ECSE 543
Assignment 3

Sean Stappas
260639512

December 7th, 2017

Contents

Introduction

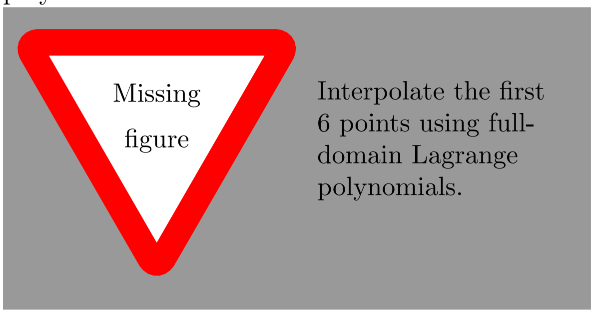
The code for this assignment was created in Python 2.7 and can be seen in Appendix A. To perform the required tasks in this assignment, the `Matrix` class from Assignment 1 was used, with useful methods such as add, multiply, transpose, etc. This package can be seen in the `matrices.py` file shown in 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

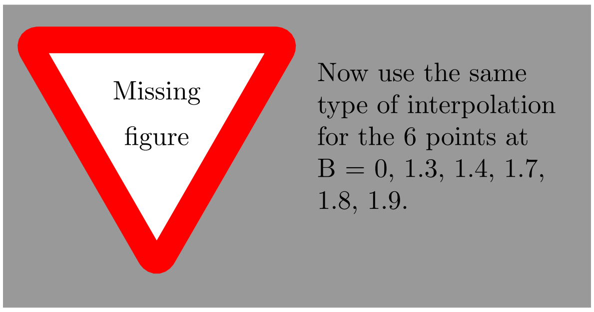
The source code for the Question 1 program can be seen in the `q1.py` file shown in Listing 2.

1.a Lagrange Polynomials

To interpolate 6 points, a 5^{th}-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).

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

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).

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

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

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

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

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

Simplifying the terms, we obtain Equation (5).

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

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.

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

2.b Newton-Raphson

$$B = \frac{\psi}{A} \tag{7}$$

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).

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

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).

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

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

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

$$I = \frac{E - v_A}{R} \tag{11}$$

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

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

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

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

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

$$\mathbf{f}(\mathbf{v_n}) = \begin{bmatrix} f_A(v_A, v_B) \\ f_B(v_A, v_B) \end{bmatrix} = \mathbf{0} \tag{14}$$

3.b Newton-Raphson

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

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

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

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

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

With these equations, the Jacobian matrix **F** is given by Equation (19).

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

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

$$\mathbf{v_n}^{(k+1)} \leftarrow \mathbf{v_n}^{(k)} - (\mathbf{F}^{(k)})^{-1} \mathbf{f}^{(k)} \tag{20}$$

The code performing this update is in the `newton_raphson.py` script and can be seen in Listing 5. The error ϵ in **f** is defined as 1×10^{-9}, where the 2-norm of **f** is used to compare to the error. The code is executed in the `q3.py` script shown in Listing 4, with output shown in Listing 9.

*Table 1: Node voltages and **f** values at every iteration of Newton-Raphson.*

1×10^{-1}

4 Function Integration

The source code for the Question 4 program can be seen in the `q4.py` file shown in Listing 6.

4.a Cosine Integration

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

$$I = \int_0^1 \cos x dx \tag{21}$$

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).

