Computational Methods Project

Thais Isabelle Parron Ruiz

EML3041 Spring 2025

Table of Contents

Introduction

For this project, I'm going to apply the concepts I learned in class using MATLAB. In real-world situations, the calculations and methods we studied are often too complex or time-consuming to do by hand. This project will help me practice solving real-world problems by combining what I learned in class with the computational tools available in MATLAB.

Part #1: Chapters 1-3

Chapter 1

For part 1 of the project we are asked to solve a flids mechanics problem using the concepts we learned in class with the help of MATLAB. The first task is to find the theoretical velocity at each radial position given in the table by using the Hagen-Poiseuille equation:

$$u(r) = \Delta P \frac{R^2}{4\mu L}(1 - \frac{r^2}{R^2})$$

To complete this task the code below was used:

```
n = size_raidio(1,2);
for i = 1:n
    velocity(1,i) = delta_p*((pipe_radious^2)/(4*viscocity_air*pipe_legth))*(1-(radio_position(1,i)^2)/(pipe_radious^2));
end
velocity;
```

Figure 1: Chapter 1, first task code

All the values used in the equation were given, and the for loop is used to create the velocity matrix for the different radio positions given.

For the second task of Chapter 1, I was asked to convert the given binary values to base 10. To complete this part, I used the following code.

```
size_binary_int = size(data_binary_int);
m = size_binary_int(1,1);
l = size_binary_int(1,2);
new_base_int = zeros(1,m);
for j = 1:m
    for k = 1:l
        term(1,k) = data_binary_int(j,k)*2^(l-k);
        new_base_int(1,j) = new_base_int(1,j) + term(1,k);
    end
end
new_base_int = new_base_int(1,1:m);

size_binary_dec = size(data_binary_dec);
p = size_binary_dec(1,1);
w = size_binary_dec(1,2);
new_base_dec = zeros(1,p);
for s = 1:p
    for d = 1:w
        term(1,d) = data_binary_dec(s,d)*2^(-d);
        new_base_dec(1,s) = new_base_dec(1,s) + term(1,d);
    end
end
new_base_dec = new_base_dec(1,1:p);
new_base = new_base_dec + new_base_int;
```

Figure 2: Chapter 1, second task code

To complete this part, I followed the same logic used when solving the problem by hand. First, I converted the integers to base 10 by organizing the binary numbers into a matrix and analyzing each one to find its base 10 equivalent. I used the same thought process to convert the decimal parts. Once both conversions were complete, I combined the two matrices, one for the integers and one for the decimals, to find the final values.

For the third task of this chapter, I was asked to find the absolute true error, expressed as a percentage, between the theoretical velocity calculated in the first task, considered the true value, and the base-10 value obtained from the binary conversion in the previous task. The code used to complete this part is the following.

```
for f = 1:11
    true_error(1,f) = abs(velocity(1,f) - new_base(1,f));
end
relative_true_error = (true_error./velocity)*100;
```

Figure 3: Chapter 1, third task code

For this part, I calculated the true error by creating a matrix of the true error values between all the points and then used this matrix to find the relative true error. Instead of using another for loop to build the answer matrix, I used element-wise operations by adding a dot (.) in MATLAB, which allowed the division to be done element by element and automatically generated the final matrix without needing a loop. The following table summarizes all of the values found on the previous steps for this chapter:

Table 1: Chapter 1 results

Radial Position [m]	Theoretical velocity [m/s]	Base-10 values [m/s]	Relative true error
-0.20	0.0000	1.2187500	N/A
-0.16	49.641	57.062500	14.949236
-0.12	88.252	85.812500	2.7637109
-0.08	115.83	116.21875	0.3355208
-0.04	132.38	134.62500	1.6979687
0.00	137.89	138.34375	0.3268875
0.04	132.38	128.75000	2.7401042
0.08	115.83	113.93750	1.6339583
0.12	88.252	87.718750	0.6036914
0.16	49.641	55.531250	11.864618
0.20	0.0000	3.4375000	N/A

After finding all the values, I created a plot of the velocities using the Hagen-Poiseuille equation, considering r values from -0.2 to 0.2 with increments of 0.001. I first calculated the velocity for each radius value, then plotted the results to visualize how the velocity changes

across the pipe's cross-section. On the same plot, I also included the curve for the base-10 values obtained from the given binary numbers. The resulting plot is shown below:

