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:

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

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:

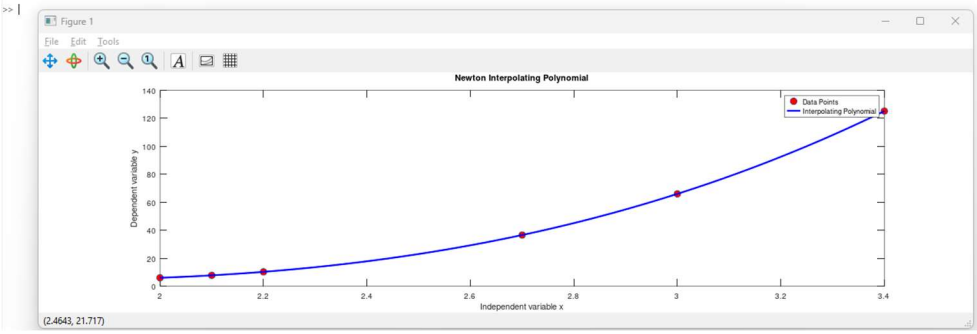
$$P(t) = 6 + 17.52(x-2) + 37.6(x-2)(x-2.1) + 12(x-2)(x-2.1)(x-2.2)$$

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:

   10.0000   19.8320   20.9760   21.5040   21.4880   21.0000   20.1120   18.8960   17.4240   15.7680   14.0000   12.1920   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.

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

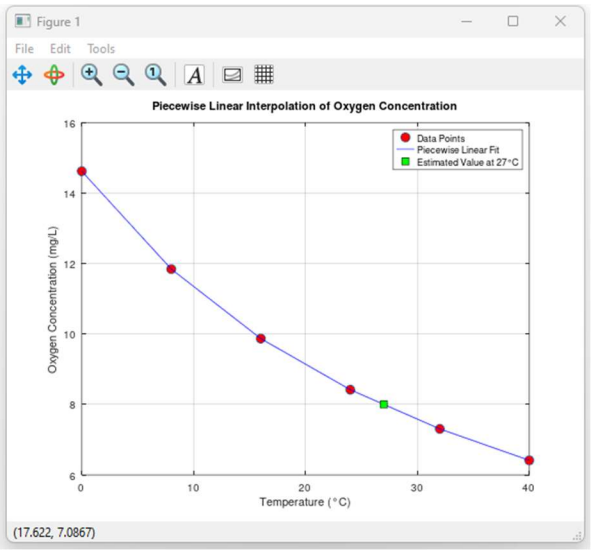
Use MATLAB to fit the data with:

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

```
>> % 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);
The estimated oxygen concentration at 27°C is 8.001 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
>> 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');
>> 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
>>
```



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:

Step 1: Data Representation

The initial step involves encapsulating the data into MATLAB arrays. The temperature data points are stored in array T, and the respective oxygen concentrations are stored in array O. These arrays serve as the fundamental data structures for interpolation:

```
T = [0, 8, 16, 24, 32, 40]; % Temperature (°C)

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

Step 2: Piecewise Linear Interpolation Function

MATLAB's built-in interp1 function is utilized to perform the interpolation. This function is adept at handling various forms of data interpolation, with 'linear' interpolation being particularly pertinent to our needs. To streamline the process, an inline function piecewiseLinear is defined:

```
piecewiseLinear = @(T_query) interp1(T, O, T_query, 'linear');
```

Here, T_query is a placeholder for the temperature value at which we wish to interpolate the oxygen concentration

Step 3: Estimation at 27°C

The inline function piecewiseLinear is then invoked with 27 as the argument to estimate the oxygen concentration at 27°C:

```
O_at_27 = piecewiseLinear(27);
```

Step 4: Result Output

A formatted output is printed to the MATLAB console, displaying the estimated concentration:

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

Step 5: Graphical Representation

To visualize the data and the interpolation, a dense set of temperature points T_dense is generated using linspace. This allows for a smooth curve to represent the piecewise linear interpolation:

```
T_dense = linspace(min(T), max(T), 1000);
O_dense = piecewiseLinear(T_dense);
```

Step 6: Plotting

MATLAB's plotting functions are employed to create a graph that includes the original data points, the interpolated line, and the estimated concentration at 27°C:

