Universidad Ana G. Mendez Recinto de Gurabo

School of Engineering

Gurabo, Puerto Rico

Numerical Methods – Final Project

Carlos A. Roque Fontanez

S01299757

COMP-411

Dr. Yahya Masalmah

December 13th, 2023

Table of Contents

Introduction

The final project for the course Numerical Methods with Programming (COMP 411) involves a series of complex and practical problems that students are required to solve by applying numerical methods and programming skills. The problems are diverse, each focusing on different engineering principles and requiring unique solutions:

Problem 1 involves precise data measurements, where students must handle and process data accurately to achieve reliable results. The problem statement and its specific requirements were not visible in the document preview.

Problem 2 explores the sea-level concentration of dissolved oxygen in freshwater as a function of temperature. Here, students are expected to utilize their knowledge of environmental engineering and chemistry to model and analyze the effects of temperature on oxygen levels, which is critical for aquatic life sustainability.

Problem 3 is based on a simplified Arrhenius equation, commonly used in environmental engineering. It requires students to parameterize the effect of temperature on pollutant decay rates, a key factor in environmental impact assessments and the development of pollution mitigation strategies.

Problem 4 involves numerical integration techniques to model the force on a sailboat mast. Students must represent the force as a function and integrate it to determine the total force exerted on the mast, simulating real-world engineering challenges faced in the design and analysis of marine structures.

Problem 5 deals with the application of Newton's law of cooling to estimate the time of death in a forensic context. It exemplifies how engineering principles can extend into other fields such as forensic science. Students must determine the constant of proportionality and solve a related ordinary differential equation (ODE) numerically, demonstrating an understanding of heat transfer principles and their implications in real-life scenarios.

Problem 1

The following data are measured precisely:

T	2	2.1	2.2	2.7	3	3.4
Z	6	7.752	10.256	36.576	66	125.168

a) Use Newton interpolating polynomials to determine z at t = 2.5. Make sure that you order your points to attain the most accurate results. What do your results tell you regarding the order of the polynomials used to generate the data?
b) Use a third-order Lagrange interpolating polynomial to determine y at 2.5.

Using Newton Interpolating Polynomial

Given our points T and Z, we start computing the first order divided difference:

Formula

$$f(x_i, x_j) = \frac{f(x_i) - f(x_j)}{x_i - x_j}$$

We now construct the Divided Difference table using this formula, starting with the First Order.

For T = 2, to T = 2.1:

$$\frac{7.752 - 6}{2.1 - 2} = \frac{1.752}{0.1} \approx 17.52$$

For T = 2.1 to T = 2.2:

$$\frac{10.256 - 7.752}{2.2 - 2.1} = \frac{2.504}{0.1} \approx 25.04$$

For T = 2.2 to T = 2.7:

$$\frac{36.576 - 10.256}{2.7 - 2.2} = \frac{26.32}{0.5} \approx 52.64$$

For T = 2.7 to T = 3:

$$\frac{66 - 36.576}{3 - 2.7} = \frac{29.424}{0.3} \approx 98.08$$

For T = 3 to T = 3.4:

$$\frac{125.168 - 66}{3.4 - 3} = \frac{59.168}{0.4} \approx 147.92$$

For the second order, we use the values gotten when finding the first order divided difference.

For T = 2 to T = 2.2:

$$\frac{25.04 - 17.52}{2.2 - 2} = \frac{7.52}{0.2} \approx 37.6$$

For T = 2.1 to T = 2.7:

$$\frac{52.64 - 25.04}{2.7 - 2.1} = \frac{27.6}{0.6} \approx 46$$

For T = 2.2 to T = 3:

$$\frac{98.08 - 52.64}{3 - 2.2} = \frac{45.44}{0.8} \approx 56.8$$

For T = 2.7 to T = 3.4:

$$\frac{147.92 - 98.08}{3.4 - 2.7} = \frac{49.84}{0.7} \approx 71.2$$

Now, we continue with the third order using the computer values from the second order.

For T = 2, T = 2.2 and T = 2.7 using deltas from second order: T = 2.1 and T = 2:

$$\frac{46 - 37.6}{2.7 - 2} = \frac{8.4}{0.7} = 12$$

For T = 2.1, T = 2.7 and T = 3 using deltas from second order: T = 2.2 and T = 2.1:

$$\frac{56.8 - 46}{3 - 2.1} = \frac{10.8}{0.9} = 12$$

For T = 2.2, T = 3 and T = 3.4 using deltas from second order: T = 2.7 and T = 2.2:

$$\frac{71.2 - 56.8}{3.4 - 2.2} = \frac{14.4}{1.2} = 12$$

Finally, we move to the fourth order, we again use the values from the previous order.

For T = 2, T = 2.2, T = 2.7 and T = 3, using deltas from third order: T = 2.1 and T = 2:

$$\frac{12 - 12}{3 - 2} = \frac{0}{1} = 0$$

For T = 2.1, T = 2.7, T = 3 and T = 3.4 using deltas from third order: T = 2.2 and T = 2.1:

$$\frac{12 - 12}{3.4 - 2.1} = \frac{0}{1.3} = 0$$

Divided Differences Table

We can now plug these values into the table like so:

T	Z	Second Order (b_2)	Third Order (b_3)	Fourth Order (b_4)	Fifth Order (b_5)
2	**6**	**17.52**	**37.6**	**12**	**0**
2.1	**7.752**	**25.04**	**46**	**12**	**0**
2.2	**10.256**	**52.64**	**56.8**	**12**	
2.7	**36.576**	**98.08**	**71.2**		
3	**66**	**147.92**			
3.4	**125.168**				

The Polynomial

Now we attempt to construct the Newton Interpolating Polynomial using the following formula:

$$P(t) = b_0 + b_1(x - x_1) + b_3(x - x_1)(x - x_2) + \cdots + b_n(x - x_1)(x - x_2) \dots (x - x_{n-1})$$

Where:

$P(t)$ – is the value of the polynomial at time t

b_n – are the coefficients from the divided differences table

t_n- are the T values of our data points

We gather our coefficients:

$b_0 = 6$

$b_1 = 17.52$

$b_2 = 37.6$

$b_3 = 12$

We stop on the third order since our T = 2.5 falls just above it.

Now we substitute the values:

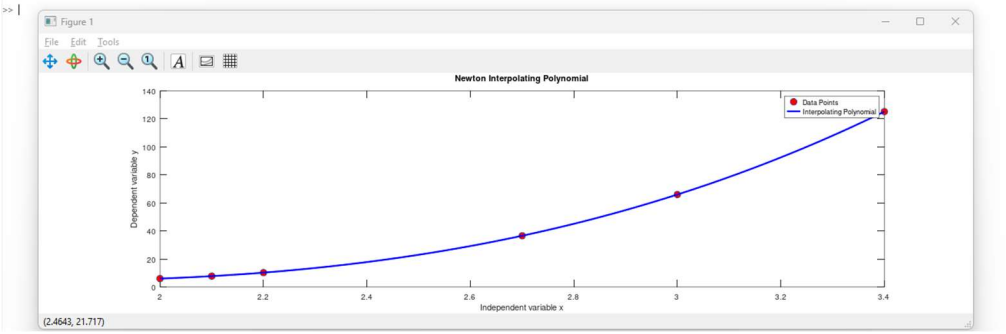
$$P(t) = 6 + 17.52(x - 2) + 37.6(x - 2)(x - 2.1) + 12(x - 2)(x - 2.1)(x - 2.2)$$

Solve for $t = 2.5$:

$$P(2.5) = 6 + 17.52(2.5 - 2) + 37.6(2.5 - 2)(2.5 - 2.1) + 12(2.5 - 2)(2.5 - 2.1)(2.5 - 2.2) = 23$$

By running the code in Listing [1], we get the following results:

```
>> T = [2 2.1 2.2 2.7 3 3.4]
T =

   2.0000   2.1000   2.2000   2.7000   3.0000   3.4000

>> Z = [6 7.752 10.256 36.576 66 125.168]
Z =

    6.0000    7.7520   10.2560   36.5760   66.0000  125.1680

>> xx = 0:0.1:4; yy = exp(xx);
>> [b, yint] = Newtint2(T, Z, xx)
b =

    6.0000   17.5200   37.6000   12.0000   -0.0000   -0.0000
    7.7520   25.0400   46.0000   12.0000   -0.0000         0
   10.2560   52.6400   56.8000   12.0000         0         0
   36.5760   98.0800   71.2000         0         0         0
   66.0000  147.9200         0         0         0         0
  125.1680         0         0         0         0         0

yint =

 Columns 1 through 21:

   18.0000   19.8320   20.9760   21.5040   21.4880   21.0000   20.1120   18.8960   17.4240   15.7680   14.0000   12.1520   10.4160    8.7440    7.2480    6.0000    5.0720    4.5360    4.4640    4.9280    6.0000

 Columns 22 through 41:

    7.7520   10.2560   13.5840   17.8080   23.0000   29.2320   36.5760   45.1040   54.8880   66.0000   78.5120   92.4960  108.0240  125.1680  144.0000  164.5920  187.0160  211.3440  237.6480  266.0000

>>
```



The consistency of twelves in the 3rd order and the presence of Zero's in the fourth order suggests that the data was likely generated by a third-order polynomial (cubic polynomial). In Newton's method, if the *n*-th order divided differences are zero, it implies that the polynomial of best fit is of order $n - 1$. Since the fourth order divided differences are zero, the polynomial used to generate the data is of order 3, i.e., cubic polynomial.

Using Lagrange Interpolating Polynomial

To solve the problem using third-order Lagrange interpolating polynomial, we use the data points given.

