ECSE 543
Assignment 3

Sean Stappas
260639512

December 7^{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 5^{th}-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).

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

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

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

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

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 20. 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.

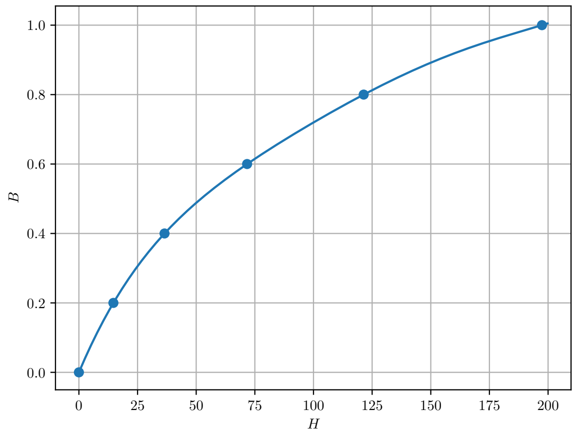

Figure 1: Lagrange interpolation of the first 6 points ($B = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$) in the B-H curve. The points are from the table and the curve is interpolated.

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 20. 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.

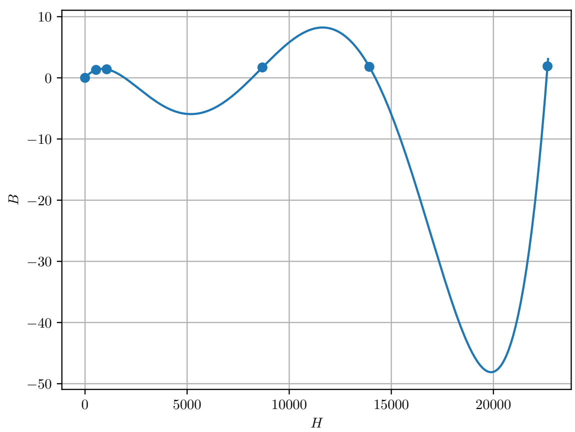

Figure 2: Lagrange interpolation of 6 points ($B = 0.0, 1.3, 1.4, 1.7, 1.8, 1.9$) in the B-H curve. The points are from the table and the curve is interpolated.

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 boundary point itself 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 $H(\psi) = H(B(\psi)) = H(\psi/A)$.

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

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 also 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.

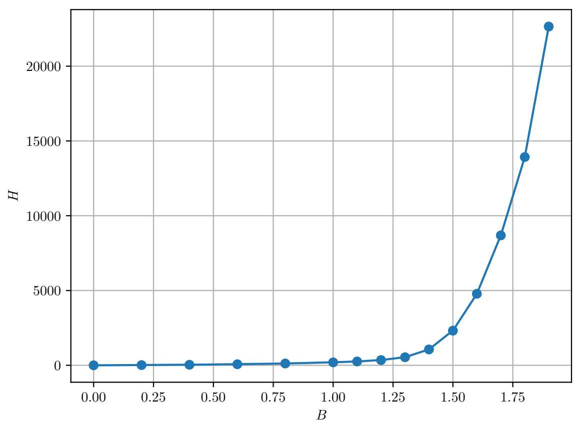

Figure 3: Piecewise-linear interpolation of the H-B curve.

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

Listing 8. The executing program is in `q2.py` shown in Listing 7, with associated output in Listing 21. The program takes 3 steps to solve for a flux of approximately $\psi = 1.613 \times 10^{-4}$ Wb.

2.c Successive Substitution

First, the Newton-Raphson update equation in Equation (11) was simply adjusted to set $f' = 1$, as shown in Equation (12).

$$\psi^{(k+1)} \leftarrow \psi^{(k)} - f^{(k)} \tag{12}$$

However, this method did not converge. Then, inspired from the fact that one must typically use the natural logarithm to solve exponential equations with successive substitution, the update equation was changed to use the inverse of the $H(B(\psi))$ function. Note that when taking the inverse, one must multiply the expression by the cross-sectional area $A = 1 \times 10^{-4}\,\mathrm{m}^2$, since $B(\psi) = \psi/A$. This update equation can be seen in Equation (13), where the inverse H function is denoted as H^{-1}.

$$\psi^{(k+1)} \leftarrow AH^{-1}\left(\frac{NI - \frac{L_a}{A\mu_o}\psi^{(k)}}{L_c}\right) \tag{13}$$

The update equation with all the plugged in values can be seen in Equation (14).

$$\psi^{(k+1)} \leftarrow 1 \times 10^{-4} H^{-1}\left(\frac{8000 - 3.98 \times 10^7 \psi^{(k)}}{0.3}\right) \tag{14}$$

The code evaluating the inverse function H^{-1} is in the `piecewise_linear_inverse.py` script shown in Listing 12. The successive substitution code is in the `successive_substitution.py` script shown in Listing 11. The executing program is in the `q2.py` file shown in Listing 7 with output in Listing 21. The method converges to the same value of $\psi = 1.613 \times 10^{-4}$ Wb as Newton-Raphson, but in 15 steps instead of 3.

3 Diode Circuit

The source code for the Question 3 program can be seen in the `q3.py` file shown in Listing 15.

3.a Voltage Equations

The current-voltage relationship for a diode is given by Equation (15).

$$I = I_s\left(\exp\left[\frac{qv}{kT}\right] - 1\right) \tag{15}$$

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 (16) and (17).

$$I = I_{sA}\left(\exp\left[\frac{q(v_A - v_B)}{kT}\right] - 1\right) \tag{16}$$

$$I = I_{sB}\left(\exp\left[\frac{qv_B}{kT}\right] - 1\right) \tag{17}$$

By KVL, we also have Equation (18), relating V_A and I.

$$I = \frac{E - v_A}{R} \tag{18}$$

Equating Equations (16) and (18), we obtain the nonlinear equation for v_A, shown in Equation (19).

$$\begin{aligned} &f_A(v_A, v_B) \\ &= v_A + RI_{sA}\left(\exp\left[\frac{q(v_A - v_B)}{kT}\right] - 1\right) - E \\ &= 0 \end{aligned} \tag{19}$$