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

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

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

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.

$$E = |I_{actual} - I_{approx}| \tag{24}$$

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

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

$$I = \int_0^1 \log x dx \tag{25}$$

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

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

Repeat part (a) for the function log(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`).

```python
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 += '{:6.3f} '.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 __div__(self, other):
        """
        Element-wise division.
        """
        if type(other) == float or type(other) == int:
```

```
            return self.scalar_divide(other)

        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)

    @property
    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)])

    @property
    def infinity_norm(self):
        if self.num_cols > 1:
            raise ValueError('Not a column vector.')
        return max([abs(x) for x in self.transpose[0]])

    @property
    def two_norm(self):
        if self.num_cols > 1:
            raise ValueError('Not a column vector.')
        return math.sqrt(sum([x ** 2 for x in self.transpose[0]]))

    @property
    def values(self):
        """
        :return: the values in this matrix, in row-major order.
        """
        vals = []
        for row in self.data:
            for val in row:
                vals.append(val)
        return tuple(vals)

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

    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 scalar_divide(self, scalar):
        return Matrix([[self[row][col] / scalar for col in range(self.num_cols)] for row in range(self.num_rows)])

    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 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)

    def save_to_csv(self, filename):
        """
        Saves the current matrix to a CSV file.

        :param filename: the name of the CSV file
        """
        with open(filename, "wb") as f:
            writer = csv.writer(f)
            for row in self.data:
                writer.writerow(row)

    def save_to_latex(self, filename):
        """
        Saves the current matrix to a latex-readable matrix.

        :param filename: the name of the CSV file
        """
        with open(filename, "wb") as f:
            for row in range(self.num_rows):
                for col in range(self.num_cols):
                    f.write('{}'.format(self.data[row][col]))
                    if col < self.num_cols - 1:
                        f.write('& ')
                if row < self.num_rows - 1:
                    f.write('\\\\\n')

    @staticmethod
    def multiply(*matrices):
        """
        Computes the product of the given matrices.

        :param matrices: the matrix objects
        :return: the product of the given matrices
        """
        n = matrices[0].rows
        product = Matrix.identity(n)
        for matrix in matrices:
```

```
                product = product * matrix
        return product

    @staticmethod
    def empty(num_rows, num_cols):
        """
        Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.

        :param num_rows: number of rows
        :param num_cols: number of columns
        :return: the empty matrix
        """
        return Matrix([[0 for _ in range(num_cols)] for _ in range(num_rows)])

    @staticmethod
    def identity(n):
        """
        Returns the identity matrix of the given size.

        :param n: the size of the identity matrix (number of rows or columns)
        :return: the identity matrix of size n
        """
        return Matrix.diagonal_single_value(1, n)

    @staticmethod
    def diagonal(values):
        """
        Returns a diagonal matrix with the given values along the main diagonal.

        :param values: the values along the main diagonal
        :return: a diagonal matrix with the given values along the main diagonal
        """
        n = len(values)
        return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])

    @staticmethod
    def diagonal_single_value(value, n):
        """
        Returns a diagonal matrix of the given size with the given value along the diagonal.

        :param value: the value of each element on the main diagonal
        :param n: the size of the matrix
        :return: a diagonal matrix of the given size with the given value along the diagonal.
        """
        return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])

    @staticmethod
    def column_vector(values):
        """
        Transforms a row vector into a column vector.

        :param values: the values, one for each row of the column vector
        :return: the column vector
        """
        return Matrix([[value] for value in values])

    @staticmethod
    def csv_to_matrix(filename):
        """
        Reads a CSV file to a matrix.

        :param filename: the name of the CSV file
        :return: a matrix containing the values in the CSV file
        """
        with open(filename, 'r') as csv_file:
            reader = csv.reader(csv_file)
            data = []
            for row_number, row in enumerate(reader):
                data.append([literal_eval(val) for val in row])
            return Matrix(data)
```

Listing 2: Question 1 (q1.py).

```python
def q1():
    print('\n=== Question 1 ===')
    q1a()


def q1a():
    pass


if __name__ == '__main__':
    q1()
```

Listing 3: Question 2 (q2.py).

```python
import math

L_a = 5e-3
L_c = 0.3
A = 1e-4
N = 1000
I = 8
mu_0 = 4e-7 * math.pi


def q2():
    print('\n=== Question 2 ===')
    q2b()


def q2b():
    print('Flux equation: ')
    coeff_1 = L_a / (A * mu_0)
    coeff_2 = L_c
    coeff_3 = N * I
    eq = 'f(\psi) = \SI{{{:1.3e}}}{{}} \psi + {}H - {} = 0'.format(coeff_1, coeff_2, coeff_3)
    print(eq)
    with open('report/latex/flux_equation.txt', 'w') as f:
        f.write(eq)


if __name__ == '__main__':
    q2()
```