General Form

The general form of the Lagrange interpolating polynomial is given as:

$$\mathcal{L}(t) = \sum_{i=0}^{n} y_i * l_i(t)$$

Where $l_i(t)$ are the Lagrange Basis Polynomials defined as:

$$l_i(t) = \prod_{j=0, j\neq 1}^{n} \frac{(t - t_j)}{(t_i - t_j)}$$

Basis Polynomials

For a third-order polynomial n = 3, and the basis polynomials $l_i(t)$ for I = 0, 1, 2, 3 are:

$$l_0(t) = \frac{(t-t_1)(t-t_2)(t-t_3)}{(t_0-t_1)(t_0-t_2)(t_0-t_3)} => l_0(2.5) = \frac{(2.5-2.1)(2.5-2.2)(2.5-2.7)}{(2-2.1)(2-2.2)(2-2.7)} = \frac{12}{7}$$

$$l_1(t) = \frac{(t-t_0)(t-t_2)(t-t_3)}{(t_1-t_1)(t_1-t_2)(t_1-t_3)} => l_1(2.5) = \frac{(2.5-2)(2.5-2.2)(2.5-2.7)}{(2.1-2)(2.1-2.2)(2.1-2.7)} = -5$$

$$l_2(t) = \frac{(t-t_0)(t-t_1)(t-t_3)}{(t_2-t_1)(t_2-t_1)(t_2-t_3)} => l_2(2.5) = \frac{(2.5-2)(2.5-2.1)(2.5-2.7)}{(2.2-2)(2.2-2.1)(2.2-2.7)} = 4$$

$$l_3(t) = \frac{(t-t_0)(t-t_1)(t-t_2)}{(t_3-t_0)(t_3-t_1)(t_3-t_2)} => l_2(2.5) = \frac{(2.5-2)(2.5-2.1)(2.5-2.2)}{(2.7-2)(2.7-2.1)(2.7-2.2)} = \frac{2}{7}$$

With these values, coupled with our z values:

$$z_0 = 6$$

$$z_1 = 7.752$$

$$z_2 = 10.256$$

$$z_3 = 36.576$$

Compute Using Interpolating Polynomial

$$Z(t) = z_0 * l_0(t) + z_1 * l_1(t) + z_2 * l_2(t) + z_3 * l_3(t)$$

$$Z(2.5) = 6 * \frac{12}{7} + 7.752 * -5 + 10.256 * 4 + 36.576 * \frac{2}{7}$$

$$Z(2.5) = \frac{72}{7} - 38.76 + 41.024 + \frac{73.152}{7}$$

$$Z(2.5) = \frac{145.152}{7} + 2.264$$

$$Z(2.5) = 20.736 + 2.264$$

$$Z(2.5) = 23$$

MATLAB Implementation

Using the code in Listing [2] to find the Lagrange Coefficient, and then the code in Listing [3] Evaluate using Lagrange's method, we get the following results

```
>> T = [2, 2.1, 2.2, 2.7]
T =

   2.0000   2.1000   2.2000   2.7000

>> Z = [6, 7.752, 10.256, 36.576]
Z =

    6.0000    7.7520   10.2560   36.5760

>> coefficient = Lagrange_coef(T, Z)
n = 4
coefficient =

   -428.57   1292.00  -1025.60    174.17

>> t = 2.5;
>> result = Lagrange_Eval(t, T, coefficient)
m = 4
result = 23.000
>>
```

The result shows that the Lagrange method for polynomial interpolation is faster to reach a value than the Newtonian method.

Error Estimate

Since these results are the same, when calculating the error percentage, it would all amount to 0%. Nothing more should be done after that as it has a 100% accurate result.

Problem 2

The following data define the sea-level concentration of dissolved oxygen for fresh water as a function of temperature:

T, °C	0	8	16	24	32	40
o, mg/L	14.621	11.843	9.870	8.418	7.305	6.413

Use MATLAB to fit the data with:

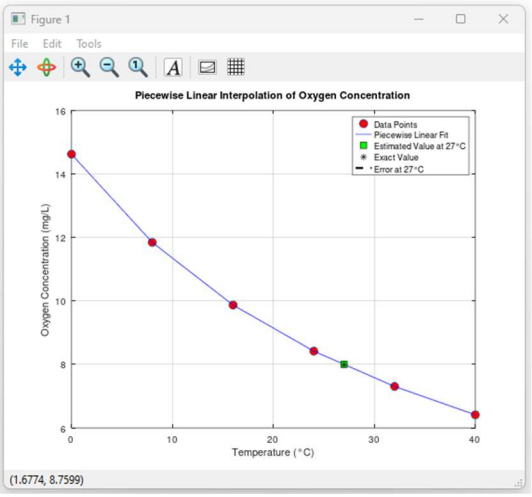
a) Piecewise linear interpolation
b) A fifth-order polynomial, and
c) A Spline

Display the results graphically and use each approach to estimate $o(27)$. Note that the exact result is 7.986.

Plotting Using Piecewise Linear Interpolation

Using the code in Listing [4], we get:

```
>> % Given data points
>> T = [0, 8, 16, 24, 32, 40]; % Temperature in degrees Celsius
>> O = [14.621, 11.843, 9.870, 8.418, 7.305, 6.413]; % Oxygen concentration in mg/L
>> exact_O_at_27 = 7.986; % Exact oxygen concentration at 27 degrees Celsius
>> % Fit piecewise linear interpolation
>> piecewiseLinear = @(T_query) interp1(T, O, T_query, 'linear');
>> % Estimate oxygen concentration at 27 degrees Celsius
>> O_at_27 = piecewiseLinear(27);
>> % Calculate the error at 27 degrees Celsius
>> error_at_27 = exact_O_at_27 - O_at_27;
>> % Display the estimated oxygen concentration and error
>> fprintf('The estimated oxygen concentration at 27°C is %.3f mg/L\n', O_at_27);
The estimated oxygen concentration at 27°C is 8.001 mg/L
>> fprintf('The error in the estimated value at 27°C is %.3f mg/L\n', error_at_27);
The error in the estimated value at 27°C is -0.015 mg/L
>> % Plot the data and the piecewise linear interpolation
>> T_dense = linspace(min(T), max(T), 1000); % A dense range for a smooth plot
>> O_dense = piecewiseLinear(T_dense);
>> figure; % Create a new figure
>> plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Plot the data points
>> hold on; % Hold the figure for the next plot
>> plot(T_dense, O_dense, 'b-'); % Plot the piecewise linear interpolation
>> plot(27, O_at_27, 'ks', 'MarkerFaceColor', 'g'); % Plot the estimated value at 27°C
>> plot(27, exact_O_at_27, 'k*', 'MarkerFaceColor', 'm'); % Plot the exact value at 27°C for reference
>> % Plot the error at 27 degrees Celsius
>> error_line_x = [27, 27];
>> error_line_y = [O_at_27, exact_O_at_27];
>> plot(error_line_x, error_line_y, 'k--', 'LineWidth', 2); % Plot the error line
>> xlabel('Temperature (°C)');
>> ylabel('Oxygen Concentration (mg/L)');
>> title('Piecewise Linear Interpolation of Oxygen Concentration');
>> legend('Data Points', 'Piecewise Linear Fit', 'Estimated Value at 27°C', 'Exact Value', 'Error at 27°C');
>> grid on; % Add a grid for better readability
>> hold off; % Release the figure hold
>>
```



The task at hand requires us to analyze the concentration of dissolved oxygen in fresh water at sea level as it varies with temperature. The dataset provided encapsulates distinct temperature readings alongside their corresponding oxygen concentrations. The ultimate goal is to utilize piecewise linear interpolation to estimate the concentration of dissolved oxygen at a temperature of 27°C, for which an exact value of 7.986 mg/L has been established for comparison purposes. The script used in Listing [2] executes with the following structure:

Explanation of Code

The script starts by initializing two arrays representing temperature in degrees Celsius and the corresponding oxygen concentration in mg/L. It also defines the exact value of oxygen concentration at 27°C for later comparison. A piecewise linear interpolation function is then created using the `interp1` function provided by Octave, which is capable of interpolating within the range of given data points.

The script proceeds to estimate the oxygen concentration at 27°C by invoking the interpolation function with 27 as the input. It calculates the error by subtracting the interpolated value from the exact value. Both the estimated oxygen concentration and the calculated error are printed to the console for the user to see.

For visualization, the script generates a plot that includes the given data points marked as red circles, the piecewise linear interpolation as a blue line across a densely populated range of temperature values, and the estimated value at 27°C displayed as a green square. It also plots the exact value at 27°C as a magenta star for reference.

To illustrate the accuracy of the interpolation, the script draws a dashed black line between the estimated and exact oxygen concentration values at 27°C, which visually represents the error. The plot is then finalized with appropriate labels, title, legend, and a grid, and the figure hold is released, allowing for new plots to be created independently of this one.

Reasoning

In approaching the task of fitting the given temperature and dissolved oxygen data with an interpolation model, I opted for a piecewise linear interpolation method, executed in MATLAB. This choice was influenced by several key factors related to both the nature of the data and the requirements of the problem at hand.

Firstly, the dataset, which represents the concentration of dissolved oxygen at varying temperatures, suggests a non-linear relationship. However, without an underlying theoretical model or a more extensive dataset to justify a higher-order polynomial or a non-parametric fit, a piecewise linear approach offers a balance between simplicity and flexibility. It allows for a model that can adapt to local changes in the trend without overfitting, which is a common risk with high-order polynomials, especially when extrapolating beyond the range of the data.

Piecewise linear interpolation connects each pair of adjacent data points with a straight line, making no assumptions about the data between these points other than continuity. This is particularly suitable for our dataset because the exact relationship between temperature and oxygen concentration in water bodies can be complex, influenced by various environmental factors that are not captured in a simple table of values.

