Computational Methods Final Project

Dexter Limcangco

EML3041 Summer 2024

Table of Contents

Introduction

Overview

This project will apply computational methods to solve engineering problems by transforming complex data into meaningful analysis and models. Please refer to the README.text in the Python repository for instructions on how to run the code and install any prerequisites.

Programming Language

For this project, I decided to use Python over MATLAB for several reasons:

- Open-Source: Unlike MATLAB, python is open-source which makes it easily and freely available without the need for expensive licenses, even though my status as a student allows me utilize MATLAB easily, I believe that developing data science skills in an open-source context will allow me to seamlessly transfer from educational environments to professional environments without the burden of licensing.

- Libraries: Python has a vast array of libraries that I was able to utilize for this project, however, to be fair, I made sure none of these libraries exceeded the functionality of MATLAB, and instead used libraries that replicated existing functions in MATLAB.

- Community Support: I found the online Python community an invaluable resource for troubleshooting; the sheer vastness of forums, tutorials, and community support was helpful for completing this project.

- Diversity of Developer Environments: While working on this project, I was often switching from my low-powered laptop to my desktop while commuting from home to school, by choosing Python, I was able to install a lightweight IDE, Visual Studio Code, on my laptop, while keeping my heavier IDE PyCharm on my desktop. I was able to synchronize settings through GitHub, making the transition between my devices seamless and optimize my productivity.

Part 1: Analysis of Electric Charge Over Time

Objective

Convert data from an old computer from binary to base-10, use this data to plot and understand how charge varies over time within a given electrical system.

Methodology

Firstly, we are given data in a text document. This data is electric charge in units of coulombs and is represented in 10-bit floated-point binary.

To work with this data, I used the 'open' function in Python to read the text file which contained the binary data. I then appended the data in a list, and used a function I defined earlier as "decimalBinaryToBaseTen(Binary)" (Appendix Figure 1), which converted all the decimal binary values into base-10. To convert integer values into base-10 I can simply use 'int(value,2)' function. However, this will still prove tricky as there each binary word contains the sign of the number, sign of the exponent, the magnitude of the exponent, and the mantissa.

To fully convert to base-10, I converted the data from the file from integers to strings, for ease of slicing and indexing. The exact Python code used can be seen in Appendix Figure 3

After converting all the values to base-10, I then used the library 'matplotlib' to plot the now converted data against time as shown in Figure 1.

Using these charge values, I then calculated the relative true error using a function I defined as "relative_true_error(TrueValue, ApproximateValue)" (Appendix Figure 2) and plotted these values in Figure 2.

