# ECSE 543: Numerical Methods

Assignment 3 Report

Wenjie Wei
260685967

December 2, 2018

# 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.

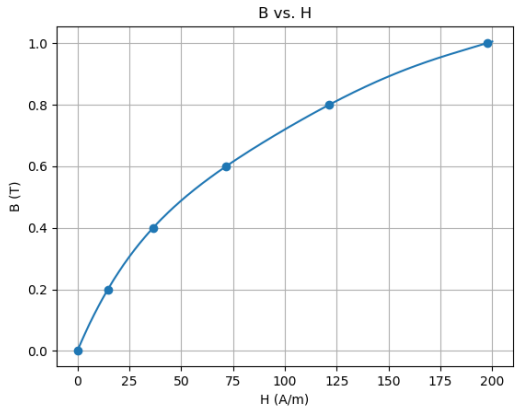


*Figure 1: Interpolation of the First Six Data Points*

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.

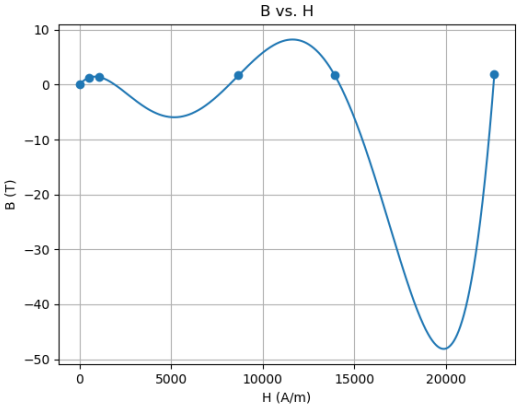


*Figure 2: Interpolation of the Second Six Data Points*

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

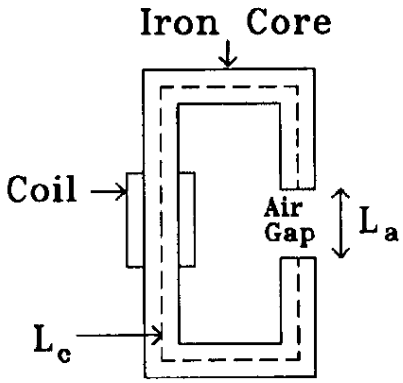


*Figure 3: The Magnetic Circuit Discussed About*

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.

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

## 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 $\psi$, 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 $1 \times 10^{-4} m^2$.

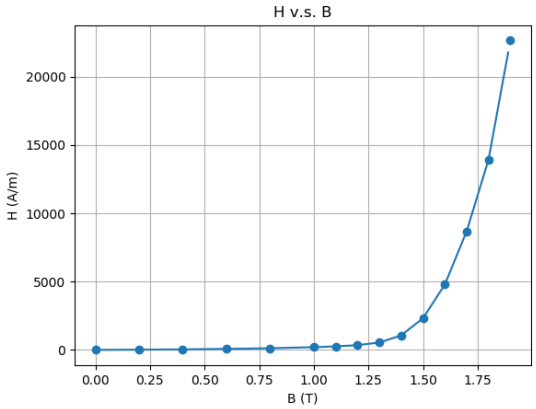


*Figure 4: Plot of the Piecewise Polynomial*

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 $5 \times 10^{-9}$. 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

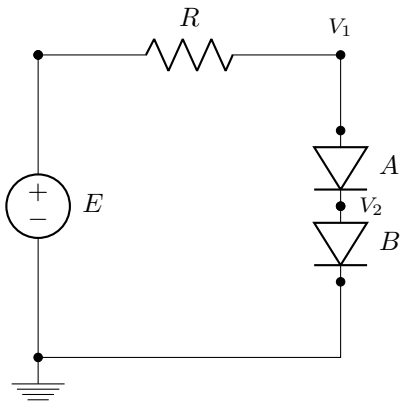


*Figure 7: The Diode Circuit to be Investigated*

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
0 0.00000000 0.00000000 0.00039063 0.00000000
1 0.19812073 0.08341926 -0.00007417 0.00004800
2 0.18413995 0.08433690 -0.00001156 0.00001154
3 0.18230749 0.08710807 -0.00000069 0.00000049
```

*Figure 8: Values During Netwon Raphson Iterations*

# A Code Listings

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

```
import math


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):
```

```
66              result_coeff = self._coeff
67              result_coeff[0] -= other
68
69          else:
70              self_has_higher_order = (max(self.order, other.order) == self.order)
71
72              if self_has_higher_order:
73                  for i in range(len(other.coefficient), len(self.coefficient)):
74                      other.coefficient.append(0)
75              else:
76                  for i in range(len(self.coefficient), len(other.coefficient)):
77                      self.coefficient.append(0)
78
79              for i in range(len(self.coefficient)):
80                  result_coeff.append(self.coefficient[i] - other.coefficient[i])
81
82          return Polynomial(result_coeff)
83
84      def __mul__(self, other):
85          result_coefficients = []
86
87          result_order = self.order + other.order
88
89          for i in range(result_order + 1):
90              coefficient = 0
91              for j in range(self.order + 1):
92                  for k in range(other.order + 1):
93                      if j + k == i:
94                          coefficient += self[j] * other[k]
95
96              result_coefficients.append(coefficient)
97
98          return Polynomial(result_coefficients)
99
100     def toString(self):
101         print("y = ", end="")
102         for i in range(self.order, 0, -1):
103             if self[i] != 1 and self[i] != -1 and self[i] != 0:
104                 if self[i] >= 0:
105                     print("+ " + str(self[i]) + "x^" + str(i), end=" ")
106                 else:
107                     print("- " + str(-self[i]) + "x^" + str(i), end=" ")
108             elif self[i] == 1:
109                 print("+ x^" + str(i), end=" ")
110             elif self[i] == -1:
111                 print("- x^" + str(i), end=" ")
112             else:
113                 pass
114
115         if self[0] < 0:
116             print("- " + str(-self[0]))
117         else:
118             print("+ " + str(self[0]))
119
120     def modify_const(self, value):
121         self._coeff[0] = value
122
123     @property
124     def order(self):
125         return self._order
126
127     @property
128     def coefficient(self):
129         return self._coeff
130
131
132 class LagrangePolynomial(object):
133     def __init__(self, n, xr, j, xj):
134         """
135         Construct a Lagrange polynomial.
```