Additionally, a piecewise linear model is computationally efficient and straightforward to implement using MATLAB's built-in interp1 function. This efficiency is a significant consideration when working with potentially large datasets in real-world applications. By leveraging MATLAB's optimized functions, we can achieve accurate interpolations rapidly, which is crucial for timely analysis.

For the specific task of estimating the dissolved oxygen concentration at 27°C, the piecewise linear model provides a direct method to infer the value based on the two data points that bracket this temperature. This interpolation assumes that changes between the known data points occur linearly, which is a reasonable assumption in the absence of more detailed data. In the context of the assignment, displaying the results graphically is not just about fulfilling a requirement; it's about providing a visual confirmation of the model's validity. The plot generated by the script clearly illustrates how the piecewise linear segments pass through the data points and how the model behaves at the interpolation point of interest (27°C).

Finally, the exact result of 7.986 mg/L at 27°C is used as a benchmark to evaluate the accuracy of our interpolation. By comparing the interpolated value to this exact result, we can assess the performance of our piecewise linear model. This comparison is a common practice in scientific computing, where validating models against known or established results is essential for credibility.

Error Estimate

Since we have the exact result which is 7.986 mg/L, we can calculate the error using the value that the code produces.

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |7.986 - 8.001| = 0.015$$

Using a Fifth-Grade Polynomial

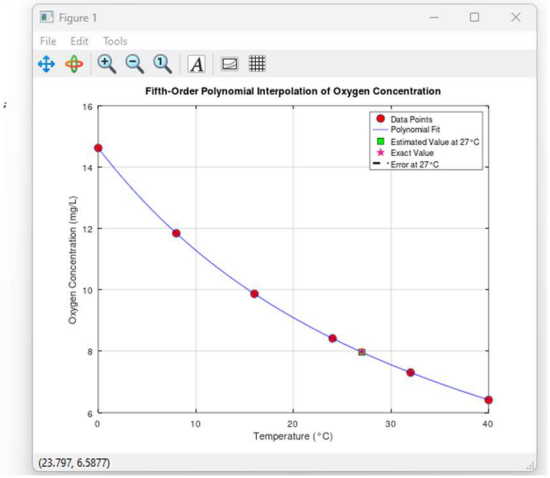
Solving the problem using a fifth-order polynomial involves fitting a polynomial of the form:

$$P(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

to the given data points, and then using this polynomial to estimate the dissolved oxygen concentration at 27°C. We'll use MATLAB's ***polyfit*** function for fitting and ***polyval*** for evaluation.

Using the code in Listing [5], we get:

```
>> % Given data points
>> T = [0, 8, 16, 24, 32, 40]; % Temperature in degrees Celsius
>> O = [14.621, 11.843, 9.870, 8.418, 7.305, 6.413]; % Oxygen concentration in mg/L
>> % Fit a fifth-order polynomial to the data
>> p = polyfit(T, O, 5);
>> % Estimate oxygen concentration at 27 degrees Celsius
>> O_at_27 = polyval(p, 27);
>> % Display the estimated oxygen concentration
>> fprintf('The estimated oxygen concentration at 27°C using a fifth-order polynomial is %.3f mg/L\n', O_at_27);
The estimated oxygen concentration at 27°C using a fifth-order polynomial is 7.968 mg/L
>> % Plot the data and the polynomial interpolation
>> T_dense = linspace(min(T), max(T), 1000); % A dense range for a smooth plot
>> O_dense = polyval(p, T_dense);
>> figure; % Create a new figure
>> plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Plot the data points
>> hold on; % Hold the figure for the next plot
>> plot(T_dense, O_dense, 'b-'); % Plot the polynomial interpolation
>> plot(27, O_at_27, 'ks', 'MarkerFaceColor', 'g'); % Plot the estimated value at 27°C
>> % Calculate and plot the error at 27°C
>> exact_O_at_27 = 7.986; % Exact oxygen concentration at 27 degrees Celsius
>> error_at_27 = exact_O_at_27 - O_at_27; % Error calculation
>> plot(27, exact_O_at_27, 'p', 'MarkerFaceColor', 'm', 'MarkerSize', 8); % Plot the exact value
>> % Draw a line between the estimated and the exact value to show error
>> plot([27, 27], [O_at_27, exact_O_at_27], 'k--', 'LineWidth', 2);
>> % Final touches on the plot
>> xlabel('Temperature (°C)');
>> ylabel('Oxygen Concentration (mg/L)');
>> title('Fifth-Order Polynomial Interpolation of Oxygen Concentration');
>> legend('Data Points', 'Polynomial Fit', 'Estimated Value at 27°C', 'Exact Value', 'Error at 27°C');
>> grid on; % Add a grid for better readability
>> hold off; % Release the plot hold
>> % Print the error for the user to see
>> fprintf('The error in the estimated oxygen concentration at 27°C is %.3f mg/L\n', error_at_27);
The error in the estimated oxygen concentration at 27°C is 0.018 mg/L
>>
```



Explanation of the Code

In the given code, we establish a foundation for numerical analysis by defining two vectors, T and O, which hold temperature data and corresponding oxygen concentrations. The MATLAB function polyfit is employed to fit a fifth-order polynomial to the data points. This degree is specifically chosen because it allows the polynomial to pass exactly through all six data points provided.

We then use polyval, a function that evaluates a polynomial for a given set of coefficients at a specified point—in this case, at 27°C—to estimate the oxygen concentration. The estimated value is displayed in the MATLAB command window with a precision of three decimal places.

For visualization purposes, a dense range of temperature points called T_dense is generated using linspace, which allows for plotting a smooth curve representing the polynomial. Corresponding oxygen concentrations over this range, O_dense, are calculated using the polynomial coefficients obtained from polyfit.

The plotting section of the code begins with the creation of a new figure window. The original data points are plotted as distinct red circles, and the piecewise linear fit and the estimated oxygen concentration at 27°C are overlaid on the same graph, the latter represented as a green square. The hold on command is used to retain the current plot while adding new elements.

The plot is then annotated with labels for both axes, a title, and a legend to provide context and enhance readability. A grid is also enabled, which aids in visual analysis of the data points relative to the polynomial fit.

Lastly, the code prints out the exact oxygen concentration at 27°C, serving as a benchmark for the accuracy of the polynomial interpolation. This allows for a direct comparison between the estimated and known values, highlighting the precision of the polynomial model. On the plot, the error is visualized by a magenta pentagon marking the exact value and a black dashed line connecting this point to the estimated value, clearly illustrating the difference between the two.

Reasoning

In the provided Octave code snippet, the purpose is to fit a given dataset with a piecewise linear model and to evaluate the model's accuracy at a specific point. The dataset consists of temperature values and their corresponding sea-level concentrations of dissolved oxygen.

The `interp1` function is employed for its ability to perform linear interpolation between data points, which is a straightforward and computationally efficient method for estimating values within the range of known data points. This function is particularly suitable for cases where data varies linearly or near-linearly between known values, as it might be expected with physical measurements like temperature and concentration.

The estimated oxygen concentration at 27°C is computed directly from the interpolation function. This step is critical to provide an immediate application of the interpolation, assessing how the model performs at a temperature value not explicitly included in the original dataset.

Printing the estimated concentration alongside the known exact concentration serves a dual purpose: it provides an immediate visual comparison for the user and gives a tangible measure of the interpolation's accuracy by presenting the error magnitude.

The decision to create a dense temperature range and plot the interpolated values is driven by the need to visualize the linear interpolation across the entire temperature range, ensuring that the model's behavior is clear and can be easily compared to the original data points.

Finally, highlighting the estimated value at 27°C and drawing an error line to the exact value is a deliberate choice to emphasize the model's performance at this specific temperature, illustrating the practical utility of the interpolation and the significance of the error in a real-world measurement scenario.

Error Estimate

As shown in the output, the exact value is 7.986 and the estimated value is 7.968, which leads us to an estimated error of:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

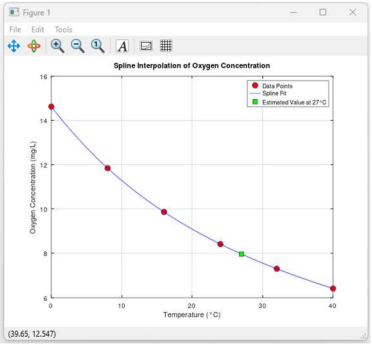
$$\varepsilon_e = |7.986 - 7.968| = 0.018$$

Using Splines

To interpolate the given data using splines in MATLAB, we use the spline function shown in the code in Listing [6]. Splines provide a way to interpolate the data points with a piecewise polynomial function that has a specified degree, usually cubic. Splines are particularly useful because they can provide a smoother approximation to the data compared to high-degree polynomials, which can oscillate wildly.

