ECSE 543: Numerical Methods

Assignment 3 Report

Wenjie Wei
260685967

December 14, 2019

Contents

Introduction

This assignment explored the use of linear interpolations and other mathematical methods. The programs are programmed and compiled using Python 3.6, and the plots are generated using package matlibplot. Listing 1 shows the implementations of polynomials including their possible maneuvers. The object classes included in this file will be used for the interpolations.

1 Linear Interpolation of BH Points

1.a Lagrange Full Domain Interpolation of First Six-Point Set

Listing 2 shows the implementation of various interpolation methods. For the first six points, the Lagrange interpolation shows an interpolated polynomial

$$\begin{aligned}B(h) =& 9.275 \times 10^{-12} h^5 - 5.951 \times 10^{-9} h^4 \\ &+ 1.469 \times 10^{-6} h^3 - 1.849 \times 10^{-4} h^2 \\ &+ 1.603 \times 10^{-2} h\end{aligned}$$

whose plot is shown in Figure 1.

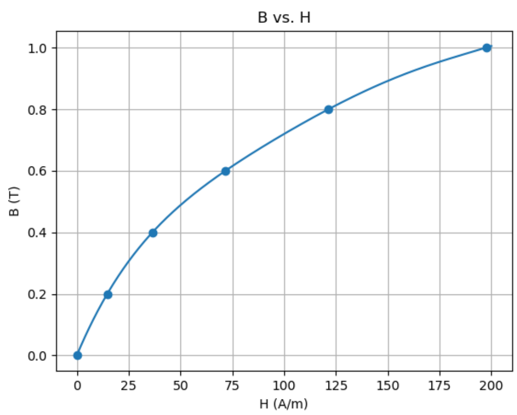


Figure 1: Interpolation of the First Six Data Points

From the figure, the interpolation has returned a plot with a **plausible** result over this range.

1.b Lagrange Full Domain Interpolation of the Second Six-Point Set

Select a second data point set, the Lagrange interpolation returned a polynomial of

$$\begin{aligned}B(h) =& 7.467 \times 10^{-19} h^5 - 3.505 \times 10^{-14} h^4 \\ &+ 5.3 \times 10^{-10} h^3 - 2.864 \times 10^{-6} h^2 \\ &+ 3.804 \times 10^{-3} h\end{aligned}$$

whose plot is shown in Figure 2.

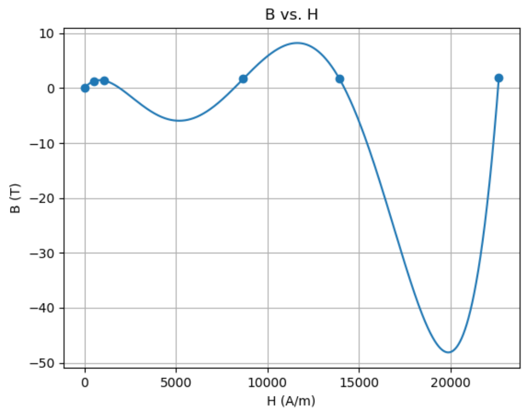


Figure 2: Interpolation of the Second Six Data Points

From this plot, we can see that the interpolation using the second set of data points is **not plausible** as the graph fluctuates violently as the value of B goes to negative at some ranges.

1.c Cubit Hermite Polynomial Interpolation

1.d Nonlinear Equation of the Magnetic Circuit

Consider the magnetic circuit shown in Figure 3.

The Magnetomotive force (MMF) can be calculated by Equation 1,

$$M = (R_g + R_c)\psi \tag{1}$$

where R_g and R_c are the reluctance of the air gap and the coil, respectively. Plug in the variables from the problem, we can transform Equation 1 to

