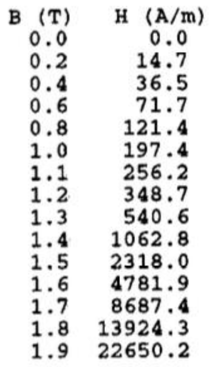
ECSE 543: Numerical Methods
Assignment 3

Student Name: Dafne Culha
Student ID: 260785524

Department of Electrical and Computer Engineering
McGill University, Canada
November, 2020

1. You are given a list of measured BH points for M19 steel (Table 1), with which to construct a continuous graph of B versus H.



B (T)	H (A/m)
0.0	0.0
0.2	14.7
0.4	36.5
0.6	71.7
0.8	121.4
1.0	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

Table 1: B-H Data for M19 Steel

(a) Interpolate the first 6 points using full-domain Lagrange polynomials. Is the result plausible, i.e. do you think it lies close to the true B versus H graph over this range?

To interpolate specified points, algorithm discussed in class was followed. To interpolate 6 points, 6 Lagrange polynomials of 5th order are needed. Each of these polynomials are given by

$$L_j(x) = \frac{Fj(x)}{Fj(x_j)}$$

and where

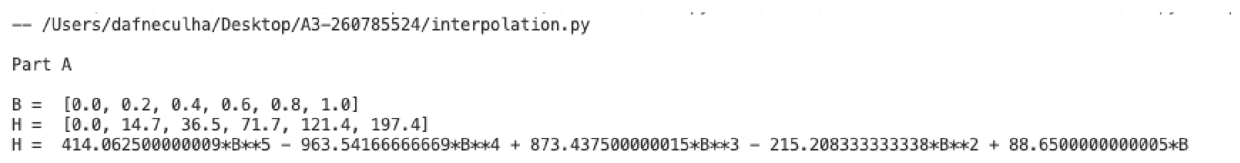
$$F_j(x) = \prod_{r=1, r \neq j}^{n} (x - x_r)$$

Finally, using these, y(x) can be approximated as

$$y(x) = \sum_{j=1}^{n=6} a_j \cdot L_j$$

where $a_j = y(x_j)$.

The whole implementation can be seen in interpolation.py. By using this program, the first 6 points were interpolated using full-domain Lagrange polynomials. The interpolated function was found to be `H = 14.062500000009*B**5 - 963.54166666669*B**4 + 873.437500000015*B**3 - 215.208333333338*B**2 + 88.6500000000005*B` as it is displayed in Figure 1.1.1. As expected, the function is a 5th degree polynomial. This function was plotted, and it's displayed in Figure 1.1.2. As it can be seen, this graph resembles a true B vs. H graph. Thus, it's concluded that the result plausible.



```
-- /Users/dafneculha/Desktop/A3-260785524/interpolation.py

Part 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]
H =  414.062500000009*B**5 - 963.54166666669*B**4 + 873.437500000015*B**3 - 215.208333333338*B**2 + 88.6500000000005*B
```

Figure 1.1.1: Interpolation of First 6 Points

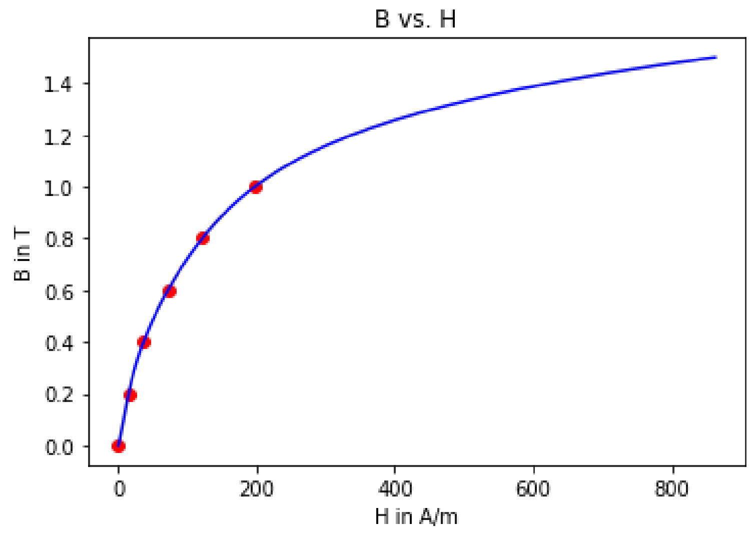
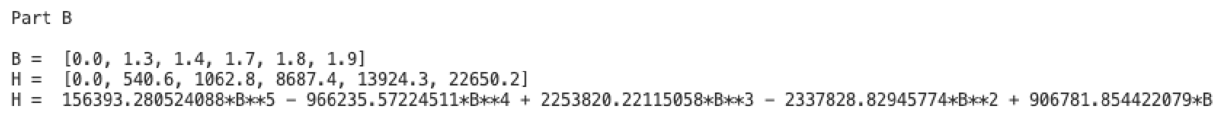