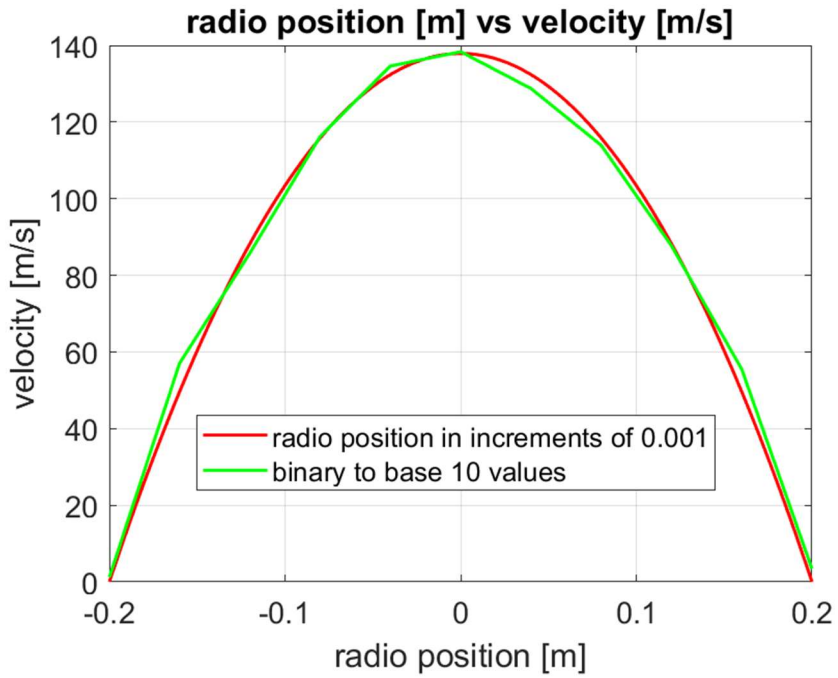

Figure 4: Chapter 1, plot for task 5

For the next part, I created a plot for the relative true error found in task 3 for this chapter:

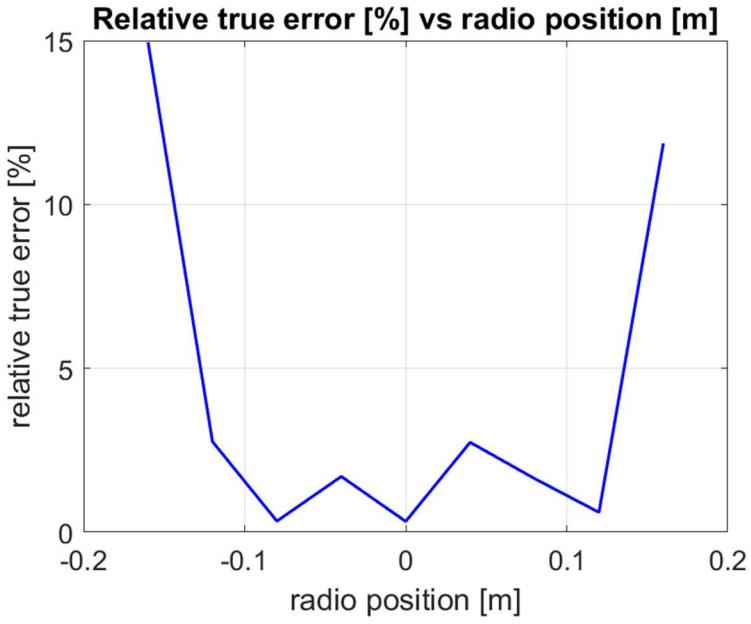

Figure 5: Chapter 1, plot for task 6

The experimental values don't match the theoretical values due to several factors. One major reason is the assumptions made when deriving the formula, such as steady, fully developed, and laminar flow, which aren't fully true under real-world conditions. These assumptions become even less accurate near the walls of the pipe, where disturbances and other effects are more significant, causing the relative true error to increase as we get closer to the walls. Additionally, the limited number of bits introduces even more errors because the values must be rounded, leading to what is known as round-off errors. This rounding makes the results even less accurate.

Chapter 2

For the first task of Chapter 2, I was asked to write a code that estimates the shear stress using central, forward, and backward divided difference methods, preferably using the central difference due to its higher accuracy. The formula for shear stress was given as:

$$\tau = \mu \frac{\partial u}{\partial r}$$

To do this in MATLAB, I used the base-10 values obtained from the binary conversions in the previous chapter. The code for this part is shown below:

```
q = size(new_base);
q1 = q(1,2);
for z = 1:q1
    if z == 1
        h(1,z) = abs(abs(radio_position(1,z))-abs(radio_position(1,z+1)));
        dif(1,z) = (new_base(1,z+1)-new_base(z))/h(1,z);
    elseif z == q1
        h(1,z) = abs(abs(radio_position(1,z))-abs(radio_position(1,z-1)));
        dif(1,z) = (new_base(1,z)-new_base(1,z-1))/h(1,z);
    elseif abs(abs(radio_position(1,z)) - abs(radio_position(1,z-1))) - abs(abs(radio_position(1,z))-abs(radio_position(1,z+1))) < 10^(-10)
        h(1,z) = abs(abs(radio_position(1,z)) - abs(radio_position(1,z-1)));
        dif(1,z) = (new_base(1,z+1)-new_base(1,z-1))/(2*h(1,z));
    elseif abs(abs(radio_position(1,z)) - abs(radio_position(1,z-1))) > abs(abs(radio_position(1,z))-abs(radio_position(1,z+1)))
        h(1,z) = abs(abs(radio_position(1,z))-abs(radio_position(1+z)));
        dif(1,z) = (new_base(1,z+1)-new_base(1,z))/h(1,z);
    elseif abs(abs(radio_position(1,z)) - abs(radio_position(1,z-1))) < abs(abs(radio_position(1,z))-abs(radio_position(1,z+1)))
        h(1,z) = abs(abs(radio_position(1,z))-abs(radio_position(1,z-1)));
        dif(1,z) = (new_base(1,z)-new_base(1,z-1))/h(1,z);
    end
    shear_stress(1,z) = viscocity_air*dif(1,z);
end
shear_stress;
```