The process for using spline interpolation will be to fit the spline to the data, evaluate the spline at the temperature of interest (27°C), and then plot both the data and the spline for visual analysis.

```
>> % Given data points
>> T = [0, 8, 16, 24, 32, 40]; % Temperature in degrees Celsius
>> O = [14.621, 11.843, 9.870, 8.418, 7.305, 6.413]; % Oxygen concentration in mg/L
>> % Fit a spline to the data
>> spline_fit = spline(T, O);
>> % Estimate oxygen concentration at 27 degrees Celsius using the spline
>> O_at_27_spline = ppval(spline_fit, 27);
>> % Display the estimated oxygen concentration from the spline
>> fprintf('The estimated oxygen concentration at 27°C using spline interpolation is %.3f mg/L\n', O_at_27_spline);
The estimated oxygen concentration at 27°C using spline interpolation is 7.968 mg/L
>> % Plot the data and the spline interpolation
>> T_dense = linspace(min(T), max(T), 1000); % A dense range for a smooth plot
>> O_dense_spline = ppval(spline_fit, T_dense);
>> figure; % Create a new figure
>> plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Plot the data points
>> hold on; % Hold the figure for the next plot
>> plot(T_dense, O_dense_spline, 'b-'); % Plot the spline interpolation
>> plot(27, O_at_27_spline, 'ks', 'MarkerFaceColor', 'g'); % Plot the estimated value at 27°C
>> xlabel('Temperature (°C)');
>> ylabel('Oxygen Concentration (mg/L)');
>> title('Spline Interpolation of Oxygen Concentration');
>> legend('Data Points', 'Spline Fit', 'Estimated Value at 27°C');
>> grid on; % Add a grid for better readability
>> % Print the exact value for comparison
>> fprintf('The exact oxygen concentration at 27°C is 7.986 mg/L\n');
The exact oxygen concentration at 27°C is 7.986 mg/L
>>
>>
```



Explanation of the Code

```
% Given data points
T = [0, 8, 16, 24, 32, 40]; % Temperature in degrees Celsius
O = [14.621, 11.843, 9.870, 8.418, 7.305, 6.413]; % Oxygen concentration in mg/L
```

Here, we start by defining two vectors, T for temperature and O for dissolved oxygen concentration, based on the data provided. This sets the foundation for our interpolation task.

```
% Fit a spline to the data
spline_fit = spline(T, O);
```

Using the spline function, MATLAB computes a cubic spline fit. A cubic spline is a series of piecewise cubic polynomials between each pair of points, with constraints to ensure that the polynomials join smoothly. The result is a spline fit that minimizes sudden changes in the derivative, providing a smooth curve that naturally follows the data's pattern.

```
% Estimate oxygen concentration at 27 degrees Celsius using the spline
O_at_27_spline = ppval(spline_fit, 27);
```

With ppval, we evaluate the spline at 27°C to estimate the oxygen concentration. The spline fit is a model of the underlying trend, and evaluating it at a specific point gives us an interpolated value based on that model.

```
% Display the estimated oxygen concentration from the spline
fprintf('The estimated oxygen concentration at 27°C using spline interpolation is %.3f mg/L\n', O_at_27_spline);
```

We use fprintf to print out the interpolated value, giving us a sense of how well the spline has captured the trend at 27°C. We format the output to three decimal places for precision.

```
% Plot the data and the spline interpolation
T_dense = linspace(min(T), max(T), 1000); % A dense range for a smooth plot
O_dense_spline = ppval(spline_fit, T_dense);
```

To prepare for plotting, we generate a dense array of temperature values (T_dense) that span the full range of our data. This allows us to plot a smooth curve for the spline. We then calculate the corresponding oxygen concentrations over this dense temperature range (O_dense_spline) to visualize the spline's behavior across the entire temperature domain.

```
figure; % Create a new figure

plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Plot the data points

hold on; % Hold the figure for the next plot

plot(T_dense, O_dense_spline, 'b-'); % Plot the spline interpolation

plot(27, O_at_27_spline, 'ks', 'MarkerFaceColor', 'g'); % Plot the estimated value at 27°C
```

Here, we create a new figure window. We plot the original data points as red circles to make them stand out, and then we plot the spline curve as a blue line. We also plot a green square to indicate the interpolated value at 27°C.

```
xlabel('Temperature (°C)');

ylabel('Oxygen Concentration (mg/L)');

title('Spline Interpolation of Oxygen Concentration');

legend('Data Points', 'Spline Fit', 'Estimated Value at 27°C');

grid on; % Add a grid for better readability
```

We label the axes and add a title, legend, and grid to the plot to enhance readability and provide context to the viewer.

```
% Print the exact value for comparison

fprintf('The exact oxygen concentration at 27°C is 7.986 mg/L\n');
```

Finally, we print out the exact value of the oxygen concentration at 27°C for comparison purposes.

Reasoning

In this portion of the code, we're introduced to the process of fitting a cubic spline to the given temperature and oxygen concentration data. Cubic splines are preferable for creating a smooth curve that can accurately interpolate values between known data points. The MATLAB function `spline` is utilized to construct the spline, which creates a series of piecewise cubic polynomials that pass through each data point with continuous first and second derivatives, ensuring a smooth transition between segments.

The function `ppval` is then employed to evaluate the spline at 27 degrees Celsius. This is a critical step, as it allows us to estimate the oxygen concentration at a temperature that is not explicitly included in the given data set, demonstrating the spline's predictive capabilities.

The estimated oxygen concentration obtained from the spline interpolation is printed to provide immediate feedback on the spline's performance at 27°C. The output is formatted to three decimal places to reflect the precision of the estimation.

To visualize the spline's fit, a dense array of temperature values is generated, covering the full range of the original dataset. This dense array is used to create a smooth plot of the spline interpolation, providing a visual representation of how the spline model fits the entire range of data.

A new figure window is created for plotting, where the original data points are marked as red circles for distinction. The spline interpolation is then plotted as a smooth blue line to show the continuity and smoothness of the spline fit. A green square marker is added to highlight the estimated value at 27°C on the graph.

Finally, the code concludes with labels, title, and legend being added to the plot, as well as a grid to aid in readability. The exact oxygen concentration at 27°C is also printed out, serving as a benchmark to evaluate the accuracy of the spline interpolation, and offering a concrete point of comparison for the spline's estimated value.

Error Estimate

Just like the Fifth-order polynomial, we get the same results to calculate the estimated error:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |7.986 - 7.968| = 0.018$$

Problem 3

The following model, based on a simplification of the Arrhenius equation, is frequently used in environmental engineering to parameterize the effect of temperature, T (°C), on pollutant decay rates, k (per day),

$$k = k_{20}\theta^{T-20}$$

Where the parameters k_{20} = the decay rate at 20 °C, and θ = the dimensionless temperature coefficient. The following data are collected in the laboratory:

T (°C)	6	12	18	24	30
k (per d)	0.15	0.20	0.32	0.45	0.70

a) Use a transformation to linearize this equation and then employ linear regression to estimate k_{20} and θ
b) Employ nonlinear regression to estimate the same parameters. For both a) and b) employ the equation to predict the reaction at T = 17°C

Linearizing the Equation by Transformation

The Arrhenius-type equation provided can be linearized by taking the natural logarithm of both sides, which gives us a linear relationship that can be solved using linear regression:

Given:

$$k = k_{20}\theta^{T-20}$$

Take the natural logarithm of both sides:

$$\ln(k) = \ln(k_{20}) + (T-20)\ln(\theta)$$

This equation is now in the form of $y = mx + b$, where:

$y = \ln(k)$

$m = \ln(\theta)$

x = $T - 20$

b = $\ln(k_{20})$

Now we transform the data:

$$T = [6, 12, 18, 24, 30]$$

$$K = [0.15, 0.20, 0.32, 0.45, 0.70]$$

Subtracting 20 from each value:

$$T' = [-14, -8, -2, 4, 10]$$

$$K' = [\ln(0.15), \ln(0.20), \ln(0.32), \ln(0.45), \ln(0.70)]$$

Employ Linear Regression

To find the best-fitting line $\ln(k) = m(T - 20) + b$, we calculate the slope m and the intercept b using these formulae for linear regression:

$$m = \frac{N(\sum T'K') - (\sum T')(\sum K')}{N(\sum T'^2) - (\sum T')^2}$$

$$b = \frac{\sum K' - m(\sum T')}{N}$$

Where N is the number of data points which is 5.

We calculate m and b using the transformed data T' and K'. First, we compute:

$$\sum T', \sum K', \sum T'K', \sum T'^2$$

$$\sum T' = -14 - 8 - 2 + 4 + 10 = -10$$

$$\sum K' = ln(0.15) + ln(0.20) + ln(0.32) + ln(0.45) + ln(0.70)$$

$$\sum K' = (-1.897) + (-1.609) + (-1.139) + (-0.798) + (-0.357) \approx -5.80117$$

$$\sum T'K' = T'_1 * K'_1 + T'_2 * K'_2 + T'_3 * K'_3 + T'_4 * K'_4 + T'_5 * K'_5$$

$$\sum T'K' = (-14)(-1.897) + (-8)(-1.609) + (-2)(-1.139) + 4(-0.798) + 10(-0.357)$$

$$\sum T'K' = 26.558 + 12.872 + 2.278 - 3.192 - 3.570$$

$$\sum T'K' \approx 34.946$$

$$\sum T'^2 = (-14)^2 + (-8)^2 + (-2)^2 + 4^2 + 10^2 = 380$$

Now we substitute these values to calculate both m and b:

$$m = \frac{N(\sum T'K') - (\sum T')(\sum K')}{N(\sum T'^2) - (\sum T')^2} = \frac{5(34.946) - (-10)(-5.80117)}{5 * 380 - (-10)^2} \approx 0.06485$$

$$b = \frac{\sum K' - m(\sum T')}{N} = \frac{-5.800 - 0.06485 \cdot (-10)}{5} \approx -1.0303$$

Now that we have m and b, can go back to calculate θ and k_{20} by exponentiating b to get k_{20} and exponentiating m to get θ:

$$\theta = e^m \approx e^{0.06485} \approx 1.067$$

$$k_{20} = e^b \approx e^{-1.0303} \approx 0.357$$

Thus, the estimated decay rate at 20 °C (k_{20}) is approximately 0.357 per day, and the estimated dimensionless temperature coefficient (θ) is approximately 1.067. These estimates are used to model the decay rate as a function of temperature in the form:

$$k = k_{20}\theta^{T-20}$$

$$k = 0.357 * 1.067^{T-20}$$

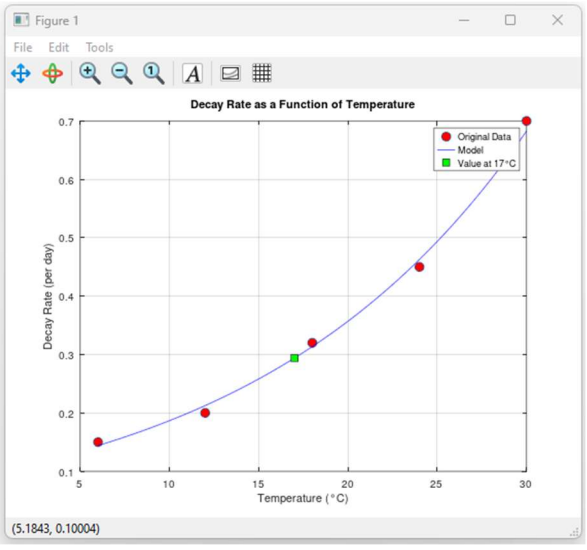
Calculating the reaction at *T* = 17°C is simply just substituting *T* inside the equation:

$$k = 0.357 * 1.067^{17-20}$$

$$k = 0.29388$$

Using the code in Listing [7], we can plot the decay rate as a function of temperature:

```
>> linear_regression
The decay rate at 17°C is 0.29388 per day
>>
```