Equating Equations (16) and (17), we obtain the nonlinear equation for v_B, shown in Equation (20).

$$\begin{aligned} f_B(v_A, v_B) = I_{sA}&\left(\exp\left[\frac{q(v_A - v_B)}{kT}\right] - 1\right) \\ - I_{sB}&\left(\exp\left[\frac{qv_B}{kT}\right] - 1\right) = 0 \end{aligned} \tag{20}$$

The total system of equations can then be expressed by Equation (21).

$$\mathbf{f}(\mathbf{v_n}) = \begin{bmatrix} f_A(v_A, v_B) \\ f_B(v_A, v_B) \end{bmatrix} = \mathbf{0} \tag{21}$$

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 (22) to (25).

$$\frac{\partial f_A}{\partial v_A} = 1 + RI_{sA}\left(\exp\left[\frac{q(v_A - v_B)}{kT}\right]\frac{q}{kT}\right) \tag{22}$$

$$\frac{\partial f_A}{\partial v_B} = -RI_{sA}\left(\exp\left[\frac{q(v_A - v_B)}{kT}\right]\frac{q}{kT}\right) \tag{23}$$

$$\frac{\partial f_B}{\partial v_A} = I_{sA}\left(\exp\left[\frac{q(v_A - v_B)}{kT}\right]\frac{q}{kT}\right) \tag{24}$$

$$\begin{aligned}\frac{\partial f_B}{\partial v_B} =& - I_{sA}\left(\exp\left[\frac{q(v_A - v_B)}{kT}\right]\frac{q}{kT}\right) \\ & - I_{sB}\left(\exp\left[\frac{qv_B}{kT}\right]\frac{q}{kT}\right)\end{aligned} \tag{25}$$

With these equations, the Jacobian matrix $\mathbf{F}$ is given by Equation (26).

$$\mathbf{F} = \begin{bmatrix} \frac{\partial f_A}{\partial v_A} & \frac{\partial f_A}{\partial v_B} \\ \frac{\partial f_B}{\partial v_A} & \frac{\partial f_B}{\partial v_B} \end{bmatrix} \tag{26}$$

With this information, we can apply the Newton-Raphson update in matrix form, shown in Equation (27).

$$\mathbf{v_n}^{(k+1)} \leftarrow \mathbf{v_n}^{(k)} - (\mathbf{F}^{(k)})^{-1}\mathbf{f}^{(k)} \tag{27}$$

The code performing this update is in the `newton_raphson_matrix.py` script and can be seen in Listing 16. The initial guess is $v_A = 0\,\mathrm{V}$ and $v_B = 0\,\mathrm{V}$. The program terminates when $\|\mathbf{f}\|/\|\mathbf{f_0}\|$ is less than some ϵ, as shown in Equation (28). The chosen value of ϵ is 1×10^{-9}. To calculate the norm, the 2-norm, i.e., the Euclidean norm, of the vectors is used.

$$\text{Error} = \frac{\|\mathbf{f}\|}{\|\mathbf{f_0}\|} < \epsilon \tag{28}$$

The code is executed in the `q3.py` script shown in Listing 15, with output shown in Listing 22. The final solved voltage values are 198.134 mV for v_A and 90.571 mV for v_B. The recorded values of v_A, v_B, f_A and f_B at each step can be seen in Table 1.

Table 1: Node voltages and $\mathbf{f}$ *values at every iteration of Newton-Raphson.*

v_A (mV)	v_B (mV)	f_A	f_B
0.000	0.000	-2.200×10^{-1}	0.000
218.254	72.751	9.906×10^{-2}	1.808×10^{-4}
205.695	81.581	2.837×10^{-2}	5.519×10^{-5}
200.110	89.250	5.100×10^{-3}	8.561×10^{-6}
198.211	90.516	1.943×10^{-4}	3.331×10^{-7}
198.134	90.571	3.088×10^{-7}	5.098×10^{-10}
198.134	90.571	7.538×10^{-13}	1.276×10^{-15}

The error at each step can be seen in Figure 4, where the error expression is given by Equation (28). It can be seen that the convergence is indeed quadratic.

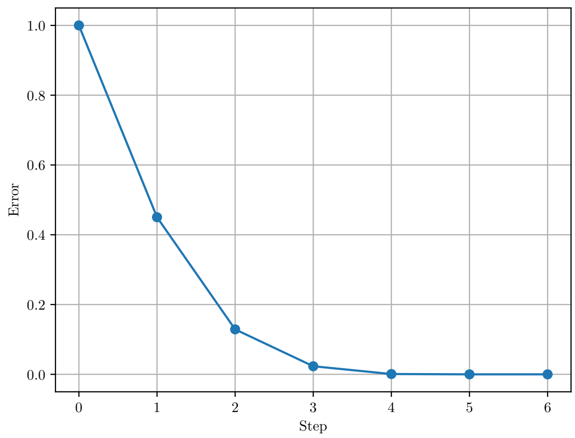

Figure 4: Error at each step of Newton-Raphson to solve the diode circuit. The error equation is given by Equation (28).

4 Function Integration

The source code for the Question 4 program can be seen in the `q4.py` file shown in Listing 17.

4.a Cosine Integration

The integral I to be solved by Gauss-Legendre integration is shown in Equation (29).

$$I = \int_{x_1}^{x_2} f(x)dx \tag{29}$$

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 (30).

$$x_i = x_{0_i} + \frac{h}{2}\zeta_i \tag{30}$$

The integral transformation from x to ζ over an interval i is then given by Equation (31).

$$\begin{aligned} I_i &= \int_{x_{0_i}-h/2}^{x_{0_i}+h/2} f(x_i)dx_i \\ &= \frac{h}{2}\int_{-1}^{1} f\left[x_{0_i} + \frac{h}{2}\zeta_i\right] d\zeta_i \end{aligned} \tag{31}$$

The one-point Gauss-Legendre approximation can then be applied for each interval, as shown in Equation (32), where $w_0 = 2$.

$$I_i = \frac{h}{2}\int_{-1}^{1} f\left[x_{0_i} + \frac{h}{2}\zeta_i\right] d\zeta_i$$
$$= \frac{h}{2} w_0 f(x_{0_i})$$
$$= h f(x_{0_i}) \tag{32}$$

The equation approximating I is then given by Equation (33).

$$I \approx \sum_{i=0}^{N-1} I_i$$
$$= \sum_{i=0}^{N-1} h f(x_{0_i})$$
$$= h \sum_{i=0}^{N-1} f(x_{0_i}) \tag{33}$$

To summarize, to solve an integral of the form shown in Equation (29) 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 (34).

$$I = \int_0^1 f(x)dx$$
$$\approx \frac{1}{N}\sum_{i=0}^{N-1} f\left[\frac{1}{N}\left(i + \frac{1}{2}\right)\right] \tag{34}$$

The code executing Equation (34) for arbitrary $f(x)$ can be seen in the `gauss_legendre.py` script shown in Listing 18.

