ECSE 543
Assignment 3

Sean Stappas
260639512

December 7$^{\text{th}}$, 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 $n = 6$ points of a function $y(x)$, six 5th-order Lagrange polynomials are needed. Each of these polynomials L_j is given by Equation (1), where each F_j is given by Equation (2).

$$L_j(x) = \frac{F_j(x)}{F_j(x_j)} \qquad (1)$$

$$F_j(x) = \prod_{r=1\ldots n, r \neq j}(x - x_r) \qquad (2)$$

The interpolation $\tilde{y}(x)$ of $y(x)$ is then given by Equation (3).

$$\tilde{y}(x) = \sum_{j=1}^{n} y(x_j)L_j(x) \qquad (3)$$

To ease the handling of these polynomials, a `Polynomial` class was created, with useful methods like `add`, `multiply` and `evaluate`. This class can be found in the `polynomial.py` file shown in Listing 3 and the associated tests can be found in the `polynomial_test.py` file shown in Listing 4.

The code evaluating the polynomials in Equations (1) to (3) for interpolation can be found in the `lagrange.py` file shown in Listing 5. The associated tests are in the `test_lagrange.py` file shown in Listing 6.

The interpolation for the first 6 points of the B-H curve is evaluated in `q1.py` shown in Listing 2, with output logged in Listing 17. The generated plot can be seen in Figure 1. It can be seen that the interpolation passes through all the data points, as expected. The curve is also relatively smooth, and

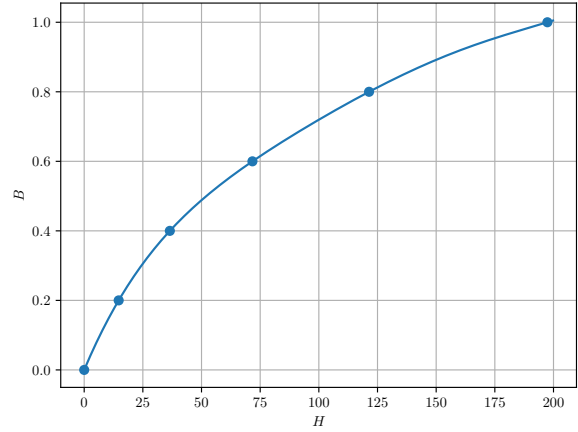should be a good approximation of the B-H curve over that range.

Figure 1: Lagrange interpolation of the first 6 points (B = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0) in the B-H curve. The points are from the table and the curve is interpolated.

1.b Full-Domain Interpolation

The interpolation for the given six points is computed in `q1.py` shown in Listing 2 with output in Listing 17. The generated plot can be seen in Figure 2. The curve passes through all the given points, but is clearly not plausible. It has the characteristic "wiggles" seen when using full-domain Lagrange polynomials over a wide range. It even shows negative B values, which do not match the actual values in between the chosen points.
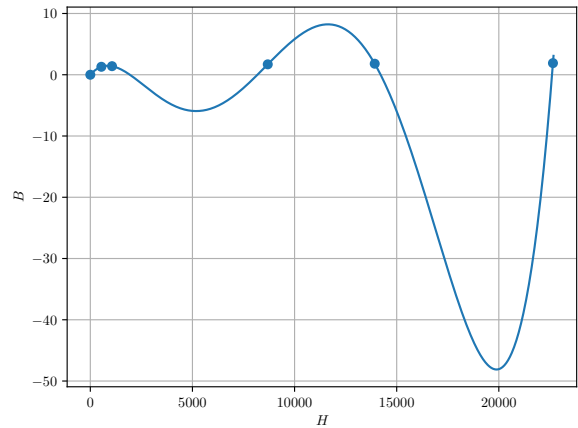

Figure 2: Lagrange interpolation of 6 points (B = 0.0, 1.3, 1.4, 1.7, 1.8, 1.9) in the B-H curve. The points are from the table and the curve is interpolated.

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 7.

2.a   Flux Equation

The magnetic analog of KVL can be seen in Equation (4).

$$(\mathcal{R}_a + \mathcal{R}_c)\psi = \mathcal{F} \tag{4}$$

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 (5).

$$\left(\frac{L_a}{A\mu_o} + \frac{L_c}{A\mu_c(\psi)}\right)\psi - NI = 0 \tag{5}$$

where $\mu_c(\psi)$ is a function of ψ given by Equation (6).

$$\mu_c(\psi) = \frac{B(\psi)}{H(\psi)} = \frac{\psi}{AH(\psi)} \tag{6}$$

Plugging Equation (6) into Equation (5), we obtain Equation (7).

$$\left(\frac{L_a}{A\mu_o} + \frac{L_c H(\psi)}{\psi}\right)\psi - NI = 0 \tag{7}$$

Simplifying the terms, we obtain Equation (8).

$$f(\psi) = \frac{L_a \psi}{A\mu_o} + L_c H(\psi) - NI = 0 \tag{8}$$

Finally, if we plug in the values from the question, we obtain Equation (9), where the coefficients of the terms are calculated in the `q2.py` script shown in Listing 2.

$$f(\psi) = 3.979 \times 10^7 \psi + 0.3H(\psi) - 8000 = 0 \tag{9}$$

In Equation (9), $H(\psi)$ can be calculated for a given ψ by first finding $B = \psi/A$, and then using the B-H curve to find H.

2.b   Newton-Raphson

To perform the Newton-Raphson update, the derivative f' of f will be needed. This can be seen in Equation (10), where the $1/A$ term comes from the fact that $B(\psi) = \psi/A$ and $H(\psi) = H(B(\psi)) = H(\psi/A)$.

$$f'(\psi) = 3.979 \times 10^7 + \frac{0.3H'(\psi)}{A}$$
$$= 3.979 \times 10^7 + 3000H' \tag{10}$$

In Equation (10), the derivative H' of H can be estimated using the slope between each of the points given in the B-H table of Question 1. The slope interpolation code is in the `slope_interpolation.py` script shown in Listing 10.

A piecewise-linear interpolation of the data in the B-H table is needed. This can easily be obtained using the Lagrange polynomial program created for Question 1. Here, we simply need to interpolate 2 points in every sub-domain using 1st-order Lagrange polynomials. We will also need to interpolate H as a function of B to evaluate Equation (9), unlike in Question 1. The linear interpolation code is in the `piecewise_linear.py` script shown in Listing 9. The generated piecewise-linear interpolation fits the B-H points, as can be seen in Figure 3.
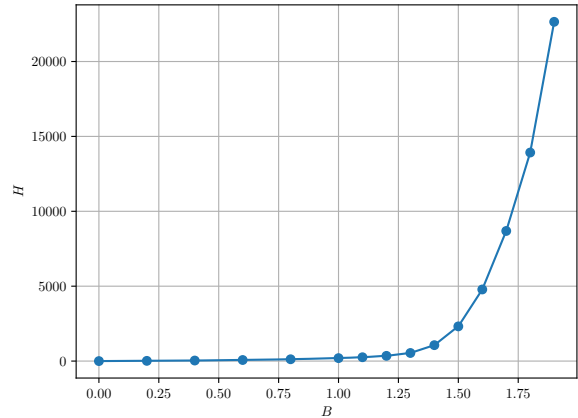

Figure 3: Piecewise-linear interpolation of the H-B curve.