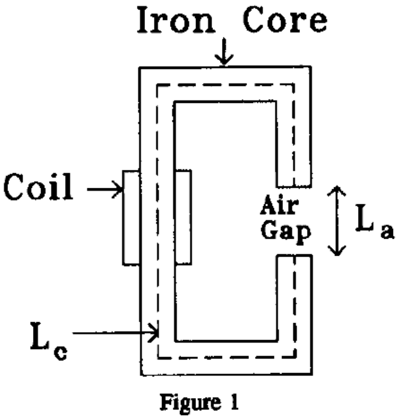


Figure 3: The Magnetic Circuit Discussed About

the equation as follows:

$$M = (\frac{l_g}{\mu_0 A} + \frac{l_c}{\mu A})\psi$$
$$NI = (\frac{l_g}{\mu_0 A} + \frac{l_c H(\psi)}{AB})\psi$$
$$NI = (\frac{l_g}{\mu_0 A} + \frac{l_c H(\psi)}{\psi})\psi$$

Simplify the equation by bringing NI to the right of the equation, and the equation will be the final formula of $f(\psi)$, as is shown in Equation

$$f(\psi) = \frac{l_g \psi}{\mu_0 A} + l_c H(\psi) - NI = 0 \quad (2)$$

Plug in the numbers, we can finalize the equation by calculating all the coefficients of the polynomial, shown in Equation 3.

$$f(\psi) = 3.979 \times 10^7 \psi + 0.3H(\psi) - 8000 \quad (3)$$

1.e Newton Raphson Method

This part of the problem implements the algorithm of Newton Raphson to solve the non-linear equation. The equation is shown in the previous section in Equation 3.

In the equation, there are two factors affecting the result of $f(\psi)$. One is the flux ψ, and the other one is the magnitude of the magnetic field $H(\psi)$. To find the magnetic field, construct a piece-wise linear interpolation shown in Figure 4.

Note that the figure is plotted with respect to H vs. B. and B is calculated as follows:

$$B = \frac{\psi}{A} \quad (4)$$

where A denotes the cross-sectional area. In this case, the area is $1 \times 10^{-4} m^2$.

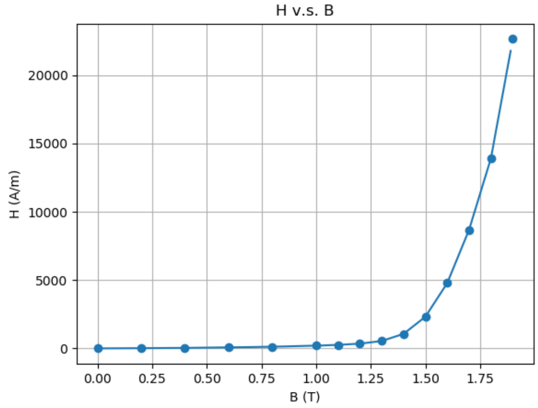
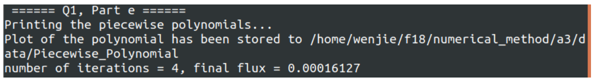


Figure 4: Plot of the Piecewise Polynomial

Using this plot, the magnetic field magnitude can be found and $f(\psi)$ can be calculated.

Listing 3 shows the implementation of the Newton-Raphson method. Run the main script of the assignment, Newton-Raphson returns with four iterations and a final flux of $\psi = 1.613 \times 10^{-4} Wb$, shown in Figure 5.



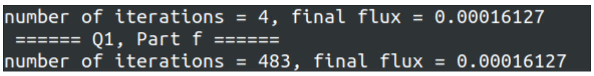
```
====== Q1, Part e ======
Printing the piecewise polynomials...
Plot of the polynomial has been stored to /home/wenjie/f18/numerical_method/a3/d
ata/Piecewise_Polynomial
number of iterations = 4, final flux = 0.00016127
```

Figure 5: Result of Newton-Raphson Run

1.f Successive Substitution

Listing 3 shows the implementation of successive substitution as well. The successive substitution turns out to be that the method is diverging to infinity. The reason is that the step has been too large. Therefore, I have reduced the step with a factor of 5×10^{-9}. Therefore, the method will run with smaller steps and does not miss the target point.