Results

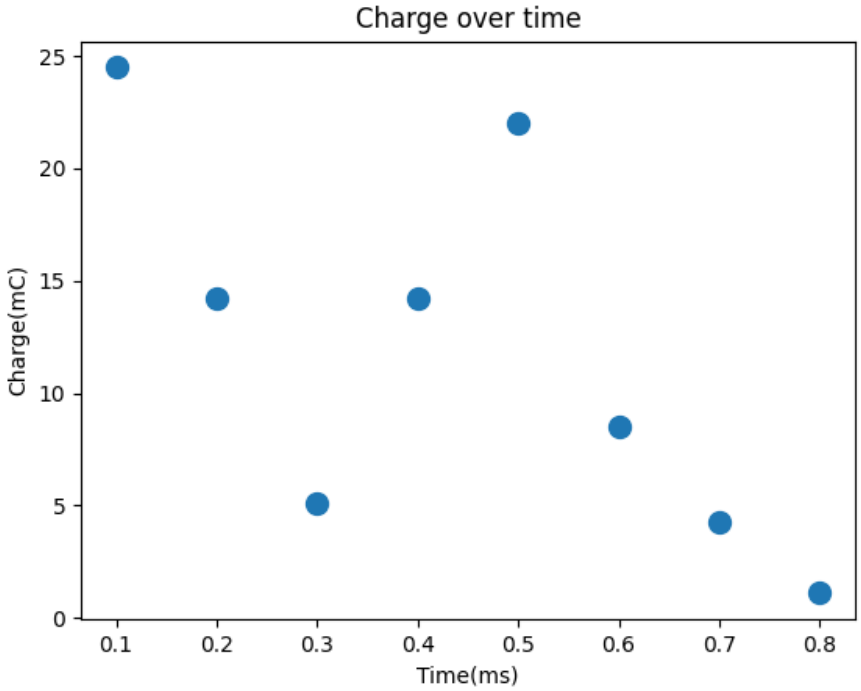

Figure 1: Scatterplot of the converted Charge values seen over time

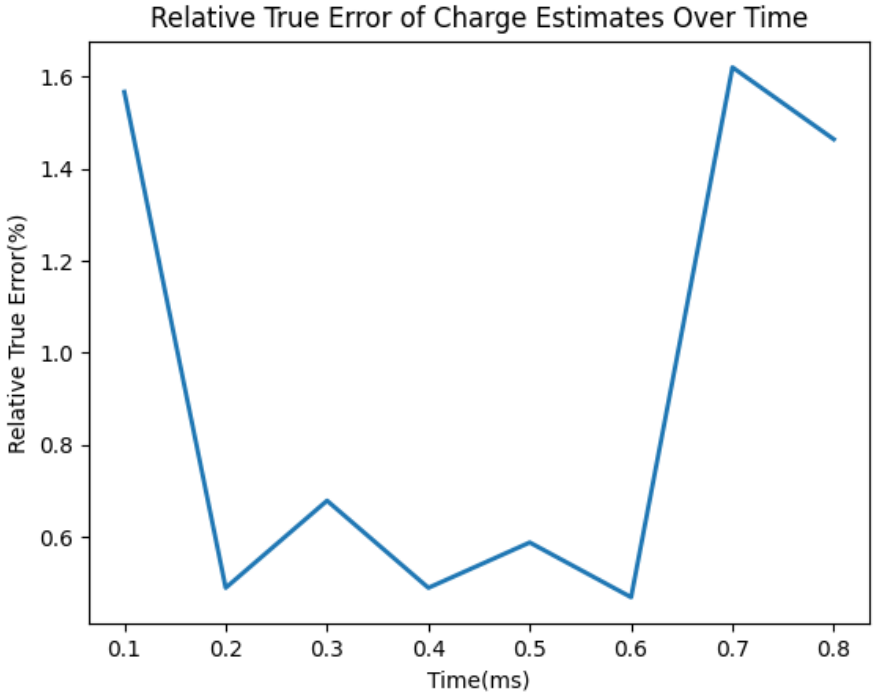

Figure 2: Line Graph of the variance of the Relative True Error

Analysis

As you can see in Figure 1 the true error is largest at 0.7 seconds.

These errors exist because of the limitations of old computers. This specific data was limited to 10 bits. This constraint reduces the precision of the values significantly.

However, there is a way you can improve accuracy. You can bias the system, this would dedicate another bit to the mantissa, which would increase the precision of the recorded values. Biasing achieves this by removing the bit from the sign of the exponent, therefore allowing more bits to represent the significant digits of the values.

Part 2: Calculation of Electrical Current

Objective

Derive the current from the charge using differentiation techniques

Methodology

I decided to use a combination of differentiation techniques to find the current at the specific time points.

Forward Difference Formula: $f'(x) = \frac{f(x+h)-f(x)}{h}$

Backward Difference Formula: $f'(x) = \frac{f(x)-f(x-h)}{h}$

Central Difference Formula: $f'(x) = \frac{f(x+h)-f(x-h)}{2h}$

Since the time points are evenly distributed, to be as accurate as possible I used the Central Difference method for all the points except the end points, where I used the forward difference method for the first point, and the backward difference method for the last. The exact code I used is shown below.

```
for index, value in enumerate(baseTenList):
    if index == 0:
        fPrime.append((baseTenList[1]-value)/0.1)
    elif index != 0 and index != len(baseTenList) - 1:
        fPrime.append((baseTenList[index + 1]-baseTenList[index - 1])/0.1)
    else:
        fPrime.append((value-baseTenList[index - 1])/0.1)
```