With f given by Equation (9) and f' given by Equation (10), the Newton-Raphson update equation for the flux ψ is given by Equation (11).

$$\psi^{(k+1)} \leftarrow \psi^{(k)} - \frac{f^{(k)}}{f'^{(k)}} \tag{11}$$

The code performing the Newton-Raphson update is in `newton_raphson.py` and can be seen in

3

Listing 8. The executing program is in `q2.py` shown in Listing 7, with associated output in Listing 18. The program takes 3 steps to solve for a flux of approximately $\psi = 161.2681\,\mu\text{Wb}$.

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 13.

3.a   Voltage Equations

The current-voltage relationship for a diode is given by Equation (12).

$$I = I_s \left(\exp\left[\frac{qv}{kT} \right] - 1 \right) \tag{12}$$

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 (13) and (14).

$$I = I_{sA} \left(\exp\left[\frac{q(v_A - v_B)}{kT} \right] - 1 \right) \tag{13}$$

$$I = I_{sB} \left(\exp\left[\frac{qv_B}{kT} \right] - 1 \right) \tag{14}$$

By KVL, we also have Equation (15), relating V_A and I.

$$I = \frac{E - v_A}{R} \tag{15}$$

Equating Equations (13) and (15), we obtain the nonlinear equation for v_A, shown in Equation (16).

$$
\begin{aligned}
& f_A(v_A, v_B) \\
&= v_A + RI_{sA} \left(\exp\left[\frac{q(v_A - v_B)}{kT} \right] - 1 \right) - E \\
&= 0
\end{aligned}
\tag{16}
$$

Equating Equations (13) and (14), we obtain the nonlinear equation for v_B, shown in Equation (17).

$$
\begin{aligned}
f_B(v_A, v_B) &= I_{sA} \left(\exp\left[\frac{q(v_A - v_B)}{kT} \right] - 1 \right) \\
&- I_{sB} \left(\exp\left[\frac{qv_B}{kT} \right] - 1 \right) = 0
\end{aligned}
\tag{17}
$$

The total system of equations can then be expressed by Equation (18).

$$\mathbf{f}(\mathbf{v_n}) = \begin{bmatrix} f_A(v_A, v_B) \\ f_B(v_A, v_B) \end{bmatrix} = \mathbf{0} \tag{18}$$

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 (19) to (22).

$$\frac{\partial f_A}{\partial v_A} = 1 + RI_{sA} \left(\exp\left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \tag{19}$$

$$\frac{\partial f_A}{\partial v_B} = -RI_{sA} \left(\exp\left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \tag{20}$$

$$\frac{\partial f_B}{\partial v_A} = I_{sA} \left(\exp\left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \tag{21}$$

$$
\begin{aligned}
\frac{\partial f_B}{\partial v_B} = &- I_{sA} \left(\exp\left[\frac{q(v_A - v_B)}{kT} \right] \frac{q}{kT} \right) \\
&- I_{sB} \left(\exp\left[\frac{qv_B}{kT} \right] \frac{q}{kT} \right)
\end{aligned}
\tag{22}
$$

With these equations, the Jacobian matrix $\mathbf{F}$ is given by Equation (23).

$$\mathbf{F} = \begin{bmatrix} \dfrac{\partial f_A}{\partial v_A} & \dfrac{\partial f_A}{\partial v_B} \\ \dfrac{\partial f_B}{\partial v_A} & \dfrac{\partial f_B}{\partial v_B} \end{bmatrix} \tag{23}$$

With this information, we can apply the Newton-Raphson update in matrix form, shown in Equation (24).

$$\mathbf{v_n}^{(k+1)} \leftarrow \mathbf{v_n}^{(k)} - (\mathbf{F}^{(k)})^{-1} \mathbf{f}^{(k)} \tag{24}$$

The code performing this update is in the `newton_raphson_matrix.py` script and can be seen in Listing 14. The initial guess is $v_A = 0\,\text{V}$ and $v_B = 0\,\text{V}$. The error ϵ in $\mathbf{f}$ is defined as 1×10^{-9}, where the 2-norm of $\mathbf{f}$ is used to compare to the error. The code is executed in the `q3.py` script shown in Listing 13, with output shown in Listing 19. The final solved voltage values are $198.134\,\text{mV}$ for v_A and $90.571\,\text{mV}$ for v_B.

4   Function Integration

The source code for the Question 4 program can be seen in the `q4.py` file shown in Listing 15.

4

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

v_A (mV)	v_B (mV)	$\|\mathbf{f}\|$
0.000	0.000	2.200×10^{-1}
218.254	72.751	9.906×10^{-2}
205.695	81.581	2.837×10^{-2}
200.110	89.250	5.100×10^{-3}
198.211	90.516	1.943×10^{-4}
198.134	90.571	3.088×10^{-7}
198.134	90.571	7.538×10^{-13}

4.a Cosine Integration

The integral I to be solved by Gauss-Legendre integration is shown in Equation (25).

$$I = \int_{x_1}^{x_2} f(x)dx \tag{25}$$

To use Gauss-Legendre integration over N equal segments, the $[x_1, x_2]$ range of x must be mapped to N intervals of size h, with each interval i having a center point x_{0_i}, with x_i ranging from $x_{0_i} - h/2$ to $x_{0_i} + h/2$. Each of these intervals must be mapped to a $[-1, 1]$ range for ζ_i. This mapping between x_i and ζ_i over an interval is given by Equation (26).

$$x_i = x_{0_i} + \frac{h}{2}\zeta_i \tag{26}$$

The integral transformation from x to ζ over an interval i is then given by Equation (27).

$$
\begin{aligned}
I_i &= \int_{x_{0_i}-h/2}^{x_{0_i}+h/2} f(x_i)dx_i \\
&= \frac{h}{2} \int_{-1}^{1} f\left[x_{0_i} + \frac{h}{2}\zeta_i\right] d\zeta_i
\end{aligned}
\tag{27}
$$

The one-point Gauss-Legendre approximation can then be applied for each interval, as shown in Equation (28), where $w_0 = 2$.

$$
\begin{aligned}
I_i &= \frac{h}{2} \int_{-1}^{1} f\left[x_{0_i} + \frac{h}{2}\zeta_i\right] d\zeta_i \\
&= \frac{h}{2} w_0 f(x_{0_i}) \\
&= h f(x_{0_i})
\end{aligned}
\tag{28}
$$

The equation approximating I is then given by Equation (29).

$$
\begin{aligned}
I &\approx \sum_{i=0}^{N-1} I_i \\
&= \sum_{i=0}^{N-1} h f(x_{0_i}) \\
&= h \sum_{i=0}^{N-1} f(x_{0_i})
\end{aligned}
\tag{29}
$$

To summarize, to solve an integral of the form shown in Equation (25) with one-point Gauss-Legendre integration over N intervals, we simply need the width h of each interval and the value $f(x_{0_i})$ of the function f at the midpoint of every interval.