Figure 1.1.2: B vs. H

(b) Now use the same type of interpolation for the 6 points at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9. Is this result plausible?

The data at the specified points was interpolated using the program written for previous part again. The interpolated function was found to be `H = 156393.280524088*B**5 - 966235.57224511*B**4 + 2253820.22115058*B**3 - 2337828.82945774*B**2 + 906781.854422079*B` as it is displayed in Figure 1.2.1. This function was plotted, and it's displayed in Figure 1.2.2. As it can be seen, this graph has too many wiggles and does not resemble the B vs. H graph. Thus, it's concluded that the result is not plausible.



```
Part 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]
H =  156393.280524088*B**5 - 966235.57224511*B**4 + 2253820.22115058*B**3 - 2337828.82945774*B**2 + 906781.854422079*B
```

Figure 1.2.1: Interpolation of Data at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9 with full-domain Lagrange polynomials

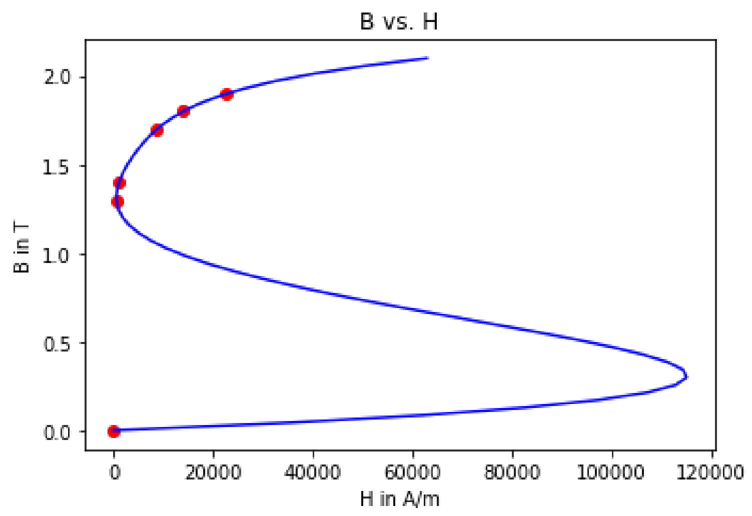


Figure 1.2.2: B vs. H

(c) An alternative to full-domain Lagrange polynomials is to interpolate using cubic Hermite polynomials in each of the 5 subdomains between the 6 points given in (b). With this approach, there remain 6 degrees of freedom - the slopes at the 6 points. Suggest ways of fixing the 6 slopes to get a good interpolation of the points. Test your suggestion and comment on the results.

$$L_j(x) = \frac{Fj(x)}{Fj(x_j)}$$

$$L_j{}'(x) = \frac{dLj}{dx}$$

$$U_j(x) = [-1 - 2L_j{}'(x_j)(x - x_j)] \cdot L_j(x) \cdot L_j(x)$$

$$V_j(x) = (x - x_j) \cdot L_j(x) \cdot L_j(x)$$

where $U_j(x)$ *and* $V_j(x)$ are polynomials of degree 2n-1 = 11.