Figure 3:Differenation of Charge

I once again used ‘matplotlib’ to plot my results as shown in Figure 4

Results

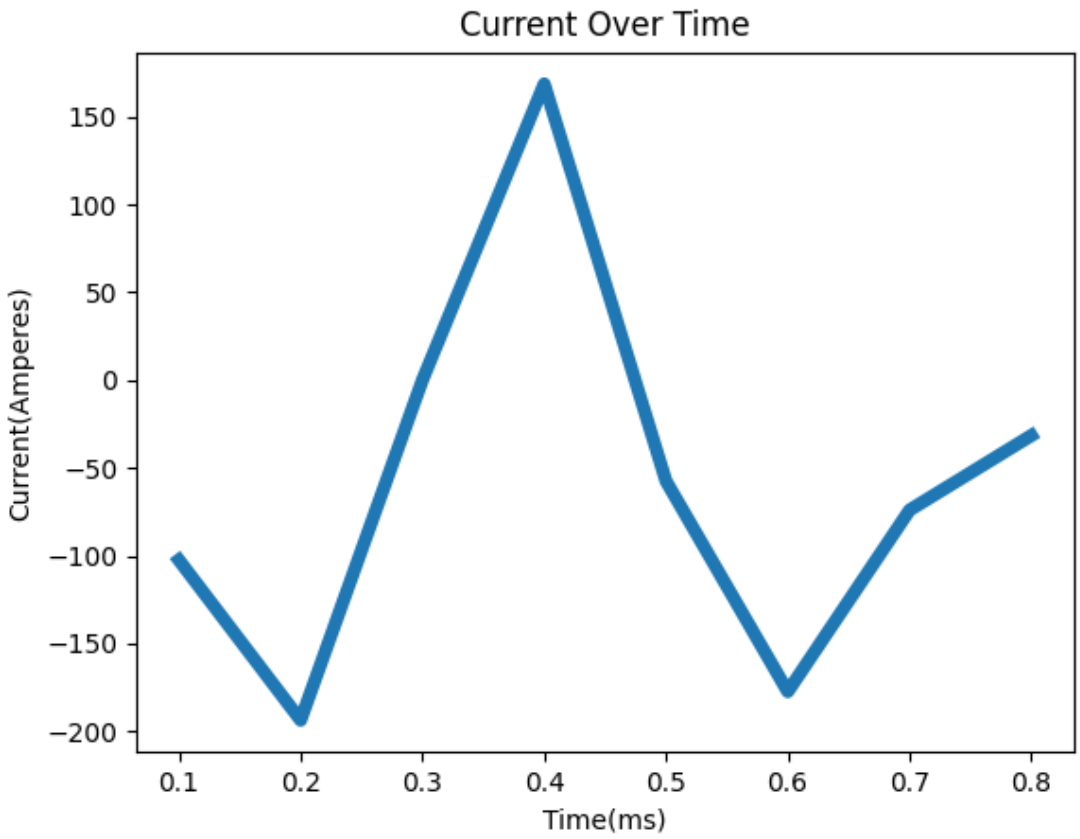

Figure 4: Results of using differentiation techniques to determine the current

Analysis

These differentiation techniques are approximations and may not have captured rapid changes or variations accurately, as you can see above, at 0.3 milliseconds, the current is zero. This suggests that the charge function reached a point of inflection where the slope is zero, where the absolute value of $f(x+h) = f(x-h)$ is likely to be zero, resulting in a zero in the numerator in the central difference formula.

Part 3: Bisection Method for Zero current

Objective

With new electric current data, use root finding techniques to find when the power spend is zero.

Methodology

Using the new data given, I will import this data into Python and clean it up for use, you can see in the code below I converted the data into floating numbers and appended them into two separate lists

```
data = open("Data_Chapter3.txt", "r")
timeList = []
currentList = []
for line in data:
    line = line.strip().split('\t')
    timeList.append(float(line[0]))
    currentList.append(float(line[1]))
```

Figure 5: Importing New Current Data

After cleaning up the data and organizing it, I used matplotlib to plot the data as shown in

I then used the Bisection method to evaluate the roots. The code runs a loop 15 times, each time refines the estimate. In each loop iteration it calculates the midpoint ('meanX') between 'lowX' and 'highX', it then rounds the midpoint to three decimal places to make sure that I can find an exact value in data given. Depending on the sign of the current value, the code with then updates either 'lowX' or 'highX' to the value of 'meanX'

```
for i in range(15): #Iterating for 15 times
    meanCurrent = None
    meanX = (lowX + highX)/2
    meanX = round(meanX,3)
    meanXList.append(meanX)
    for index, line in enumerate(timeList): #Checks every time value
in TimeList and assigns the time values to corresponding current
values
        if line == lowX:
            lowXCurrent = currentList[index]
        if line == highX:
            highXCurrent = currentList[index]
        if line == meanX:
            meanCurrent = currentList[index]
    if meanCurrent == None:
        print(f'Mean time value of {meanX} not found in data set')
        break
    if lowXCurrent < 0 and meanCurrent < 0 or lowXCurrent > 0 and
meanCurrent > 0:
        lowX = meanX
    elif highXCurrent < 0 and meanCurrent < 0 or highXCurrent > 0 and
meanCurrent > 0:
        highX = meanX
```