In the context of this question, $x_1 = 0$ and $x_2 = 1$. The width of each interval is $h = 1/N$ and the midpoint of each interval is $x_{0_i} = 1/(2N) + i/N$. This yields the equation shown in Equation (30).

$$
\begin{aligned}
I &= \int_{0}^{1} f(x)dx \\
&\approx \frac{1}{N} \sum_{i=0}^{N-1} f\left[\frac{1}{N}\left(i + \frac{1}{2}\right)\right]
\end{aligned}
\tag{30}
$$

The code executing Equation (30) for arbitrary $f(x)$ can be seen in the `gauss_legendre.py` script shown in Listing 16.

If we use the fact that $f(x) = \cos x$ as well for this question, we obtain Equation (31).

$$I \approx \frac{1}{N} \sum_{i=0}^{N-1} \cos\left[\frac{1}{N}\left(i + \frac{1}{2}\right)\right] \tag{31}$$

To evaluate the estimation, it can be compared to actual value of the integral, which is given by Equation (32).

$$\int_{0}^{1} \cos x\, dx = \sin 1 - \sin 0 = \sin 1 \approx 0.841 \tag{32}$$

The equation for the absolute error used is shown in Equation (33), where I_{actual} is the actual value of the integral, and I_{approx} is the approximate value computed by Gauss-Legendre integration.

$$E = |I_{actual} - I_{approx}| \tag{33}$$

The integral of $\cos x$ is computed in the `q4.py` script shown in Listing 15, with output shown in Listing 20. The logarithmic plot of the error versus

N can be seen in Figure 4. The straight-line slope of the plot is indicative of the fact that first-order Gauss-Legendre integration was used.
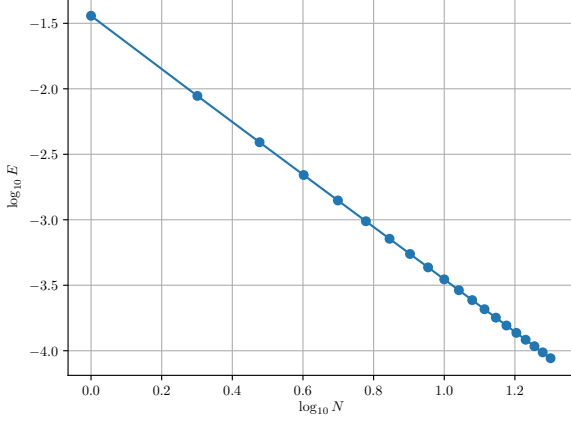
Figure 4: Logarithmic plot of the error E versus number of intervals N for $f(x) = \cos x$.

4.b Log Integration

The integral I to be evaluated is shown in Equation (34).

$$I = \int_0^1 \log_e x \, dx \qquad (34)$$

Using Equation (30) for $f(x) = \log_e x$, we obtain the integral shown in Equation (35).

$$I \approx \frac{1}{N} \sum_{i=0}^{N-1} \log_e \left[\frac{1}{N} \left(i + \frac{1}{2} \right) \right] \qquad (35)$$

The actual value of the integral to which the estimation is compared is shown in Equation (36).

$$\int_0^1 \log_e x \, dx = -1 \qquad (36)$$

The integral of $\log_e x$ is computed in the `q4.py` script shown in Listing 15, with output shown in Listing 20. The logarithmic plot of the error versus N can be seen in Figure 5. The straight-line slope of the plot is indicative of the fact that first-order Gauss-Legendre integration was used.

4.c Log Integration Improvement

To have arbitrary interval widths, Equation (29) must be adjusted, as shown in Equation (37), where h_i is the width of interval i.
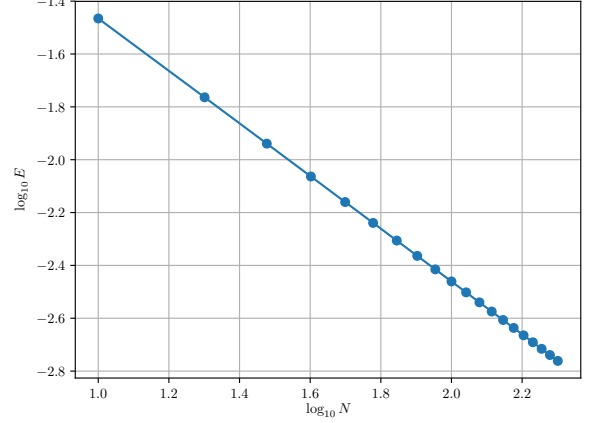
Figure 5: Logarithmic plot of the error E versus number of intervals N for $f(x) = \log_e x$.

$$I \approx \sum_{i=0}^{N-1} h_i f(x_{0_i}) \qquad (37)$$

The relative widths used are shown in Equation (38).

$$(h_{rel_i})_{i=0}^{N-1} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) \qquad (38)$$

These relative widths are converted to actual widths and then used to compute the integral in the `gauss_legendre.py` script shown in Listing 16. The code is executed in the `q4.py` script shown in Listing 15, with output shown in Listing 20. How closely the widths approximate the $\log_e x$ curve can be seen in Figure 6. The estimated value of the integral is -0.988377 with absolute error of 0.011622. This is much more accurate than the equal-segment version in part (b), which obtained a value of -0.965759 and error of 0.034241.

Figure 6: Sizes of intervals used for Question 4(c), with $\log_e x$ shown as reference.

