Thomas Christinck
260601793

ECSE 543 Assignment 3

1. (a) The first six points provided in the handout were interpolated using full-domain Lagrange polynomials. The interpolation is shown in Figure 1. We see that the result is plausible; however, the curve underestimates *B*-values values for large *H*-values. The code implementing this interpolation can be found in Appendix 1.1 on page 8 as *lagrange_interpolator.m*. Running this code will produce the results discussed in sections 1.a. and 1.b.

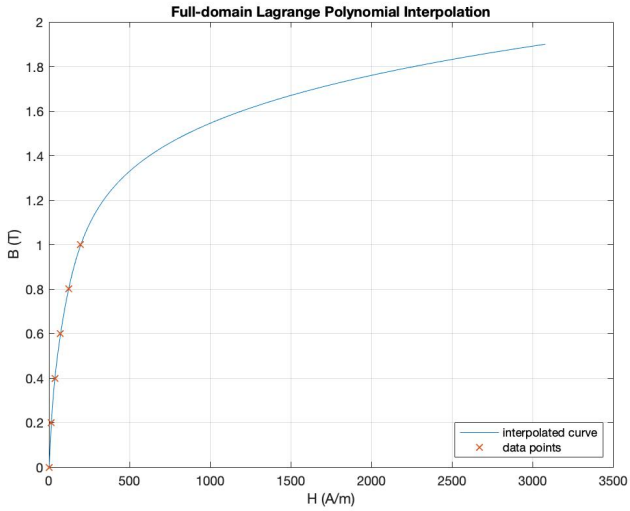

Figure 1 - Full-domain Lagrange interpolation using the first six points provided

1. (b) Now, the 6 points at *B* = 0, 1.3, 1.4, 1.7, 1.8, 1.9 were used in the full-domain Lagrange polynomial interpolation. The interpolation is shown in Figure 2. Clearly, the result is *not* plausible (not a plausible B-H curve).

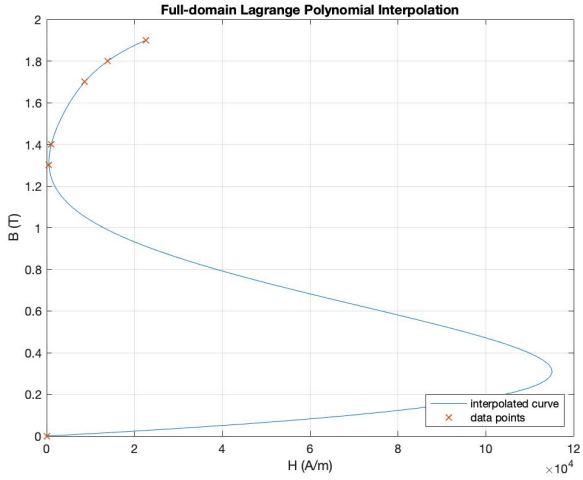

Figure 2 - Full domain Lagrange polynomial interpolation for the six points at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9

1. (c) We interpolate the 6 points from 1.b. using cubic Hermite polynomials. The slopes at the 6 points are approximated using the slopes between the 6 points. Code is provided as *hermite_interpolator.m* in Appendix 1.2 on page 9. The derivative approximations are computed in the sub-function *get_coefficient_matrix.m* on

page 10. The resulting curve is shown in Figure 3. Clearly, the curve is a better fit than the curves interpolated using full-domain Lagrange polynomial interpolation, especially for large *H*-values.

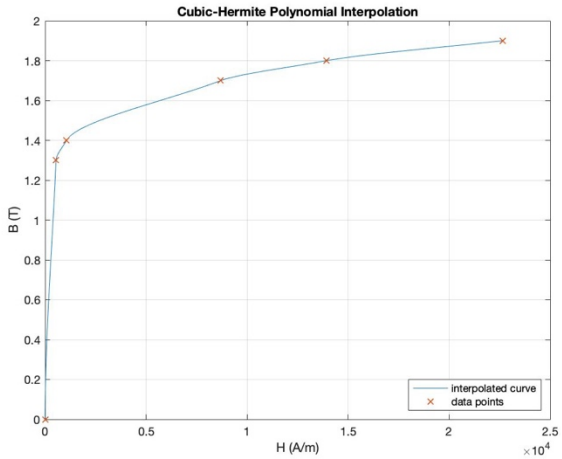

Figure 3 - Cubic Hermite polynomial interpolation for the six points at B = 0,1.3,1.4,1.7,1.8,1.9