Figure 6: Bisection Code

Results

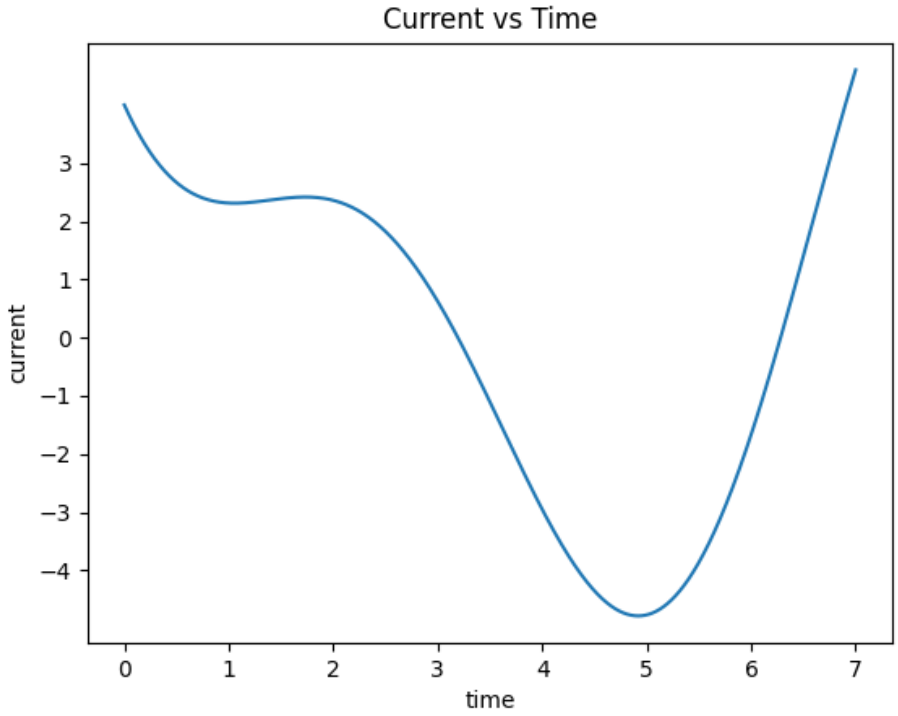

Figure 7: Given Current Data Cleaned up and Plotted

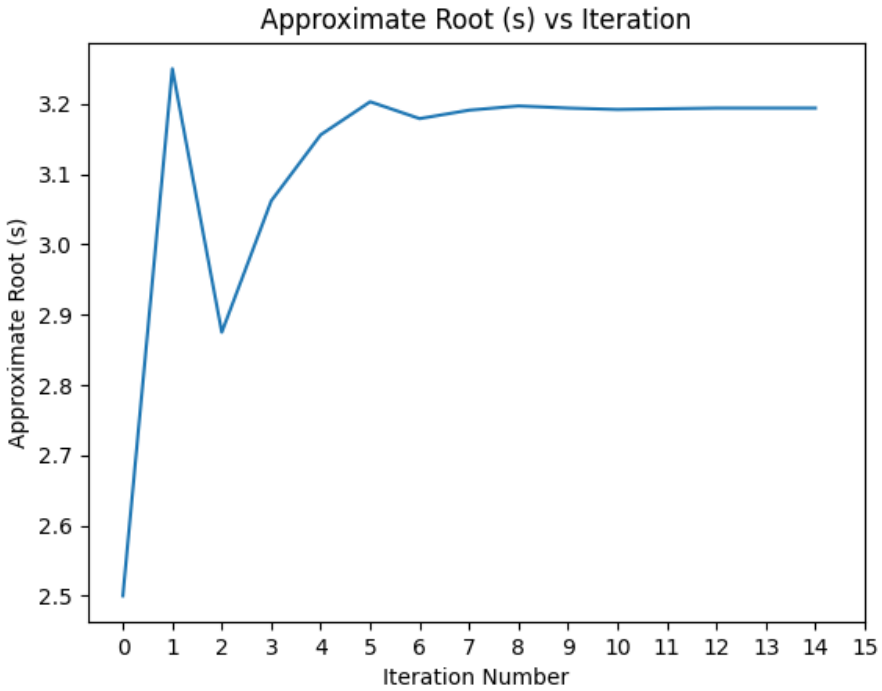

Figure 8: Showing how each iteration converges on the root

Analysis

As you can see, the bisection method converges around 3.2 seconds from Figure 8 this is about what we expect if we look at Figure 7. We can see that the current goes from positive to negative between 3 to 4 seconds.

You might notice the approximate root oscillating in Figure 8, this is because the bisection method operates by bracketing the root, after each iteration the interval between the two points we are using ‘lowX’ and ‘highX’ is essentially halved. This will converge to the true root by ‘squeezing’ the range.