After the modification, the method returns with an iteration step of 483 and a flux of $1.161 \times 10^{-4} Wb$, which is similar to the result returned by Newton-Raphson, but with a much larger number of iterations, shown in Figure 6.



```
number of iterations = 4, final flux = 0.00016127
 ====== Q1, Part f ======
number of iterations = 483, final flux = 0.00016127
```

Figure 6: Result of Successive Substitution Run

2 The Problem of the Diode Circuit

2.a Derivation of Circuit Equation

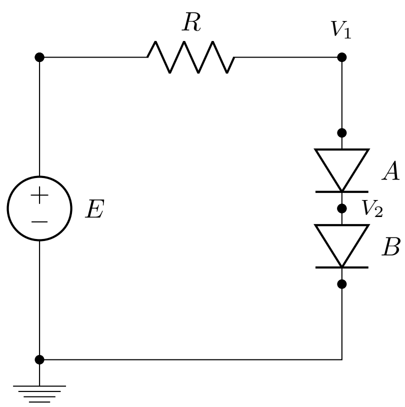


Figure 7: The Diode Circuit to be Investigated

Figure 7 shows the diode circuit to be investigated in this problem. Define the current flowing in the circuit to be I, and the current is expressed with:

$$I = \frac{E - V_1}{R} \tag{5}$$

In the circuit, the current flowing through the two diodes are identical to the current flowing through the resistor. Therefore, using the diode characteristic current, the following relations can be derived:

$$I = I_{s,A}(e^{q(V_1 - V_2)/(kT)} - 1) \tag{6}$$

and

$$I = I_{s,B}(e^{qV_2/(kT)} - 1) \tag{7}$$

From the above equations, we can derive the following two entries for the $\boldsymbol{f}$ matrix, represented explicitly in terms of the variables:

$$\begin{aligned} f_1 &= (5) - (6) \\ &= \frac{E - V_1}{R} - I_{s,A}(e^{q(V_1 - V_2)/(kT)} - 1) \end{aligned}$$

$$\begin{aligned} f_2 &= (6) - (7) \\ &= I_{s,A}(e^{q(V_1 - V_2)/(kT)} - 1) - I_{s,B}(e^{qV_2/(kT)} - 1) \end{aligned}$$

The $\boldsymbol{f}$ matrix is then expressed as follows:

$$\vec{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

2.b Solution using Newton-Raphson

Since this equation has output a multi-variable vector, the first step will be finding the Jacobian matrix $\boldsymbol{F}$ and the multi-variable Newton Raphson update formula will be changed to:

$$\boldsymbol{V}_n^{(k+1)} = \boldsymbol{V}_n^{(k)} - \boldsymbol{F}^{-1(k)}\boldsymbol{f}^{(k)}$$

The Jacobian matrix will be calculated following Equation as follows:

$$\boldsymbol{F} = \begin{bmatrix} \frac{\partial f_1}{\partial V_1} & \frac{\partial f_1}{\partial V_2} \\ \frac{\partial f_2}{\partial V_1} & \frac{\partial f_2}{\partial V_2} \end{bmatrix} \tag{8}$$

From the previous calculations for f_1 and f_2, we can derive the following expression for the four entries in the $\boldsymbol{F}$ matrix:

$$\frac{\partial f_1}{\partial V_1} = -\frac{1}{R} - I_{s,A}\frac{q}{kT}exp(\frac{q(V_1 - V_2)}{kT})$$

$$\frac{\partial f_1}{\partial V_2} = I_{s,A}\frac{q}{kT}exp(\frac{q(V_1 - V_2)}{kT})$$

$$\frac{\partial f_2}{\partial V_1} = I_{s,A}\frac{q}{kT}exp[\frac{q(V_1 - V_2)}{kT}]$$

$$\frac{\partial f_2}{\partial V_2} = -I_{s,A}\frac{q}{kT}exp[\frac{q(V_1 - V_2)}{kT}] - I_{s,B}\frac{q}{kT}exp(\frac{qV_2}{kT})$$

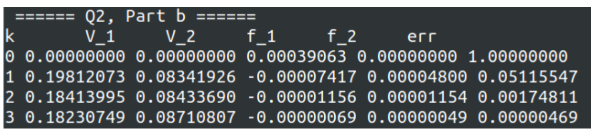
As the Jacobian matrix is a 2-by-2 matrix, its inverse can be easily calculated by:

$$\boldsymbol{F}^{-1} = det(\boldsymbol{F})\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{9}$$

where $det(\boldsymbol{F})$ is calculated by:

$$det(\boldsymbol{F}) = \frac{1}{ad - bc}$$

The code in the main script shows the implementation of the Newton Raphson update. The error measurement is selected to be $\varepsilon_k = 1 \times 10^{-6}$, and the program three iterations to converge. By running the main script, the detailed information during the iterations are shown in Figure 8.



```
====== Q2, Part b ======
k       V_1        V_2       f_1       f_2       err
0 0.00000000 0.00000000 0.00039063 0.00000000 1.00000000
1 0.19812073 0.08341926 -0.00007417 0.00004800 0.05115547
2 0.18413995 0.08433690 -0.00001156 0.00001154 0.00174811
3 0.18230749 0.08710807 -0.00000069 0.00000049 0.00000469
```