Figure 6: Chapter 2, first task code

In this code, I used if loops to check which divided difference method was the best to use at each point to find the shear stress, and a for loop was used to make sure that all of the points were addressed.

The next task was to create a plot showing the found shear stresses as a function of radial position. The plot created can be seen below.

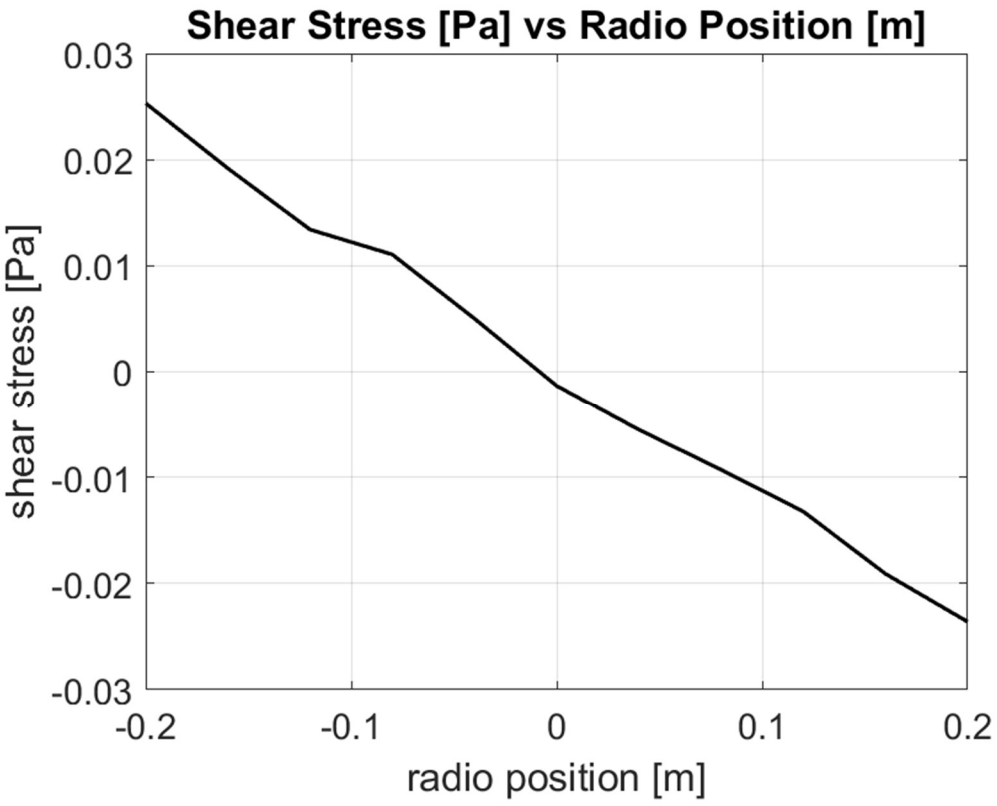

Figure 7: Chapter 2, second task plot

Based on this estimate, the shear stress is the largest at a radio position of -0.2, having a magnitude of 0.0253.

Chapter 3

For the first task of this report, I was asked to write a code that applies the bisection method to find the radial position corresponding to the velocities provided. The specifications were that the code needed to continue running until the relative approximate error dropped below 0.00005%, and that I had to keep track of the number of iterations performed during the process. To complete the task, I used the theoretical velocity equation and set up the bisection method accordingly, making sure the code updated the error and iteration count at each step. The final code for this part is shown below:

```
found_root = zeros(1,3);
real_aprox_error = [100 100 100];
x_high = [0.2 0.2 0.2];
x_low = [0 0 0];
a = 0;
b = 0;
c = 0;
for v = 62.1
        while real_aprox_error(1,1) >= 0.00005
            a = a + 1;
            func_high = delta_p*((pipe_radious^2)/(4*viscocity_air*pipe_legth))*(1-(x_high(1)^2)/(pipe_radious^2))-v;
            func_low = delta_p*((pipe_radious^2)/(4*viscocity_air*pipe_legth))*(1-(x_low(1)^2)/(pipe_radious^2))-v;
            if func_high*func_low < 0
                x_mean = (x_high(1)+x_low(1))/2;
                func_mean = delta_p*((pipe_radious^2)/(4*viscocity_air*pipe_legth))*(1-(x_mean^2)/(pipe_radious^2))-v;
                if func_mean*func_high < 0
                    real_aprox_error(1) = abs((x_mean-x_low(1))/x_mean)*100;
                    x_low(1) = x_mean;
                elseif func_mean*func_low < 0
                    real_aprox_error(1) = abs((x_mean-x_high(1))/x_mean)*100;
                    x_high(1) = x_mean;
                elseif func_mean*func_low == 0 || func_mean*func_high == 0
                    found_root(1) = x_mean;
                    real_aprox_error = 0;
                    break
                end
            else
                disp('bad assumption');
            end
        end
end
found_root(1) = x_mean;
```