If we use the fact that $f(x) = \cos x$ as well for this question, we obtain Equation (35).

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

To evaluate the estimation, it can be compared to actual value of the integral, which is given by Equation (36).

$$\int_0^1 \cos x dx = \sin 1 - \sin 0 = \sin 1 \approx 0.841 \tag{36}$$

The equation for the absolute error used is shown in Equation (37), where I_{actual} is the actual value of the integral, and I_{approx} is the approximate value computed by Gauss-Legendre integration.

$$E = |I_{actual} - I_{approx}| \tag{37}$$

The integral of $\cos x$ is computed in the `q4.py` script shown in Listing 17, with output shown in Listing 23. 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. It can also be seen that there are diminishing returns to using a high value of N.

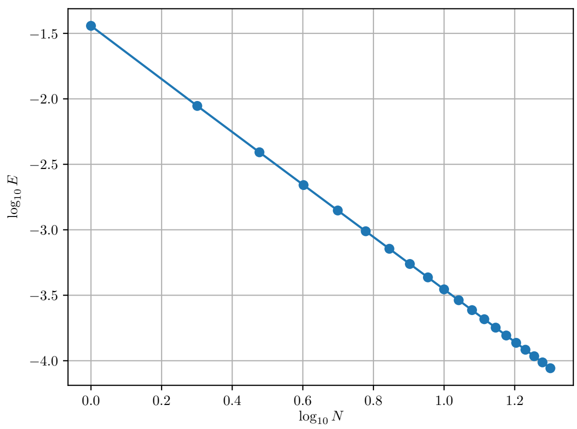

Figure 5: Logarithmic plot of the error E versus number of intervals N for $f(x) = \cos x$.

4.b Log Integration

The integral I to be evaluated is shown in Equation (38).

$$I = \int_0^1 \log_e x dx \tag{38}$$

Using Equation (34) for $f(x) = \log_e x$, we obtain the integral shown in Equation (39).

$$I \approx \frac{1}{N}\sum_{i=0}^{N-1} \log_e\left[\frac{1}{N}\left(i + \frac{1}{2}\right)\right] \tag{39}$$

The actual value of the integral to which the estimation is compared is shown in Equation (40).

$$\int_0^1 \log_e x dx = -1 \tag{40}$$

The integral of $\log_e x$ is computed in the `q4.py` script shown in Listing 17, with output shown in

Listing 23. The logarithmic plot of the error versus N can be seen in Figure 6. Once again, the straight-line slope of the plot is indicative of the fact that first-order Gauss-Legendre integration was used. It can also be seen that there are diminishing returns to using a high value of N.

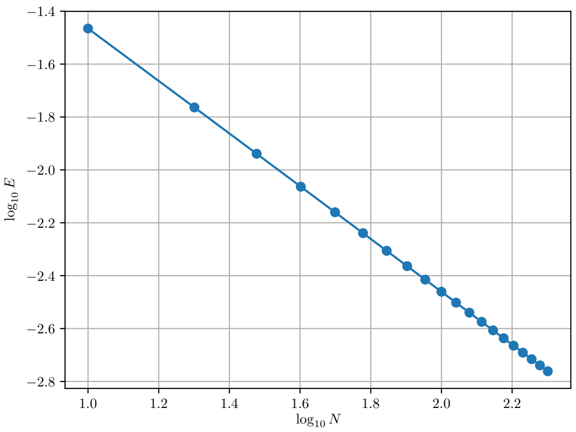

Figure 6: Logarithmic plot of the error E versus number of intervals N for $f(x) = \log_e x$.

4.c Log Integration Improvement

To have arbitrary interval widths, Equation (33) must be adjusted, as shown in Equation (41), where h_i is the width of interval i.

$$I \approx \sum_{i=0}^{N-1} h_i f(x_{0_i}) \tag{41}$$

It is more convenient to adjust relative interval widths to find a suitable combination. The relative widths used are shown in Equation (42).

$$(h_{rel_i})_{i=0}^{N-1} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) \tag{42}$$

These relative widths are converted to actual widths and then used to compute the integral in the `gauss_legendre.py` script shown in Listing 18. The code is executed in the `q4.py` script shown in Listing 17, with output shown in Listing 23. How closely the widths approximate the $\log_e x$ curve can be seen in Figure 7. 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.

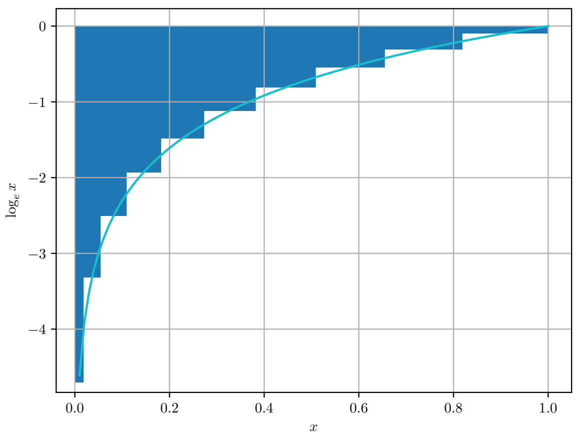

Figure 7: Sizes of intervals used for Question 4(c) in blue, with $\log_e x$ shown as reference in light blue.