Finally, using these, y(x) can be obtained as

$$y(x) = \sum_{j=1}^{n=6} a_j \cdot U_j(x) + b_j \cdot V_j(x)$$

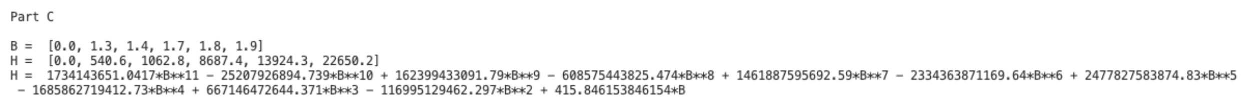
where $a_j = y(x_j)$

and

$$b_j = y'(x_j) = \frac{y(x_{j+1}) - y(x_j)}{x_{j+1} - x_j}$$

The whole implementation can be seen in interpolation.py. Using this program, the data at the specified p oints was interpolated using cubic Hermite polynomials. The interpolated function was found to be `H =`

`1734143651.0417*B**11 - 25207926894.739*B**10 + 162399433091.79*B**9 - 608575443825.474*B**8 + 1461887595692.59*B**7 - 2334363871169.64*B**6 + 2477827583874.83*B**5 - 1685862719412.73*B**4 + 667146472644.371*B**3 - 116995129462.297*B**2 + 415.846153846154*B` as it is presented in Figure 1.3.1. A s expected, the fuction is a 11th degree polynomial. This function was plotted, and it's displayed in Figure 1.3.2. As it can be seen, this graph resembles the B vs. H graph. Thus, it's concluded that the result is plau sible.



```
Part C

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]
H =  1734143651.0417*B**11 - 25207926894.739*B**10 + 162399433091.79*B**9 - 608575443825.474*B**8 + 1461887595692.59*B**7 - 2334363871169.64*B**6 + 2477827583874.83*B**5
 - 1685862719412.73*B**4 + 667146472644.371*B**3 - 116995129462.297*B**2 + 415.846153846154*B
```

Figure 1.3.1: Interpolation of Data at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9 with Cubic Hermite polynomials

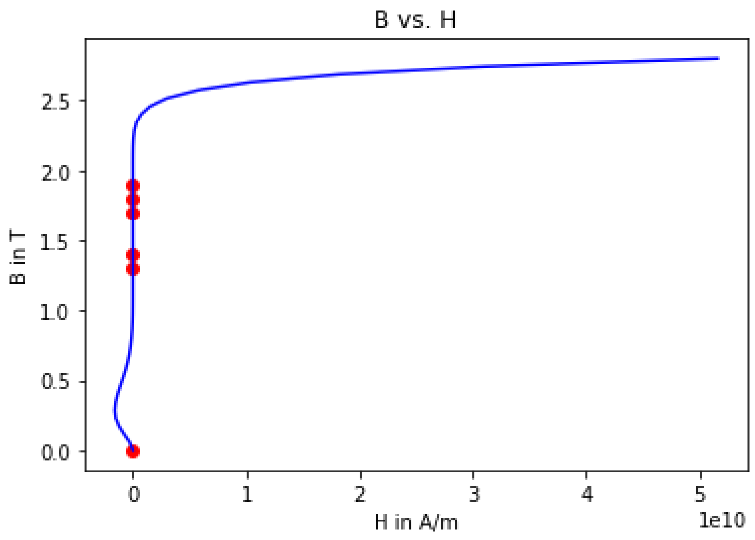


Figure 1.3.2: B vs. H

(d) The magnetic circuit of Figure 1 has a core made of Ml9 steel, with a cross-sectional area of 1 cm^2. L_C=30cm and L_A =0.5cm. The coil has N=800 turns and carries a current I=10A. Derive a (nonlinear) equation for the flux ψ in the core, of the form f(ψ) = 0.