Figure 8: Values During Netwon Raphson Iterations

To inspect if the convergence is quadratic, make a plot of the four error points shown in Figure 9.

From the plot, we can show the convergence to be quadratic.

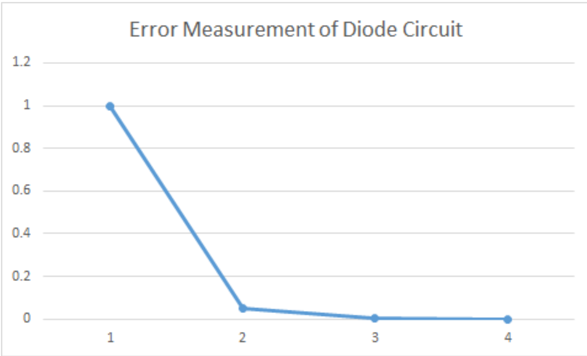


Figure 9: Plot of Error Measurement

3 Gauss-Legendre Integration

The implementation of the one-point Gauss-Legendre Integration is shown in Listing 4.

The implementation has been straight forward. The function requires the target function, the lower and upper limit, and number of rectangular segments to be used. The function also requires an input of the real value calculated by MATLAB, and the function returns the result of the integration, as well as the absolute error.

The integration performs if the width of each rectangular segment ζ is provided. The algorithm performs an integration following the equation below:

$$\int_{x_0}^{x_1} f(x)dx \approx \sum_{i=1}^{n} \zeta_i f(x_0 + \frac{\zeta_i}{2}) \quad (10)$$

where x_0 is the central value of each rectangular segment.

3.a One-Point Gauss-Legendre Integration of $sin(x)$

The one-point Gauss-Legendre integration is firstly tested using a sine function from 0 to 1. The real value passed into the function is 0.4597, which was calculated by MATLAB. The program is tested with different number of rectangle segments, varying from 1 to 20. The plot of the absolute error on the log scale is shown in Figure 10.

The plot shows a straight line of the absolute error on the logarithm scale. The straight line indicates a first-order Gauss-Legendre integration

3.b One-Point Gauss-Legendre Integration of $ln(x)$

The one-point Gauss-Legendre integration is then used to calculate $\int_0^1 ln(x)dx$. Since the one-point Gauss-Legendre methods uses the value on

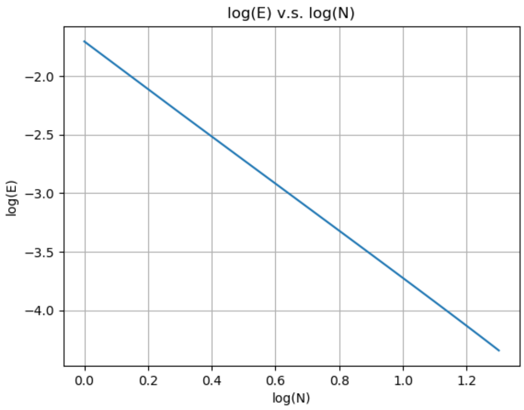


Figure 10: Absolute Error of $\int_0^1 sin(x)dx$

the right of the rectangular segment, integrating the logarithm function from 0 will not produce the problem of an undefined number.

The test for the logarithm uses 10 to 200 rectangular segments for the integration. Similar to the sine function test, the real value -1 is used for the calculation of absolute errors, and the plot for the errors on the logarithm scale is shown in Figure 11.