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:
```

```python
            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)

    def scaled_values(self, scale):
        """
        :return: the values in this matrix, in row-major order.
        """
        vals = []
        for row in self.data:
            for val in row:
                vals.append('{:.3f}'.format(val * scale))
        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 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
```

```python
    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.
```

```python

        :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
from __future__ import division
from lagrange import lagrange_interpolation

import matplotlib.pyplot as plt
from matplotlib import rc
rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)

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]


def q1():
    print('\n=== Question 1 ===')
    q1a()
    q1b()


def q1a():
    print('\n=== Question 1(a) ===')
    num_points = 6
    y_values = B[:num_points]
    x_values = H[:num_points]

    print('B: {}'.format(y_values))
    print('H: {}'.format(x_values))

    lagrange_interpolation_polynomial = lagrange_interpolation(x_values, y_values)

    print('Interpolation polynomial: {}'.format(lagrange_interpolation_polynomial))

    plot_polynomial(0, 200, lagrange_interpolation_polynomial, x_values, y_values, filename='q1a')


def q1b():
    print('\n=== Question 1(b) ===')
    y_values = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
    x_values = [H[B.index(b)] for b in y_values]

    print('B: {}'.format(y_values))
    print('H: {}'.format(x_values))

    lagrange_interpolation_polynomial = lagrange_interpolation(x_values, y_values)

    print('Interpolation polynomial: {}'.format(lagrange_interpolation_polynomial))

    plot_polynomial(0, 22700, lagrange_interpolation_polynomial, x_values, y_values, filename='q1b')


def plot_polynomial(x_min, x_max, polynomial, data_x_points, data_y_points, num_points=1000,
 filename='q1a'):
    subdivision = (x_max - x_min) / num_points
    x_range = [x_min + i * subdivision for i in range(num_points)]
    y_range = [polynomial.evaluate(x) for x in x_range]
    f = plt.figure()
```

```python
    plt.plot(x_range, y_range)
    plt.plot(data_x_points, data_y_points, 'oC0')
    plt.xlabel('$H$')
    plt.ylabel('$B$')
    plt.grid(True)
    f.savefig('report/plots/{}.pdf'.format(filename), bbox_inches='tight')


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

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

```python
class Polynomial:
    """
    Polynomial object used for multiplication and addition of polynomials.
    """

    def __init__(self, coefficients):
        """
        :param coefficients: the polynomial coefficients, in increasing order
        """
        self.coefficients = coefficients
        self.order = len(coefficients) - 1

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

    def __add__(self, other):
        result_coefficients = []
        common_order = min(self.order, other.order)
        for i in range(common_order + 1):
            result_coefficients.append(self[i] + other[i])
        if self.order > other.order:
            for i in range(common_order + 1, self.order + 1):
                result_coefficients.append(self[i])
        elif other.order > self.order:
            for i in range(common_order + 1, other.order + 1):
                result_coefficients.append(other[i])
        return Polynomial(result_coefficients)

    def __sub__(self, other):
        result_coefficients = []
        common_order = min(self.order, other.order)
        for i in range(common_order + 1):
            result_coefficients.append(self[i] - other[i])
        if self.order > other.order:
            for i in range(common_order + 1, self.order + 1):
                result_coefficients.append(self[i])
        elif other.order > self.order:
            for i in range(common_order + 1, other.order + 1):
                result_coefficients.append(-other[i])
        return Polynomial(result_coefficients)

    def __mul__(self, other):
        result_coefficients = []
        max_result_order = self.order + other.order
        for result_order in range(max_result_order + 1):
            coefficient = 0
            for order1 in range(self.order + 1):
                for order2 in range(other.order + 1):
                    if order1 + order2 == result_order:
                        coefficient += self[order1] * other[order2]
            result_coefficients.append(coefficient)
        return Polynomial(result_coefficients)

    def __str__(self):
        return ' + '.join('{}x^{}'.format(coefficient, power) for power, coefficient in
         ↪  enumerate(self.coefficients))
```

```python

    def scalar_multiply(self, scalar):
        return Polynomial([scalar * coefficient for coefficient in self.coefficients])

    def evaluate(self, x):
        """
        Evaluate the polynomial at the given value of x.

        :param x: the x value to evaluate the polynomial at
        :return: the evaluated polynomial
        """
        result = 0
        for power, coefficient in enumerate(self.coefficients):
            result += coefficient * (x ** power)
        return result
```

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

```python
import unittest

from polynomial import Polynomial


class TestPolynomial(unittest.TestCase):
    def test___add__(self):
        p1 = Polynomial([4, 5, 6])
        p2 = Polynomial([4, 5, 6, 7, 8])
        expected_coefficients = [8, 10, 12, 7, 8]

        p3 = (p1 + p2)
        actual_coefficients = p3.coefficients

        self.assertEqual(expected_coefficients, actual_coefficients)
        print('({}) + ({}) = ({})'.format(p1, p2, p3))

    def test___sub__(self):
        p1 = Polynomial([4, 5, 6])
        p2 = Polynomial([4, 5, 6, 7, 8])
        expected_coefficients = [0, 0, 0, -7, -8]

        p3 = (p1 - p2)
        actual_coefficients = p3.coefficients

        self.assertEqual(expected_coefficients, actual_coefficients)
        print('({}) - ({}) = ({})'.format(p1, p2, p3))

    def test___mul__(self):
        p1 = Polynomial([4, 5, 6])
        p2 = Polynomial([4, 5, 6, 7, 8])
        expected_coefficients = [16, 40, 73, 88, 103, 82, 48]

        p3 = (p1 * p2)
        actual_coefficients = p3.coefficients

        self.assertEqual(expected_coefficients, actual_coefficients)
        print('({}) * ({}) = ({})'.format(p1, p2, p3))

    def test_evaluate_0(self):
        p1 = Polynomial([4, 5, 6])
        expected = 4

        actual = p1.evaluate(0)

        self.assertEqual(expected, actual)

    def test_evaluate_1(self):
        p1 = Polynomial([4, 5, 6])
        expected = 4 + 5 + 6

```

```python
        actual = p1.evaluate(1)

        self.assertEqual(expected, actual)

    def test_evaluate_2(self):
        p1 = Polynomial([4, 5, 6])
        expected = 4 + 10 + 24

        actual = p1.evaluate(2)

        self.assertEqual(expected, actual)


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

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

```python
from __future__ import division

from polynomial import Polynomial


def lagrange_interpolation(x_values, y_values):
    """
    Creates a polynomial interpolating the given x and y values using multiple Lagrange polynomials.