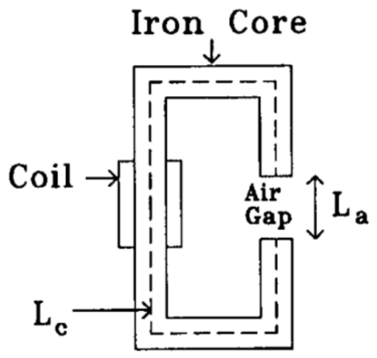


Figure 1.4.1: M19 Steel

By using knowledge learnt in Power Engineering and revised in ECSE 543 lectures,

$$MMF = \mathcal{F} = N \cdot I = R_{Total} \times \psi$$

$$R_{Total} = R_A + R_C$$

$$R_A = \frac{LA}{\mu_A \cdot A_A}$$

$$\mu_c = \frac{B_C}{H_C}$$

$$B_c = \frac{\psi}{A_c}$$

$$R_c = \frac{Lc}{\mu_c \cdot A_c} = \frac{H_C}{B_C} \cdot \frac{Lc}{A_c} = \frac{H_C}{\psi} \cdot Lc$$

$$\mathcal{F} = N \cdot I = 800\ t \cdot 10A = 8000\ A \cdot t$$

$$\mu_A = \mu_0 = 4\pi \times 10^{-7}$$

$$R_A = \frac{L_A}{\mu_A \cdot A_A} = \frac{0.5 \times 10^{-2}}{4\pi \times 10^{-7} \times 1 \times 10^{-4}} = 39\ 788\ 735.77$$

$$R_c = \frac{Lc}{\mu_c \cdot A_c} = \frac{H_C(\psi)}{B_C} \cdot \frac{Lc}{A_c} = \frac{H_C(\psi)}{\psi} \cdot Lc = \frac{H_C(\psi)}{\psi} \cdot 30 \times 10^{-2} = 0.3 \times \frac{H_C(\psi)}{\psi}$$

$$R_{Total} = R_A + R_C = \frac{H_C(\psi)}{\psi} \cdot 30 \times 10^{-2} + 39\ 788\ 735.77$$

$$\mathcal{F} = N \cdot I = R_{Total} \times \psi$$

$$8000 = \psi \cdot (\frac{H_C(\psi)}{\psi} \cdot Lc + R_A) = H_C \times 0.3 + 39\ 788\ 735.77 \cdot \psi$$

$$f(\psi) = 0 = H_C(\psi) \cdot Lc + R_A \cdot \psi - N \cdot I$$

$$f(\psi) = 0 = H_C(\psi) \times 0.3 + 39\ 788\ 735.77 \cdot \psi - 8000$$

(e) Solve the nonlinear equation using Newton-Raphson. Use a piecewise-linear interpolation of the data in Table 1. Start with zero flux and finish when | f(ψ) / f(0) | < 10^{-6}. Record the final flux, and the number of steps taken.

To use Newton-Raphson method, first $f(\psi)$ *and* $f'(\psi)$ need to be found.

$$f(\psi) = 0 = H_C(\psi) \times 0.3 + 39\ 788\ 735.77 \cdot \psi - 8000 = H_C(\psi) \cdot L_C + R_A \cdot \psi - 8000$$

and

$$f'(\psi) = 0 = H_C'(\psi) \times 0.3 + 39\ 788\ 735.77 = H_C{}'(\psi) \times L_C + R_A$$

To get $H_C(\psi)$ *and* $H_C{}'(\psi)$ piecewise linear approximation of data in Table 1 was used. To find these,

$$\mu_c = \frac{\Delta B_c}{\Delta H_c} = \frac{B_i - B_{i-1}}{H_i - H_{i-1}}$$

$$B_c(\psi) = \psi / Area$$

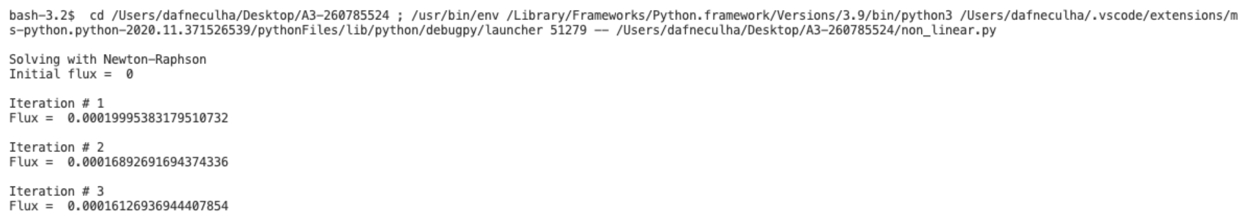
$$H_c(\psi) = \frac{B_c(\psi) - B_{i-1}}{\mu_c} + H_{i-1}$$