Listing 4: Question 3 (q3.py).

```python
from __future__ import division

from csv_saver import save_rows_to_csv
from newton_raphson import newton_raphson_solve


def q3():
    print('\n=== Question 3 ===')
    v_n, values = newton_raphson_solve()
    print('Solution: {}'.format(v_n))
    v_a, v_b = v_n.values
    print('v_a: {:.3f} mV'.format(v_a * 1000))
    print('v_b: {:.3f} mV'.format(v_b * 1000))

    save_rows_to_csv('report/csv/q3.csv', values, header=('Iteration', 'v_A', 'v_B', 'f_A', 'f_B', '|f|'))


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

Listing 5: Newton-Raphson (`newton_raphson.py`).

```
from __future__ import division

from math import exp

from matrices import Matrix

E = 220e-3
R = 500
I_SA = 0.6e-6
I_SB = 1.2e-6
kT_q = 25e-3

EPSILON = 1e-9


def newton_raphson_solve():
    values = []

    iteration = 1
    v_n = Matrix.empty(2, 1)
    f = Matrix.empty(2, 1)
    F = Matrix.empty(2, 2)
    update_f(f, v_n)
    update_jacobian(F, v_n)
    values.append((iteration,) + v_n.values + f.values + (f.two_norm, ))
    while f.two_norm > EPSILON:
        v_n -= inverse_2x2(F) * f
        update_f(f, v_n)
        update_jacobian(F, v_n)
        iteration += 1
        values.append((iteration,) + v_n.values + f.values + (f.two_norm, ))
    return v_n, values


def update_f(f, v_n):
    v_a, v_b = v_n.values
    f[0][0] = f_a(v_a, v_b)
    f[1][0] = f_b(v_a, v_b)


def update_jacobian(F, v_n):
    v_a, v_b = v_n.values
    F[0][0] = dfa_dva(v_a, v_b)
    F[0][1] = dfa_dvb(v_a, v_b)
    F[1][0] = dfb_dva(v_a, v_b)
    F[1][1] = dfb_dvb(v_a, v_b)


def f_a(v_a, v_b):
    return v_a + R * I_SA * exp_f_term(v_a, v_b) - E


def f_b(v_a, v_b):
    return I_SA * exp_f_term(v_a, v_b) - I_SB * exp_f_term(0, -v_b)


def dfa_dva(v_a, v_b):
    return 1 + R * I_SA * exp_df_term(v_a, v_b)


def dfa_dvb(v_a, v_b):
    return - R * I_SA * exp_df_term(v_a, v_b)


def dfb_dva(v_a, v_b):
    return I_SA * exp_df_term(v_a, v_b)

```

```python

def dfb_dvb(v_a, v_b):
    return - I_SA * exp_df_term(v_a, v_b) - I_SB * exp_df_term(0, -v_b)


def exp_f_term(v_a, v_b):
    return exp((v_a - v_b) / kT_q) - 1


def exp_df_term(v_a, v_b):
    return exp((v_a - v_b) / kT_q) / kT_q


def inverse_2x2(A):
    a = A[0][0]
    b = A[0][1]
    c = A[1][0]
    d = A[1][1]
    inverse = Matrix([
        [d, -b],
        [-c, a]
    ])
    return inverse.scalar_divide(a * d - b * c)
```

Listing 6: Question 4 (q4.py).

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


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

B Output Logs

Listing 7: Output of Question 1 program (q1.txt).

Listing 8: Output of Question 2 program (q2.txt).

Listing 9: Output of Question 3 program (q3.txt).

```
=== Question 3 ===
Solution:
 0.198
 0.091
v_a: 198.134 mV
v_b: 90.571 mV
```

Listing 10: Output of Question 4 program (q4.txt).