Part 4: Kirchoff's Voltage Law and Circuit Analysis

Objective

After performing Kirchoff's Voltage Law on three loops to solve an electrical circuit, we have created a system of equations. Our objective is to solve this system of equations.

$$-V + I_1 R_1 + (I_1 - I_2) R_2 = 0$$

$$I_2 R_3 + (I_2 - I_3) R_4 + (I_2 - I_1) R_2 = 0$$

$$I_3 R_5 + (I_3 - I_2) R_4 = 0$$

Methodology

This is the first time I utilized a library other than matplotlib. There are several libraries that assist in linear algebra, but I decided to use NumPy. However, NumPy does not evaluate equations symbolically, therefore I must rewrite the system equations in the form of **Ax = B**. I have done this by hand below

$$\begin{bmatrix} R_1 + R_2 & -R_2 & 0 \\ -R_2 & R_2 + R_3 + R_4 & R_4 \\ 0 & R_4 & R_4 + R_5 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} v \\ 0 \\ 0 \end{bmatrix}$$

After rewriting the system of equations, I then used the NumPy function 'numpy.linalg.solve' to solve for the ***I*** values below.

```
r1, r2, r3, r4, r5, v = 10, 20, 30, 40, 50, 15
aMatrix = np.array([[r1 + r2,-r2,0],[r2,-(r2 + r3 + r4),-r4],[0,-r4,r5+r4]])
xMatix =  np.array([[v],[0],[0]])
print(np.linalg.solve(aMatrix,xMatix))
```

Figure 9: Solving Matrix Equations using Linear AlgebraThis NumPy function utilizes LU Decomposition.

Results

After executing the code above we get:

- $I_1 = 0.5705\ \Omega$
- $I_2 = 0.1058\ \Omega$
- $I_3 = 0.04705\ \Omega$

Analysis

If I wanted to do further analysis, NumPy also has a function to find the determinant and the inverse of a matrix, these functions are ‘numpy.lingalg.det()’ and ‘numpy.linalg.inv()’. In order to use these functions, I would simply put the matrix within the parenthesis.

When changing the resistor values in the problem you may need to pivot the rows in the matrix to produce a more accurate answer. By rearranging the rows to place the largest value on the diagonal, we can avoid division by zero and reduce numerical errors when using Gaussian elimination or LU Decomposition. Thankfully no changes to the code are necessary because NumPy does this automatically.

Part 5: Quadratic Splines for Capacitor Discharge

Objective

Given experimental data of the voltage across a capacitor, we want to create quadratic splines between the data points.

Methodology

The experimental data points are shown below:

Time(seconds)	Voltage(V)
0	50
0.5	30.33
1	18.39
1.5	11.6
2	6.77

Since we have 5 points, we need to create 4 splines. There will be 3 unknowns per quadratic spline, so we'll have to produce 12 equations.
We will create 8 equations by simply plugging in points:

$$f_1(0) = a_1 0^2 + b_1 0 + c_1$$

$$f_1(0.5) = a_1 0.5^2 + b_1 0.5 + c_1$$

$$f_2(0.5) = a_2 0.5^2 + b_2 0.5 + c_2$$

$$f_2(1) = a_2 1^2 + b_2 1 + c_2$$

$$f_3(1) = a_3 1^2 + b_3 1 + c_3$$

$$f_3(1.5) = a_3 1.5^2 + b_3 1.5 + c_3$$

$$f_4(1.5) = a_4 1.5^2 + b_4 1.5 + c_4$$

$$f_4(2) = a_4 2^2 + b_4 2 + c_4$$

We will then find 3 other equations by ensuring the slopes at the points where the splines touch is equal, this will also avoid jagged edges.

$$f_1'(0.5) = f_2'(0.5)$$

$$f_2'(1) = f_3'(1)$$

$$f_3'(1.5) = f_4'(1.5)$$

To find the last equation we will assume that the last spline is linear since it has a smaller interval compared to the first spline.

$$a_4 = 0$$

After we have established our equations, we can write them in the form of **Ax=B** shown below

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.5^2 & 0.5 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5^2 & 0.5 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1.5^2 & 1.5 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.5^2 & 1.5 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2^2 & 2 & 1 \\ 2*0.5 & 1 & 0 & -2*0.5 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2*1 & 1 & 0 & -2*1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2*1.5 & 1 & 0 & -2*1.5 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \\ a_3 \\ b_3 \\ c_3 \\ a_4 \\ b_4 \\ c_4 \end{bmatrix} = \begin{bmatrix} 50 \\ 30.33 \\ 30.33 \\ 18.39 \\ 18.39 \\ 11.6 \\ 11.6 \\ 6.77 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Instead of doing all of this in my code, I simply used another library called SciPy, which provides functions for optimization, integration, and interpolation, perfect for this problem.