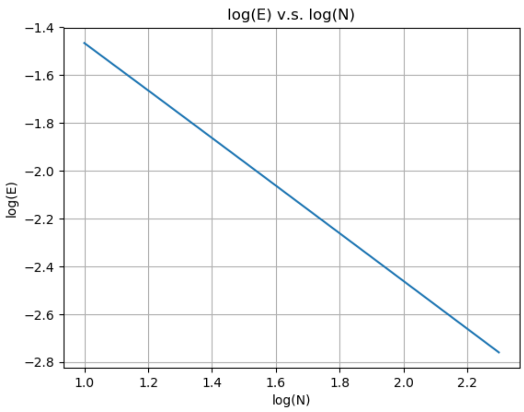


Figure 11: Plot of Absolute Errors on Log Scale for the Logarithm Function

Similar to the sine function, the straight line of the error plot indicates the use of one-point Gauss-Legendre integration, and it can also be seen that with more rectangular segments used, the error is reduced to achieve higher accuracy.

3.c One-Point Integration Test for $\int_0^1 ln(0.2|sin(x)|)dx$

The tricky part for this problem is to perform a substitution to do the integration. By substituting

$0.2sin(x)$ by u, we can obtain a new integration function below:

$$\int_0^1 ln(0.2|sin(x)|)dx = \int_0^{0.2sin(1)} \frac{ln(u)}{\sqrt{0.04 - u^2}} du$$

By performing the new integration with 10 to 200 rectangular segments, we obtain a new plot shown in Figure 12.

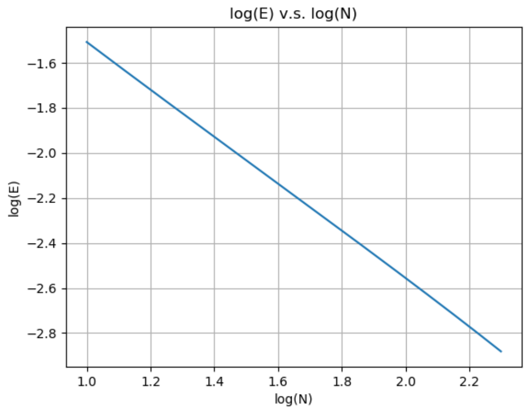


Figure 12: Plot on Absolute Errors of the New Integral

The logarithm calculations returns a plot with a similar behavior comparing with the plots shown in the previous sections.

3.d Varying Segment Width

The fixed-width integration does not give a good approximation when the rate of change of the function is high at certain domain. Therefore, if the width could be varied according to the rate of change, the result of the integration will become more accurate.

In the function implemented for this problem, using a ten-segment integration, I manually decided a width ratio of $[1 : 2 : 3 : \cdots : 9 : 10]$ for the ten rectangular segments. The testing result of such modification is shown in Figure 13 below:

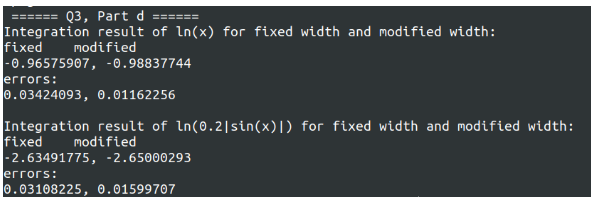


Figure 13: Testing Result of the Modification

The testing result shows that with a varying width, the integration approximation becomes much more accurate comparing to the fixed-width integration approximation.