Figure 8: Chapter 3, first task code

The code above was repeated for the different velocities given. The while loop was included to keep track of the approximate error so that the code would only stop running once the error was below the required value. The variable "a" takes care of the number of iterations for this specific velocity value. The if functions are used to check if the low or high values are going to be substituted by the found mean value. Additionally, a break function was also added in case the root is found to make the code stop running at that point. The table below shows the results of the code shown above.

Table 2:Results for Chapter 3

Final r values [m]	Number of Iterations	Velocity at r [m/s]
0.14827685	22	62.100023
0.18703370	22	17.300037
0.19941168	21	0.81006120

The bisection method is a much slower method compared to others, and it heavily depends on the initial guesses because you need to properly bracket the root. Sometimes, depending on the shape of the function, you might think you didn't bracket the root correctly even if you did. On the other hand, the Newton-Raphson method is much faster and doesn't rely as much on having perfect initial guesses. Because of its faster convergence and less dependency on bracketing, Newton-Raphson is often preferred over bisection.

Part #2: Chapters 4-6

Chapter 4

The second part of this project focuses on heat transfer. For the first task of this chapter, I was asked to transform the values given in the following matrix:

$$Ax = B$$

$$\underset{A}{\begin{bmatrix} 0.015833 & -0.0075 & 0 \\ -0.0625 & 0.070833 & -0.00833 \\ 0 & -0.1 & 0.1625 \end{bmatrix}} \quad \underset{x}{\begin{bmatrix} T1 \\ T2 \\ T3 \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} 6.667 \\ 0 \\ 0 \end{bmatrix}}$$

In the second task this matrix was solved in MATLAB using the following code:

```
X = A\B;
T1 = X(1);
T2 = X(2);
T3 = X(3);
L1 = 0.03;
L2 = 0.05;
L3 = 0.06;
L4 = 0.07;
tch4 = [t(1) T1 T2 T3 t(5)];
lch4 = [l(1) L1 L2 L3 L4];
```

Figure 9: Chapter 4, second task code

The first line of the code is the one that solves the equation and the following lines are just for organization, arranging the temperature and length values into two matrices to help with the next parts. The values found in this task as shown in the table below.

Table 3: Chapter 4 task 2 results

T1	766.355
T2	728.971
T3	448.598

After finding these values, I was asked to create a scatter plot of the temperatures as a function of distance. The resulting graph is shown below:

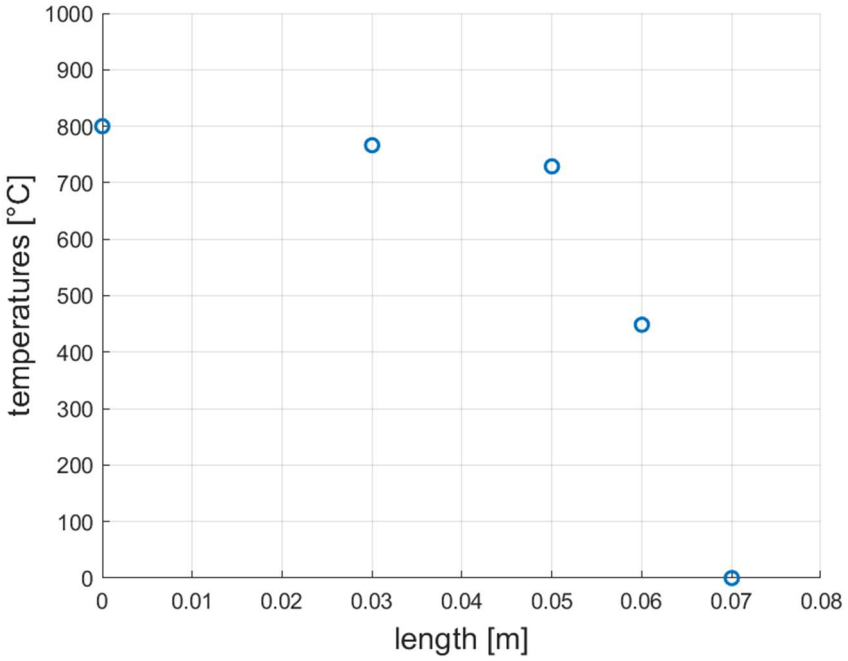

Figure 10: Chapter 4, fourth task plot

This problem could be solved by hand using either Gaussian elimination, LU decomposition, or by using the inverse matrix.