I used the function InterpolatedUnivariateSpline from SciPy to solve the problem below, the 'k=2' parameter defines the spline as being quadratic, the results of this function are shown in Figure 11

```
time = [0,0.5,1,1.5,2]
voltage = [50,30.33,18.39,11.16,6.77]
spline = InterpolatedUnivariateSpline(time,voltage,k=2)
```

Figure 10: Interpolation Code

Results

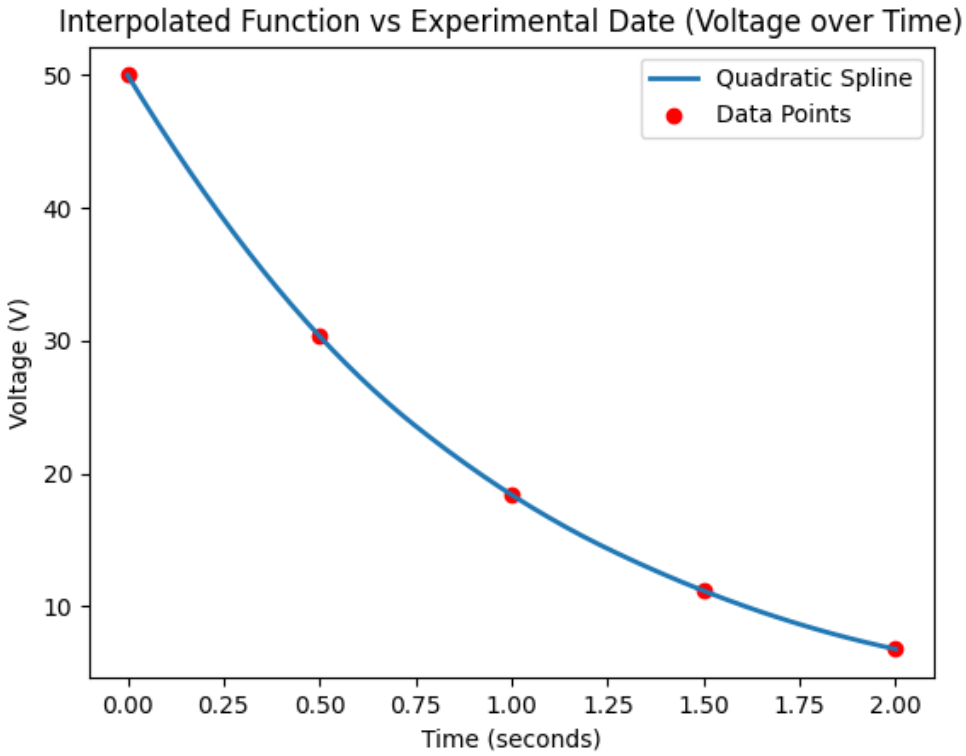

Figure 11: The plot returned by using the InterpolatedUnivariateSpline

To return the coefficients of the splines I tried using the SciPy function '.get_coeffs()', however it only returns 4 coefficients, there is only a little documentation on this function, but I have deciphered that these coefficients are not the direct coefficients of the polynomial segments, but are instead related to something in the underlying mathematical formulation of the function.

So instead of using SciPy to get coefficients I used NumPy's basic linear algebra solver, which resulted in the following coefficients

$$\begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \\ a_3 \\ b_3 \\ c_3 \\ a_4 \\ b_4 \\ c_4 \end{bmatrix} = \begin{bmatrix} 18.16 \\ -48.42 \\ 50 \\ 12.76 \\ -43.02 \\ 48.65 \\ 7.84 \\ -33.18 \\ 43.73 \\ 0 \\ -9.66 \\ 26.09 \end{bmatrix}$$

Part 6: Exponential Fit for Capacitor Discharge

Objective

Given data from a capacitor discharging, use regression techniques to find the exponential function of best fit

Methodology

First, I will transform the exponential function into a linear function for ease of work. The exponential relationship of the discharge rate of a capacitor is: $V(t) = ae^{bt}$

To do this I will utilize the least-squares method, which is a statistical technique to find the best-fit line/curve for a set number of data points. This works by minimizing the sum of the square of the residuals (or the vertical difference between the data point, and our model value.)