1. (d) The magnetic circuit described in the handout can be modeled with gap reluctance R_g, and core reluctance R_c. The coil is modeled as a voltage source *NI*, where *N = 800* is the number of turns. We are interested in an expression for the flux ψ. We have the following equations:

$$R_c = \frac{l_c}{\mu A} \tag{1.1}$$

$$R_g = \frac{l_g}{\mu_0 A} \tag{1.2}$$

Where l_g and l_c are the gap and core lengths, respectively, *A* is the cross-sectional area, and μ is the core permeability, which is given by equation (1.3).

$$\mu = \frac{B}{H} \tag{1.3}$$

Further, we have the following expressions for the flux:

$$\psi = \frac{(R_g + R_C)}{NI} \tag{1.4}$$

$$\psi = A * B \tag{1.5}$$

Now, substituting equation (1.5) into (1.3), and substituting the resulting expression into (1.4), we get the following expression.

$$NI = \psi(R_g + \frac{Hl_c}{\psi}) \tag{1.6}$$

Simplifying the above expression, we get the non-linear expression for the flux in the core (1.7).

$$\psi R_g + Hl_c - NI = 0$$

$$\boldsymbol{f(\psi) = (3.97887 * 10^7)\psi + 0.3H - 8000 = 0} \tag{1.7}$$

1. (e) Using Newton-Raphson, we solve the nonlinear equation. The code is included in *newton_raphson.m* under Appendix 1.3 on page 12. Code for piecewise-linear interpolations can be found as sub-functions of *newton_raphson.m* as *h_der.m* and *h_val.m*, on pages 13 and 12, respectively. Running Newton-Raphson, we find the final flux to be **1.613e-4 Wb, after 3 iterations** (Figure 4).

```
>> newton_raphson
Iterations:

i =

     3

Flux calculated:

psi =

   1.6127e-04
```

Figure 4 - Output for Newton-Raphson method

1. (f) If we try solving the same problem with successive substitution, the method does not converge. However, if we add a scaling factor to the iteration step updates, such that for iteration step $i + 1$ we have:

$$\psi_{i+1} = \psi_i - \alpha * f(\psi_i)$$

A scaling factor of $\alpha = 6.5 * 10^{-9}$ achieves convergence after 9 iterations. The code for this is included in *successive_substitution.m*, a sub-function of *newton_raphson.m* under Appendix 1.3 on page 13. Further, running the provided *newton_raphson.m* function will produce results for ψ using both the successive substitution method and the Newton Raphson method. Using the modified successive substitution, we find the final flux to be **1.613e-4 Wb, after 9 iterations** (the same result as in (e); however, it takes more iterations to achieve this result) (Figure 5).

```
------------ Successive substitution ----------------------
Iterations:

i =

     9

Flux calculated:

psi =

   1.6127e-04
```

Figure 5 - Output for successive substitution method

2. (a) We are interested in deriving a set of nonlinear equations for a vector of node voltages. First, we write expressions for the current in diodes A and B as equations (2.1) and (2.2). Note that these currents are equivalent.

$$I_A = I_{sA}\left(e^{\frac{q(V_1 - V_2)}{Kt}} - 1\right) \tag{2.1}$$

$$I_B = I_{sB}\left(e^{\frac{qV_2}{Kt}} - 1\right) \tag{2.2}$$

Further, we have the have the current as:

$$I = \frac{E - V_D}{R} \tag{2.3}$$

Where V_D is the voltage across the diodes A and B. Now, substituting equation (2.3) into (2.1), we can get expressions for the node voltages in terms of the variables I_{sA}, I_{sB}, E, R, and the node voltages V_1 and V_2, where $V_1 = V_D$. First, we get equation (2.4).

$$\frac{E - V_1}{R} = I_{sA}\left(e^{\frac{q(V_1 - V_2)}{Kt}} - 1\right)$$

$$V_1 = E - RI_{sA}\left(e^{\frac{q(V_1 - V_2)}{Kt}} - 1\right)$$

$$\boldsymbol{f_1(V_1, V_2) = 0 = V_1 - E + RI_{sA}\left(e^{\frac{q(V_1 - V_2)}{Kt}} - 1\right)} \tag{2.4}$$

Recall that the expressions in equations (2.1) and (2.2) are equivalent. This gives us:

$$I_{sA}\left(e^{\frac{q(V_1 - V_2)}{Kt}} - 1\right) = I_{sB}\left(e^{\frac{qV_2}{Kt}} - 1\right) \tag{2.5}$$

Now, rearranging equation (2.5) we get the expression in equation (2.6):

$$\boldsymbol{f_2(V_1, V_2) = 0 = I_{sA}\left(e^{\frac{q(V_1 - V_2)}{Kt}} - 1\right) - I_{sB}\left(e^{\frac{qV_2}{Kt}} - 1\right)} \tag{2.6}$$

