

ECSE 543

Assignment 3

Sean Stappas
260639512

December 7th, 2017

Contents

1	BH Interpolation	2
1.a	Lagrange Polynomials	2
1.b	Full-Domain Lagrange Polynomials	2
1.c	Cubic Hermite Polynomials	2
2	Magnetic Circuit	2
2.a	Flux Equation	2
2.b	Newton-Raphson	2
2.c	Successive Substitution	3
3	Diode Circuit	3
3.a	Voltage Equations	3
3.b	Newton-Raphson	3
4	Function Integration	3
4.a	Cosine Integration	3
4.b	Log Integration	4
4.c	Log Integration Improvement	4
	Appendix A Code Listings	5
	Appendix B Output Logs	9

Introduction

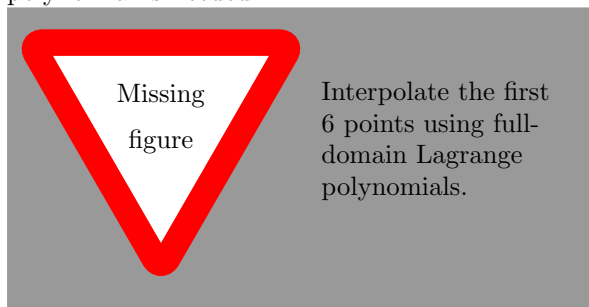
The code for this assignment was created in Python 2.7 and can be seen in Appendix A. To perform the required tasks in this assignment, the `Matrix` class from Assignment 1 was used, with useful methods such as `add`, `multiply`, `transpose`, etc. This package can be seen in the `matrices.py` file shown in Listing 1. The only packages used that are not built-in are those for creating the plots for this report, i.e., `matplotlib` for plotting. The structure of the rest of the code will be discussed as appropriate for each question. Output logs of the program are provided in Appendix B.

1 BH Interpolation

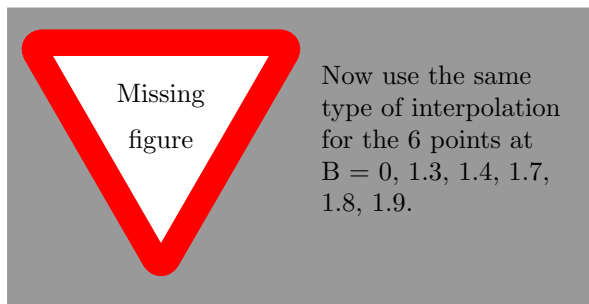
The source code for the Question 1 program can be seen in the `q1.py` file shown in Listing 2.

1.a Lagrange Polynomials

To interpolate 6 points, a 5th-order Lagrange polynomial is needed.



1.b Full-Domain Lagrange Polynomials



The result is not plausible because of the characteristic “wiggles” seen when using full-domain Lagrange polynomials over a wide range.

1.c Cubic Hermite Polynomials

The slopes at each of the 6 points can be approximated by the slope of the straight line passing through the two adjacent points, i.e., the point immediately before and the point after the point of interest. For the boundary points of 0 T and 1.9 T, the slope of the line formed by the point and one adjacent point can be used.

2 Magnetic Circuit

The source code for the Question 2 program can be seen in the `q2.py` file shown in Listing 3.

2.a Flux Equation

The magnetic analog of KVL can be seen in Equation (1).

$$(\mathcal{R}_a + \mathcal{R}_c)\psi = \mathcal{F} \quad (1)$$

where \mathcal{R}_a is the reluctance of the air gap, \mathcal{R}_c is the reluctance of the coil, and \mathcal{F} is the magnetomotive force. Plugging in the relevant variables from the problem, we obtain Equation (2).

$$\left(\frac{L_a}{A\mu_o} + \frac{L_c}{A\mu_c(\psi)} \right) \psi - NI = 0 \quad (2)$$

where $\mu_c(\psi)$ is a function of ψ given by Equation (3).

$$\mu_c(\psi) = \frac{B}{H} = \frac{\psi}{AH} \quad (3)$$

Plugging Equation (3) into Equation (2), we obtain Equation (4).