A   Code Listings

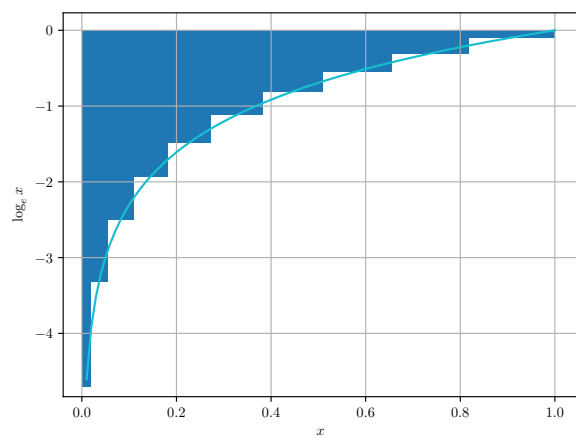*Listing 1: Custom matrix package (`matrices.py`).*

```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:
```

```
63                    return self.scalar_divide(other)
64
65            if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
66                raise ValueError('Incompatible matrix sizes.')
67            return Matrix([[self[row][col] / other[row][col] for col in range(self.num_cols)]
68                           for row in range(self.num_rows)])
69
70        def __neg__(self):
71            return Matrix([[-self[row][col] for col in range(self.num_cols)] for row in range(self.num_rows)])
72
73        def __deepcopy__(self, memo):
74            return Matrix(copy.deepcopy(self.data))
75
76        def __getitem__(self, item):
77            return self.data[item]
78
79        def __len__(self):
80            return len(self.data)
81
82        @property
83        def transpose(self):
84            """
85            :return: the transpose of the current matrix
86            """
87            return Matrix([[self.data[row][col] for row in range(self.num_rows)] for col in
88 ↪           range(self.num_cols)])
89        @property
90        def infinity_norm(self):
91            if self.num_cols > 1:
92                raise ValueError('Not a column vector.')
93            return max([abs(x) for x in self.transpose[0]])
94
95        @property
96        def two_norm(self):
97            if self.num_cols > 1:
98                raise ValueError('Not a column vector.')
99            return math.sqrt(sum([x ** 2 for x in self.transpose[0]]))
100
101        @property
102        def values(self):
103            """
104            :return: the values in this matrix, in row-major order.
105            """
106            vals = []
107            for row in self.data:
108                for val in row:
109                    vals.append(val)
110            return tuple(vals)
111
112        def scaled_values(self, scale):
113            """
114            :return: the values in this matrix, in row-major order.
115            """
116            vals = []
117            for row in self.data:
118                for val in row:
119                    vals.append('{:.3f}'.format(val * scale))
120            return tuple(vals)
121
122        @property
123        def item(self):
124            """
125            :return: the single element contained by this matrix, if it is 1x1.
126            """
127            if not (self.num_rows == 1 and self.num_cols == 1):
128                raise ValueError('Matrix is not 1x1')
129            return self.data[0][0]
130
131        def integer_string(self):
```

```python
            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 is 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
```

```
200         def multiply(*matrices):
201             """
202             Computes the product of the given matrices.
203
204             :param matrices: the matrix objects
205             :return: the product of the given matrices
206             """
207             n = matrices[0].rows
208             product = Matrix.identity(n)
209             for matrix in matrices:
210                 product = product * matrix
211             return product
212
213         @staticmethod
214         def empty(num_rows, num_cols):
215             """
216             Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
217
218             :param num_rows: number of rows
219             :param num_cols: number of columns
220             :return: the empty matrix
221             """
222             return Matrix([[0 for _ in range(num_cols)] for _ in range(num_rows)])
223
224         @staticmethod
225         def identity(n):
226             """
227             Returns the identity matrix of the given size.
228
229             :param n: the size of the identity matrix (number of rows or columns)
230             :return: the identity matrix of size n
231             """
232             return Matrix.diagonal_single_value(1, n)
233
234         @staticmethod
235         def diagonal(values):
236             """
237             Returns a diagonal matrix with the given values along the main diagonal.
238
239             :param values: the values along the main diagonal
240             :return: a diagonal matrix with the given values along the main diagonal
241             """
242             n = len(values)
243             return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
244
245         @staticmethod
246         def diagonal_single_value(value, n):
247             """
248             Returns a diagonal matrix of the given size with the given value along the diagonal.
249
250             :param value: the value of each element on the main diagonal
251             :param n: the size of the matrix
252             :return: a diagonal matrix of the given size with the given value along the diagonal.
253             """
254             return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
255
256         @staticmethod
257         def column_vector(values):
258             """
259             Transforms a row vector into a column vector.
260
261             :param values: the values, one for each row of the column vector
262             :return: the column vector
263             """
264             return Matrix([[value] for value in values])
265
266         @staticmethod
267         def csv_to_matrix(filename):
268             """
269             Reads a CSV file to a matrix.
```

```
270
271            :param filename: the name of the CSV file
272            :return: a matrix containing the values in the CSV file
273            """
274        with open(filename, 'r') as csv_file:
275            reader = csv.reader(csv_file)
276            data = []
277            for row_number, row in enumerate(reader):
278                data.append([literal_eval(val) for val in row])
279            return Matrix(data)
```

Listing 2: Question 1 (`q1.py`).

```
1   from __future__ import division
2   from lagrange import lagrange_interpolation
3
4   import matplotlib.pyplot as plt
5   from matplotlib import rc
6   rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
7   rc('text', usetex=True)
8
9   B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
10  H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8, 2317.0, 4781.9, 8687.4, 13924.3,
    ↪   22650.2]
11
12
13  def q1():
14      print('\n=== Question 1 ===')
15      q1a()
16      q1b()
17
18
19  def q1a():
20      print('\n=== Question 1(a) ===')
21      num_points = 6
22      y_values = B[:num_points]
23      x_values = H[:num_points]
24
25      print('B: {}'.format(y_values))
26      print('H: {}'.format(x_values))
27
28      lagrange_interpolation_polynomial = lagrange_interpolation(x_values, y_values)
29
30      print('Interpolation polynomial: {}'.format(lagrange_interpolation_polynomial))
31
32      plot_polynomial(0, 200, lagrange_interpolation_polynomial, x_values, y_values, filename='q1a')
33
34
35  def q1b():
36      print('\n=== Question 1(b) ===')
37      y_values = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
38      x_values = [H[B.index(b)] for b in y_values]
39
40      print('B: {}'.format(y_values))
41      print('H: {}'.format(x_values))
42
43      lagrange_interpolation_polynomial = lagrange_interpolation(x_values, y_values)
44
45      print('Interpolation polynomial: {}'.format(lagrange_interpolation_polynomial))
46
47      plot_polynomial(0, 22700, lagrange_interpolation_polynomial, x_values, y_values, filename='q1b')
48
49
50  def plot_polynomial(x_min, x_max, polynomial, data_x_points, data_y_points, num_points=1000,
    ↪   filename='q1a'):
51      subdivision = (x_max - x_min) / num_points
52      x_range = [x_min + i * subdivision for i in range(num_points)]
53      y_range = [polynomial.evaluate(x) for x in x_range]
54      f = plt.figure()
```

```
55        plt.plot(x_range, y_range)
56        plt.plot(data_x_points, data_y_points, 'oC0')
57        plt.xlabel('$H$')
58        plt.ylabel('$B$')
59        plt.grid(True)
60        f.savefig('report/plots/{}.pdf'.format(filename), bbox_inches='tight')
61
62
63    if __name__ == '__main__':
64        q1()
```