2. (b) Equation (2.6) was solved using the Newton-Raphson method. Convergence was achieved when the error measure $\varepsilon_k < 1 * 10^{-14}$. At each step, f and the voltage across each diode was recorded in Table 1. Code can be found in Appendix 2.1 on page 14 as *test_q2.m*. Results can be reproduced by running *test_q2.m* in MATLAB. All matrix operation code (*matrix_multiply*, *matrix_add_or_subtract*, etc.) is from Assignment 1.

k	f_1	f_2	V_1	V_2
0	0.0380	4.80e-05	0.1981	0.0834
1	0.0059	1.15e-05	0.1841	0.0843
2	3.54e-04	4.87e-07	0.1823	0.0871
3	9.41e-07	1.63e-09	0.1821	0.0872
4	8.46e-12	1.24e-14	0.1821	0.0872
5	3.47e-18	0	0.1821	0.0872

Table 1 – f1 and f2 with corresponding values for V1 and V2 as number of iterations increases

Further, the error is plotted in Figure 6. We can see from Figure 6 that the convergence is approximately quadratic.

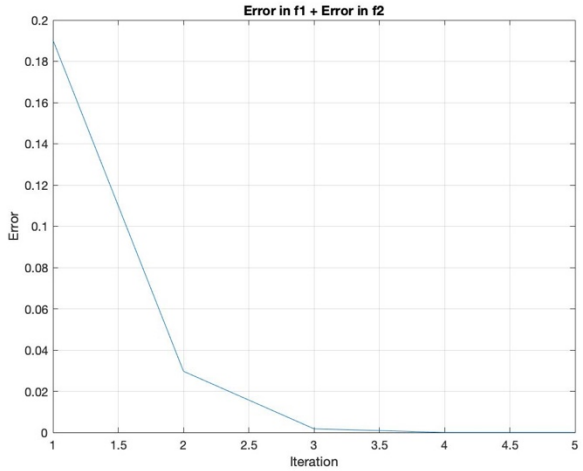

Figure 6 - Error in solutions to f1 and f2. We can see from the curve that convergence is approximately quadratic

3. Code that implements Gauss-Legendre integration for N evenly spaced segments can be found in Appendix 3.1 as *get_integral.m* on page 17. Code that implements Gauss-Legendre integration for N unevenly spaced segments can be found in Appendix 3.2 as *get_uneven_integral.m* on page 17. The included code submission groups these two functions into a single function, *test_q3.m*. Running this function will produce the results discussed in this section.

   (a) Table 2 shows the approximate integral of *sin(x)* from 0 to 1, and corresponding error as the number of segments varies from 1 to 20 (selected values shown). Figure 7 shows the log of the error versus the log of the number of segments. The figure shows that the logarithmically scaled error decreases linearly as the number of segments increases. The true value of the integral is 0.4597.

N (number of segments)	*Integral*	*Error*
1	0.00	0.4597
5	0.4609	0.0012
10	0.4599	0.0002
15	0.4598	0.0001
20	0.4598	0.0001

Table 2 - The integral approximation and corresponding error as the number of segments increases

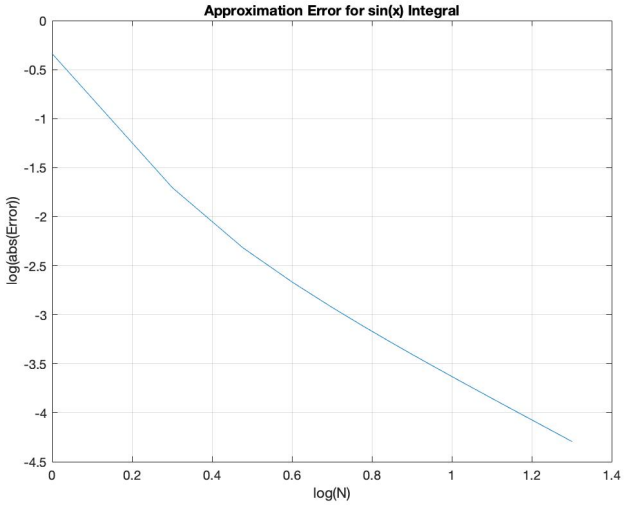

Figure 7 - The log of the error as the log of the number of segments increases

3. (b) Table 3 shows the approximate integral of *ln(x)* from 0 to 1, and the corresponding error as the number of segments varies from 10 to 200 (selected values shown). Figure 8 shows the log of the error versus the log of the number of segments. As in 3.a., the logarithmically scaled error decreases linearly as the number of segments increases. The true value of the integral is -1.0.