$$\left(\frac{L_a}{A\mu_o} + \frac{L_c H}{\psi} \right) \psi - NI = 0 \quad (4)$$

Simplifying the terms, we obtain Equation (5).

$$f(\psi) = \frac{L_a \psi}{A\mu_o} + L_c H - NI = 0 \quad (5)$$

Finally, if we plug in the values from the question, we obtain Equation (6), where the coefficients of the terms are calculated in the `q2.py` script shown in Listing 2.

$$f(\psi) = 3.979 \times 10^7 \psi + 0.3H - 8000 = 0 \quad (6)$$

2.b Newton-Raphson

$$B = \frac{\psi}{A} \quad (7)$$

2.c Successive Substitution

3 Diode Circuit

The source code for the Question 3 program can be seen in the `q3.py` file shown in Listing 4.

3.a Voltage Equations

The current-voltage relationship for a diode is given by Equation (8).

$$I = I_s \left(\exp \left[\frac{qv}{kT} \right] - 1 \right) \quad (8)$$

Let the nodal voltage at the anode of the A diode be denoted by v_A and that of the B diode by v_B . Let the current through the circuit be denoted by I . The diode equations for A and B can be seen in Equations (9) and (10).

$$I = I_{sA} \left(\exp \left[\frac{q(v_A - v_B)}{kT} \right] - 1 \right) \quad (9)$$

$$I = I_{sB} \left(\exp \left[\frac{qv_B}{kT} \right] - 1 \right) \quad (10)$$

By KVL, we also have Equation (11), relating V_A and I .

$$I = \frac{E - v_A}{R} \quad (11)$$

Equating Equations (9) and (11), we obtain the nonlinear equation for v_A , shown in Equation (12).

$$\begin{aligned} f_A(v_A, v_B) &= v_A + RI_{sA} \left(\exp \left[\frac{q(v_A - v_B)}{kT} \right] - 1 \right) - E \\ &= 0 \end{aligned} \quad (12)$$

Equating Equations (9) and (10), we obtain the nonlinear equation for v_B , shown in Equation (13).

$$\begin{aligned} f_B(v_A, v_B) &= I_{sA} \left(\exp \left[\frac{q(v_A - v_B)}{kT} \right] - 1 \right) \\ &\quad - I_{sB} \left(\exp \left[\frac{qv_B}{kT} \right] - 1 \right) = 0 \end{aligned} \quad (13)$$

The total system of equations can then be expressed by Equation (14).

$$\mathbf{f}(\mathbf{v}_n) = \begin{bmatrix} f_A(v_A, v_B) \\ f_B(v_A, v_B) \end{bmatrix} = \mathbf{0} \quad (14)$$

3.b Newton-Raphson

To find an expression for the Jacobian matrix \mathbf{F} , we must first find expressions for all the partials of f_A and f_B . These are shown in Equations (15) to (18).

$$\frac{\partial f_A}{\partial v_A} = 1 + RI_{sA} \left(\exp \left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \quad (15)$$

$$\frac{\partial f_A}{\partial v_B} = -RI_{sA} \left(\exp \left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \quad (16)$$

$$\frac{\partial f_B}{\partial v_A} = I_{sA} \left(\exp \left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \quad (17)$$

$$\begin{aligned} \frac{\partial f_B}{\partial v_B} &= -I_{sA} \left(\exp \left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \\ &\quad - I_{sB} \left(\exp \left[\frac{qv_B}{kT} \right] \frac{q}{kT} \right) \end{aligned} \quad (18)$$

With these equations, the Jacobian matrix \mathbf{F} is given by Equation (19).

$$\mathbf{F} = \begin{bmatrix} \frac{\partial f_A}{\partial v_A} & \frac{\partial f_A}{\partial v_B} \\ \frac{\partial f_B}{\partial v_A} & \frac{\partial f_B}{\partial v_B} \end{bmatrix} \quad (19)$$

With this information, we can apply the Newton-Raphson update in matrix form, shown in Equation (20).

$$\mathbf{v}_n^{(k+1)} \leftarrow \mathbf{v}_n^{(k)} - (\mathbf{F}^{(k)})^{-1} \mathbf{f}^{(k)} \quad (20)$$

4 Function Integration

The source code for the Question 4 program can be seen in the `q4.py` file shown in Listing 5.