$$S_r = \sum E_i^2$$

Where E_i is the residual

$$E_i = [y_i - f(x_i)]$$

This method involves using a model ($f(x_i)$) that will represent our curve, the voltage discharge rate of a capacitor in based on an exponential relationship, but since that is difficult to work with, we will instead transform it into a linear relationship, and use the model $f(x_i) = a_1 x_1 + a_0$

To transform this model, we take the natural log of both sides:

$$\ln\big(V(t)\big) = \ln\,(ae^{bt})$$

$$\ln\big(V(t)\big) = \ln(a) + bt$$

And then we apply our transformation to new variables:

$$z_i = a_0 + a_1 x_1$$

Then we can use the equations for linear regression:

$$a_1 = \frac{N\sum x_i y_i - \sum x_i \cdot \sum y_i}{N\Sigma x_i^2 - (\sum x_i)^2}$$

$$a_0 = \bar{y} - a_1 \bar{x}$$

I did these summations in my code by breaking the equations into chunks, for example in my code my a_1 equation looked like this:

$$a_1 = \frac{first - second}{third - forth}$$

The numbers represent variables, where I evaluated each summation individually, as shown in Figure 12

```
time = [0.5, 1, 1.5, 2, 2.5]
voltage = [30.33,18.39,11.16,6.77,4.10]
first, second, third, forth = 0,0,0,0
zAverage = 0
for index, unit in enumerate(time):
    first = first + unit*math.log((voltage[index]))
    third = third + unit**2
    zAverage = zAverage + math.log(voltage[index])
first = first * len(time)
second = sum(time) * sum([math.log(v) for v in voltage])
third = third * len(time)
forth = sum(time)**2
zAverage = zAverage/len(voltage)
a1 = (first - second)/(third - forth)
a0 = zAverage - a1*np.average(time)
```

Figure 12: Finding the a_1 and a_0 Values using summations

I also utilized the Math library that usually comes preinstalled with Python. This made it easier to use logarithms.

After finding the a_1 and a_0 values we have to convert back to our original variables, *a* and *b,* using the math function .exp() to get rid of the natural log.

```
b = a1
a = math.exp(a0)
```

Figure 13: Transforming Back

Results

After executing my code, the value of 'a' is approximately 50, while the value of 'b' is around -1. This means that our curve of best fit is given by the function:

$$V(t) = 50e^{-t}$$

You can see this graph as compared to the data points in Figure 15.

Below is a table of the residual values, as you can see the residuals are very small, indicating that our curve is a good fit.

Time (s)	Measured	Function	Residuals
0.5	30.33	30.33418	-0.00418
1	18.39	18.39565	-0.00565
1.5	11.16	11.15573	0.004265
2	6.77	6.765208	0.004792
2.5	4.1	4.102647	-0.00265