A Code Listings

Listing 1: Polynomials Implementation (`polynomial.py`).

```python
import math
# this is a git test


class Polynomial(object):
    def __init__(self, coeff):
        self._coeff = coeff
        self._order = len(coeff) - 1

    def calculate(self, value):
        """
        This function calculates the result of the polynomial.

        :param value: value of x
        :return: value of y
        """
        result = 0
        for i in range(len(self._coeff)):
            result += self._coeff[i] * math.pow(value, i)

        return result

    def derive(self, der_order):
        result_coeff = []
        counter = 0

        for i in range(1, len(self._coeff)):
            result_coeff.append(i * self[i])
        result_poly = Polynomial(result_coeff)
        counter += 1

        if counter < der_order:
            return result_poly.derive(der_order - 1)
        else:
            return result_poly

    def __getitem__(self, item):
        return self._coeff[item]

    def __add__(self, other):
        result_coeff = []

        if isinstance(other, int):
            result_coeff = self._coeff
            result_coeff[0] += other
        else:
            self_has_higher_order = (max(self.order, other.order) == self.order)

            if self_has_higher_order:
                big_coeff = self.coefficient
                small_coeff = other.coefficient
            else:
                big_coeff = other.coefficient
                small_coeff = self.coefficient

            for i in range(len(small_coeff), len(big_coeff)):
                small_coeff.append(0)

            for i in range(len(big_coeff)):
                result_coeff.append(small_coeff[i] + big_coeff[i])

        return Polynomial(result_coeff)

    def __sub__(self, other):
        result_coeff = []
```

```
        if isinstance(other, int):
            result_coeff = self._coeff
            result_coeff[0] -= other

        else:
            self_has_higher_order = (max(self.order, other.order) == self.order)

            if self_has_higher_order:
                for i in range(len(other.coefficient), len(self.coefficient)):
                    other.coefficient.append(0)
            else:
                for i in range(len(self.coefficient), len(other.coefficient)):
                    self.coefficient.append(0)

            for i in range(len(self.coefficient)):
                result_coeff.append(self.coefficient[i] - other.coefficient[i])

        return Polynomial(result_coeff)

    def __mul__(self, other):
        result_coefficients = []

        result_order = self.order + other.order

        for i in range(result_order + 1):
            coefficient = 0
            for j in range(self.order + 1):
                for k in range(other.order + 1):
                    if j + k == i:
                        coefficient += self[j] * other[k]

            result_coefficients.append(coefficient)

        return Polynomial(result_coefficients)

    def toString(self):
        print("y = ", end="")
        for i in range(self.order, 0, -1):
            if self[i] != 1 and self[i] != -1 and self[i] != 0:
                if self[i] >= 0:
                    print("+ " + str(self[i]) + "x^" + str(i), end=" ")
                else:
                    print("- " + str(-self[i]) + "x^" + str(i), end=" ")
            elif self[i] == 1:
                print("+ x^" + str(i), end=" ")
            elif self[i] == -1:
                print("- x^" + str(i), end=" ")
            else:
                pass

        if self[0] < 0:
            print("- " + str(-self[0]))
        else:
            print("+ " + str(self[0]))

    def modify_const(self, value):
        self._coeff[0] = value

    @property
    def order(self):
        return self._order

    @property
    def coefficient(self):
        return self._coeff


class LagrangePolynomial(object):
    def __init__(self, n, xr, j, xj):
        """
```

```
        Construct a Lagrange polynomial.

        :param n: how many points are on the x axis
        :param xr: the values of x
        :param j: the position of the current x
        :param xj: the value of x at position j
        """
        self._order = n
        self._j = j
        self._xr = []
        self._xj = xj

        self._x = 0

        for i in range(len(xr)):
            self._xr.append(-xr[i])

        self._numerator = self._create_numerator()
        self._denominator = self._create_denominator(xj)

        self._polynomial = self._create_polynomial()

    def _create_numerator(self):
        """
        This method creates the list of the parameters x_r.

        :return: no return value
        """
        i = 0
        result_numerator = Polynomial([1])

        while i < self._order:
            if i == self.j:
                i += 1

            if i >= self._order:
                break

            result_numerator *= Polynomial([self._xr[i], 1])
            i += 1

        return result_numerator

    def _create_denominator(self, x):
        """
        This method calculates the numerical result of the denominator.

        :return: the value in decimal of the denominator.
        """

        return self._numerator.calculate(x)

    def _create_polynomial(self):
        """
        This method creates the general form of the lagrange polynomial.
        :return:
        """
        denom = Polynomial([1 / self._denominator])

        return denom * self._numerator

    def set_x(self, value):
        self._x = value

    @property
    def j(self):
        return self._j

    @property
    def xj(self):
```

```
        return self._xj

    @property
    def denominator(self):
        return self._denominator

    @property
    def numerator(self):
        return self._numerator

    @property
    def poly(self):
        return self._polynomial


if __name__ == "__main__":
    coeff1 = Polynomial([2])
    coeff2 = Polynomial([4, 5, 7, 8])

    coeff2.toString()
    (coeff2 - 3).toString()
```