4.a Cosine Integration

The integral I we wish to solve is shown in Equation (21).

$$I = \int_0^1 \cos x dx \quad (21)$$

To use Gauss-Legendre integration, the $[0, 1]$ range of x must be mapped to the $[-1, 1]$ range of ζ . This mapping between x and ζ is given by Equation (22).

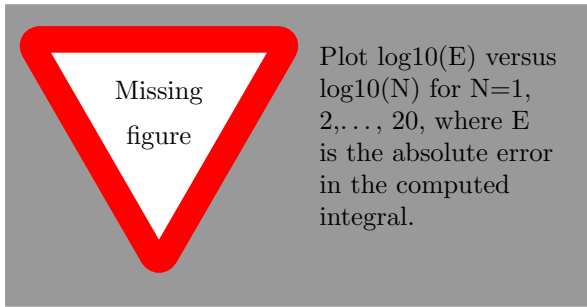
$$x = \frac{1}{2}(\zeta + 1) \quad (22)$$

The updated integral equation is then given by Equation (23).

$$I = \frac{1}{2} \int_{-1}^1 \cos \left[\frac{1}{2}(\zeta + 1) \right] d\zeta \quad (23)$$

The equation for the absolute error used is shown in Equation (24), where I_{actual} is the actual value of the integral, and I_{approx} is the approximate value computed by Gauss-Legendre integration.

$$E = |I_{actual} - I_{approx}| \quad (24)$$



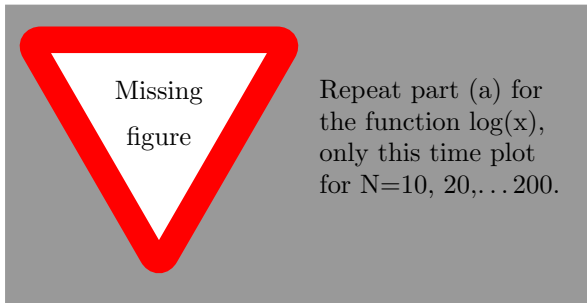
4.b Log Integration

The integral I we would wish to solve is shown in Equation (25).

$$I = \int_0^1 \log x dx \quad (25)$$

Using Equation (22) for the log function, we obtain the integral shown in Equation (26).

$$I = \frac{1}{2} \int_{-1}^1 \log \left[\frac{1}{2}(\zeta + 1) \right] d\zeta \quad (26)$$