Listing 3: Custom polynomial class (`polynomial.py`).

```
1     class Polynomial:
2         def __init__(self, coefficients):
3             self.coefficients = coefficients
4             self.order = len(coefficients) - 1
5
6         def __getitem__(self, item):
7             return self.coefficients[item]
8
9         def __add__(self, other):
10            result_coefficients = []
11            common_order = min(self.order, other.order)
12            for i in range(common_order + 1):
13                result_coefficients.append(self[i] + other[i])
14            if self.order > other.order:
15                for i in range(common_order + 1, self.order + 1):
16                    result_coefficients.append(self[i])
17            elif other.order > self.order:
18                for i in range(common_order + 1, other.order + 1):
19                    result_coefficients.append(other[i])
20            return Polynomial(result_coefficients)
21
22        def __sub__(self, other):
23            result_coefficients = []
24            common_order = min(self.order, other.order)
25            for i in range(common_order + 1):
26                result_coefficients.append(self[i] - other[i])
27            if self.order > other.order:
28                for i in range(common_order + 1, self.order + 1):
29                    result_coefficients.append(self[i])
30            elif other.order > self.order:
31                for i in range(common_order + 1, other.order + 1):
32                    result_coefficients.append(-other[i])
33            return Polynomial(result_coefficients)
34
35        def __mul__(self, other):
36            result_coefficients = []
37            max_result_order = self.order + other.order
38            for result_order in range(max_result_order + 1):
39                coefficient = 0
40                for order1 in range(self.order + 1):
41                    for order2 in range(other.order + 1):
42                        if order1 + order2 == result_order:
43                            coefficient += self[order1] * other[order2]
44                result_coefficients.append(coefficient)
45            return Polynomial(result_coefficients)
46
47        def __str__(self):
48            return ' + '.join('{}x^{}'.format(coefficient, power) for power, coefficient in
          ↪  enumerate(self.coefficients))
49
50        def scalar_multiply(self, scalar):
51            return Polynomial([scalar * coefficient for coefficient in self.coefficients])
52
53        def evaluate(self, x):
54            result = 0
55            for power, coefficient in enumerate(self.coefficients):
```

13

```
56            result += coefficient * (x ** power)
57        return result
```

Listing 4: Tests of the custom polynomial class (`test_polynomial.py`).

```python
1   import unittest
2
3   from polynomial import Polynomial
4
5
6   class TestPolynomial(unittest.TestCase):
7       def test___add__(self):
8           p1 = Polynomial([4, 5, 6])
9           p2 = Polynomial([4, 5, 6, 7, 8])
10          expected_coefficients = [8, 10, 12, 7, 8]
11
12          p3 = (p1 + p2)
13          actual_coefficients = p3.coefficients
14
15          self.assertEqual(expected_coefficients, actual_coefficients)
16          print('({}) + ({}) = ({})'.format(p1, p2, p3))
17
18      def test___sub__(self):
19          p1 = Polynomial([4, 5, 6])
20          p2 = Polynomial([4, 5, 6, 7, 8])
21          expected_coefficients = [0, 0, 0, -7, -8]
22
23          p3 = (p1 - p2)
24          actual_coefficients = p3.coefficients
25
26          self.assertEqual(expected_coefficients, actual_coefficients)
27          print('({}) - ({}) = ({})'.format(p1, p2, p3))
28
29      def test___mul__(self):
30          p1 = Polynomial([4, 5, 6])
31          p2 = Polynomial([4, 5, 6, 7, 8])
32          expected_coefficients = [16, 40, 73, 88, 103, 82, 48]
33
34          p3 = (p1 * p2)
35          actual_coefficients = p3.coefficients
36
37          self.assertEqual(expected_coefficients, actual_coefficients)
38          print('({}) * ({}) = ({})'.format(p1, p2, p3))
39
40      def test_evaluate_0(self):
41          p1 = Polynomial([4, 5, 6])
42          expected = 4
43
44          actual = p1.evaluate(0)
45
46          self.assertEqual(expected, actual)
47
48      def test_evaluate_1(self):
49          p1 = Polynomial([4, 5, 6])
50          expected = 4 + 5 + 6
51
52          actual = p1.evaluate(1)
53
54          self.assertEqual(expected, actual)
55
56      def test_evaluate_2(self):
57          p1 = Polynomial([4, 5, 6])
58          expected = 4 + 10 + 24
59
60          actual = p1.evaluate(2)
61
62          self.assertEqual(expected, actual)
63
64
```

```python
65  if __name__ == '__main__':
66      unittest.main()
```

Listing 5: Lagrange interpolation (`lagrange.py`).

```python
1   from __future__ import division
2
3   from polynomial import Polynomial
4
5
6   def lagrange_interpolation(x_values, y_values):
7       n = len(x_values)
8       result_polynomial = Polynomial([])
9       for j in range(n):
10          result_polynomial += lagrange_lj_polynomial(j, x_values).scalar_multiply(y_values[j])
11      return result_polynomial
12
13
14  def lagrange_lj_polynomial(j, x_values):
15      fj_x = lagrange_fj_polynomial(j, x_values)
16      fj_xj = lagrange_fj_constant_denominator(j, x_values)
17      return fj_x.scalar_multiply(1 / fj_xj)
18
19
20  def lagrange_fj_polynomial(j, x_values):
21      result_polynomial = Polynomial([1])
22      for r in range(len(x_values)):
23          if r != j:
24              result_polynomial *= Polynomial([-x_values[r], 1])
25      return result_polynomial
26
27
28  def lagrange_fj_constant_denominator(j, x_values):
29      product = 1
30      for r, x_r in enumerate(x_values):
31          if r != j:
32              product *= (x_values[j] - x_r)
33      return product
```

Listing 6: Tests of the Lagrange interpolation (`test_lagrange.py`).

```python
1   import unittest
2
3   from lagrange import lagrange_fj_polynomial, lagrange_interpolation
4
5
6   class TestLagrange(unittest.TestCase):
7       def test_lagrange_fj_polynomial(self):
8           expected_coefficients = [-1, 1]
9
10          actual_coefficients = (lagrange_fj_polynomial(1, [1, 2])).coefficients
11
12          self.assertEqual(expected_coefficients, actual_coefficients)
13
14      def test_lagrange_interpolation(self):
15          x_values = [1, 2, 3, 4, 5]
16          y_values = [4, 5, 1, 6, 10]
17
18          polynomial = lagrange_interpolation(x_values, y_values)
19
20          for x, y in zip(x_values, y_values):
21              self.assertAlmostEqual(y, polynomial.evaluate(x))
22
23
24  if __name__ == '__main__':
25      unittest.main()
```

```python
1    import math
2
3    import matplotlib.pyplot as plt
4    from matplotlib import rc
5
6    from piecewise_linear import PiecewiseLinearInterpolator
7
8    rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
9    rc('text', usetex=True)
10
11   from newton_raphson import newton_raphson_solve
12
13   B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
14   H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8, 2317.0, 4781.9, 8687.4, 13924.3,
     ↪   22650.2]
15   L_a = 5e-3
16   L_c = 0.3
17   A = 1e-4
18   N = 1000
19   I = 8
20   mu_0 = 4e-7 * math.pi
21
22
23   def q2():
24       print('\n=== Question 2 ===')
25       q2a()
26       q2b()
27
28
29   def q2a():
30       print('\n=== Question 2(a) ===')
31       print('Flux equation: ')
32       coeff_1 = L_a / (A * mu_0)
33       coeff_2 = L_c
34       coeff_3 = N * I
35       eq = 'f(\psi) = \SI{{{:1.3e}}}{{}} \psi + {}H(\psi) - {} = 0'.format(coeff_1, coeff_2, coeff_3)
36       print(eq)
37       with open('report/latex/flux_equation.txt', 'w') as f:
38           f.write(eq)
39
40
41   def q2b():
42       print('\n=== Question 2(b) ===')
43       flux, iterations = newton_raphson_solve()
44       print('Solved flux: {} Wb'.format(flux))
45       print('Number of iterations: {}'.format(iterations))
46       # plot_interpolation(0.0, 1.9, PiecewiseLinearInterpolator(), B, H)
47
48
49   def plot_interpolation(x_min, x_max, interpolator, data_x_points, data_y_points, num_points=1000,
     ↪   filename='q2b'):
50       subdivision = (x_max - x_min) / num_points
51       x_range = [x_min + i * subdivision for i in range(num_points)]
52       y_range = [interpolator.evaluate_b(x) for x in x_range]
53       f = plt.figure()
54       plt.plot(x_range, y_range)
55       plt.plot(data_x_points, data_y_points, 'oC0')
56       plt.xlabel('$B$')
57       plt.ylabel('$H$')
58       plt.grid(True)
59       f.savefig('report/plots/{}.pdf'.format(filename), bbox_inches='tight')
60
61
62   if __name__ == '__main__':
63       q2()
```