N (number of segments)	*Integral*	*Error*
10	-0.9620	0.0380
50	-0.9929	0.0071
100	-0.9965	0.0035
150	-0.9977	0.0023
200	-0.9983	0.0017

Table 3 - The integral approximation and corresponding error as the number of segments increases

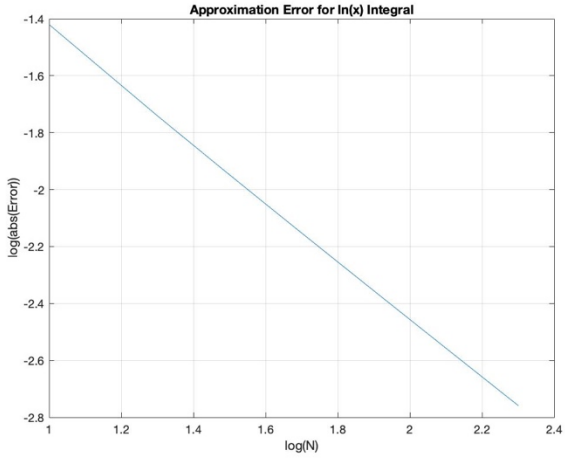

Figure 8 - The log of the error as the log of the number of segments increases

3. (c) Table 4 shows the approximate integral of *ln(0.2|sin(x)|)* from 0 to 1, and the corresponding error as the number of segments varies from 10 to 200 (selected values shown). Figure 9 shows the log of the error versus the log of the number of segments. As in 3.a. and 3.b., the logarithmically scaled error decreases approximately linearly as the number of segments increases. The true value of the integral is -2.666.

N (number of segments)	*Integral*	*Error*
10	0.0000	2.6662
50	-2.5812	0.0850
100	-2.6280	0.0382
150	-2.6415	0.0246
200	-2.6480	0.0182

Table 4 – The integral approximation and corresponding error as the number of segments increases

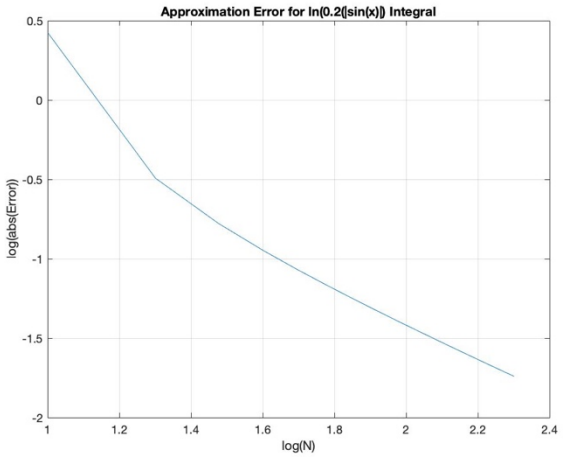

Figure 9 - The log of the error as the log of the number of segments increases

3. (d) An interval spacing with smaller intervals closer to 0 (in regions where the function changes more rapidly), and larger intervals closer to 1 was used. Code is under *get_uneven_integral.m* in Appendix 3.2 on page 17. To reproduce results, run *test_q3.m* in the code submission. The uneven interval spacing approximation with only 10 segments achieves better results than the even interval spacing approximation with 10 segments. The results are shown in Figure 10.

```
 ---------------------- Uneven integrals --------------------
Integral of ln(x):
Integral is -0.9803
 Error is 0.0197

Integral of ln(0.2(|sin(x)|):
Integral is -2.6442
 Error is 0.0219
```

Figure 10 - Integral approximations (from 0 to 1) using 10 segments with uneven interval spacing