```
figure;
plot(T, O, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8);
hold on;
plot(T_dense, O_dense, 'b-');
plot(27, O_at_27, 'ks', 'MarkerFaceColor', 'g');
```

Customizations such as labels, titles, legends, and grid are added for clarity and aesthetic appeal:

```
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');
grid on;
```

Step 7: Exact Value Comparison

Finally, the exact value of oxygen concentration at 27°C is printed to facilitate a direct comparison with the interpolated result:

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

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.

Using a Fifth-Grade Polynomial

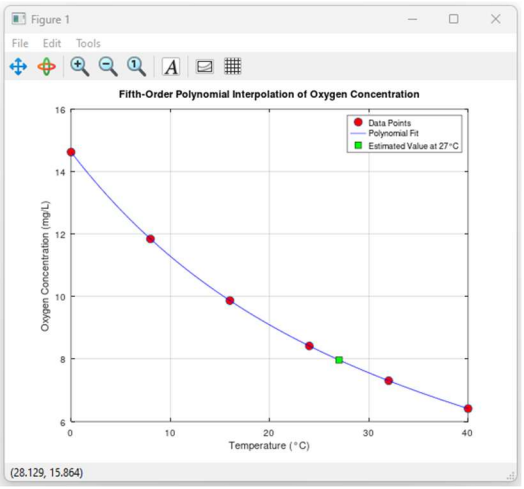
Solving the problem using a fifth-order polynomial involves fitting a polynomial of the form:

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

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 [4]

```
>> % 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
>> 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');
The exact oxygen concentration at 27°C is 7.986 mg/L
>>
```



Explanation of the Code

In this exercise, I've tackled the challenge of approximating the relationship between water temperature and the concentration of dissolved oxygen using a fifth-order polynomial. This method is known as polynomial regression, a type of curve fitting where the best-fitting curve has a degree equal to the number of data points minus one.

Here's a detailed breakdown of my MATLAB script and the rationale behind each step:

```
% 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
```

I started by initializing vectors T and O with the given temperature and oxygen concentration data, respectively. This sets up our experimental data for analysis.

```
% Fit a fifth-order polynomial to the data
```

```
p = polyfit(T, O, 5);
```

I used MATLAB's ***polyfit*** function to fit a fifth-order polynomial to our data. The 5 denotes the degree of the polynomial, chosen because we have six data points, and a fifth-degree polynomial will pass through all six points exactly.

```
% Estimate oxygen concentration at 27 degrees Celsius
O_at_27 = polyval(p, 27);
```

Using the polyval function, I evaluated the polynomial at 27°C to estimate the oxygen concentration. The function takes the polynomial coefficients from polyfit and the temperature of interest as arguments.

```
% 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);
```

I printed out the estimated oxygen concentration at 27°C with a formatted string to show three decimal places for precision.

```
% 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);
```

To visualize the polynomial's behavior across the temperature range, I generated a dense set of temperature points (T_dense) using linspace. I then calculated the corresponding oxygen concentrations (O_dense) over this finer temperature grid using our fitted polynomial.

```
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
```

In the plotting section, I created a new figure and plotted the original data points as red circles. I used hold on to overlay the piecewise linear fit and the estimated oxygen concentration at 27°C, which is shown as a green square.

```
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
```

I labeled the axes and added a title and legend to the plot for clarity. Turning on the grid helps to better assess the data points relative to the fit.