Listing 8: Newton-Raphson (`newton_raphson.py`).

```python
from __future__ import division

from piecewise_linear import PiecewiseLinearInterpolator
from slope_interpolation import SlopeInterpolation

EPSILON = 1e-6


def newton_raphson_solve():
    h_interpolator = PiecewiseLinearInterpolator()
    h_prime_interpolator = SlopeInterpolation()

    flux = 0
    f_0 = update_f(flux, h_interpolator)
    f_prime = update_f_prime(flux, h_prime_interpolator)
    f = f_0
    iterations = 0
    while abs(f / f_0) >= EPSILON:
        print('Flux: {} Wb at iteration {}'.format(flux, iterations))
        flux -= f / f_prime
        f = update_f(flux, h_interpolator)
        f_prime = update_f_prime(flux, h_prime_interpolator)
        iterations += 1
    return flux, iterations


def update_f(flux, h_interpolator):
    return 3.979e7 * flux + 0.3 * h_interpolator.evaluate_flux(flux) - 8000


def update_f_prime(flux, h_prime_interpolator):
    return 3.979e7 + 3000 * h_prime_interpolator.evaluate_flux(flux)
```

Listing 9: Piecewise-linear interpolation (`piecewise_linear.py`).

```python
from __future__ import division

from lagrange import lagrange_interpolation

B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8, 2317.0, 4781.9, 8687.4, 13924.3,
     22650.2]
A = 1e-4


class PiecewiseLinearInterpolator:
    def __init__(self):
        self.piecewise_linear_polynomials = []
        for i in range(len(B) - 1):
            x_values = B[i:i + 2]
            y_values = H[i:i + 2]
            lagrange_interpolation_polynomial = lagrange_interpolation(x_values, y_values)
            self.piecewise_linear_polynomials.append(lagrange_interpolation_polynomial)

    def evaluate_flux(self, flux):
        b = flux / A
        return self.evaluate_b(b)

    def evaluate_b(self, b):
        if b > B[-1]:
            return self.piecewise_linear_polynomials[-1].evaluate(b)
        elif b < B[0]:
            return self.piecewise_linear_polynomials[0].evaluate(b)
        for i in range(len(B) - 1):
            if B[i] <= b <= B[i + 1]:
                return self.piecewise_linear_polynomials[i].evaluate(b)
        return None
```

Listing 10: Slope interpolation (`slope_interpolation.py`).

```python
1  from __future__ import division
2
3  B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
4  H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8, 2317.0, 4781.9, 8687.4, 13924.3,
   ↪  22650.2]
5  A = 1e-4
6
7
8  class SlopeInterpolation:
9      def __init__(self):
10         self.slopes = []
11         for i in range(len(B) - 1):
12             h_slope = (H[i + 1] - H[i]) / (B[i + 1] - B[i])
13             self.slopes.append(h_slope)
14
15     def evaluate_flux(self, flux):
16         b = flux / A
17         return self.evaluate_b(b)
18
19     def evaluate_b(self, b):
20         if b > B[-1]:
21             return self.slopes[-1]
22         elif b < B[0]:
23             return self.slopes[0]
24         for i in range(len(B) - 1):
25             if B[i] <= b <= B[i + 1]:
26                 return self.slopes[i]
27         return None
```

Listing 11: Piecewise-linear interpolation tests (`test_piecewise_linear.py`).

```python
1  import unittest
2
3  from piecewise_linear import PiecewiseLinearInterpolator
4
5  B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
6  H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8, 2317.0, 4781.9, 8687.4, 13924.3,
   ↪  22650.2]
7  A = 1e-4
8
9
10 class TestPiecewiseLinearInterpolation(unittest.TestCase):
11     def test_evaluate(self):
12         interpolator = PiecewiseLinearInterpolator()
13         for b, h in zip(B, H):
14             self.assertAlmostEqual(h, interpolator.evaluate_b(b))
15             self.assertAlmostEqual(h, interpolator.evaluate_flux(b * A))
16
17
18 if __name__ == '__main__':
19     unittest.main()
```

Listing 12: Slope interpolation tests (`test_slope_interpolation.py`).

```python
1  import unittest
2
3  from slope_interpolation import SlopeInterpolation
4
5  B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
6  H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8, 2317.0, 4781.9, 8687.4, 13924.3,
   ↪  22650.2]
7  A = 1e-4
8
9
```

```
10    class TestPiecewiseLinearInterpolation(unittest.TestCase):
11        def test_evaluate(self):
12            interpolator = SlopeInterpolation()
13            for i in range(len(B) - 1):
14                h_slope = (H[i + 1] - H[i]) / (B[i + 1] - B[i])
15                self.assertAlmostEqual(h_slope, interpolator.evaluate_b(B[i] + 0.01))
16                self.assertAlmostEqual(h_slope, interpolator.evaluate_flux((B[i] + 0.01) * A))
17
18
19    if __name__ == '__main__':
20        unittest.main()
```

Listing 13: Question 3 (`q3.py`).

```
1     from __future__ import division
2
3     from data_saver import save_rows_to_latex
4     from newton_raphson_matrix import newton_raphson_matrix_solve
5
6
7     def q3():
8         print('\n=== Question 3 ===')
9         v_n, values = newton_raphson_matrix_solve()
10        print('Solution: {}'.format(v_n))
11        v_a, v_b = v_n.values
12        print('v_a: {:.3f} mV'.format(v_a * 1000))
13        print('v_b: {:.3f} mV'.format(v_b * 1000))
14        save_rows_to_latex('report/latex/q3.txt', values)
15
16
17    if __name__ == '__main__':
18        q3()
```