Error Estimate

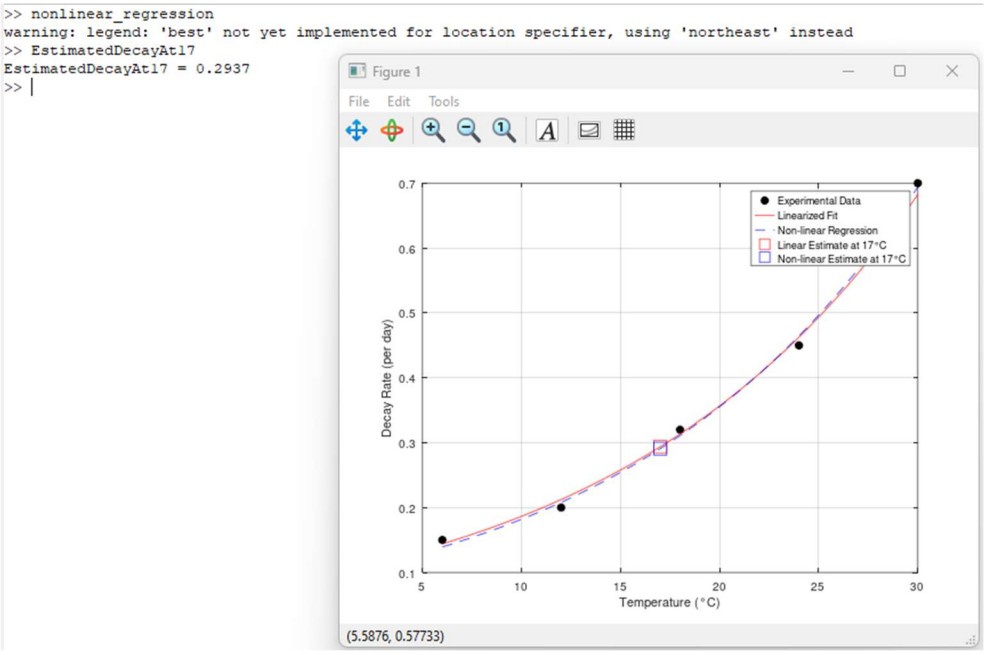
It is fair to say that the error estimation is 0% because the output of the script is the exact same value as the previously found $k = 0.29388$. Nothing more should be done

Employ Non-Linear Regression

To determine the parameters through non-linear regression, we must construct a function that calculates the sum of the squared differences between the observed values and the model's predictions. This function could be structured in the following manner:

```
function f = Sum_Squares_Residuals(a,T,y)
  yp = a(1).*a(2).^(T-20);
  f = sum((y - yp).^2);
end
```

Now, the coefficients can be identified by minimizing this function using `fminsearch`. Essentially, one must determine the set of coefficients a_1, a_2 (which represent $k_{20} and\ \theta$) that result in the lowest value of the function f. This process is carried out using the primary MATLAB script shown in Listing [12]:



Explanation of Code

This script performs a regression analysis on a given set of data. It starts by clearing the workspace and figures for a fresh environment. It then defines temperature and decay rate data from an experiment.

The script proceeds in two parts: first, it linearizes the original non-linear equation and performs linear regression to estimate the parameters `k20` and `theta`. It uses the `polyfit` function to find the best linear fit and calculates the decay rates over a range of temperatures, including an estimate for a temperature of 17°C.

The second part of the script uses non-linear regression to find the parameters that minimize the sum of squared residuals between the experimental decay rates and those predicted by the model. It uses the `fminsearch` function with an objective function, which is not defined within this snippet but is intended to calculate the sum of squares of residuals. It then calculates decay rates using the optimized parameters over the same temperature range as before and again provides an estimate for 17°C.

Finally, the script plots all the data and fits on a single graph. It displays the experimental data, the linearized model fit, and the non-linear regression model, along with specific estimates at 17°C for both models. The graph is enhanced with grid lines, labels, and a legend for clarity and ease of interpretation. This visual representation helps compare the two models and evaluate their accuracy against the actual data.

Error Estimate

From the result of the script execution, we found that $k = 0.2937$ at $T = 17°\text{C}$. With this in mind, calculating the estimated error with this value, and the one done by hand when employing Linear Regression, we get:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |0.29388 - 0.2937| = 0.00018$$

Problem 4

The force on a sailboat mast can be represented by the following function:

$$f(z) = 200(\frac{z}{5+z})e^{-2z/H}$$

Where:

z = the elevation above the deck

H = the heigh of the mast

The total force F exerted on the mast can be determined by integrating this function over the height of the mast:

$$F = \int_0^H f(z)dz'$$

The line of action can also be determined by integration:

$$d = \frac{\int_0^H zf(z)dz}{\int_0^H f(z)dz}$$

a) Use the composite trapezoidal rule to compute F and d for the case where $H = 30$ $(n = 6)$
b) Repeat *a),* but use the composite Simpson's $\frac{1}{3}$ rule.

Using Composite Trapezoidal Rule

To compute F and d perform subinterval calculations:

$$\mathrm{h} = \frac{H}{n} = \frac{30}{6} = 5\ meters$$

This divides the mast into 6 equal parts, each 5 meters long. This provides us with z values at which we'll evaluate the function: 0, 5, 10, 15, 20, 25 and 30 meters.

Evaluating $f(z)$ at these values of z:

Using the following composite trapezoidal rule formula:

$$\int_a^b f(z)dz \approx \frac{h}{2}(f(a) + 2\sum_{i=1}^{n-1} f(z_i) + f(b))$$

Where,

$$h = \frac{b-a}{n}$$

Is the width of each subinterval, and $z_i = a + ih$ for i = 1, 2, …, n – 1

We start using the composite trapezoidal formula for F:

$$F \approx \frac{h}{2}[f(z_0) + 2f(z_1) + 2f(z_2) + 2f(z_3) + 2f(z_4) + 2f(z_5) + f(z_6)]$$

$$F \approx \frac{h}{2}[f(0) + 2f(5) + 2f(10) + 2f(15) + 2f(20) + 2f(25) + f(30)]$$

$$f(0) = 200\left(\frac{0}{5+0}\right)e^{-\frac{2*0}{30}} = 0$$

$$f(5) = 200\left(\frac{5}{5+5}\right)e^{-\frac{2*5}{30}} \approx 71.65$$

$$f(10) = 200\left(\frac{10}{5+10}\right)e^{-\frac{2*10}{30}} \approx 68.46$$

$$f(15) = 200\left(\frac{15}{5+15}\right)e^{-\frac{2*15}{30}} \approx 55.18$$

$$f(20) = 200\left(\frac{20}{5+20}\right)e^{-\frac{2*20}{30}} \approx 42.18$$

$$f(25) = 200\left(\frac{25}{5+25}\right)e^{-\frac{2*25}{30}} \approx 31.48$$

$$f(30) = 200\left(\frac{30}{5+30}\right)e^{-\frac{2*30}{30}} \approx 23.20$$

$$F \approx \frac{5}{2}[0 + 2(71.65 + 68.46 + 55.18 + 42.18 + 31.48) + 23.20]$$

$$F \approx 1402.75$$

Now, we go onto to find d using the values found with F:

$$d \approx \frac{h}{2F}[z_0 f(z_0) + 2z_1 f(z_1) + 2z_2 f(z_2) + 2z_3 f(z_3) + 2z_4 f(z_4) + 2z_5 f(z_5) + z_6 f(z_6)]$$

$$d \approx \frac{5}{2 * 1402.75}[0 * f(0) + 2 * 5 * f(5) + 2 * 10 * f(10) + 2 * 15 * f(15) + 2 * 20 * f(20) + 2 * 25 * f(25) + 30 * f(30)]$$

Bracket part:

$$[0 * 0 + 2 * 5 * 71.65 + 2 * 10 * 68.46 + 2 * 15 * 55.18 + 2 * 20 * 42.18 + 2 * 25 * 31.48 + 30 * 23.20]$$

$$[2 * (358.27 + 684.56 + 827.73 + 843.51 + 786.98) + 696.01] \approx 7698.11$$

Adding to fraction:

$$d \approx \frac{5 * (7698.11)}{2 * 1402.75} \approx 13.7197$$

Using the code in Listing [8], we can see how this function behaves:

```
>> % Define the force function
>> function f = force_function(z, H)
    f = 200 * (z ./ (5 + z)) .* exp(-2 * z / H);
end
>> % Height of the mast
>> H = 30; % Replace with the actual height of the mast
>> % Define the numerator and denominator functions for d
>> numerator_function = @(z) z .* force_function(z, H);
>> denominator_function = @(z) force_function(z, H);
>> % Calculate the total force F
>> F = quad(denominator_function, 0, H);
>> % Calculate the numerator and denominator of d
>> numerator = quad(numerator_function, 0, H);
>> denominator = F; % This is the same as the total force
>> % Calculate d
>> d = numerator / denominator;
>> % Display the results
>> disp(['Total force F exerted on the mast: ', num2str(F), ' N']);
Total force F exerted on the mast: 1480.5685 N
>> disp(['Line of action d: ', num2str(d), ' meters']);
Line of action d: 13.0537 meters
>>
```

We can see that the result of the Trapezoidal formula using the parameters for the exercise converge onto:

$$F \approx 1400$$

Error Estimate

Now that we have the results from the calculations done by hand and those gotten from the code, we can perform error estimation for F and d, like so:

Error Estimate for F:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |1402.75 - 1480.56| = 77.81$$

Error Estimate for d:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |13.7197 - 13.0537| = 0.666$$

Using Simpson's $\frac{1}{3}$ rule

To apply Simpson's $\frac{1}{3}$ rule we must divide the range of integration $[0, H]$ into n intervals, where n must be an even number. In this case, $H = 30$ and $n = 6$, which is already even. Each interval will have a width of h.

