ECSE 543
Assignment 3

Sean Stappas
260639512

December 7th, 2017

Contents

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

Interpolate the first 6 points using full-domain Lagrange polynomials.

1.b Full-Domain Lagrange Polynomials

Now use the same type of interpolation for the 6 points at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9.

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

Plot log10(E) versus log10(N) for N=1, 2,. . . , 20, where E is the absolute error in the computed integral.

4.b Log Integration

Repeat part (a) for the function loge(x), only this time plot for N=10, 20,. . . 200.

4.c Log Integration Improvement

A Code Listings

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

```
from __future__ import division

import copy
import csv
from ast import literal_eval

import math


class Matrix:
    def __init__(self, data):
        self.data = data
        self.num_rows = len(data)
        self.num_cols = len(data[0])

    def __str__(self):
        string = ''
        for row in self.data:
            string += '\n'
            for val in row:
                string += '{:6.2f} '.format(val)
        return string

    def integer_string(self):
        string = ''
        for row in self.data:
            string += '\n'
            for val in row:
                string += '{:3.0f} '.format(val)
        return string

    def __add__(self, other):
        if len(self) != len(other) or len(self[0]) != len(other[0]):
            raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is
             ↪  {}x{}.'
                             .format(len(self), len(self[0]), len(other), len(other[0])))

        return Matrix([[self[row][col] + other[row][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def __sub__(self, other):
        if len(self) != len(other) or len(self[0]) != len(other[0]):
            raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
             ↪  is {}x{}.'
                             .format(len(self), len(self[0]), len(other), len(other[0])))

        return Matrix([[self[row][col] - other[row][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def __mul__(self, other):
        if type(other) == float or type(other) == int:
            return self.scalar_multiply(other)

        if self.num_cols != other.num_rows:
            raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
             ↪  B is {}x{}.'
                             .format(self.num_rows, self.num_cols, other.num_rows, other.num_cols))

        # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
        product = Matrix.empty(self.num_rows, other.num_cols)
        for i in range(self.num_rows):
            for j in range(other.num_cols):
                row_sum = 0
                for k in range(self.num_cols):
                    row_sum += self[i][k] * other[k][j]
```

```
                product[i][j] = row_sum
        return product

    def scalar_multiply(self, scalar):
        return Matrix([[self[row][col] * scalar for col in range(self.num_cols)] for row in
        ↪  range(self.num_rows)])

    def __div__(self, other):
        """
        Element-wise division.
        """
        if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
            raise ValueError('Incompatible matrix sizes.')
        return Matrix([[self[row][col] / other[row][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def __neg__(self):
        return Matrix([[-self[row][col] for col in range(self.num_cols)] for row in range(self.num_rows)])

    def __deepcopy__(self, memo):
        return Matrix(copy.deepcopy(self.data))

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

    def __len__(self):
        return len(self.data)

    def item(self):
        """
        :return: the single element contained by this matrix, if it is 1x1.
        """
        if not (self.num_rows == 1 and self.num_cols == 1):
            raise ValueError('Matrix is not 1x1')
        return self.data[0][0]

    def is_positive_definite(self):
        """
        :return: True if the matrix if positive-definite, False otherwise.
        """
        A = copy.deepcopy(self.data)
        for j in range(self.num_rows):
            if A[j][j] <= 0:
                return False
            A[j][j] = math.sqrt(A[j][j])
            for i in range(j + 1, self.num_rows):
                A[i][j] = A[i][j] / A[j][j]
                for k in range(j + 1, i + 1):
                    A[i][k] = A[i][k] - A[i][j] * A[k][j]
        return True

    def transpose(self):
        """
        :return: the transpose of the current matrix
        """
        return Matrix([[self.data[row][col] for row in range(self.num_rows)] for col in
        ↪  range(self.num_cols)])

    def mirror_horizontal(self):
        """
        :return: the horizontal mirror of the current matrix
        """
        return Matrix([[self.data[self.num_rows - row - 1][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def empty_copy(self):
        """
        :return: an empty matrix of the same size as the current matrix.
        """
        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
4
5   if __name__ == '__main__':
6       q1()
```

Listing 3: Question 2 (`q2.py`).

```
1   def q2():
2       print('\n=== Question 2 ===')
3
4
5   if __name__ == '__main__':
6       q2()
```

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

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


if __name__ == '__main__':
    q3()
```

Listing 5: Question 4 (`q4.py`).

```
def q4():
    print('\n=== Question 4 ===')


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`).

```

```