Listing 14: Newton-Raphson (`newton_raphson_matrix.py`).

```
1     from __future__ import division
2
3     from math import exp
4
5     from matrices import Matrix
6
7     E = 220e-3
8     R = 500
9     I_SA = 0.6e-6
10    I_SB = 1.2e-6
11    kT_q = 25e-3
12
13    EPSILON = 1e-9
14
15
16    def newton_raphson_matrix_solve():
17        values = []
18
19        iteration = 1
20        v_n = Matrix.empty(2, 1)
21        f = Matrix.empty(2, 1)
22        F = Matrix.empty(2, 2)
23        update_f(f, v_n)
24        update_jacobian(F, v_n)
25        values.append(v_n.scaled_values(1000) + ('{:.3e}'.format(f.two_norm), ))
26        while f.two_norm > EPSILON:
27            v_n -= inverse_2x2(F) * f
28            update_f(f, v_n)
29            update_jacobian(F, v_n)
30            iteration += 1
31            values.append(v_n.scaled_values(1000) + ('{:.3e}'.format(f.two_norm), ))
32        return v_n, values
33
```

```python
34
35   def update_f(f, v_n):
36       v_a, v_b = v_n.values
37       f[0][0] = f_a(v_a, v_b)
38       f[1][0] = f_b(v_a, v_b)
39
40
41   def update_jacobian(F, v_n):
42       v_a, v_b = v_n.values
43       F[0][0] = dfa_dva(v_a, v_b)
44       F[0][1] = dfa_dvb(v_a, v_b)
45       F[1][0] = dfb_dva(v_a, v_b)
46       F[1][1] = dfb_dvb(v_a, v_b)
47
48
49   def f_a(v_a, v_b):
50       return v_a + R * I_SA * exp_f_term(v_a, v_b) - E
51
52
53   def f_b(v_a, v_b):
54       return I_SA * exp_f_term(v_a, v_b) - I_SB * exp_f_term(0, -v_b)
55
56
57   def dfa_dva(v_a, v_b):
58       return 1 + R * I_SA * exp_df_term(v_a, v_b)
59
60
61   def dfa_dvb(v_a, v_b):
62       return - R * I_SA * exp_df_term(v_a, v_b)
63
64
65   def dfb_dva(v_a, v_b):
66       return I_SA * exp_df_term(v_a, v_b)
67
68
69   def dfb_dvb(v_a, v_b):
70       return - I_SA * exp_df_term(v_a, v_b) - I_SB * exp_df_term(0, -v_b)
71
72
73   def exp_f_term(v_a, v_b):
74       return exp((v_a - v_b) / kT_q) - 1
75
76
77   def exp_df_term(v_a, v_b):
78       return exp((v_a - v_b) / kT_q) / kT_q
79
80
81   def inverse_2x2(A):
82       a = A[0][0]
83       b = A[0][1]
84       c = A[1][0]
85       d = A[1][1]
86       inverse = Matrix([
87           [d, -b],
88           [-c, a]
89       ])
90       return inverse.scalar_divide(a * d - b * c)
```

Listing 15: Question 4 (`q4.py`).

```python
1    from __future__ import division
2
3    from math import cos, log10, sin, log
4    from matplotlib.patches import Rectangle
5    from gauss_legendre import one_point_gauss_legendre, one_point_gauss_legendre_arbitrary_widths, \
6        convert_relative_widths_to_widths
7
8    import matplotlib.pyplot as plt
9    from matplotlib import rc
```

```python
10    rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
11    rc('text', usetex=True)
12
13
14    def q4():
15        print('\n=== Question 4 ===')
16        q4a()
17        q4b()
18        q4c()
19
20
21    def q4a():
22        print('\n=== Question 4(a) ===')
23        n_values = []
24        integrals = []
25        n_max = 20
26        actual_integral = sin(1)
27        print('Actual integral of cos(x): {}'.format(actual_integral))
28        for n in range(1, n_max + 1):
29            integral = one_point_gauss_legendre(n, func=cos)
30            n_values.append(n)
31            integrals.append(integral)
32            print('Integral of cos(x) with N={}: {}'.format(n, integral))
33            print('Error: {}'.format(abs(actual_integral - integral)))
34        plot_error(n_values, integrals, actual_integral, func=cos, filename='q4a')
35
36
37    def q4b():
38        print('\n=== Question 4(b) ===')
39        n_values = []
40        integrals = []
41        n_max = 200
42        actual_integral = -1
43        print('Actual integral of ln(x): {}'.format(actual_integral))
44        for n in range(10, n_max + 1, 10):
45            integral = one_point_gauss_legendre(n, func=log)
46            n_values.append(n)
47            integrals.append(integral)
48            print('Integral of ln(x) with N={}: {}'.format(n, integral))
49            print('Error: {}'.format(abs(actual_integral - integral)))
50        plot_error(n_values, integrals, actual_integral, func=log, filename='q4b')
51
52
53    def q4c():
54        print('\n=== Question 4(c) ===')
55        actual_integral = -1
56        print('Actual integral of ln(x): {}'.format(actual_integral))
57        relative_widths = [x for x in range(1, 11)]
58        print('Relative widths: {}'.format(relative_widths))
59        widths = convert_relative_widths_to_widths(relative_widths)
60        print('Actual widths: {}'.format(widths))
61        integral = one_point_gauss_legendre_arbitrary_widths(widths, func=log)
62        print('Estimated Integral of ln(x): {}'.format(integral))
63        print('Error: {}'.format(abs(actual_integral - integral)))
64        plot_log_widths(widths)
65
66
67    def plot_error(n_values, integrals, actual_integral, func, filename='q4a'):
68        x_range = [log10(n) for n in n_values]
69        y_range = [log10(abs(actual_integral - integral)) for integral in integrals]
70        f = plt.figure()
71        plt.plot(x_range, y_range, 'o-')
72        plt.xlabel('$\log_{10}{N}$')
73        plt.ylabel('$\log_{10}{E}$')
74        plt.grid(True)
75        f.savefig('report/plots/{}.pdf'.format(filename), bbox_inches='tight')
76
77
78    def plot_log_widths(widths):
79        x_range = [i / 100 for i in range(1, 101)]
```

```
80        y_range = [log(x) for x in x_range]
81        f = plt.figure()
82        plt.plot(x_range, y_range, 'C9')
83        axis = plt.gca()
84        width_sum = 0
85        for w in widths:
86            axis.add_patch(Rectangle((width_sum, 0), w, log(width_sum + w / 2), facecolor='C0'))
87            width_sum += w
88
89        plt.xlabel('$x$')
90        plt.ylabel('$\log_e x$')
91        plt.grid(True)
92        f.savefig('report/plots/q4c.pdf', bbox_inches='tight')
93
94
95    if __name__ == '__main__':
96        q4()
```