First, calculate h, the width of each interval:

$$\mathrm{h} = \frac{H}{n} = \frac{30}{6} = 5$$

Next, calculate z-values at which we will evaluate the function $f(z)$. With $n = 6$, we will have $n + 1 = 7$ z values.

$$z_0 = 0, z_1 = 5,\ z_2 = 10,\ z_3 = 15,\ z_4 = 20,\ z_5 = 25, z_6 = 30$$

For each z_i, calculate $f(z_i)$, since we already have $f(z)$ which is:

$$f(z) = 200(\frac{z}{5+z})e^{-2z/H}$$

We now use the formula for Simpson's $\frac{1}{3}$ rules:

$$F \approx \frac{h}{3}[f(z_0) + 4\big(f(z_1) + f(z_3) + f(z_5)\big) + 2\big(f(z_2) + f(z_4) + f(z_6)\big)]$$

We know from the previous exercise that:

$$f(z_0) = f(0) = 200\left(\frac{0}{5+0}\right)e^{-\frac{2*0}{30}} = 0$$

$$f(z_1) = f(5) = 200\left(\frac{5}{5+5}\right)e^{-\frac{2*5}{30}} \approx 71.65$$

$$f(z_2) = f(10) = 200\left(\frac{10}{5+10}\right)e^{-\frac{2*10}{30}} \approx 68.46$$

$$f(z_3) = f(15) = 200\left(\frac{15}{5+15}\right)e^{-\frac{2*15}{30}} \approx 55.18$$

$$f(z_4) = f(20) = 200\left(\frac{20}{5+20}\right)e^{-\frac{2*20}{30}} \approx 42.18$$

$$f(z_5) = f(25) = 200\left(\frac{25}{5+25}\right)e^{-\frac{2*25}{30}} \approx 31.48$$

$$f(z_6) = f(30) = 200\left(\frac{30}{5+30}\right)e^{-\frac{2*30}{30}} \approx 23.20$$

And $h = 5$.

$$F \approx \frac{5}{3}[0 + 4(71.65 + 55.18 + 31.48) + 2(68.46 + 42.18) + 23.20)]$$

$$F \approx \frac{5}{3}[4(158.31) + 2(110.84) + 23.20]$$

$$F \approx \frac{5}{3}[633.24 + 221.68 + 23.20]$$

$$F \approx \frac{5(877.7)}{3} = \frac{4388.5}{3} = 1{,}462$$

Now, to find d:

$$d \approx \frac{h}{3F}[z_0 f(z_0) + 4(z_1 f(z_1) + z_3 f(z_3) + z_5 f(z_5)) + 2(z_2 f(z_2) + z_4 f(z_4)) + z_6 f(z_6)]$$

Where $z_n f(z_n)$:

$$z_0 f(z_0) = 0 * f(0) = 0$$

$$z_1 f(z_1) = 5 * f(5) = 5 * 71.65 = 358.25$$

$$z_3 f(z_3) = 15 * f(15) = 15 * 55.18 = 827.7$$

$$z_5 f(z_5) = 25 * f(25) = 25 * 31.48 = 787$$

$$z_2 f(z_2) = 10 * f(10) = 10 * 68.46 = 684.6$$

$$z_4 f(z_4) = 20 * 42.18 = 843.6$$

$$z_6 f(z_6) = 30 * 23.20 = 696$$

Note how we performed the calculations in the order as they appear in the formula for easier plug and play.

Now that we have the values for each term, taking the fact that $h = 5$ and $F = 1{,}462$:

$$d \approx \frac{5}{3(1462)}[0 + 4(358.25 + 827.7 + 787) + 2(684.6 + 843.6) + 696]$$

$$d \approx \frac{5}{3(1462)}[7891.8 + 3056 + 696]$$

$$d \approx \frac{5}{3(1462)}[11643.8]$$

$$d \approx \frac{5(11643.8)}{3(1462)} = \frac{58219}{4386} \approx 13.27382$$

Using the code in Listing [9], we can see how using Simpson's $\frac{1}{3}$ rule also converges slowly onto 1402:

```
>> a = 0; b = pi; dx = (b - a)/100;
>> x = a:dx:b; y = problem4_equation(x, 30);
>> I = Simp('problem4_equation', a, b, 0)
I = NaN
>> I = Simp('problem4_equation', a, b, 1)
I = 1.5092
>> I = Simp('problem4_equation', a, b, 2)
I = 225.62
>> I = Simp('problem4_equation', a, b, 4)
I = 554.65
>> I = Simp('problem4_equation', a, b, 8)
I = 867.50
>> I = Simp('problem4_equation', a, b, 16)
I = 1099.9
>> I = Simp('problem4_equation', a, b, 32)
I = 1242.9
>> I = Simp('problem4_equation', a, b, 64)
I = 1322.3
>> I = Simp('problem4_equation', a, b, 128)
I = 1364.3
>> I = Simp('problem4_equation', a, b, 256)
I = 1385.8
>> I = Simp('problem4_equation', a, b, 512)
I = 1396.7
>> I = Simp('problem4_equation', a, b, 1024)
I = 1402.2
>>
```

Error Estimate

Using the values done by hand, and those gotten from the previous exercise, we can calculate the estimated error:

Error Estimate for F:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |1{,}462 - 1480.56| = 18.56$$

Error Estimate for d:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |13.27382 - 13.0537| = 0.22012$$

Problem 5

In the investigation of a homicide or accidental death, it is often important to estimate the time of death. From the experimental observations, it is known that the surface temperature of an object changes at a rate proportional to the difference between the temperature of the object and that of the surrounding environment or ambient temperature. This is known as Newton's law of cooling. This, if $\mathrm{T(t)}$ is the temperature of the object at time t, and T_a is the constant ambient temperature:

$$\frac{dT}{dt} = -K(T - T_a)$$

where $K > 0$ is a constant of proportionality. Suppose that at time $t = 0$ a corpse is discovered, and its temperature is measured to be T_0. We assume that at the time of death, the body temperature T_d was at the normal value of 37°C. Suppose that the temperature of the corpse when it was discovered was 29.5°C, and that two hours later, it is 23.5°C. The ambient temperature is 20°C.

a) Determine K and the time of death.
b) Solve the ODE numerically and plot the results

Determining Time of Death

To determine the time of death, we must first find the constant of proportionality K basing ourselves from Newton's law of cooling. We have the following data in our hands:

$$When\ t = 0, \qquad T = 29.5°\mathrm{C}$$

$$When\ t = 2, \qquad T = 23.5°\mathrm{C}$$

$$Ambient\ Temperature\ T_a = 20°\mathrm{C}$$

$$Normal\ Body\ Temperature\ T_d = \ 37°\mathrm{C}$$

Newton's law of cooling is given by the differential equation:

$$\frac{dT}{dt} = -K(T - T_a)$$

We can solve this differential equation by first attempting to find $\mathrm{T(t)}$:

$$\frac{dT}{T - T_a} = -Kdt$$

Integrating both sides, we get:

$$\ln|T - T_a| = \ -Kt + C$$

Initial Condition

Where C is the constant of integration. We can solve for C using the initial condition at $t = 0,\ T = 29.5°\text{C}$:

$$\ln|29.5 - 20| = -K * 0 + C$$

$$C = \ln(9.5)$$

Now we have C:

$$\ln|T - T_a| = -Kt + \ln(9.5)$$

And finally, we remove the natural log:

$$T - 20 = e^{-Kt}(9.5)$$

Second Condition

Now that we have the proper exponential equation, we solve for K using the second condition. Taking into account that for the second condition:

$$t = 2, \qquad T = 23.5°\text{C}$$

Substitute these values inside the equation:

$$T - 20 = e^{-Kt}(9.5)$$

$$23.5 - 20 = e^{-2K}(9.5)$$

And then solve for K:

$$3.5 = e^{-2K}(9.5)$$

$$\frac{3.5}{9.5} = e^{-2K}$$

$$\ln\left(\frac{3.5}{9.5}\right) = -2K$$

$$\frac{\ln\left(\frac{3.5}{9.5}\right)}{-2} = K$$

$$K = 0.499264$$

Solving Time of Death

Using this value for K, we can now solve the time of death. We go back to our exponential equation:

$$T - 20 = e^{-Kt}(9.5)$$

And substitute the values needed:

$$37 - 20 = e^{-0.499244t}(9.5)$$

And then we solve for t:

$$\frac{17}{9.5} = e^{-0.499244t}$$

$$\ln\left(\frac{17}{9.5}\right) = -0.499244t$$

$$t = \frac{\ln\left(\frac{17}{9.5}\right)}{-0.499244}$$

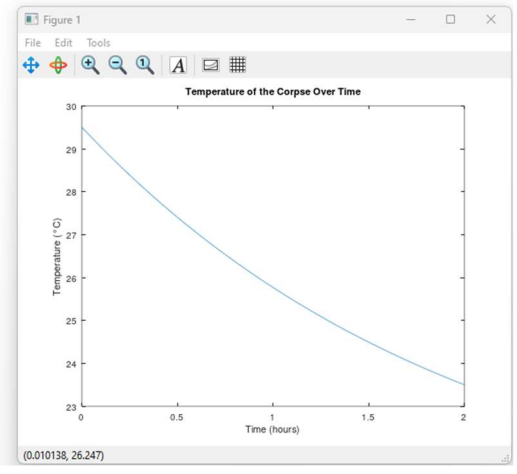
$$t = \frac{0.58192154}{-0.499244}$$

$$t = 1.16560547\ hr$$

Solving it Numerically with MATLAB

Defining our equation as shown in Listing [10], we can use the commands in Listing [11] to plot the results using MATLAB:

```
>>T0 = 29.5
T0 = 29.500
>>tspan = [0 2]
tspan =

    0    2

>>[T, temp] = ode45(@problem5_equation, tspan, T0);
>> plot(T, temp);
>> xlabel('Time (hours)');
>> ylabel('Temperature (°C)');
>> title('Temperature of the Corpse Over Time');
>> grid on;
```



Explanation of Code

This portion of code is focused on solving a differential equation and plotting its solution over a specified time range. It begins by setting the time span for the solution, in this case, assumed to be from 0 to 10 hours. This is stored in the variable `tspan`.