4.c Log Integration Improvement

A Code Listings

Listing 1: Custom matrix package (*matrices.py*).

```
1  from __future__ import division
2
3  import copy
4  import csv
5  from ast import literal_eval
6
7  import math
8
9
10 class Matrix:
11     def __init__(self, data):
12         self.data = data
13         self.num_rows = len(data)
14         self.num_cols = len(data[0])
15
16     def __str__(self):
17         string = ''
18         for row in self.data:
19             string += '\n'
20             for val in row:
21                 string += '{:6.2f} '.format(val)
22         return string
23
24     def integer_string(self):
25         string = ''
26         for row in self.data:
27             string += '\n'
28             for val in row:
29                 string += '{:3.0f} '.format(val)
30         return string
31
32     def __add__(self, other):
33         if len(self) != len(other) or len(self[0]) != len(other[0]):
34             raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is
35                 ↪ {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
36
37         return Matrix([[self[row][col] + other[row][col] for col in range(self.num_cols)]
38             for row in range(self.num_rows)])
39
40     def __sub__(self, other):
41         if len(self) != len(other) or len(self[0]) != len(other[0]):
42             raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
43                 ↪ is {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
44
45         return Matrix([[self[row][col] - other[row][col] for col in range(self.num_cols)]
46             for row in range(self.num_rows)])
47
48     def __mul__(self, other):
49         if type(other) == float or type(other) == int:
50             return self.scalar_multiply(other)
51
52         if self.num_cols != other.num_rows:
53             raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
54                 ↪ B is {}x{}.'.format(self.num_rows, self.num_cols, other.num_rows, other.num_cols))
55
56         # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
57         product = Matrix.empty(self.num_rows, other.num_cols)
58         for i in range(self.num_rows):
59             for j in range(other.num_cols):
60                 row_sum = 0
61                 for k in range(self.num_cols):
62                     row_sum += self[i][k] * other[k][j]
```

```

63         product[i][j] = row_sum
64     return product
65
66 def scalar_multiply(self, scalar):
67     return Matrix([[self[row][col] * scalar for col in range(self.num_cols)] for row in
        ↪ range(self.num_rows)])
68
69 def __div__(self, other):
70     """
71     Element-wise division.
72     """
73     if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
74         raise ValueError('Incompatible matrix sizes.')
75     return Matrix([[self[row][col] / other[row][col] for col in range(self.num_cols)]
76                    for row in range(self.num_rows)])
77
78 def __neg__(self):
79     return Matrix([[-self[row][col] for col in range(self.num_cols)] for row in range(self.num_rows)])
80
81 def __deepcopy__(self, memo):
82     return Matrix(copy.deepcopy(self.data))
83
84 def __getitem__(self, item):
85     return self.data[item]
86
87 def __len__(self):
88     return len(self.data)
89
90 def item(self):
91     """
92     :return: the single element contained by this matrix, if it is 1x1.
93     """
94     if not (self.num_rows == 1 and self.num_cols == 1):
95         raise ValueError('Matrix is not 1x1')
96     return self.data[0][0]
97
98 def is_positive_definite(self):
99     """
100     :return: True if the matrix is positive-definite, False otherwise.
101     """
102     A = copy.deepcopy(self.data)
103     for j in range(self.num_rows):
104         if A[j][j] <= 0:
105             return False
106         A[j][j] = math.sqrt(A[j][j])
107         for i in range(j + 1, self.num_rows):
108             A[i][j] = A[i][j] / A[j][j]
109             for k in range(j + 1, i + 1):
110                 A[i][k] = A[i][k] - A[i][j] * A[k][j]
111     return True
112
113 def transpose(self):
114     """
115     :return: the transpose of the current matrix
116     """
117     return Matrix([[self.data[row][col] for row in range(self.num_rows)] for col in
        ↪ range(self.num_cols)])
118
119 def mirror_horizontal(self):
120     """
121     :return: the horizontal mirror of the current matrix
122     """
123     return Matrix([[self.data[self.num_rows - row - 1][col] for col in range(self.num_cols)]
124                    for row in range(self.num_rows)])
125
126 def empty_copy(self):
127     """
128     :return: an empty matrix of the same size as the current matrix.
129     """
130     return Matrix.empty(self.num_rows, self.num_cols)

```

```

131
132 def infinity_norm(self):
133     if self.num_cols > 1:
134         raise ValueError('Not a column vector.')
135     return max([abs(x) for x in self.transpose()[0]])
136
137 def two_norm(self):
138     if self.num_cols > 1:
139         raise ValueError('Not a column vector.')
140     return math.sqrt(sum([x ** 2 for x in self.transpose()[0]]))
141
142 def save_to_csv(self, filename):
143     """
144     Saves the current matrix to a CSV file.
145
146     :param filename: the name of the CSV file
147     """
148     with open(filename, "wb") as f:
149         writer = csv.writer(f)
150         for row in self.data:
151             writer.writerow(row)
152
153 def save_to_latex(self, filename):
154     """
155     Saves the current matrix to a latex-readable matrix.
156
157     :param filename: the name of the CSV file
158     """
159     with open(filename, "wb") as f:
160         for row in range(self.num_rows):
161             for col in range(self.num_cols):
162                 f.write('{}'.format(self.data[row][col]))
163                 if col < self.num_cols - 1:
164                     f.write('& ')
165             if row < self.num_rows - 1:
166                 f.write('\n')
167
168 @staticmethod
169 def multiply(*matrices):
170     """
171     Computes the product of the given matrices.
172
173     :param matrices: the matrix objects
174     :return: the product of the given matrices
175     """
176     n = matrices[0].rows
177     product = Matrix.identity(n)
178     for matrix in matrices:
179         product = product * matrix
180     return product
181
182 @staticmethod
183 def empty(num_rows, num_cols):
184     """
185     Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
186
187     :param num_rows: number of rows
188     :param num_cols: number of columns
189     :return: the empty matrix
190     """
191     return Matrix([[0 for _ in range(num_cols)] for _ in range(num_rows)])
192
193 @staticmethod
194 def identity(n):
195     """
196     Returns the identity matrix of the given size.
197
198     :param n: the size of the identity matrix (number of rows or columns)
199     :return: the identity matrix of size n
200     """