Gaussian elimination:

To do Gaussian elimination, you need to make all the elements below the main diagonal of the matrix equal to zero so you can solve the system using back substitution. This part is called Forward Elimination. You do it by taking a row and multiplying it by a certain factor so that, when you subtract it from the row below, one of the elements in the lower triangle becomes zero. You repeat this process for each pivot row, going down the matrix. This is done on both matrix A (the coefficients) and matrix B (the constants), since you're applying the same row operations to the entire system. Once the matrix is in upper triangular form, you can easily solve for the variables starting from the last row and working your way up.

LU decomposition:

To perform LU decomposition, you also need to use Forward Elimination, but this time you only apply it to matrix A. The goal is to break A into two matrices: L (lower triangular) and U (upper triangular). As you eliminate the lower triangle of A, you need to keep track of the values you're using to multiply the rows, that's because those values will form the entries of matrix L. L will have zeros above the diagonal, and U will be the result of the Forward

Elimination step, where all the elements below the diagonal are zero. To complete this method, you have to follow the steps below:

$$Ax = B \rightarrow A = LU \rightarrow LUx = B \rightarrow Lz = B \rightarrow Ux = z$$

Finding the Inverse:

This problem can also be solved by finding matrix H, which is the inverse of matrix A. When using this method, you must be careful with the order of the matrices during multiplication, since matrix multiplication is not commutative, changing the order can give you a completely different result. To use this method correctly, you just need to follow the equations shown below.

$$A = LU \rightarrow AH = I \rightarrow LUH = I \rightarrow Lz = I \rightarrow UH = z$$

After you find the inverse of matrix A, you multiply it by matrix B to get the values of matrix X.

Chapter 5

The order of the interpolant for this problem is fourth because five points were given. You can find the order by subtracting 1 from the total number of points, since the order of the polynomial is always one less than the number of data points used.

To find the matrices "A" and "B" from the formula $Ax = B$ for this problem, I used the code below and used the backslash function to solve the system of equations.

```
A2 = [0 0 0 0 1; L1^4 L1^3 L1^2 L1 1; L2^4 L2^3 L2^2 L2 1; L3^4 L3^3 L3^2 L3 1; L4^4 L4^3 L4^2 L4 1];
B2 = [t(1); T1; T2; T3; t(5)];
X2 = A2\B2;
v = 0;
for n = 0:0.000001:0.07
    v = v + 1;
    y_interp(1,v) = n^4*X2(1)+n^3*X2(2)+n^2*X2(3)+n*X2(4)+X2(5);
end
hold on
plot(0:0.000001:0.07, y_interp, 'LineWidth',2,'Color','k');
```

Figure 11: Chapter 5 Interpolant Code

The code above also includes the commands for the plot of the interpolant on top of the scatter graph created in the previous section. The resulting plot is the one shown below.

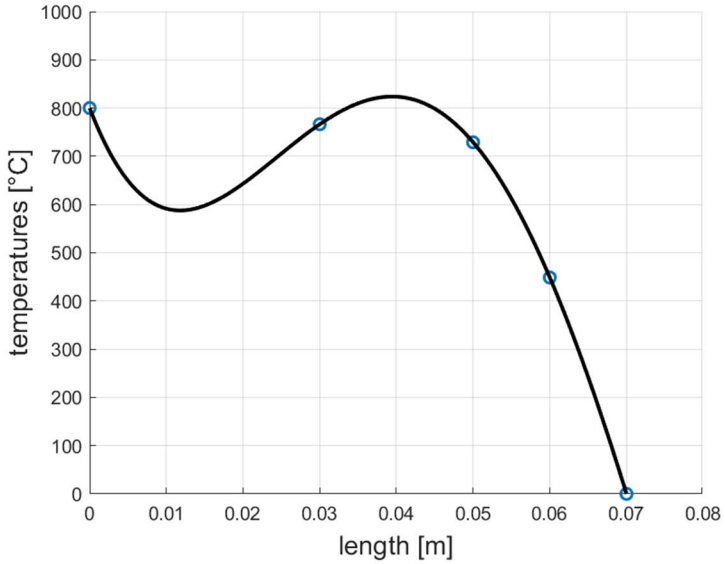

Figure 12: Chapter 5, Interpolant Plot

By analyzing the plotted interpolant above, it's clear that it's not a good representation of the actual function. The curve diverges too much, especially near the edges, which is a common issue with high-order polynomials. This happens because increasing the order to fit more points can cause the interpolant to swing wildly between them. So even though the interpolant passes through all the points, it doesn't accurately reflect the behavior of the original function.

For the next part, I derive the equations to connect the points using a quadratic spline. I did that by hand, so the image below shows all the work done to find the system of equations and its matrix representation.