    :param x_values: the x values
    :param y_values: the y values
    :return: the interpolated polynomial
    """
    n = len(x_values)
    result_polynomial = Polynomial([])
    for j in range(n):
        result_polynomial += lagrange_lj_polynomial(j, x_values).scalar_multiply(y_values[j])
    return result_polynomial


def lagrange_lj_polynomial(j, x_values):
    """
    Computes the Lj Lagrange polynomial.

    :param j: the j index
    :param x_values: the x values
    :return: the Lj Lagrange polynomial
    """
    fj_x = lagrange_fj_polynomial(j, x_values)
    fj_xj = lagrange_fj_constant_denominator(j, x_values)
    return fj_x.scalar_multiply(1 / fj_xj)


def lagrange_fj_polynomial(j, x_values):
    """
    Computes the Fj polynomial.

    :param j: the j index
    :param x_values: the x values
    :return: the Fj polynomial
    """
    result_polynomial = Polynomial([1])
    for r in range(len(x_values)):
        if r != j:
            result_polynomial *= Polynomial([-x_values[r], 1])
    return result_polynomial


def lagrange_fj_constant_denominator(j, x_values):
    """
    Computes the Fj polynomial which evaluates to a constant in the denominator of Lj.
```

```python

    :param j: the j index
    :param x_values: the x values
    :return: the Fj polynomial
    """
    product = 1
    for r, x_r in enumerate(x_values):
        if r != j:
            product *= (x_values[j] - x_r)
    return product
```

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

```python
import unittest

from lagrange import lagrange_fj_polynomial, lagrange_interpolation


class TestLagrange(unittest.TestCase):
    def test_lagrange_fj_polynomial(self):
        expected_coefficients = [-1, 1]

        actual_coefficients = (lagrange_fj_polynomial(1, [1, 2])).coefficients

        self.assertEqual(expected_coefficients, actual_coefficients)

    def test_lagrange_interpolation(self):
        x_values = [1, 2, 3, 4, 5]
        y_values = [4, 5, 1, 6, 10]

        polynomial = lagrange_interpolation(x_values, y_values)

        for x, y in zip(x_values, y_values):
            self.assertAlmostEqual(y, polynomial.evaluate(x))


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

Listing 7: Question 2 (`q2.py`).

```python
import math

from piecewise_linear import PiecewiseLinearInterpolator
from successive_substitution import successive_substitution_solve
from newton_raphson import newton_raphson_solve

import matplotlib.pyplot as plt
from matplotlib import rc

rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)


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


def q2():
    print('\n=== Question 2 ===')
    q2a()
```

```python
    q2b()
    q2c()


def q2a():
    print('\n=== Question 2(a) ===')
    print('Flux equation: ')
    coeff_1 = L_a / (A * mu_0)
    print(coeff_1)
    coeff_2 = L_c
    coeff_3 = N * I
    eq = 'f(\psi) = \SI{{{:1.3e}}}{{}} \psi + {}H(\psi) - {} = 0'.format(coeff_1, coeff_2, coeff_3)
    print(eq)
    with open('report/latex/flux_equation.txt', 'w') as f:
        f.write(eq)


def q2b():
    print('\n=== Question 2(b) ===')
    flux, iterations = newton_raphson_solve()
    print('Solved flux: {:1.3e} Wb'.format(flux))
    print('Number of iterations: {}'.format(iterations))
    plot_interpolation(0.0, 1.9, PiecewiseLinearInterpolator(), B, H)


def q2c():
    print('\n=== Question 2(c) ===')
    flux, iterations = successive_substitution_solve()
    print('Solved flux: {:1.3e} Wb'.format(flux))
    print('Number of iterations: {}'.format(iterations))


def plot_interpolation(x_min, x_max, interpolator, data_x_points, data_y_points, num_points=1000,
↪   filename='q2b'):
    subdivision = (x_max - x_min) / num_points
    x_range = [x_min + i * subdivision for i in range(num_points)]
    y_range = [interpolator.evaluate_b(x) for x in x_range]
    f = plt.figure()
    plt.plot(x_range, y_range)
    plt.plot(data_x_points, data_y_points, 'oC0')
    plt.xlabel('$B$')
    plt.ylabel('$H$')
    plt.grid(True)
    f.savefig('report/plots/{}.pdf'.format(filename), bbox_inches='tight')


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

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

```python
from __future__ import division

import math

from piecewise_linear import PiecewiseLinearInterpolator
from slope_interpolation import SlopeInterpolator

L_a = 5e-3
A = 1e-4
mu_0 = 4e-7 * math.pi
EPSILON = 1e-6


def newton_raphson_solve():
    """
    Solves for the flux of the magnetic circuit in Q2 by Newton-Raphson.

    :return: the solved flux and number of steps
```

```python
    """
    h_interpolator = PiecewiseLinearInterpolator()
    h_prime_interpolator = SlopeInterpolator()

    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):
    """
    Updates the f vector to perform a Newton-Raphson step.

    :param flux: the old flux
    :param h_interpolator: the interpolation for the H curve
    :return: the new f vector
    """
    return L_a / (A * mu_0) * flux + 0.3 * h_interpolator.evaluate_flux(flux) - 8000


def update_f_prime(flux, h_prime_interpolator):
    """
    Updates the f  vector to perform a Newton-Raphson step.

    :param flux: the old flux
    :param h_prime_interpolator: the interpolation for the H curve derivative
    :return: the new f vector
    """
    return L_a / (A * mu_0) + 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:
    """
    Piecewise-linear interpolator for the H-B curve.
    """

    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`).

```
from __future__ import division

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 SlopeInterpolator:
    """
    Interpolator for the slope of the H-B curve.
    """

    def __init__(self):
        self.slopes = []
        for i in range(len(B) - 1):
            h_slope = (H[i + 1] - H[i]) / (B[i + 1] - B[i])
            self.slopes.append(h_slope)

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

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

Listing 11: Successive substitution (`successive_substitution.py`).