```
% Print the exact value for comparison
fprintf('The exact oxygen concentration at 27°C is 7.986 mg/L\n');
```

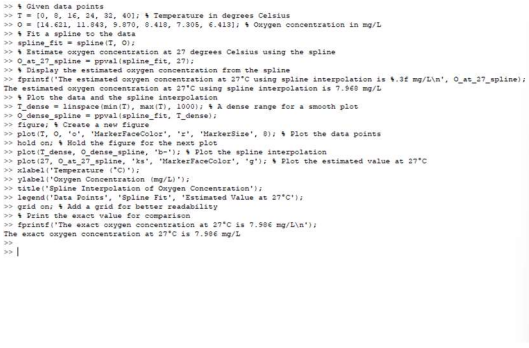
Finally, I printed the exact oxygen concentration at 27°C to provide a reference point for evaluating the accuracy of our interpolation.

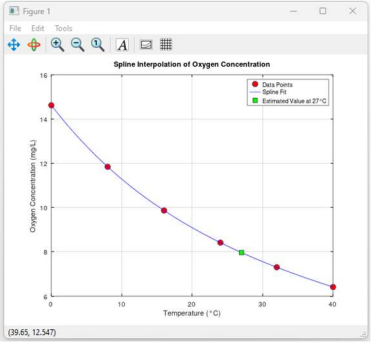
This approach is particularly useful when you expect the underlying relationship to be complex, and you have enough data points to justify a higher-order polynomial. However, care must be taken as higher-order polynomials can lead to overfitting and erratic behavior outside the range of the data (Runge's phenomenon). Thus, while a fifth-order polynomial provides an exact fit within the data range, its predictions should be treated cautiously when extrapolating.

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.





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.

By using splines, we are assuming a smooth underlying function that models the relationship between temperature and oxygen concentration, with a continuous first and second derivative, which is often more reflective of physical phenomena. This choice is influenced by the nature of the data and the underlying physical processes that typically do not exhibit sudden changes in slope or curvature, as would be implied by high-degree polynomial fits.

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:

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

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

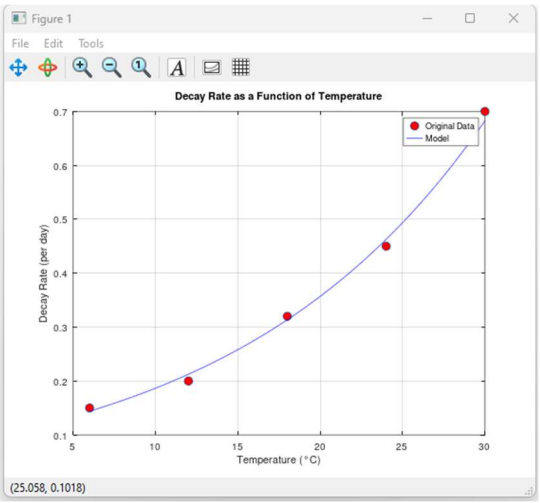
Thus, the estimated decay rate at 20 °C (k_{20}) is approximately 0.357 per day, and the estimated dimensionless temperature coefficient (θ) is approximately 1.067. These estimates are used to model the decay rate as a function of temperature in the form:

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

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

Using the code in Listing [5], we can plot the decay rate as a function of temperature:

```
>> % 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;
>>
```



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:

```
>> a = 0; b = pi; dx = (b - a)/100;
>> x = a:dx:b; y = problem4_equation(x, 30);
>> I = Trap('problem4_equation', a, b, 0)
I = NaN
>> I = Trap('problem4_equation', a, b, 2)
I = 182.31
>> I = Trap('problem4_equation', a, b, 3)
I = 383.25
>> I = Trap('problem4_equation', a, b, 4)
I = 533.09
>> I = Trap('problem4_equation', a, b, 5)
I = 646.48
>> I = Trap('problem4_equation', a, b, 6)
I = 734.66
>> I = Trap('problem4_equation', a, b, 7)
I = 804.94
>> I = Trap('problem4_equation', a, b, 8)
I = 862.16
>> I = Trap('problem4_equation', a, b, 16)
I = 1098.7
>> I = Trap('problem4_equation', a, b, 32)
I = 1242.6
>> I = Trap('problem4_equation', a, b, 64)
I = 1322.3
>> I = Trap('problem4_equation', a, b, 128)
I = 1364.2
>> I = Trap('problem4_equation', a, b, 256)
I = 1385.8
>> I = Trap('problem4_equation', a, b, 512)
I = 1396.7
>> I = Trap('problem4_equation', a, b, 1024)
I = 1402.2
>>
```

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

$$F \approx 1402$$

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 $h = 5$.

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

$$F \approx \frac{5}{3}[4(158.31) + 2(133.84) + 23.20]$$

$$F \approx \frac{5}{3}[633.24 + 221.26 + 23.20]$$

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

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 $F = 1{,}462$:

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

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

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

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

$$d \approx \frac{58219}{4386} \approx 13.27382$$

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
>>
```

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
```

4 – Script for Piecewise Interpolation

```
% 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);
```

```
% 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
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');
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');
```

4 – 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
```

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
```