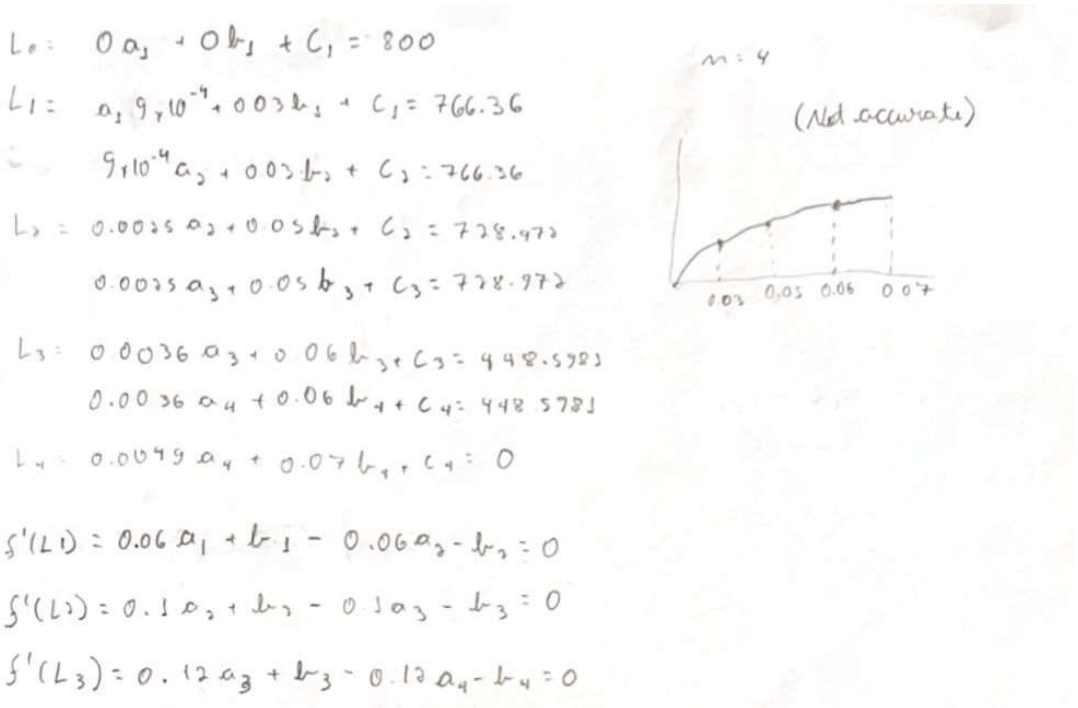

Figure 13: Chapter 5, Quadratic Spline Calculations

The final matrices in the form $Ax = B$ are the ones below

A

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9*10^{-4} & 0.03 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9*10^{-4} & 0.03 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.0025 & 0.05 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0025 & 0.05 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0036 & 0.06 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0036 & 0.06 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0049 & 0.07 & 1 \\ 0.06 & 1 & 0 & -0.06 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 1 & 0 & -0.1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.12 & 1 & 0 & -0.12 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

x

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \\ a_3 \\ b_3 \\ c_3 \\ a_4 \\ b_4 \\ c_4 \end{bmatrix}$$

$=$

B

$$\begin{bmatrix} 800 \\ 766.36 \\ 766.36 \\ 728.972 \\ 728.972 \\ 448.5981 \\ 448.5981 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

I found the coefficients by setting up the equations in MATLAB and using the backslash function, since it's an easy way to solve systems of equations with one line of code.

The next step was to plot the graph of the spline on top of the previous plots. The resulting figure is the following:

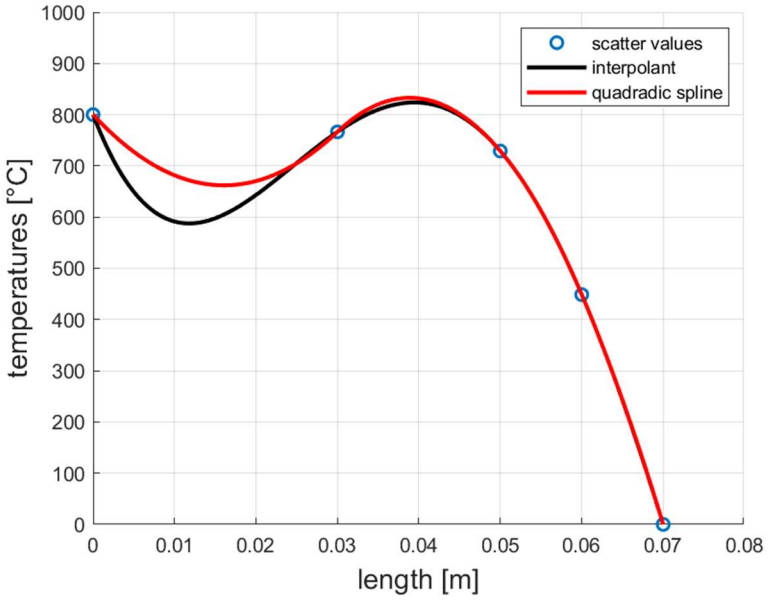

Figure 14: Chapter 5, Quadratic Spline Plot

This figure shows that using splines is a better way to approximate the behavior of a function compared to using an interpolant, especially when dealing with a large number of points. The quadratic spline stays much more stable and doesn't oscillate as much as the interpolant, making it a more accurate and reliable estimate.