$$H_C'(\psi) = \frac{1}{\mu_c}$$

Next, $H_c(\psi)$ *and* $H'_c(\psi)$ were plugged into $f(\psi)$ and $f'(\psi)$. Then, for Newton Raphson, ψ was updated at each iteration using

$$\psi_{i+1} = \psi_i - \frac{f_i}{f'_i}$$

The whole implementation can be seen in non_linear.py. Program took 3 iterations to reach $| f(\psi) / f(0) | < 10^{-6}$ and ψ was found as 0.00016126936944407854 Wb as displayed in Figure 1.5.1.



```
bash-3.2$  cd /Users/dafneculha/Desktop/A3-260785524 ; /usr/bin/env /Library/Frameworks/Python.framework/Versions/3.9/bin/python3 /Users/dafneculha/.vscode/extensions/m
s-python.python-2020.11.371526539/pythonFiles/lib/python/debugpy/launcher 51279 -- /Users/dafneculha/Desktop/A3-260785524/non_linear.py

Solving with Newton-Raphson
Initial flux =  0

Iteration # 1
Flux =  0.00019995383179510732

Iteration # 2
Flux =  0.00016892691694374336

Iteration # 3
Flux =  0.00016126936944407854
```

Figure 1.5.1: Solution of non-linear equation using Newton-Raphson

(f) Try solving the same problem with successive substitution. If the method does not converge, suggest and test a modification of the method that does converge.

For successive substitution, ψ was updated at each iteration using

$$\psi_{i+1} = \psi_i - f_i$$

Using this, with the method didn't converge, and the program kept running forever. Thus, program was modified using the steps explained below.

$$\mathcal{F} = N \cdot I = R_{Total} \times \psi$$

$$8000 = \psi \cdot (\frac{H_C(\psi)}{\psi} \cdot L_C + R_A)$$

From this, $H_C(\psi)$ can be derived as

$$H_C(\psi) = \frac{8000 - R_A \cdot \psi}{L_C} = \frac{8000 - 39\ 788\ 735.77 \cdot \psi}{0.3}$$

Using Table 1 values, $B_C(\psi)$ was obtained as

$$\mu_c = \frac{\Delta B_c}{\Delta H_c} = \frac{B_{i+1} - B_i}{H_{i+1} - H_i}$$

$$B_c(\psi) = \mu_c(H_i(\psi) - H_{i-1}) + B_{i-1}$$

And using $B_c(\psi)$ values, ψ was obtained as

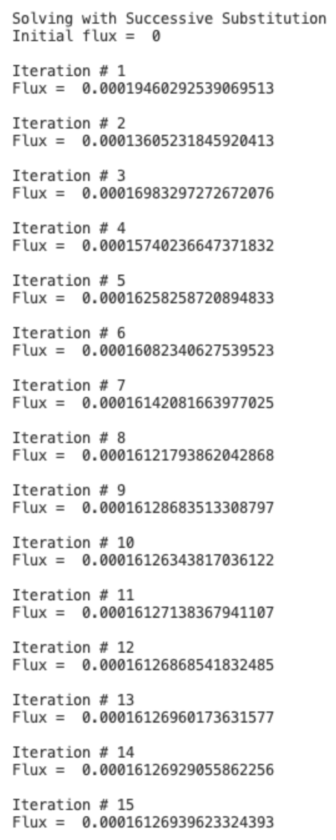
$$\psi = Area \cdot B_c(\psi)$$

These values were plugged into $f(\psi)$

$$f(\psi) = 0 = H_C(\psi) \times 0.3 + 39\ 788\ 735.77 \cdot \psi - 8000$$

and successive substitution was run again. The whole implementation can be seen in non_linear.py