Listing 2: Lagrange Interpolation Implementation (`interpolation.py`).

```
from polynomial import Polynomial, LagrangePolynomial


TOLERANCE = 1e-6

def lagrange_full_domain(xr, y, points=None):
    """
    This is the method for the lagrange full domain interpolation.
    X is the variable that varies.
    Y is the variable that varies with respect to X.

    :param X: X vector of type Matrix
    :param Y: Y vector of type Matrix
    :param points: select the range of data to be interpolated if needed.
    :return: Polynomial expression for y(x)
    """
    result_polynomial = Polynomial([0])

    if points is None:
        for j in range(len(xr)):
            xj = xr[j]
            aj = y[j]

            temp_lagrange_poly = LagrangePolynomial(len(xr), xr, j, xj)

            result_polynomial += Polynomial([aj]) * temp_lagrange_poly.poly

    else:
        pass

    return result_polynomial


def cubic_hermite(xr, y, slopes):
    result = Polynomial([0])

    for j in range(len(xr)):
        xj = xr[j]
        aj = y[j]
        bj = slopes[j]

        temp = LagrangePolynomial(len(xr), xr, j, xj).poly
        lagrange_backup = LagrangePolynomial(len(xr), xr, j, xj).poly

        # Calculate the polynomial u(x)
```

```
        temp = (temp.derive(1) * Polynomial([-xj, 1])) * Polynomial([-2])
        temp = temp + 1

        square = lagrange_backup * lagrange_backup
        uj = temp * square

        # Calculate the polynomial v(x)
        vj = Polynomial([-xj, 1]) * square

        aj_poly = Polynomial([aj])
        bj_poly = Polynomial([bj])

        result += uj * aj_poly + vj * bj_poly

    return result

def piecewise_linear_interpolate(xr, y):
    polynomials = []

    for i in range(1, len(xr)):
        a = (y[i] - y[i - 1]) / (xr[i] - xr[i - 1])
        b = y[i] - a * xr[i]

        temp_poly = Polynomial([b, a])
        polynomials.append(temp_poly)

    return polynomials
```