```
from __future__ import division

import math

from piecewise_linear import PiecewiseLinearInterpolator
from piecewise_linear_inverse import PiecewiseLinearInterpolatorInverse

L_a = 5e-3
A = 1e-4
mu_0 = 4e-7 * math.pi
EPSILON = 1e-6


def successive_substitution_solve():
    """
    Solves for the flux in the magnetic circuit of Q2 using successive substitution.

    :return: the solved flux and the number of steps to solve
    """
    h_interpolator = PiecewiseLinearInterpolator()
    b_interpolator = PiecewiseLinearInterpolatorInverse()

```

```python
    flux = 1e-6
    f_0 = update_f(flux, h_interpolator)
    f = f_0
    iterations = 0
    while abs(f / f_0) >= EPSILON:
        print('Flux: {} Wb at iteration {}'.format(flux, iterations))
        flux = update_flux(flux, b_interpolator)
        f = update_f(flux, h_interpolator)
        iterations += 1
    return flux, iterations


def update_f(flux, h_interpolator):
    return L_a / (A * mu_0) * flux + 0.3 * h_interpolator.evaluate_flux(flux) - 8000


def update_flux(flux, b_interpolator):
    return A * b_interpolator.evaluate_h((8000 - (L_a / (A * mu_0)) * flux) / 0.3)
```

Listing 12: B-H piecewise-linear interpolation (`piecewise_linear_inverse.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 PiecewiseLinearInterpolatorInverse:
    """
    Piecewise-linear interpolator for the B-H curve.
    """

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

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

Listing 13: Piecewise-linear interpolation tests (`test_piecewise_linear.py`).

```python
import unittest

from piecewise_linear import PiecewiseLinearInterpolator

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 TestPiecewiseLinearInterpolation(unittest.TestCase):
    def test_evaluate(self):
```

```
        interpolator = PiecewiseLinearInterpolator()
        for b, h in zip(B, H):
            self.assertAlmostEqual(h, interpolator.evaluate_b(b))
            self.assertAlmostEqual(h, interpolator.evaluate_flux(b * A))


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

Listing 14: Slope interpolation tests (`test_slope_interpolation.py`).

```
import unittest

from slope_interpolation import SlopeInterpolator

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 TestPiecewiseLinearInterpolation(unittest.TestCase):
    def test_evaluate(self):
        interpolator = SlopeInterpolator()
        for i in range(len(B) - 1):
            h_slope = (H[i + 1] - H[i]) / (B[i + 1] - B[i])
            self.assertAlmostEqual(h_slope, interpolator.evaluate_b(B[i] + 0.01))
            self.assertAlmostEqual(h_slope, interpolator.evaluate_flux((B[i] + 0.01) * A))


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

Listing 15: Question 3 (`q3.py`).

```
from __future__ import division

from data_saver import save_rows_to_latex
from newton_raphson_matrix import newton_raphson_matrix_solve


import numpy as np
import numpy.polynomial.polynomial as poly
import sympy as sp
import matplotlib.pyplot as plt
from matplotlib import rc

rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)


def q3():
    print('\n=== Question 3 ===')
    v_n, error_values, norm_values, vA_values, vB_values, fA_values, fB_values = newton_raphson_matrix_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_latex('report/latex/q3.txt', zip(vA_values, vB_values, fA_values, fB_values))
    plot_error(error_values)
    plot_error_quadratic_fit(error_values)


def plot_error(error_values):
    x_range = [i for i in range(len(error_values))]
    y_range = error_values
    f = plt.figure()
```

```
    plt.plot(x_range, y_range, 'o-')
    plt.xlabel('Step')
    plt.ylabel('Error')
    plt.grid(True)
    f.savefig('report/plots/q3.pdf', bbox_inches='tight')


def plot_error_quadratic_fit(error_values):
    f = plt.figure()

    x_range = [i for i in range(len(error_values))]
    y_range = [float(error) for error in error_values]
    plt.plot(x_range, y_range, 'o')

    x_new = np.linspace(x_range[0], x_range[-3], num=len(x_range) * 10)
    polynomial_coeffs = poly.polyfit(x_range, y_range, deg=2)
    polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
    N = sp.symbols("1/h")
    poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
    equation = '${}$'.format(sp.printing.latex(poly_label))
    plt.plot(x_new, polynomial_fit, 'C0-', label=equation)

    plt.xlabel('Step')
    plt.ylabel('Error')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q3_fit.pdf', bbox_inches='tight')


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

Listing 16: Newton-Raphson (`newton_raphson_matrix.py`).

```
from __future__ import division

import copy
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_matrix_solve():
    """
    Solves for the nodal voltages of the nonlinear diode in Q3 by Newton-Raphson.

    :return: the solved voltages, as well stepwise values for the error, voltage and f vector
    """
    error_values = []
    norm_values = []
    vA_values = []
    vB_values = []
    fA_values = []
    fB_values = []

    iteration = 1
    v_n = Matrix.empty(2, 1)
    f = Matrix.empty(2, 1)
    F = Matrix.empty(2, 2)
    update_f(f, v_n)
    f_0 = copy.deepcopy(f)
```

```
    update_jacobian(F, v_n)

    error_values.append('{:.3e}'.format(f.two_norm / f_0.two_norm))
    norm_values.append('{:.3e}'.format(f.two_norm))
    vA_values.append(v_n.scaled_values(1000)[0])
    vB_values.append(v_n.scaled_values(1000)[1])
    fA_values.append('{:.3e}'.format(f.values[0]))
    fB_values.append('{:.3e}'.format(f.values[1]))

    while f.two_norm / f_0.two_norm >= EPSILON:
        v_n -= inverse_2x2(F) * f
        update_f(f, v_n)
        update_jacobian(F, v_n)
        iteration += 1

        error_values.append('{:.3e}'.format(f.two_norm / f_0.two_norm))
        norm_values.append('{:.3e}'.format(f.two_norm))
        vA_values.append(v_n.scaled_values(1000)[0])
        vB_values.append(v_n.scaled_values(1000)[1])
        fA_values.append('{:.3e}'.format(f.values[0]))
        fB_values.append('{:.3e}'.format(f.values[1]))
    return v_n, error_values, norm_values, vA_values, vB_values, fA_values, fB_values


def update_f(f, v_n):
    """
    Updates the f vector.

    :param f: the f vector to update
    :param v_n: the nodal voltages
    """
    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):
    """
    Updates the Jacobian matrix.

    :param F: the Jacobian matrix to update
    :param v_n: the nodal voltages
    """
    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):
    """
    Inverts a 2x2 matrix and returns a copy.

    :param A: the matrix to invert
    :return: the inverted matrix
    """
    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 17: Question 4 (q4.py).