APPENDIX 1

1.1 lagrange_interpolator.m

```
function [y_test] = lagrange_interpolator()
% Lagrange interpolator


   hb_data = [0.0 0.0;0.2 14.7; 0.4,36.5;0.6 71.7; 0.8 121.4; 1 197.4; 1.1 256.2; 1.2 348.7; 1.3 540.6; 1.4 1062.8; 1.5
2318.0; 1.6 4781.9; 1.7,8687.4; 1.8 13924.3; 1.9 22650.2];
   X = hb_data(1:6,1);
   Y = hb_data(1:6,2);
   x_test = 0:(1.9/99):1.9;

   y_test = interpolate(x_test, X, Y);

   figure(1)
   plot(y_test, x_test)
   legend("interpolated curve")
   hold on
   plot(Y, X, 'x')
   legend("interpolated curve", "data points", 'location', 'southeast')
   title('Full-domain Lagrange Polynomial Interpolation')
   hold off
   xlabel('H (A/m)')
   ylabel('B (T)')
   grid

   X = hb_data([1 9 10 13 14 15],1);
   Y = hb_data([1 9 10 13 14 15],2);

   y_test = interpolate(x_test, X, Y);


   figure(2)
   plot(y_test, x_test)
   legend("interpolated curve")
   hold on
   plot(Y, X, 'x')
   title('Full-domain Lagrange Polynomial Interpolation')
   legend("interpolated curve", "data points", 'location', 'southeast')
   hold off
   xlabel('H (A/m)')
   ylabel('B (T)')
   grid


   function[y] = multiplier(j, x_test, X)
   %   Function that finds the multiplier given row index and x_test
      [rows_x_test, cols_x_test] = size(x_test);
      [rows_X, cols_X] = size(X)
      if rows_x_test > 1 || cols_x_test > 1
         y(1:rows_x_test, 1:cols_x_test) = 1;
```

```
    else
      y = 1.0;
    end
    for i = 1:rows_X
      if i == j
        continue
      end
      check = (x_test - X(i, 1)) / (X(j,1) - X(i, 1));
      y = y .* (x_test - X(i, 1)) / (X(j,1) - X(i, 1));
    end

  function[y] = interpolate(x_test, X, Y)
    [rows_x_test, cols_x_test] = size(x_test);
    [rows_X, cols_X] = size(X)
    if rows_x_test > 1 || cols_x_test > 1
      y(1:rows_x_test, 1:cols_x_test) = 1;
    else
      y = 1.0;
    end
    for j = 1:rows_X
      y = y + multiplier(j, x_test, X) * Y(j, 1);
    end
```

1.2. hermite_interpolator.m

```
function [y_test] = hermite_interpolator()
% Interpolate using cubic Hermite polynomials

  hb_data = [0.0 0.0;0.2 14.7; 0.4,36.5;0.6 71.7; 0.8 121.4; 1 197.4; 1.1 256.2; 1.2 348.7; 1.3 540.6; 1.4 1062.8; 1.5
2318.0; 1.6 4781.9; 1.7,8687.4; 1.8 13924.3; 1.9 22650.2];
  X = hb_data([1 9 10 13 14 15],1);
  Y = hb_data([1 9 10 13 14 15],2);
  x_test = 0:(1.9/99):1.9;
  y_test(1:100) = 0;
  y_test = interpolate(x_test, X, Y);


  figure(2)
  plot(y_test, x_test)
  legend("interpolated curve")
  hold on
  plot(Y, X, 'x')
  title('Cubic-Hermite Polynomial Interpolation')
  legend("interpolated curve", "data points", 'location', 'southeast')
  hold off
  xlabel('H (A/m)')
  ylabel('B (T)')
  grid

  function[coefficients] = get_coefficients(x0, x1, y0, y1, dy0, dy1)
    A = [x0^3 x0^2 x0 1.0;
      x1^3 x1^2 x1 1.0;
      (3 * x0^2) (2 * x0) 1.0 0.0;
```

```
    (3 * x1^2) (2 * x1) 1.0 0.0];
  b = [y0; y1; dy0; dy1];
  disp("Matrix solver")
  coefficients = mat_solve(A, b);

function[coefficient_matrix] = get_coefficient_matrix(X, Y)
  % Returns a coefficient matrix for the given X and Y (with rows
  % specified by get_coefficients function)
  dy1 = 0;
  [X_rows, X_cols] = size(X);
  coefficient_matrix(1:(X_rows -1), 1:4) = 0;
  for i = 1:X_rows - 1
    x0 = X(i, 1);
    x1 = X(i + 1, 1);
    y0 = Y(i, 1);
    y1 = Y(i + 1, 1);

    % Derivative approximation
    dy0 = dy1;
    dy1 = (y1 - y0) / (x1 - x0);

    coefficient_matrix(i,:) = get_coefficients(x0, x1, y0, y1, dy0, dy1);
  end;

function[x] = mat_solve(A, b)
% Matrix solver for non-symmetric matrices
  det = 1;
  [A_rows, A_cols] = size(A);
  [b_rows, b_cols] = size(b);
  for i = 1:A_rows - 1
    k = i;
    for j=i + 1:A_rows
      if abs(A(j, i)) > abs(A(k, i))
        k = j;
      end
    end
    if k ~= i
      % If k is not equal to i, then transpose ith entry and kth
      % entry of A and b
      A([i k],:)=A([k i],:)
      b([i k]) = b([k i]);
      det = -1 * det;
    end

    for j = (i + 1):A_rows
      t = A(j, i) / A(i, i);
      for k = (i + 1): A_rows
        A(j, k) = A(j,k) - t * A(i, k);
      end
      for k = 1: b_cols
        b(j, k) = b(j, k) - t * b(i, k);
      end
    end
```

```
    end

    for i = A_rows:-1:1
        for j=i + 1: A_rows
            for k=1:b_cols
                b(i, k) = b(i,k) - A(i, j) * b(j, k);
            end
        end
        t = 1.0 / A(i, i);
        det = det * A(i, i);
        for j=1:b_cols
            b(i, j) = b(i, j) * t;
        end
    end

    x = b;

function[y] = apply(coefficient_matrix, x)
%   Apply coefficients to x to get y
    [rows_x, cols_x] = size(x);
    x = [x.^3 x^2 x 1.0];
    % Dot product of the coefficient matrix with x
    y = coefficient_matrix .* x;
    y = sum(y);


function[y] = interpolate(x_test, X, Y)
    [rows_x_test, cols_x_test] = size(x_test);
    [rows_X, cols_X] = size(X);
    coefficient_matrix = get_coefficient_matrix(X, Y);
    y(1:rows_x_test, 1:cols_x_test) = 0;
    for i = 1:cols_x_test
        for j = 1:rows_X - 1
            if x_test(:,i) <= X(j + 1, 1)
                break;
            end;
        end;
        y(:,i) = apply(coefficient_matrix(j,:), x_test(:,i));
    end
```