Listing 3: Newton Raphson (`nonlinear.py`).

```
from polynomial import Polynomial
from matrix import Matrix
import math

TOLERANCE = 1e-6
MAX_ITERATIONS = 1000
r = 512
e = 0.2
isa = 0.8e-6
isb = 1.1e-6
ktq = 0.025


def calc_newton_raphson(equation, data_x, data_y):
    """
    calculates the newton raphson
    :param equation: either a polynomial or a list of piecewise linear polynomials
    :param data_x: the list of data on the x axis
    :param data_y: the list of data on the y axis
    :return: number of iterations and the final result
    """
    if isinstance(equation, list):
        """
            This condition will be taken if equation is a list of linear polynomials.
        """
        area = 1e-4
        flux_list = []
        coefficients = [3.9790e7, 0.3, -8000]

        # Calculate f(0)
        k = 0
        flux = 0
        convergent = False
        fk = -8000
        prev_fk = -8000

        while not convergent:
            if abs(fk / prev_fk) < TOLERANCE or k >= MAX_ITERATIONS:
                break
```

```
            prev_fk = fk
            # Find the piecewise polynomial segment of the current flux
            for i in range(1, len(data_x)):
                if data_x[i - 1] <= (flux / area) < data_x[i]:
                    temp_poly = equation[i - 1]
                    start_H = data_y[i - 1]
                    start_B = data_x[i - 1]
                    break
                else:
                    temp_poly = equation[len(equation) - 1]
                    start_H = data_y[len(equation)]
                    start_B = data_x[len(equation)]

            # The polynomial segment is located at location i - 1
            # Calculating stuff at k
            slope = temp_poly[1]
            H = slope * (flux - (start_B * area)) / area + start_H
            fk = coefficients[0] * flux + coefficients[1] * H + coefficients[2]
            fk_prime = coefficients[0] + coefficients[1] * temp_poly[1] / area

            k += 1
            flux = flux - fk / fk_prime
            flux_list.append(flux)

        return k, flux_list


def calc_successive_subs(equation, data_x, data_y):
    if isinstance(equation, list):
        area = 1e-4
        coefficients = [3.979e7, 0.3, -8000]
        flux_list = []

        # Calculate f(0)
        k = 0
        flux = 0
        convergent = False
        f0 = -8000 * 5e-9
        fk = -8000 * 5e-9

        while not convergent:
            if abs(fk / f0) < TOLERANCE or k >= MAX_ITERATIONS:
                break
            # Find the piecewise polynomial segment of the current flux
            for i in range(1, len(data_x)):
                if data_x[i - 1] <= (flux / area) < data_x[i]:
                    temp_poly = equation[i - 1]
                    start_H = data_y[i - 1]
                    start_B = data_x[i - 1]
                    break
                else:
                    temp_poly = equation[len(equation) - 1]
                    start_H = data_y[len(equation)]
                    start_B = data_x[len(equation)]

            # The polynomial segment is located at location i - 1
            # Calculating stuff at k
            slope = temp_poly[1]
            H = slope * (flux - (start_B * area)) / area + start_H
            fk = coefficients[0] * flux + coefficients[1] * H + coefficients[2]
            fk /= 5e9

            k += 1
            flux -= fk
            flux_list.append(flux)

        return k, flux_list


def calc_jacobian(voltages):
```

```python
    if not isinstance(voltages, Matrix):
        raise ValueError("The input must be the list of V1 and V2.")

    j_vec = [[0, 0], [0, 0]]
    jacobian = Matrix(j_vec, 2, 2)

    jacobian[0][0] = - 1 / r - (isa / ktq * math.exp((voltages[0][0] - voltages[1][0]) / ktq))
    jacobian[0][1] = isa / ktq * math.exp((voltages[0][0] - voltages[1][0]) / ktq)
    jacobian[1][0] = jacobian[0][1]
    jacobian[1][1] = - isa / ktq * math.exp((voltages[0][0] - voltages[1][0]) / ktq) \
                     - isb / ktq * math.exp(voltages[1][0] / ktq)

    inv_jacobian = jacobian.inv()

    return jacobian, inv_jacobian


def calc_f1(voltages):
    return (e - voltages[0][0]) / r - isa * (math.exp((voltages[0][0] - voltages[1][0]) / ktq) - 1)


def calc_f2(voltages):
    return isa * (math.exp((voltages[0][0] - voltages[1][0]) / ktq) - 1) - isb * (math.exp(voltages[1][0]
↪  / ktq) - 1)


def calc_norm_vec(vector):
    if vector.cols > 1:
        raise ValueError("The vector must be a one-column one!")

    result = 0
    for i in range(vector.rows):
        result += pow(vector[i][0], 2)

    return result
```

Listing 4: Gauss-Legendre Integration (`integration.py`).

```python
import math
from math import log


def gauss_legendre_integration(function, lim_low, lim_high, n, real_value):
    width = (lim_high - lim_low) / n

    result = 0
    x = lim_low
    for i in range(n):
        result += function(x + width / 2) * width
        x += width

    error = abs(result - real_value)
    return result, error


def nested_integration(lim_low, lim_high, n, real_value):
    width = (lim_high - lim_low) / n

    result = 0
    x = lim_low
    for i in range(n):
        x_0 = x + width / 2
        result += (log(x_0) / math.sqrt(0.04 - x_0 ** 2)) * width
        x += width

    error = abs(result - real_value)
    return result, error


```

```python
def modified_width(function, lim_low, lim_high, n, real_value):
    width = []
    for i in range(n):
        temp = (lim_high - lim_low) * (i + 1) / (5.5 * n)
        width.append(temp)

    result = 0
    x = lim_low
    for i in range(n):
        result += function(x + width[i] / 2) * width[i]
        x += width[i]

    error = abs(result - real_value)
    return result, error


def modified_width_nested_integration(lim_low, lim_high, n, real_value):
    width = []
    for i in range(n):
        temp = (lim_high - lim_low) * (i + 1) / (5.5 * n)
        width.append(temp)

    result = 0
    x = lim_low
    for i in range(n):
        x_0 = x + width[i] / 2
        result += (log(x_0) / math.sqrt(0.04 - x_0 ** 2)) * width[i]
        x += width[i]

    error = abs(result - real_value)
    return result, error
```