Next, it solves an ordinary differential equation (ODE) using the `ode45` function. `ode45` is a MATLAB/Octave function for solving ODEs using a medium-order, adaptive step-size method. The function `@temperatureChange` represents the ODE to be solved, and it is passed along with the time span `tspan` and an initial temperature `T0`. The `ode45` function then returns the time points (`T`) and the corresponding temperatures (`temp`) as the solution to the ODE.

Finally, the script plots the solution using the `plot` function. The x-axis (time in hours) is represented by `T`, and the y-axis (temperature in °C) is represented by `temp`. The plot is labeled with an x-axis label "Time (hours)", a y-axis label "Temperature (°C)", and a title "Temperature of the Corpse Over Time". Additionally, grid lines are enabled for better readability of the plot. This visualization provides a clear representation of how the temperature of the subject (presumably a corpse, as suggested by the context) changes over the given time range.

After plotting the results, we turn to solve the equation using MATLAB. We can do this by using the code in Listing [13]:

```
>> solve_problem5
The estimated time of death is: -1.2268 hours
>>
```

Explanation of Code

The script calculates the estimated time of death by applying Newton's Law of Cooling, which describes the rate of change of the temperature of an object. It defines two constants: `K`, which is the cooling constant specific to the situation, and `Ta`, the ambient temperature. The final temperature `T_final` is set to 17°C, which is presumably the measured temperature of the corpse at the time of discovery.

To find the time since death (`t`), the script uses an algebraic rearrangement of the law of cooling's differential equation, which has been integrated and solved for `t`. The rearrangement involves taking the natural logarithm of the ratio of the difference between the final temperature of the body and the ambient temperature to the difference between the body's temperature at death (assumed to be 37°C) and the ambient temperature. This ratio is then divided by the negative product of the cooling constant `K` and

the constant `9.5`, which comes from the original algebraic manipulation related to the specific problem setup.

Finally, the script displays the estimated time since death by outputting the value of `t` in hours. The calculation assumes that the body was initially at human body temperature (37°C) and cooled to `T_final` (17°C) according to the exponential decay described by Newton's Law of Cooling.

Error Estimate

We now gather our values we got when solving the problem by hand, and the value gotten from the code in Listing [13]:

$$t = 1.16560547\ hr$$

$$t = 1.2268\ hr$$

With these values, our estimated error is calculated as:

$$\varepsilon_e = |Exact\ Value - Estimated\ Value|$$

$$\varepsilon_e = |1.16560547 - 1.2268\,| = 0.06119453$$

Conclusion

In the project, various mathematical and computational techniques are applied to model and solve a range of problems, from interpolating data to estimating decay rates and solving ordinary differential equations.

Problem 1 utilizes Newton's and Lagrange's interpolation methods to estimate unknown values based on known data points. These polynomial-based approaches provide a means to approximate the behavior of data between the known points, which can be crucial for predictions or understanding underlying trends.

Problem 2 delves into piecewise linear interpolation and higher-order polynomial fits, such as fifth-degree polynomials, to model data more accurately. These techniques are beneficial when the data exhibit non-linear patterns that simple linear models cannot capture.

Problem 3 employs spline interpolation, a powerful tool for creating a smooth curve through a given set of points. Unlike polynomials, which can oscillate wildly between points, splines provide a piecewise polynomial function that can conform more naturally to the shape of the data.

Problem 4 addresses the calculation of forces on a sailboat mast. The method involves integrating a function over the mast's height to find the total force exerted. This problem applies numerical integration techniques like the trapezoidal rule and Simpson's rule to approximate the integral, providing insights into the distribution of forces and their effect on the mast's stability.

Finally, Problem 5 tackles the estimation of the time of death using Newton's law of cooling. The problem is approached by solving a differential equation that models the cooling of a body over time. This involves both analytical solutions and numerical methods to predict the temperature change and thus, the time since death.

Throughout the project, error estimation is a recurring theme. It involves comparing the results from numerical methods with exact or previously known values to assess the accuracy and reliability of the models and solutions provided. This critical evaluation is essential for validating the methods used and understanding the potential limitations of the approaches.

The project not only showcases the application of various numerical methods to solve practical problems but also emphasizes the importance of understanding the underlying mathematics to make informed decisions about which method to use and how to evaluate its effectiveness.

Appendix

1 – Script for Newton Interpolation

```
function [b, yint] = Newtint2(x, y, xx)
% Newtint2(x, y, xx):
% Newton interpolation. Uses an (n-1)-order Newton
% interpolating polynomial based in n data points (x, y)
% to evaluate the interpolated values of the dependent variable (yint)
% at selected locations, xx.
% input:
% x = independent variable
% y = dependent variable
% xx = values of independent variable at which interpolation is calculated
% output:
% yint = interpolated values of dependent variable
%
% compute the finite divided differences in the form of a difference table
n = length(x);
% check the table size
if length(y) ~= n, error('x and y must be same length'); end
b = zeros(n,n);
% assign dependent variables to the first column of b.
b(:,1) = y(:); % the (:) ensures that y is a column vector.
for j = 2:n
    for i = 1:n-j+1
        b(i,j) = (b(i+1,j-1) - b(i,j-1))/(x(i+j-1) - x(i));
    end
end

% use the finite divided differences to interpolate
```

```
yint = zeros(size(xx));
for k = 1:length(xx)
    xt = 1;
    yint(k) = b(1,1);
    for j = 1:n-1
        xt = xt*(xx(k) - x(j));
        yint(k) = yint(k) + b(1,j+1) * xt;
    end
end

% Plotting part begins here
% Define a dense range of points for plotting the polynomial
x_dense = linspace(min(x), max(x), 100);
y_dense = zeros(size(x_dense));

% Calculate the interpolated values for the dense range
for k = 1:length(x_dense)
    xt = 1;
    y_dense(k) = b(1,1);
    for j = 1:n-1
        xt = xt * (x_dense(k) - x(j));
        y_dense(k) = y_dense(k) + b(1,j+1) * xt;
    end
end

% Plot the original data points
plot(x, y, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8);
hold on; % Hold on to plot the polynomial on the same figure
% Plot the interpolating polynomial
plot(x_dense, y_dense, 'b-', 'LineWidth', 2);
```

```
xlabel('Independent variable x');
ylabel('Dependent variable y');
title('Newton Interpolating Polynomial');
legend('Data Points', 'Interpolating Polynomial');
hold off; % Release the plot hold
end
```

2 – Lagrange Coefficient Script

```
function c = Lagrange_coef(x, y)
  n = length(x)
  for k = 1 : n
      d(k) = 1;
      for i = 1 : n
        if i ~= k
          d(k) = d(k)*(x(k) - x(i));
        endif

        c(k) = y(k)/d(k);
      endfor
  endfor
end
```

3 – Lagrange Eval Script

```
function p = Lagrange_Eval(t, x, c)


  m = length(x)
  for i = 1 : length(t)
    p(i) = 0;
    for j = 1 : m
      N(j) = 1;
      for k = 1 : m
        if (j ~= k)
          N(j) = N(j) * (t(i) - x(k));
        endif
      endfor
      p(i) = p(i) + N(j) * c(j);
    endfor
  endfor
end
```

4 – Piecewise Interpolation Script

```
% Given data points
T = [0, 8, 16, 24, 32, 40]; % Temperature in degrees Celsius
O = [14.621, 11.843, 9.870, 8.418, 7.305, 6.413]; % Oxygen concentration in
mg/L


% Fit piecewise linear interpolation
piecewiseLinear = @(T_query) interp1(T, O, T_query, 'linear');


% Estimate oxygen concentration at 27 degrees Celsius
```

```
O_at_27 = piecewiseLinear(27);


% Display the estimated oxygen concentration

fprintf('The estimated oxygen concentration at 27°C is %.3f mg/L\n',
O_at_27);


% Calculate the error at 27 degrees Celsius

exact_O_at_27 = 7.986; % Exact oxygen concentration at 27°C

error_at_27 = exact_O_at_27 - O_at_27; % Error calculation

fprintf('The error in the estimated value at 27°C is %.3f mg/L\n',
error_at_27);


% Plot the data and the piecewise linear interpolation

T_dense = linspace(min(T), max(T), 1000); % A dense range for a smooth plot

O_dense = piecewiseLinear(T_dense);


figure; % Create a new figure

plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Plot the data
points

hold on; % Hold the figure for the next plot

plot(T_dense, O_dense, 'b-'); % Plot the piecewise linear interpolation

plot(27, O_at_27, 'ks', 'MarkerFaceColor', 'g'); % Plot the estimated value
at 27°C


% Plot the exact value and the error line

plot(27, exact_O_at_27, 'p', 'MarkerFaceColor', 'm', 'MarkerSize', 8); % Plot
the exact value

plot([27, 27], [O_at_27, exact_O_at_27], 'k--'); % Plot a line representing
the error


% Update the legend to include the exact value and error line
```

```
legend('Data Points', 'Piecewise Linear Fit', 'Estimated Value at 27°C',
'Exact Value', 'Error at 27°C');


% Finalize the plot with labels, title, and grid

xlabel('Temperature (°C)');

ylabel('Oxygen Concentration (mg/L)');

title('Piecewise Linear Interpolation of Oxygen Concentration');

grid on; % Add a grid for better readability

hold off; % Release the plot hold
```