1.3 newton_raphson.m

```
function [psi] = newton_raphson()
   %   Function that optimizes to some tolerance
   psi = 0;
   tolerance = 1e-6;
   hb_data = [0.0 0.0;0.2 14.7; 0.4 36.5;0.6 71.7; 0.8 121.4; 1 197.4; 1.1 256.2; 1.2 348.7; 1.3 540.6; 1.4 1062.8; 1.5
2318.0; 1.6 4781.9; 1.7,8687.4; 1.8 13924.3; 1.9 22650.2];
   i = 0;
   while abs(flux_expression(psi, hb_data)/flux_expression(0, hb_data)) > tolerance %&& i < 10
           i = i + 1;
           psi = psi - (flux_expression(psi, hb_data) / flux_expression_der(psi, hb_data));
   end
   disp("Iterations: ")
   i
   disp("Flux calculated: ")
   psi

   % Now try successive substitution
   psi_successive_sub = successive_substitution(1e-6, hb_data, tolerance);

           function[flux_ex] = flux_expression(flux, hb_data)
                   % Evaluates the expression for flux found in 1d)

                   flux_ex = 3.978873577e7 * flux + 0.3 * h_val(flux, hb_data) - 8000;

           function[flux_ex_der] = flux_expression_der(flux, hb_data)
                   % Evaluates the expression for flux derivative found in 1d)
                   flux_ex_der = 3.978873577e7 + 0.3 * h_der(flux, hb_data)/(1/(100 ^ 2));

            function [successive_ex] = successive_sub_expression(flux, hb_data)
                   successive_ex = 8000 / (39.78873577e6 + 0.3 * h_val(flux, hb_data) / flux);

           function[flux_ex] = flux_expression(flux, hb_data)
                   % Evaluates the expression for flux found in 1d)
                   flux_ex = 3.978873577e7 * flux + 0.3 * h_val(flux, hb_data) - 8000;

           function[flux_ex_der] = flux_expression_der(flux, hb_data)
                   % Evaluates the expression for flux derivative found in 1d)
                   flux_ex_der = 3.978873577e7 + 0.3 * h_der(flux, hb_data)/(1/(100 ^ 2));


           function[h] = h_val(flux, hb_data)
                   B = flux / (1.0 / (100 ^ 2));
                   [hb_rows, hb_cols] = size(hb_data);
                   % Interpolate for values outside domain
                   if B > hb_data(end,1)
                      slope = (hb_data(end,2) - hb_data(end-1,2)) / (hb_data(end,1) - hb_data(end -1,1));
                      h = (B - hb_data(end, 1)) * slope + hb_data(end, 2);
                      return
                   end

                   for i = 1:hb_rows
```

```
        if hb_data(i,1) > B
            slope = (hb_data(i,2) - hb_data(i-1, 2)) / (hb_data(i, 1) - hb_data(i-1, 1));
            h = (B - hb_data(i-1,1)) * slope + hb_data(i-1, 2);
            return
        end
    end

    % Must be smaller
    slope = (hb_data(2, 2) - hb_data(1, 2)) / (hb_data(2, 1) - hb_data(1,1));
    h = (B - hb_data(1, 1)) * slope + hb_data(1,2);
    return

function[h_der] = h_der(flux, hb_data)
    B = flux / (1.0 / (100 ^ 2));
    [hb_rows, hb_cols] = size(hb_data);
    % Interpolate for values outside domain
    if B > hb_data(end,1)
        h_der = (hb_data(end, 2) - hb_data(end-1, 2)) /  (hb_data(end,1) - hb_data(end-1, 1));
        return
    end

    for i = 1:hb_rows
        if hb_data(i,1) > B
            slope = (hb_data(i,2) - hb_data(i-1, 2)) / (hb_data(i, 1) - hb_data(i-1, 1));
            h_der = slope;
            return
        end
    end

    % Must be smaller
    slope = (hb_data(2, 2) - hb_data(1, 2)) / (hb_data(2, 1) - hb_data(1,1));
    h_der = slope;
    return

function[psi] = successive_substitution(psi, hb_data, tolerance)
    i = 0;

    while abs(flux_expression(psi, hb_data) / flux_expression(0, hb_data)) > tolerance
        i = i + 1;
        psi = psi - (6.5e-9) * flux_expression(psi, hb_data);

    end
    disp("---------- Successive substitution ----------")
    disp("Iterations: ")
    i
    disp("Flux calculated: ")
    psi
```