However, for this specific problem, we know that conduction follows a linear behavior, so even the quadratic spline won't be a good approximation of the actual function. That's because it connects the points using quadratic curves, which don't really match how the function behaves.

Chapter 6

For this chapter the same values given and found in chapter 4 are used, but this time to find a quadradic regression model for this function. To find the coefficients of this model the following code was used.

```
t_ch6 = [t(1) T1 T2 T3 t(5)];
l_ch6 = [l(1) L1 L2 L3 L4];
x_sum = 0;
x_sqr_sum = 0;
x_cub_sum = 0;
x_4_sum = 0;
y_sum = 0;
yx_sum = 0;
yx2_sum = 0;
r = size(t_ch6);
for n = 1:5
    x_sum = x_sum + l_ch6(n);
    x_sqr_sum = x_sqr_sum + l_ch6(n)^2;
    x_cub_sum = x_cub_sum + l_ch6(n)^3;
    x_4_sum = x_4_sum + l_ch6(n)^4;
    y_sum = y_sum + t_ch6(n);
    yx_sum = yx_sum + t_ch6(n)*l_ch6(n);
    yx2_sum = yx2_sum + t_ch6(n)*l_ch6(n)^2;
end
A_ch6 = [r(2) x_sum x_sqr_sum; x_sum x_sqr_sum x_cub_sum; x_sqr_sum x_cub_sum x_4_sum];
B_ch6 = [y_sum; yx_sum; yx2_sum];
X_ch6 = A_ch6\B_ch6;
```

Figure 15: Chapter 6 Quadratic Regression Code

To check the values found in the code above, I wrote a second code using a MATLAB function called “polyfit,” which finds the coefficients of the quadratic regression for me in one line. After verifying that the results were the same, I created a plot for this model on top of the scatter points found in Chapter 4.

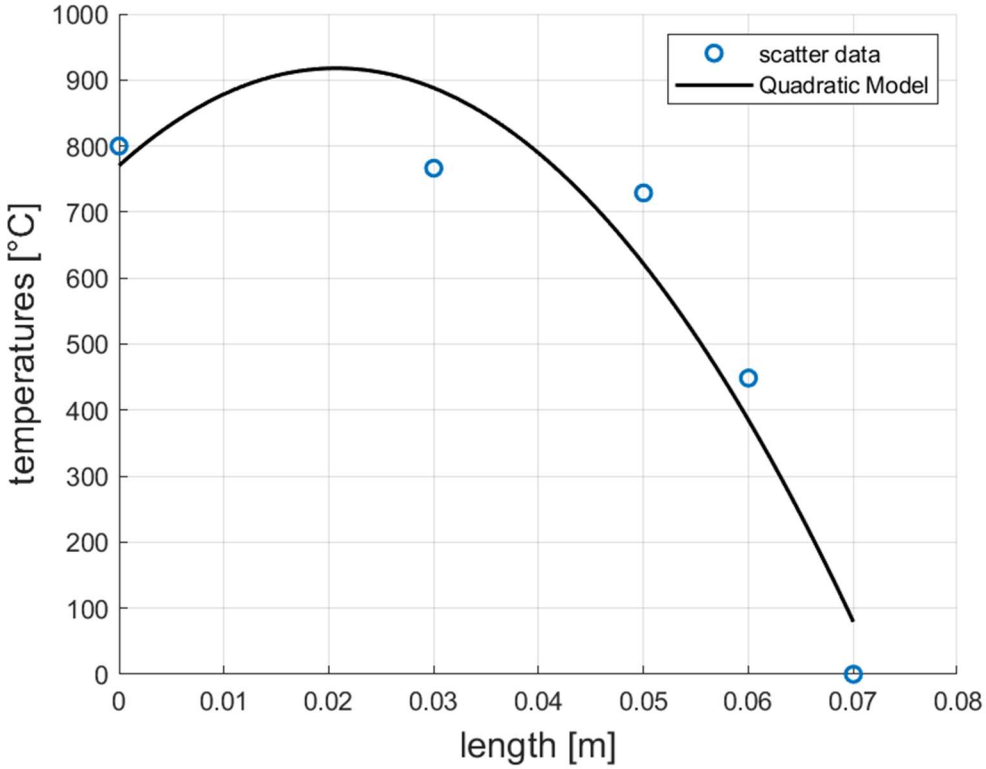

Figure 16: Chapter 6 Quadratic Regression Plot

After finding the quadratic model above, I used code to calculate the residuals for the plot. Then, using those values, I created the plot below showing the residuals as a function of distance.