```python
from __future__ import division

from math import cos, log10, sin, log
from matplotlib.patches import Rectangle
from gauss_legendre import one_point_gauss_legendre, one_point_gauss_legendre_arbitrary_widths, \
    convert_relative_widths_to_widths

import matplotlib.pyplot as plt
from matplotlib import rc
rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)


def q4():
    print('\n=== Question 4 ===')
    q4a()
    q4b()
    q4c()


def q4a():
    print('\n=== Question 4(a) ===')
    n_values = []
    integrals = []
    n_max = 20
    actual_integral = sin(1)
    print('Actual integral of cos(x): {}'.format(actual_integral))
    for n in range(1, n_max + 1):
        integral = one_point_gauss_legendre(n, func=cos)
        n_values.append(n)
        integrals.append(integral)
        print('Integral of cos(x) with N={}: {}'.format(n, integral))
        print('Error: {}'.format(abs(actual_integral - integral)))
    plot_error(n_values, integrals, actual_integral, func=cos, filename='q4a')


def q4b():
    print('\n=== Question 4(b) ===')
```

```
    n_values = []
    integrals = []
    n_max = 200
    actual_integral = -1
    print('Actual integral of ln(x): {}'.format(actual_integral))
    for n in range(10, n_max + 1, 10):
        integral = one_point_gauss_legendre(n, func=log)
        n_values.append(n)
        integrals.append(integral)
        print('Integral of ln(x) with N={}: {}'.format(n, integral))
        print('Error: {}'.format(abs(actual_integral - integral)))
    plot_error(n_values, integrals, actual_integral, func=log, filename='q4b')


def q4c():
    print('\n=== Question 4(c) ===')
    actual_integral = -1
    print('Actual integral of ln(x): {}'.format(actual_integral))
    relative_widths = [x for x in range(1, 11)]
    print('Relative widths: {}'.format(relative_widths))
    widths = convert_relative_widths_to_widths(relative_widths)
    print('Actual widths: {}'.format(widths))
    integral = one_point_gauss_legendre_arbitrary_widths(widths, func=log)
    print('Estimated Integral of ln(x): {}'.format(integral))
    print('Error: {}'.format(abs(actual_integral - integral)))
    plot_log_widths(widths)


def plot_error(n_values, integrals, actual_integral, func, filename='q4a'):
    x_range = [log10(n) for n in n_values]
    y_range = [log10(abs(actual_integral - integral)) for integral in integrals]
    f = plt.figure()
    plt.plot(x_range, y_range, 'o-')
    plt.xlabel('$\log_{10}{N}$')
    plt.ylabel('$\log_{10}{E}$')
    plt.grid(True)
    f.savefig('report/plots/{}.pdf'.format(filename), bbox_inches='tight')


def plot_log_widths(widths):
    x_range = [i / 100 for i in range(1, 101)]
    y_range = [log(x) for x in x_range]
    f = plt.figure()
    plt.plot(x_range, y_range, 'C9')
    axis = plt.gca()
    width_sum = 0
    for w in widths:
        axis.add_patch(Rectangle((width_sum, 0), w, log(width_sum + w / 2), facecolor='C0'))
        width_sum += w

    plt.xlabel('$x$')
    plt.ylabel('$\log_e x$')
    plt.grid(True)
    f.savefig('report/plots/q4c.pdf', bbox_inches='tight')


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

Listing 18: Gauss-Legendre integration (`gauss_legendre.py`).

```
from __future__ import division


def one_point_gauss_legendre(n, func):
    """
    Approximates the integral of the given function from 0 to 1 with one-point Gauss-Legendre integration.

    :param n: the number of segments
```

```
    :param func: the function to integrate
    :return: the approximate integral
    """
    integral = 0
    for i in range(n):
        integral += func((i + 0.5) / n)
    return integral / n


def one_point_gauss_legendre_arbitrary_widths(widths, func):
    """
    Approximates the integral of the given function from 0 to 1 with one-point Gauss-Legendre integration with
    arbitrary widths.

    :param widths: the widths of the intervals
    :param func: the function to integrate
    :return: the approximate integral
    """
    integral = 0
    width_sum = 0
    for h in widths:
        integral += h * func(width_sum + h / 2)
        width_sum += h
    return integral


def convert_relative_widths_to_widths(relative_widths):
    """
    Converts the given relative interval widths to actual widths.

    :param relative_widths: the relative widths to convert
    :return: the actual widths
    """
    sum_relative_widths = sum(relative_widths)
    return [r / sum_relative_widths for r in relative_widths]
```

Listing 19: Utility to save rows to CSV or LaTeX. (`data_saver.py`).

```
import csv


def save_rows_to_csv(filename, rows, header=None):
    with open(filename, "wb") as f:
        writer = csv.writer(f)
        if header is not None:
            writer.writerow(header)
        for row in rows:
            writer.writerow(['{:.3f}'.format(v) for v in row])


def save_rows_to_latex(filename, rows, header=None):
    with open(filename, "wb") as f:
        if header is not None:
            for i, val in enumerate(header):
                f.write('{}'.format(val))
                if i < len(header) - 1:
                    f.write('&')
            f.write('\\\\ \\hline  \n')
        for j, row in enumerate(rows):
            for i, val in enumerate(row):
                f.write('\SI{{{}}}{{{}}}'.format(val))
                if i < len(row) - 1:
                    f.write('&')
            if j < len(rows) - 1:
                f.write('\\\\\n')
```

B Output Logs

Listing 20: Output of Question 1 program (`q1.txt`).

```
=== Question 1 ===

=== Question 1(a) ===
B: [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
H: [0.0, 14.7, 36.5, 71.7, 121.4, 197.4]
Interpolation polynomial: 0.0x^0 + 0.0160250789192x^1 + -0.000184937790227x^2 + 1.46891994659e-06x^3 + -5.95091845404e-09x^4 + 9.27493520842e-12x^5

=== Question 1(b) ===
B: [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
H: [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]
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 21: Output of Question 2 program (`q2.txt`).