```



```

201         return Matrix.diagonal_single_value(1, n)
202
203     @staticmethod
204     def diagonal(values):
205         """
206         Returns a diagonal matrix with the given values along the main diagonal.
207
208         :param values: the values along the main diagonal
209         :return: a diagonal matrix with the given values along the main diagonal
210         """
211         n = len(values)
212         return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
213
214     @staticmethod
215     def diagonal_single_value(value, n):
216         """
217         Returns a diagonal matrix of the given size with the given value along the diagonal.
218
219         :param value: the value of each element on the main diagonal
220         :param n: the size of the matrix
221         :return: a diagonal matrix of the given size with the given value along the diagonal.
222         """
223         return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
224
225     @staticmethod
226     def column_vector(values):
227         """
228         Transforms a row vector into a column vector.
229
230         :param values: the values, one for each row of the column vector
231         :return: the column vector
232         """
233         return Matrix([[value] for value in values])
234
235     @staticmethod
236     def csv_to_matrix(filename):
237         """
238         Reads a CSV file to a matrix.
239
240         :param filename: the name of the CSV file
241         :return: a matrix containing the values in the CSV file
242         """
243         with open(filename, 'r') as csv_file:
244             reader = csv.reader(csv_file)
245             data = []
246             for row_number, row in enumerate(reader):
247                 data.append([literal_eval(val) for val in row])
248         return Matrix(data)

```

Listing 2: Question 1 (q1.py).

```

1  def q1():
2      print('\n=== Question 1 ===')
3      q1a()
4
5
6  def q1a():
7      pass
8
9
10 if __name__ == '__main__':
11     q1()

```

Listing 3: Question 2 (q2.py).

```

1  import math
2
3  L_a = 5e-3

```

```

4  L_c = 0.3
5  A = 1e-4
6  N = 1000
7  I = 8
8  mu_0 = 4e-7 * math.pi
9
10
11 def q2():
12     print('\n=== Question 2 ===')
13     q2b()
14
15
16 def q2b():
17     print('Flux equation: ')
18     coeff_1 = L_a / (A * mu_0)
19     coeff_2 = L_c
20     coeff_3 = N * I
21     eq = 'f(\psi) = \SI{1.3e}{\psi} + {}H - {} = 0'.format(coeff_1, coeff_2, coeff_3)
22     print(eq)
23     with open('report/latex/flux_equation.txt', 'w') as f:
24         f.write(eq)
25
26
27 if __name__ == '__main__':
28     q2()

```

Listing 4: Question 3 (q3.py).

```

1  def q3():
2      print('\n=== Question 3 ===')
3
4
5  if __name__ == '__main__':
6      q3()

```

Listing 5: Question 4 (q4.py).

```

1  def q4():
2      print('\n=== Question 4 ===')
3
4
5  if __name__ == '__main__':
6      q4()

```

B Output Logs

Listing 6: Output of Question 1 program (q1.txt).

1

Listing 7: Output of Question 2 program (q2.txt).

1

Listing 8: Output of Question 3 program (q3.txt).

1

Listing 9: Output of Question 4 program (q4.txt).

1