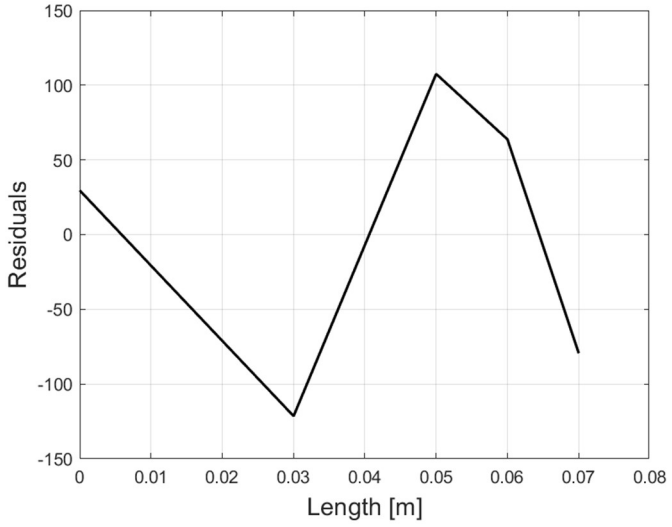

Figure 17: Chapter 6 Residuals Plota

Based on this plot, I believe the quadratic model is not a very good representation of the data because the residuals range from about 107 to -121. For the approximation to be accurate, the residuals should stay as close to zero as possible. The variation in the residuals happens because the regression model is trying to minimize the overall error across all the points. That means it sometimes overestimates the actual values, resulting in negative residuals, and sometimes underestimates them, which leads to positive residuals.

Chapter 7

For the first part of this chapter, I was asked to plot the function $k(x) = x^{-x}\ln(x + 1.2)$ from 0 to 4 in increments of 0.001 and the result is shown below.

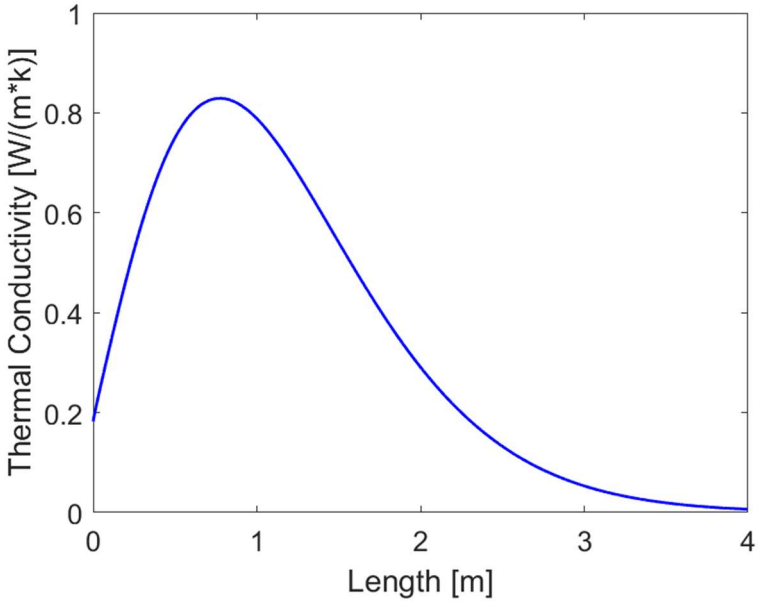

Figure 18: Chapter 7 First Task Plot

For the second task, I was asked to compute the true value of the integral from 0 to 4 using MATLAB. The code used is shown in the image below.

```
syms x
f = x^(-x)*log(x + 1.2);
f1 = int(f, x, 0, 4);
f2 = subs(f1,x,4);
f3 = double(f2);
```

Figure 19: Chapter 7 Second Task Code

After this part was completed, I was asked to write a code that finds how many trapezoids of equal width are required to have an absolute relative true error less than 0.001%, and the following image shows the code I used to complete this part.

```
R = 0;
c = zeros(30000,30000);
area = zeros(30000,30000);
final_value = 0;
error = 1;
while error >= 0.001
    R = R + 1;
    delta_x = (4-0)/R;
    for y = 1:R
        c(y+1) = c(y) + delta_x;
        c(y+2) = c(y+1) + delta_x;
        area(y) = (1/2)*(c(y+1)^(-c(y+1))*log(c(y+1)+1.2) + c(y+2)^(-c(y+2))*log(c(y+2)+1.2))*delta_x;
        final_value = final_value + area(y);
    end
    error = abs((f3-final_value)/f3)*100;
    final_value = 0;
end
```

Figure 20: Chapter 7 Third Task Code

And, with the code above, I found that the number of Trapezoids necessary to complete the requirements of this problem is 51489.

Conclusion

This project focused on applying the concepts we learned in class to real-world problems that mechanical engineers will likely face in their careers. It helped me see how the math and theory we practice in class show up in real engineering situations. I also learned how programming can make solving these problems a lot easier and faster, especially when dealing with complex calculations or large amounts of data. Completing this project also helped me get more comfortable with reading and interpreting graphs. I started to understand not just how to make them, but what they're telling me about the data. It also gave me a better sense of how each approximation method works and how the choice of method can affect the results. Some methods gave better estimates than others, depending on the shape of the data, which was interesting to see in action.