diff, s_diff, g_diff]

plt.figure(figsize=(8, 6))
plt.bar(methods, diffs, color=['blue', 'orange', 'green'])
plt.xlabel('Integration Method')
plt.ylabel('Absolute Difference')
plt.title('Absolute Difference in Approximations for erf(0.5)')
plt.axhline(0, color='grey', linewidth=0.8, linestyle='--')  # Add a horizontal line at y=0
plt.yscale('log')
plt.show()
```

SOLUTION:

$$\int_{x_0=a}^{x_n=b} f(x)dx \approx \sum_{i=2}^{n} \frac{h}{2}[f(x_i) + f(x_{i+1})] \quad where\ h = \frac{b-a}{n} \tag{11}$$

$$\int_{b}^{a} f(x)dx \approx \frac{3h}{8}\left[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b)\right] \tag{12}$$

$$\int_{a}^{b} f(x)dx \approx \sum_{i=1}^{n} w_i \cdot f(x_i) \tag{13}$$

PROBLEM 4

SOURCE CODE:

```
def func(x):
    return (np.sin(x)/ (1 + x**2)**0.5)

def adaptive_quadrature(func, a, b, to):
    def plot_curve(f, a, b, total_area):

        # Initialize list to store intervals
        intervals = [(a, b)]
```

```
        while intervals:
            left, right = intervals.pop()
            mid = (left + right) / 2
            approx_full = simpsons_rule(func, left, right)
            approx_split = simpsons_rule(func, left, mid) + simpsons_rule(func, mid, right)

            if abs(approx_full - approx_split) > tolerance:
                intervals.extend([(left, mid), (mid, right)])
            else:
                x = np.linspace(left, right, 1000)
                y = func(x)
                plt.plot(x, y, color='blue')
                plt.fill_between(x, 0, y, where=((x >= left) & (x <= right)), alpha=0.3)


        plt.legend([f'Total Area: {total_area:.4f}'], loc='upper right')
        plt.xlabel('x')
        plt.ylabel('f(x)')
        plt.title('Adaptive Quadrature: Function and Shaded Area')
        plt.grid(True)
        plt.show()

    def simpsons_rule(f, a, b):
        return (b - a) / 6 * (f(a) + 4 * f((a + b) / 2) + f(b))

    def adaptive_integration(f, a, b, tol, whole):
        c = (a + b) / 2
        left = simpsons_rule(f, a, c)
        right = simpsons_rule(f, c, b)
        approx = left + right

        if abs(approx - whole) <= 15 * tol:
            return approx + (approx - whole) / 15

        return adaptive_integration(f, a, c, tol / 2, left) + adaptive_integration(f, c, b, tol / 2,
right)

    plot_curve(func, a, b, adaptive_integration(func, a, b, tol, simpsons_rule(func, a, b)))
    return adaptive_integration(func, a, b, tol, simpsons_rule(func, a, b))

# Define the interval and tolerance
a = 0
b = 3
tolerance = 1e-6

result = adaptive_quadrature(func, a, b, tolerance)
```

```
print("Approximated integral using adaptive quadrature:", result)
```

SOLUTION:

$$S(a,b) = \int_a^b f(x)dx = \frac{h}{3}[f(a) + 4f(c) + f(b) \quad where\ h = \frac{b-a}{2} \tag{14}$$

$$\int_a^b f(x)dx = S(a,b) - \frac{16}{15}(S(a_1,b_1) + S(a_2,b_2) - S(a,b)) \quad where\ c = \frac{a+b}{2} \tag{15}$$