```
136
137             :param n: how many points are on the x axis
138             :param xr: the values of x
139             :param j: the position of the current x
140             :param xj: the value of x at position j
141             """
142             self._order = n
143             self._j = j
144             self._xr = []
145             self._xj = xj
146
147             self._x = 0
148
149             for i in range(len(xr)):
150                 self._xr.append(-xr[i])
151
152             self._numerator = self._create_numerator()
153             self._denominator = self._create_denominator(xj)
154
155             self._polynomial = self._create_polynomial()
156
157         def _create_numerator(self):
158             """
159             This method creates the list of the parameters x_r.
160
161             :return: no return value
162             """
163             i = 0
164             result_numerator = Polynomial([1])
165
166             while i < self._order:
167                 if i == self.j:
168                     i += 1
169
170                 if i >= self._order:
171                     break
172
173                 result_numerator *= Polynomial([self._xr[i], 1])
174                 i += 1
175
176             return result_numerator
177
178         def _create_denominator(self, x):
179             """
180             This method calculates the numerical result of the denominator.
181
182             :return: the value in decimal of the denominator.
183             """
184
185             return self._numerator.calculate(x)
186
187         def _create_polynomial(self):
188             """
189             This method creates the general form of the lagrange polynomial.
190             :return:
191             """
192             denom = Polynomial([1 / self._denominator])
193
194             return denom * self._numerator
195
196         def set_x(self, value):
197             self._x = value
198
199         @property
200         def j(self):
201             return self._j
202
203         @property
204         def xj(self):
205             return self._xj
```

```
206
207     @property
208     def denominator(self):
209         return self._denominator
210
211     @property
212     def numerator(self):
213         return self._numerator
214
215     @property
216     def poly(self):
217         return self._polynomial
218
219
220 if __name__ == "__main__":
221     coeff1 = Polynomial([2])
222     coeff2 = Polynomial([4, 5, 7, 8])
223
224     coeff2.toString()
225     (coeff2 - 3).toString()
```

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

```
1  from polynomial import Polynomial, LagrangePolynomial
2
3
4  TOLERANCE = 1e-6
5
6  def lagrange_full_domain(xr, y, points=None):
7      """
8      This is the method for the lagrange full domain interpolation.
9      X is the variable that varies.
10     Y is the variable that varies with respect to X.
11
12     :param X: X vector of type Matrix
13     :param Y: Y vector of type Matrix
14     :param points: select the range of data to be interpolated if needed.
15     :return: Polynomial expression for y(x)
16     """
17     result_polynomial = Polynomial([0])
18
19     if points is None:
20         for j in range(len(xr)):
21             xj = xr[j]
22             aj = y[j]
23
24             temp_lagrange_poly = LagrangePolynomial(len(xr), xr, j, xj)
25
26             result_polynomial += Polynomial([aj]) * temp_lagrange_poly.poly
27
28     else:
29         pass
30
31     return result_polynomial
32
33
34 def cubic_hermite(xr, y, slopes):
35     result = Polynomial([0])
36
37     for j in range(len(xr)):
38         xj = xr[j]
39         aj = y[j]
40         bj = slopes[j]
41
42         temp = LagrangePolynomial(len(xr), xr, j, xj).poly
43         lagrange_backup = LagrangePolynomial(len(xr), xr, j, xj).poly
44
45         # Calculate the polynomial u(x)
46         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):
    if not isinstance(voltages, Matrix):
```

```
111             raise ValueError("The input must be the list of V1 and V2.")
112
113         j_vec = [[0, 0], [0, 0]]
114         jacobian = Matrix(j_vec, 2, 2)
115
116         jacobian[0][0] = - 1 / r - (isa / ktq * math.exp((voltages[0][0] - voltages[1][0]) / ktq))
117         jacobian[0][1] = isa / ktq * math.exp((voltages[0][0] - voltages[1][0]) / ktq)
118         jacobian[1][0] = jacobian[0][1]
119         jacobian[1][1] = - isa / ktq * math.exp((voltages[0][0] - voltages[1][0]) / ktq) \
120                          - isb / ktq * math.exp(voltages[1][0] / ktq)
121
122         inv_jacobian = jacobian.inv()
123
124         return jacobian, inv_jacobian
125
126
127     def calc_f1(voltages):
128         return (e - voltages[0][0]) / r - isa * (math.exp((voltages[0][0] - voltages[1][0]) / ktq) - 1)
129
130
131     def calc_f2(voltages):
132         return isa * (math.exp((voltages[0][0] - voltages[1][0]) / ktq) - 1) - isb * (math.exp(voltages[1][0]
        ↪  / ktq) - 1)
133
134
135     def calc_norm_vec(vector):
136         if vector.cols > 1:
137             raise ValueError("The vector must be a one-column one!")
138
139         result = 0
140         for i in range(vector.rows):
141             result += pow(vector[i][0], 2)
142
143         return result
```