## ECSE 543 Assignment 3

Sean Stappas 260639512

December  $7^{\text{th}}$ , 2017

## Contents

1	$\mathbf{BH}$	Interpolation	2
	1.a	Lagrange Polynomials	2
		Full-Domain Lagrange Polynomials	
		Cubic Hermite Polynomials	
2		gnetic Circuit	2
	2.a	Flux Equation	2
	2.b	Newton-Raphson	2
		Successive Substitution	
3	Diode Circuit		
	3.a	Voltage Equations	2
		Newton-Raphson	
4	Fun	ction Integration	2
		Cosine Integration	2
		Log Integration	
		Log Integration Improvement	
$\mathbf{A}_{\mathbf{J}}$	ppen	dix A Code Listings	3
${f A}_1$	Appendix B Output Logs		

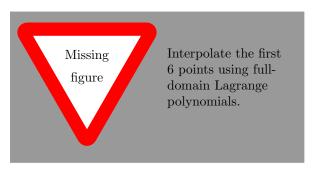
#### Introduction

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.

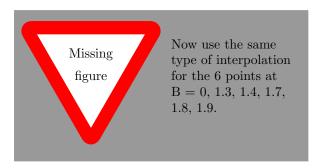
#### 1 BH Interpolation

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

#### 1.a Lagrange Polynomials



# 1.b Full-Domain Lagrange Polynomials



#### 1.c Cubic Hermite Polynomials

### 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

$$f(\psi) = 0 \tag{1}$$

- 2.b Newton-Raphson
- 2.c Successive Substitution

#### 3 Diode Circuit

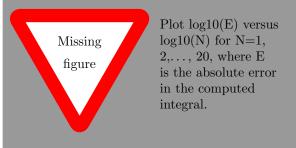
The source code for the Question 3 program can be seen in the q3.py file shown in Listing 4.

- 3.a Voltage Equations
- 3.b Newton-Raphson

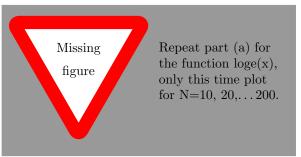
#### 4 Function Integration

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

#### 4.a Cosine Integration



#### 4.b Log Integration



#### 4.c Log Integration Improvement

#### A Code Listings

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

```
product[i][j] = row_sum
63
             return product
64
65
         def scalar_multiply(self, scalar):
66
              return Matrix([[self[row][col] * scalar for col in range(self.num_cols)] for row in
67

    range(self.num_rows)])

 68
69
         def __div__(self, other):
70
71
             Element-wise division.
72
             if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
73
                 raise ValueError('Incompatible matrix sizes.')
             return Matrix([[self[row][col] / other[row][col] for col in range(self.num_cols)]
75
76
                             for row in range(self.num_rows)])
77
         def __neg__(self):
78
79
              return Matrix([[-self[row][col] for col in range(self.num_cols)] for row in range(self.num_rows)])
80
         def __deepcopy__(self, memo):
81
82
             return Matrix(copy.deepcopy(self.data))
83
84
         def __getitem__(self, item):
             return self.data[item]
85
86
87
         def __len__(self):
             return len(self.data)
88
89
         def item(self):
90
91
             :return: the single element contained by this matrix, if it is 1x1.
92
93
             if not (self.num_rows == 1 and self.num_cols == 1):
94
                 raise ValueError('Matrix is not 1x1')
95
             return self.data[0][0]
96
97
         def is_positive_definite(self):
98
99
              : return: \ \textit{True if the matrix if positive-definite, False otherwise}.
100
101
             A = copy.deepcopy(self.data)
102
103
             for j in range(self.num_rows):
                  if A[j][j] <= 0:
104
                      return False
105
                  A[j][j] = math.sqrt(A[j][j])
                  for i in range(j + 1, self.num_rows):
107
                      A[i][j] = A[i][j] / A[j][j]
108
                      for k in range(j + 1, i + 1):
109
                          A[i][k] = A[i][k] - A[i][j] * A[k][j]
110
111
             return True
112
         def transpose(self):
113
114
              :return: the transpose of the current matrix
115
116
             return Matrix([[self.data[row][col] for row in range(self.num_rows)] for col in
117

    range(self.num_cols)])

118
119
         def mirror_horizontal(self):
120
              :return: the horizontal mirror of the current matrix
121
122
             return Matrix([[self.data[self.num_rows - row - 1][col] for col in range(self.num_cols)]
123
                             for row in range(self.num_rows)])
124
125
126
         def empty_copy(self):
127
              :return: an empty matrix of the same size as the current matrix.
128
129
             return Matrix.empty(self.num_rows, self.num_cols)
130
```

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

```
201
              return Matrix.diagonal_single_value(1, n)
202
          @staticmethod
203
         def diagonal(values):
204
205
              Returns a diagonal matrix with the given values along the main diagonal.
206
207
208
              :param values: the values along the main diagonal
              :return: a diagonal matrix with the given values along the main diagonal
209
210
              n = len(values)
211
             return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
212
         Ostaticmethod
214
215
         def diagonal_single_value(value, n):
216
              Returns a diagonal matrix of the given size with the given value along the diagonal.
217
218
              :param value: the value of each element on the main diagonal
219
              :param n: the size of the matrix
220
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
         @staticmethod
225
226
         def column_vector(values):
227
              Transforms a row vector into a column vector.
228
              :param values: the values, one for each row of the column vector
230
231
              :return: the column vector
232
             return Matrix([[value] for value in values])
233
234
          Ostaticmethod
235
         def csv_to_matrix(filename):
236
237
              Reads a CSV file to a matrix.
238
239
240
              :param filename: the name of the CSV file
              : return: \ a \ \textit{matrix} \ \textit{containing} \ \textit{the values} \ \textit{in the CSV} \ \textit{file}
241
242
              with open(filename, 'r') as csv_file:
243
                 reader = csv.reader(csv_file)
244
                  data = []
                  for row_number, row in enumerate(reader):
246
                      data.append([literal_eval(val) for val in row])
247
                  return Matrix(data)
                                             Listing 2: Question 1 (q1.py).
     def q1():
         print('\n=== Question 1 ===')
 2
 3
     if __name__ == '__main__':
 5
         q1()
                                             Listing 3: Question 2 (q2.py).
     def q2():
         print('\n=== Question 2 ===')
 2
 3
     if __name__ == '__main__':
 5
         q2()
```

```
Listing 4: Question 3 (q3.py).

def q3():
    print('\n=== Question 3 ===')

if __name__ == '__main__':
    q3()

Listing 4: Question 3 (q3.py).

Listing 5: Question 4 (q4.py).

Listing 5: Question 4 (q4.py).

if __name__ == '__main__':
    q4()
```

### B Output Logs

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

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

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

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