APPENDIX 2

2.1 test_q2.m

```
function[V] = test_q2()

   V1 =  0;
   V2 = 0;
   E = 0.2;
   R = 512.0;
   Isa = 0.0000008;
   Isb = 0.0000011;
   V1 = 0.0;
   V2 = 0.0;
   k = 0.025;

   iterations = 0;

   V = vector_newton_raphson(Isa,Isb,k,E,R,V1,V2,iterations);
   f1 = V(1,1) - E + R * Isa * (exp((V(1,1) - V(2,1)) / k) - 1.0);
   f2 = Isa * ((exp((V(1,1) - V(2,1)) / k) - 1.0)) - Isb * (exp(V(2,1) / k) - 1.0);

    % Set up storage for errors
   npoints = 10;
   error_list = cell(npoints, 2);

   % Get the error
   fV1 = V(1,1) - E + R * Isa * (exp((V(1,1) - V(2,1)) / k) - 1.0);
   fV2 = Isa * ((exp((V(1,1) - V(2,1)) / k) - 1.0)) - Isb * (exp(V(2,1) / k) - 1.0);
   f01 = -1 * E + R * Isa * (exp(- 1.0));
   f02 = Isa * ((exp(- 1.0)) - Isb * (exp(- 1.0)));
   error_list(1, :) = {1, ((abs(fV1) + abs(fV2)) / (abs(f01) + abs(f02)))};

   while abs(f1) + abs(f2) > 1e-14
      % Update
      iterations = iterations + 1;
      V = vector_newton_raphson(Isa,Isb,k,E,R,V(1,1),V(2,1),iterations);
      f1 = V(1,1) - E + R * Isa * (exp((V(1,1) - V(2,1)) / k) - 1.0);
      f2 = Isa * ((exp((V(1,1) - V(2,1)) / k) - 1.0)) - Isb * (exp(V(2,1) / k) - 1.0);

      % Get the error
      fV1 = V(1,1) - E + R * Isa * (exp((V(1,1) - V(2,1)) / k) - 1.0);
      fV2 = Isa * ((exp((V(1,1) - V(2,1)) / k) - 1.0)) - Isb * (exp(V(2,1) / k) - 1.0);
      f01 = -1 * E + R * Isa * (exp(- 1.0));
      f02 = Isa * ((exp(- 1.0)) - Isb * (exp(- 1.0)));
      error_list(iterations + 1, :) = {iterations + 1, ((abs(fV1) + abs(fV2)) / (abs(f01) + abs(f02)))};

   end

   disp("Number of iterations: ")
   iterations

   disp("V1 : ")
```

```
V(1,1)
disp("V2 : ")
V(2,1)


error_list(iterations + 1:end, :) = [];
errors = cell2mat(error_list);
fprintf('\n Plotting graph for Error... \n ')

figure(1)
plot(errors(:, 1), errors(:,2))
xlabel('Iteration')
ylabel('Error')
grid

function[V] = vector_newton_raphson(Isa, Isb, k, E, R, V1, V2, iterations)
   % Vectorized Newton Raphson implementation
   f1 = V1 - E + R * Isa * (exp((V1 - V2) / k) - 1.0);
   f2 = Isa * ((exp((V1 - V2) / k) - 1.0)) - Isb * (exp(V2 / k) - 1.0);
   f1V1Prime = 1.0 + (R * Isa / k) * (exp((V1 - V2) / k));
   f1V2Prime = -1.0 * (R * Isa / k) * (exp((V1 - V2) / k));
   f2V1Prime = (Isa / k) * (exp((V1 - V2) / k));
   f2V2Prime = -1.0 * (Isa / k) * (exp((V1 - V2) / k)) - (Isb / k) * (exp(V2 / k));
   V = [V1; V2];
   f = [f1; f2];
   J(1:2,1:2) = 0;
   J(1,1) = f1V1Prime;
   J(1,2) = f1V2Prime;
   J(2,1) = f2V1Prime;
   J(2,2) = f2V2Prime;
   J_inverse = mat_inverse(J);

   negative_product = -1 * matrix_multiply(J_inverse, f);
   V = matrix_add_or_subtract(negative_product, V, 'a');

function[A_inv] = mat_inverse(A)
%   Function that inverts a 2 x 2 matrix

   A_det = A(1,1) * A(2,2) - A(1,2) * A(2,1);
   A_inv(1:2, 1:2) = 0;
   A_inv(1,1) = A(2,2) / A_det;
   A_inv(2,2) = A(1,1) / A_det;
   A_inv(1,2) = -1 * A(1,2) / A_det;
   A_inv(2,1) = -1 * A(2,1) / A_det;

function[C] = matrix_add_or_subtract(A,B,operation)
%   Function that performs vector/matrix addition/subtraction
%   Matrices A and B must be the same size. Operation should be specified
%   as 's' for subtract and 'a' for add. Returns the result C

   size_A = size(A);
   size_B = size(B);
   rows_A = size_A(1);
```

```
  cols_A = size_A(2);
  rows_B = size_B(1);
  cols_B = size_B(2);
  if rows_A == rows_B & cols_A == cols_B

    C(1:rows_A,1:cols_A)=0;
    if operation == 'a'
      for i = 1:rows_A
        for j = 1:cols_A
          C(i,j) = A(i,j)+B(i,j);
        end;
      end;
    elseif operation == 's'
      for i = 1:rows_A
        for j = 1:cols_A
          C(i,j) = A(i,j)-B(i,j);
        end;
      end;
    end;
  else
    error("The two matrices to be added have to be equal in size")
  end;

function[C] = matrix_multiply(A, B)
%  Function that multiplies two matricies A and B
%  The number of columns in matrix A must equal the number of rows in matrix B. The
%  product matrix, C, is returned.

  size_A = size(A);
  size_B = size(B);
  rows_A = size_A(1);
  cols_A = size_A(2);
  rows_B = size_B(1);
  cols_B = size_B(2);

  if cols_A == rows_B
    C(1:rows_A,1:cols_B)=0;
    for i = 1:rows_A
      for j = 1:cols_B
        for k = 1:cols_A
          C(i,j) = C(i,j) + (A(i, k) * B(k,j));
        end;
      end;
    end;
  else
    error("Cannot multiply the two matrices. Cols_A must equal Rows_B")
  end;
```