Listing 16: Gauss-Legendre integration (`gauss_legendre.py`).

```
1    from __future__ import division
2
3
4    def one_point_gauss_legendre(n, func):
5        integral = 0
6        for i in range(n):
7            integral += func((i + 0.5) / n)
8        return integral / n
9
10
11   def one_point_gauss_legendre_arbitrary_widths(widths, func):
12       integral = 0
13       width_sum = 0
14       for h in widths:
15           integral += h * func(width_sum + h / 2)
16           width_sum += h
17       return integral
18
19
20   def convert_relative_widths_to_widths(relative_widths):
21       sum_relative_widths = sum(relative_widths)
22       return [r / sum_relative_widths for r in relative_widths]
```

B   Output Logs

Listing 17: Output of Question 1 program (`q1.txt`).

```
1    === Question 1 ===
2
3    === Question 1(a) ===
4    B: [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
5    H: [0.0, 14.7, 36.5, 71.7, 121.4, 197.4]
6    Interpolation polynomial: 0.0x^0 + 0.0160250789192x^1 + -0.000184937790227x^2 + 1.46891994659e-06x^3 +
     ↪  -5.95091845404e-09x^4 + 9.27493520842e-12x^5
7
8    === Question 1(b) ===
9    B: [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
10   H: [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]
11   Interpolation polynomial: 0.0x^0 + 0.00380360001675x^1 + -2.8640158825e-06x^2 + 5.30016232425e-10x^3 +
     ↪  -3.50510926118e-14x^4 + 7.46724167973e-19x^5
```

Listing 18: Output of Question 2 program (`q2.txt`).

```
1    === Question 2 ===
2
```

```
 3   === Question 2(a) ===
 4   Flux equation:
 5   f(\psi) = \SI{3.979e+07}{} \psi + 0.3H(\psi) - 8000 = 0
 6
 7   === Question 2(b) ===
 8   Flux: 0 Wb at iteration 0
 9   Flux: 0.000199947513778 Wb at iteration 1
10   Flux: 0.00016892620877 Wb at iteration 2
11   Solved flux: 0.000161268070466 Wb
12   Number of iterations: 3
```

Listing 19: Output of Question 3 program (`q3.txt`).

```
 1   === Question 3 ===
 2   Solution:
 3    0.198
 4    0.091
 5   v_a: 198.134 mV
 6   v_b: 90.571 mV
```

Listing 20: Output of Question 4 program (`q4.txt`).

```
 1   === Question 4 ===
 2
 3   === Question 4(a) ===
 4   Actual integral of cos(x): 0.841470984808
 5   Integral of cos(x) with N=1: 0.87758256189
 6   Error: 0.0361115770825
 7   Integral of cos(x) with N=2: 0.850300645292
 8   Error: 0.00882966048434
 9   Integral of cos(x) with N=3: 0.845379345845
10   Error: 0.00390836103756
11   Integral of cos(x) with N=4: 0.843666316703
12   Error: 0.00219533189465
13   Integral of cos(x) with N=5: 0.84287507437
14   Error: 0.00140408956194
15   Integral of cos(x) with N=6: 0.842445699196
16   Error: 0.00097471438853
17   Integral of cos(x) with N=7: 0.842186947503
18   Error: 0.000715962695571
19   Integral of cos(x) with N=8: 0.842019067246
20   Error: 0.000548082438602
21   Integral of cos(x) with N=9: 0.841903996167
22   Error: 0.000433011359186
23   Integral of cos(x) with N=10: 0.841821700007
24   Error: 0.000350715199399
25   Integral of cos(x) with N=11: 0.841760817405
26   Error: 0.000289832597425
27   Integral of cos(x) with N=12: 0.841714515321
28   Error: 0.000243530512976
29   Integral of cos(x) with N=13: 0.841678483879
30   Error: 0.000207499070943
31   Integral of cos(x) with N=14: 0.841649895569
32   Error: 0.000178910761171
33   Integral of cos(x) with N=15: 0.84162683297
34   Error: 0.000155848162437
35   Integral of cos(x) with N=16: 0.841607958582
36   Error: 0.000136973773665
37   Integral of cos(x) with N=17: 0.841592316399
38   Error: 0.000121331591133
39   Integral of cos(x) with N=18: 0.841579208411
40   Error: 0.000108223603482
41   Integral of cos(x) with N=19: 0.841568115345
42   Error: 9.71305373565e-05
43   Integral of cos(x) with N=20: 0.841558644427
44   Error: 8.76596193864e-05
45
46   === Question 4(b) ===
```

```
47  Actual integral of ln(x): -1
48  Integral of ln(x) with N=10: -0.965759065346
49  Error: 0.0342409346539
50  Integral of ln(x) with N=20: -0.982775471974
51  Error: 0.0172245280263
52  Integral of ln(x) with N=30: -0.988493840287
53  Error: 0.0115061597127
54  Integral of ln(x) with N=40: -0.99136170096
55  Error: 0.00863829903958
56  Integral of ln(x) with N=50: -0.993085194472
57  Error: 0.00691480552777
58  Integral of ln(x) with N=60: -0.994235347382
59  Error: 0.00576465261812
60  Integral of ln(x) with N=70: -0.99505745201
61  Error: 0.00494254798958
62  Integral of ln(x) with N=80: -0.995674340479
63  Error: 0.00432565952117
64  Integral of ln(x) with N=90: -0.996154326326
65  Error: 0.0038456736739
66  Integral of ln(x) with N=100: -0.99653843074
67  Error: 0.00346156926044
68  Integral of ln(x) with N=110: -0.996852774507
69  Error: 0.00314722549297
70  Integral of ln(x) with N=120: -0.997114780254
71  Error: 0.00288521974554
72  Integral of ln(x) with N=130: -0.99733651478
73  Error: 0.00266348521974
74  Integral of ln(x) with N=140: -0.997526600199
75  Error: 0.00247339980084
76  Integral of ln(x) with N=150: -0.997691361245
77  Error: 0.00230863875481
78  Integral of ln(x) with N=160: -0.997835542661
79  Error: 0.00216445733879
80  Integral of ln(x) with N=170: -0.997962773572
81  Error: 0.00203722642786
82  Integral of ln(x) with N=180: -0.998075877171
83  Error: 0.00192412282897
84  Integral of ln(x) with N=190: -0.998177082672
85  Error: 0.00182291732836
86  Integral of ln(x) with N=200: -0.998268173714
87  Error: 0.00173182628625
88
89  === Question 4(c) ===
90  Actual integral of ln(x): -1
91  Relative widths: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
92  Actual widths: [0.01818181818181818, 0.03636363636363636, 0.05454545454545454, 0.07272727272727272,
    ↪  0.09090909090909091, 0.10909090909090909, 0.12727272727272726, 0.14545454545454545,
    ↪  0.16363636363636364, 0.18181818181818182]
93  Estimated Integral of ln(x): -0.988377436631
94  Error: 0.0116225633689
```