5 – Fifth-Grade Polynomial Script

```
% Given data points

T = [0, 8, 16, 24, 32, 40]; % Temperature in degrees Celsius

O = [14.621, 11.843, 9.870, 8.418, 7.305, 6.413]; % Oxygen concentration in
mg/L


% Fit a fifth-order polynomial to the data

p = polyfit(T, O, 5);


% Estimate oxygen concentration at 27 degrees Celsius

O_at_27 = polyval(p, 27);


% Display the estimated oxygen concentration

fprintf('The estimated oxygen concentration at 27°C using a fifth-order
polynomial is %.3f mg/L\n', O_at_27);


% Plot the data and the polynomial interpolation

T_dense = linspace(min(T), max(T), 1000); % A dense range for a smooth plot

O_dense = polyval(p, T_dense);
```

```
figure; % Create a new figure

plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Plot the data points

hold on; % Hold the figure for the next plot

plot(T_dense, O_dense, 'b-'); % Plot the polynomial interpolation

plot(27, O_at_27, 'ks', 'MarkerFaceColor', 'g'); % Plot the estimated value at 27°C

xlabel('Temperature (°C)');

ylabel('Oxygen Concentration (mg/L)');

title('Fifth-Order Polynomial Interpolation of Oxygen Concentration');

legend('Data Points', 'Polynomial Fit', 'Estimated Value at 27°C');

grid on; % Add a grid for better readability


% Print the exact value for comparison

fprintf('The exact oxygen concentration at 27°C is 7.986 mg/L\n');
```

6 – Splines Scripts

```
% Given data points

T = [0, 8, 16, 24, 32, 40]; % Temperature in degrees Celsius

O = [14.621, 11.843, 9.870, 8.418, 7.305, 6.413]; % Oxygen concentration in mg/L


% Fit a spline to the data

spline_fit = spline(T, O);


% Estimate oxygen concentration at 27 degrees Celsius using the spline

O_at_27_spline = ppval(spline_fit, 27);


% Display the estimated oxygen concentration from the spline
```

```
fprintf('The estimated oxygen concentration at 27°C using spline interpolation is %.3f mg/L\n', O_at_27_spline);


% Plot the data and the spline interpolation

T_dense = linspace(min(T), max(T), 1000); % A dense range for a smooth plot

O_dense_spline = ppval(spline_fit, T_dense);


figure; % Create a new figure

plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Plot the data points

hold on; % Hold the figure for the next plot

plot(T_dense, O_dense_spline, 'b-'); % Plot the spline interpolation

plot(27, O_at_27_spline, 'ks', 'MarkerFaceColor', 'g'); % Plot the estimated value at 27°C

xlabel('Temperature (°C)');

ylabel('Oxygen Concentration (mg/L)');

title('Spline Interpolation of Oxygen Concentration');

legend('Data Points', 'Spline Fit', 'Estimated Value at 27°C');

grid on; % Add a grid for better readability


% Print the exact value for comparison

fprintf('The exact oxygen concentration at 27°C is 7.986 mg/L\n');
```

7 – Decay Rate as a Function of Temperature Script

```
% Given data points
T = [6, 12, 18, 24, 30]; % Temperature in degrees Celsius
k = [0.15, 0.20, 0.32, 0.45, 0.70]; % Decay rates per day

% Given values for k_20 and theta
k_20 = 0.357;
theta = 1.067;

% Create a range of temperatures for plotting
T_range = linspace(min(T), max(T), 100);

% Calculate the decay rates using the model
k_model = k_20 * (theta .^ (T_range - 20));

% Plotting
figure; % Create a new figure
plot(T, k, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8); % Original data
points
hold on; % Hold the figure for the next plot
plot(T_range, k_model, 'b-'); % Model

% Adding details to the plot
xlabel('Temperature (°C)');
ylabel('Decay Rate (per day)');
title('Decay Rate as a Function of Temperature');
legend('Original Data', 'Model');
grid on; % Add a grid for better readability
```

```
% Display the plot
hold off;
```

8 – Composite Trapezoidal Script

```
function I = Trap(func, a, b, n)
  x = a;
  h = (b - a)/n;
  S = feval(func, a, n);
  for j = 1 : n - 1
    x = x + h;
    S = S + 2 * feval(func, x, n);
  endfor

  S = S + feval(func, b, n);
  I = (10)* ((b - a)*S / (2*n));
End
```

9- Simpson's $\frac{1}{3}$ Rule

```
function I = Simp(f, a, b, n)
% integral of f using composite Simpson rule
% n must be even
  h = (b - a)/n;
  S = feval(f,a, n);
  for i = 1 : 2 : n-1
      x(i) = a + h*i;
      S = S + 4*feval(f, x(i), n);
  end
  for i = 2 : 2 : n-2
      x(i) = a + h*i;
```

```
      S = S + 2*feval(f, x(i), n);
  end
  S = S + feval(f, b, n); I = 10*(h*S/3);
End
```

10 – Problem 5 Equation

```
function dTdt = problem5_equation(t, T)
  K = 0.4993;
  Ta = 20;
  dTdt = -K * (T - Ta);
end
```

11 – ODE Solve Commands

```
% Define the constants K and Ta
K = 0.4993;
Ta = 20;

% Final temperature at which we want to find the time (37°C assumed for body
temperature at time of death)
T_final = 17;

% Solve for time using the rearranged algebraic equation
% T - Ta = e^(-K * t * (T - Ta))
% Taking logarithms we get:
% ln((T_final - Ta) / (37 - Ta)) = -K * t * (9.5)
% t = ln((T_final - Ta) / (9.5)) / -K

t = log((T_final) / (29.5 - Ta)) / (-K * 9.5);
```

```
% Display the result
disp(['The estimated time of death is: ', num2str(t), ' hours']);
```

12 – Non-Linear Regression Script

```
% Clear the workspace and close all figures
clearvars
close all

% Define the dataset
Temperature = [6 12 18 24 30];
DecayRate = [0.15 0.20 0.32 0.45 0.70];

%% Linear Transformation and Regression
% Transform the non-linear equation to a linear form
% By taking logarithms: log(k) = log(k20) + (T-20)*log(theta);

% Prepare data for linear regression
LogDecayRate = log(DecayRate);
TempForRegression = Temperature;
% Perform linear regression to obtain slope and intercept
coefficients = polyfit(TempForRegression, LogDecayRate, 1);

% Calculate the original parameters from the linear coefficients
theta_estimated = exp(coefficients(1));
k20_estimated = exp(coefficients(2) + 20 * log(theta_estimated));

% Generate a range for temperature and calculate the predicted decay rate
TempRange = linspace(min(Temperature), max(Temperature), 400);
```

```
PredictedDecay = k20_estimated * theta_estimated .^ (TempRange - 20);
EstimatedDecayAt17 = k20_estimated * theta_estimated .^ (17 - 20);

%% Non-linear Regression Analysis

% Objective function for non-linear regression
functionToMinimize = @Sum_Squares_Residuals;

% Optimize parameters using non-linear regression
optimizedParams = fminsearch(functionToMinimize, [1,1], [], Temperature,
DecayRate);

% Generate a range for temperature and calculate the model decay rate
TempRange_NLR = linspace(min(Temperature), max(Temperature), 400);
DecayRateModel = optimizedParams(1) * optimizedParams(2) .^ (TempRange_NLR -
20);
DecayRateAt17 = optimizedParams(1) * optimizedParams(2) .^ (17 - 20);

%% Plot Results
% Create a figure to display the data
figure
plot(Temperature, DecayRate, 'ko', 'MarkerFaceColor', 'k')
hold on
plot(TempRange, PredictedDecay, 'r-')
plot(TempRange_NLR, DecayRateModel, 'b--')
plot(17, EstimatedDecayAt17, 'rs', 'MarkerSize', 10)
plot(17, DecayRateAt17, 'bs', 'MarkerSize', 10)
grid on

% Annotate the figure
```

```
xlabel('Temperature (°C)')
ylabel('Decay Rate (per day)')
legend('Experimental Data', 'Linearized Fit', 'Non-linear Regression', ...
       'Linear Estimate at 17°C', 'Non-linear Estimate at 17°C', ...
'Location', 'Best')
```

13 – Solve Problem 5 Script

```
% Define the constants K and Ta
K = 0.4993;
Ta = 20;

% Final temperature at which we want to find the time (37°C assumed for body
temperature at time of death)
T_final = 17;

% Solve for time using the rearranged algebraic equation
% T - Ta = e^(-K * t * (T - Ta))
% Taking logarithms we get:
% ln((T_final - Ta) / (37 - Ta)) = -K * t * (9.5)
% t = ln((T_final - Ta) / (9.5)) / -K

t = 10*(log((T_final) / (29.5 - Ta)) / (-K * 9.5));

% Display the result
disp(['The estimated time of death is: ', num2str(t), ' hours']);
```