APPENDIX 3

3.1 get_integral.m

```
function [sum_approximation] = get_integral(f1, f2, n, a, b)
    % Approximates the function f2(f1) using n points from a to b
    % f1 and f2 should be function handles
    sum_approximation = 0;
    segments = linspace(a,b,n);
    if ~isa(f2,'function_handle')
      for i = 1:n-1
        lower_limit = segments(i);
        higher_limit = segments(i+1);
        sum_approximation = sum_approximation + (higher_limit - lower_limit) * f1((higher_limit + lower_limit) /
                2.0);
      end
    else
      for i = 1:n-1
        lower_limit = segments(i);
        higher_limit = segments(i+1);
        sum_approximation = sum_approximation + (higher_limit - lower_limit) * f2(0.2 * f1((higher_limit +
                lower_limit)/2.0));
      end
    end
```

3.2 get_uneven_integral.m

```
function [sum_approximation] = get_uneven_integral(f1, f2, a, b)
    % Approximates the function f2(f1) using points from a to b
    % f1 and f2 should be function handles
    % First we want the relative widths between points. We use a relative width scale of
    % [2^0, 2^1, ...., 2^9] for
    relative_widths(1:10) = 0;
    for i = 0:9
      relative_widths(i+1) = 2^i;
    end
    scale = (b - a) / sum(relative_widths);
    [widths_rows, widths_len] = size(relative_widths);
    widths(1:widths_len) = 0;
    for i = 1:widths_len
      widths(i) = relative_widths(i) * scale;
    end
    width = 0;
    sum_approximation = 0;

    if ~isa(f2,'function_handle')

      for i = 1:widths_len
        sum_approximation = sum_approximation + f1(widths(i)/2 + width) * widths(i);
        width = width + widths(i);
      end
    else
      for i = 1:widths_len
```

```
    % f2(0.2(|f1(x)|) - in this question f2 is ln(), f1 is sin()
    sum_approximation = sum_approximation + f2(0.2 * abs(f1(widths(i)/2 + width))) * widths(i);
    width = width + widths(i);
  end

end
```