Figure 14: Residual values after Regression

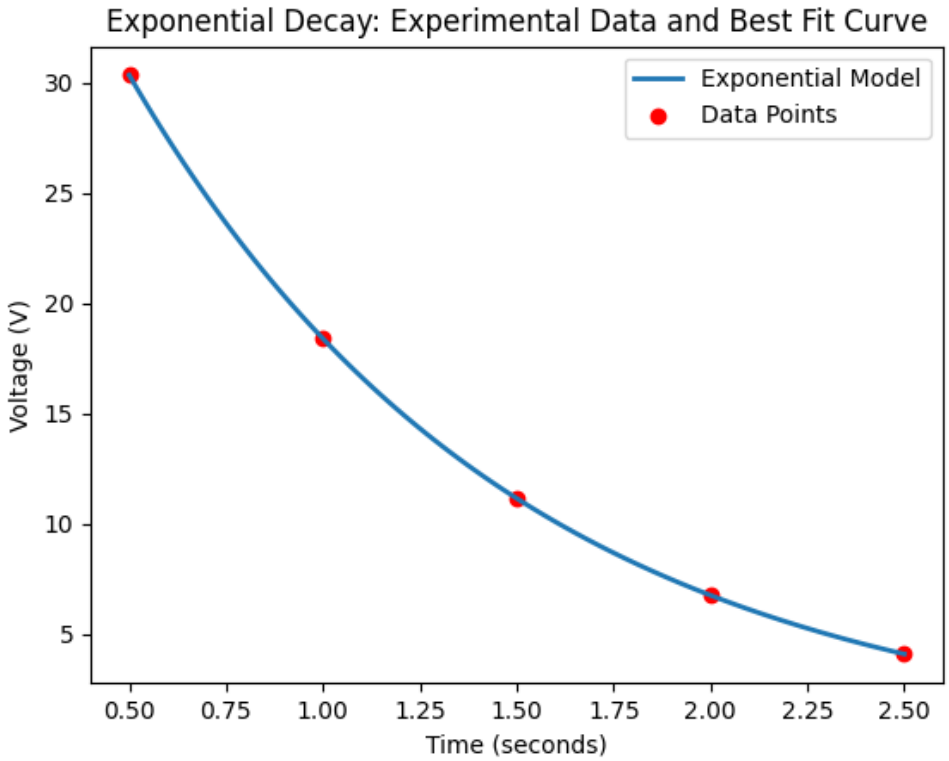

Figure 15: Exponential Decay of Voltage in a Capacitor Discharge

Analysis

Suppose I wanted to find the exponential regression without transforming the model, in that case I would define the residual as:

$$E_i = [y_i - V(t)]$$

$$E_i = [y_i - ae^{bt}]$$

I would then sum the squares of the residuals, then take the partial derivatives with respect to *a* and *b,* then solve for a and b. This will minimize the function, but in return we will have very complex equations that we would have to use root finding methods to solve.

Conclusion:

In this project, I used various computational methods techniques to solve electrical engineering problems. Furthermore, I utilized Python to transform complex data into meaningful analysis. The key findings of this project are summarized as follows:

1. Analysis of Electric Charge Over Time: By transforming binary data to base-10 and plotting charge values over time, we observed and recorded errors in the original recording system, we then discussed how these errors can be minimized by biasing the original system

2. Calculation of Electrical Current: We used several differentiation techniques to analyize electrical current given charge data.

3. Bisection Method for Zero Current: We used the bisection method effectily to converge on the point where the electrical current was zero. This demonstrated the effectiveness of this root-finding method

4. Kirchoff's Voltage Law and Circuit Analysis: We formulated a set of linear equations in a matrix and used LU Decomposition within NumPy to accurately analysis complex circuits.

5. Quadratic Splines for Capacitor Discharge: Quadratic splines were created to model the voltage across a capacitor over time.

6. Exponential Regression for Capacitor Discharge: Regression techniques were used to graph a exponential function that best fits voltage discharge data. We transformed the exponential model into a linear form and applied least-squares regression to obtain an accurate fit.

Overall, this project has improved my understanding of data science within Python, I used several libraries that I was unfamiliar with such as SciPy. I also gained a better understanding of numerical methods, and how to apply such techniques to a range of engineering problems.

Appendix:

```
def decimalBinaryToBaseTen(Binary):
    value = 0
    for index, Units in enumerate(Binary):
        value = value + int(Units) * 2 ** ((index + 1) * (-1))
    return value
```

Appendix Figure 1:#This function will convert binary values that are right of the decimal point to base-10, I can use the function int(value, 2) to convert integer values to base 10 point to base-10,

```
def relative_true_error(TrueValue, ApproximateValue):
    RteList = []
    for index, value in enumerate(TrueValue):
        RteList.append(((value - ApproximateValue[index]) / value) * 10
    return RteList
```

Appendix Figure 2: This Function will find the Relative True Error and append it to a list

```
biList = []
for line in data:
    line = line.strip()
    line = line.replace(" ", "")
    biList.append(line)

#Indexing the floating point binary data from biList and converting it
then placing it in baseTenList

baseTenList = []
for line in biList:  #This will iterate for every float point binary nu

    sign = int(line[0])  #Checks the sign of the mantissa
    if sign == 0:
        sign = 1
    else:
        sign = -1

    exp_sign = int(line[1])  #Checks the sign of the exponent
    if exp_sign == 0:
        exp_sign = 1
    else:
        exp_sign = -1

    expMag = int(line[2:5], 2)  #Finds the integer value of the exponen
    mantissa = line[5:10]

    # We need to move the decimal point expMag times to the right or le
    if exp_sign == 1:  # This will run if the exponent is positive
        if expMag <= len(mantissa):  # If the length of the exponent is
the length of the mantissa we need to add a "." in the middle of the ma
            mantissa = "1" + mantissa[0:expMag] + "." +
mantissa[expMag:len(mantissa) + 1]
        else:
            # noinspection PyTypeChecker
            for i in len(
                    expMag):  # If the length of the exponent is greate
mantissa we need to add zeros to the end on the mantissa
                mantissa = mantissa + "0"
            mantissa = "1" + mantissa
    else:
        for i in len(expMag):
            mantissa = "0" + mantissa

    valueLeftOfPoint = ""
    valueRightOfPoint = ""
```

Appendix Figure 3: Converting Binary into Base-10