```
=== Question 2 ===

=== Question 2(a) ===
Flux equation:
39788735.773
f(\psi) = \SI{3.979e+07}{} \psi + 0.3H(\psi) - 8000 = 0

=== Question 2(b) ===
Flux: 0 Wb at iteration 0
Flux: 0.000199953831795 Wb at iteration 1
Flux: 0.000168926916944 Wb at iteration 2
Solved flux: 1.613e-04 Wb
Number of iterations: 3

=== Question 2(c) ===
Flux: 1e-06 Wb at iteration 0
Flux: 0.000194450930617 Wb at iteration 1
Flux: 0.000136438357006 Wb at iteration 2
Flux: 0.000169701875678 Wb at iteration 3
Flux: 0.000157473959843 Wb at iteration 4
Flux: 0.000162558274405 Wb at iteration 5
Flux: 0.0001608316628 Wb at iteration 6
Flux: 0.000161418012759 Wb at iteration 7
Flux: 0.000161218890806 Wb at iteration 8
Flux: 0.000161286511775 Wb at iteration 9
Flux: 0.000161263547981 Wb at iteration 10
Flux: 0.000161271346388 Wb at iteration 11
Flux: 0.000161268698082 Wb at iteration 12
Flux: 0.000161269597436 Wb at iteration 13
Flux: 0.000161269292019 Wb at iteration 14
Solved flux: 1.613e-04 Wb
Number of iterations: 15
```

Listing 22: 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 23: Output of Question 4 program (`q4.txt`).

```
=== Question 4 ===

```

```
=== Question 4(a) ===
Actual integral of cos(x): 0.841470984808
Integral of cos(x) with N=1: 0.87758256189
Error: 0.0361115770825
Integral of cos(x) with N=2: 0.850300645292
Error: 0.00882966048434
Integral of cos(x) with N=3: 0.845379345845
Error: 0.00390836103756
Integral of cos(x) with N=4: 0.843666316703
Error: 0.00219533189465
Integral of cos(x) with N=5: 0.84287507437
Error: 0.00140408956194
Integral of cos(x) with N=6: 0.842445699196
Error: 0.00097471438853
Integral of cos(x) with N=7: 0.842186947503
Error: 0.000715962695571
Integral of cos(x) with N=8: 0.842019067246
Error: 0.000548082438602
Integral of cos(x) with N=9: 0.841903996167
Error: 0.000433011359186
Integral of cos(x) with N=10: 0.841821700007
Error: 0.000350715199399
Integral of cos(x) with N=11: 0.841760817405
Error: 0.000289832597425
Integral of cos(x) with N=12: 0.841714515321
Error: 0.000243530512976
Integral of cos(x) with N=13: 0.841678483879
Error: 0.000207499070943
Integral of cos(x) with N=14: 0.841649895569
Error: 0.000178910761171
Integral of cos(x) with N=15: 0.84162683297
Error: 0.000155848162437
Integral of cos(x) with N=16: 0.841607958582
Error: 0.000136973773665
Integral of cos(x) with N=17: 0.841592316399
Error: 0.000121331591133
Integral of cos(x) with N=18: 0.841579208411
Error: 0.000108223603482
Integral of cos(x) with N=19: 0.841568115345
Error: 9.71305373565e-05
Integral of cos(x) with N=20: 0.841558644427
Error: 8.76596193864e-05

=== Question 4(b) ===
Actual integral of ln(x): -1
Integral of ln(x) with N=10: -0.965759065346
Error: 0.0342409346539
Integral of ln(x) with N=20: -0.982775471974
Error: 0.0172245280263
Integral of ln(x) with N=30: -0.988493840287
Error: 0.0115061597127
Integral of ln(x) with N=40: -0.99136170096
Error: 0.00863829903958
Integral of ln(x) with N=50: -0.993085194472
Error: 0.00691480552777
Integral of ln(x) with N=60: -0.994235347382
Error: 0.00576465261812
Integral of ln(x) with N=70: -0.99505745201
Error: 0.00494254798958
Integral of ln(x) with N=80: -0.995674340479
Error: 0.00432565952117
Integral of ln(x) with N=90: -0.996154326326
Error: 0.0038456736739
Integral of ln(x) with N=100: -0.99653843074
Error: 0.00346156926044
Integral of ln(x) with N=110: -0.996852774507
Error: 0.00314722549297
Integral of ln(x) with N=120: -0.997114780254
Error: 0.00288521974554
Integral of ln(x) with N=130: -0.99733651478
```

```
Error: 0.00266348521974
Integral of ln(x) with N=140: -0.997526600199
Error: 0.00247339980084
Integral of ln(x) with N=150: -0.997691361245
Error: 0.00230863875481
Integral of ln(x) with N=160: -0.997835542661
Error: 0.00216445733879
Integral of ln(x) with N=170: -0.997962773572
Error: 0.00203722642786
Integral of ln(x) with N=180: -0.998075877171
Error: 0.00192412282897
Integral of ln(x) with N=190: -0.998177082672
Error: 0.00182291732836
Integral of ln(x) with N=200: -0.998268173714
Error: 0.00173182628625

=== Question 4(c) ===
Actual integral of ln(x): -1
Relative widths: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Actual widths: [0.01818181818181818, 0.03636363636363636, 0.05454545454545454, 0.07272727272727272, 0.09090909090909091, 0.10909090909090909, 0.12727272727272726, 0.14545454545454545, 0.16363636363636364, 0.18181818181818182]
Estimated Integral of ln(x): -0.988377436631
Error: 0.0116225633689
```