New program took 15 iterations to reach $| f(\psi) / f(0) | < 10^{-6}$ and ψ was found as 0.00016126939623324393 Wb as displayed in Figure 1.6.1.



```
Solving with Successive Substitution
Initial flux =  0

Iteration # 1
Flux =  0.00019460292539069513

Iteration # 2
Flux =  0.00013605231845920413

Iteration # 3
Flux =  0.00016983297272672076

Iteration # 4
Flux =  0.00015740236647371832

Iteration # 5
Flux =  0.00016258258720894833

Iteration # 6
Flux =  0.00016082340627539523

Iteration # 7
Flux =  0.00016142081663977025

Iteration # 8
Flux =  0.00016121793862042868

Iteration # 9
Flux =  0.00016128683513308797

Iteration # 10
Flux =  0.00016126343817036122

Iteration # 11
Flux =  0.00016127138367941107

Iteration # 12
Flux =  0.00016126868541832485

Iteration # 13
Flux =  0.00016126960173631577

Iteration # 14
Flux =  0.00016126929055862256

Iteration # 15
Flux =  0.00016126939623324393
```

Figure 1.6.1: Solution of non-linear equation using Successive Substitution

2. For the circuit shown in Figure 2.1.1 below, the DC voltage E is 200 mV, the resistance R is 512 Ω, the reverse saturation current for diode A is I_{SA} = 0.8 μA, the reverse saturation current for diode B is I_{SB} = 1.1 μA and assume kT/q = 25 mV.

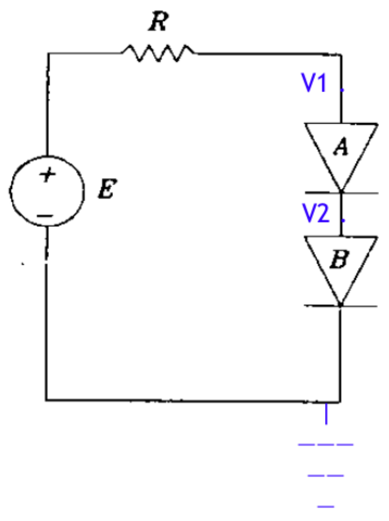


Figure 2.1.1: Circuit with Non-Linear Elements (Diodes)

(a) Derive nonlinear equations for a vector of nodal voltages, v_n, in the form $f(v_n) = 0$. Give f explicitly in terms of the variables I_{sA}, I_{sB}, E, R and v_n.

By Ohm's Law,

$$I = \frac{E - V1}{R}$$

By diode equations learnt in Microelectronics,

$$Vt = \frac{k \cdot T}{q} = 0.025\text{V}$$

KCL in Diode A,

$$I = \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{Vt}\right) - 1\right] = \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right) - 1\right]$$

$$\frac{E - V1}{R} = \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right) - 1\right]$$

$$E - V1 = \text{R} \times \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right) - 1\right]$$

$$f(V1) = 0 = \text{V1} - \text{E} + \text{R} \times \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right) - 1\right]$$

KCL in Diode B,

$$I = \text{I}_{\text{sB}} \times \left[\exp\left(\frac{V2}{Vt}\right) - 1\right] = \text{I}_{\text{sB}} \times \left[\exp\left(\frac{V2}{0.025}\right) - 1\right]$$

$$\frac{E - V1}{R} = \text{I}_{\text{sB}} \times \left[\exp\left(\frac{V2}{0.025}\right) - 1\right] = \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{Vt}\right) - 1\right]$$

$$f(V2) = \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{Vt}\right) - 1\right] - \text{I}_{\text{sB}} \times \left[\exp\left(\frac{V2}{0.025}\right) - 1\right]$$

$$f(Vn) = 0 = \begin{bmatrix} \text{f(V1)} \\ \text{f(V2)} \end{bmatrix} = \begin{bmatrix} \text{V1} - \text{E} + \text{R} \times \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right) - 1\right] \\ \text{I}_{\text{sA}} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right) - 1\right] - \text{I}_{\text{sB}} \times \left[\exp\left(\frac{V2}{0.025}\right) - 1\right] \end{bmatrix}$$

(b) Solve the equation f = 0 by the Newton-Raphson method. At each step, record f and the voltage across each diode. Is the convergence quadratic? [Hint: define a suitable error measure εk].

To design this program, f(Vn) found in previous part was used. Partial derivatives of f1 and f2 were taken with respect to V1 and V2 were taken and they were used to define the Jacobi matrix as presented below.

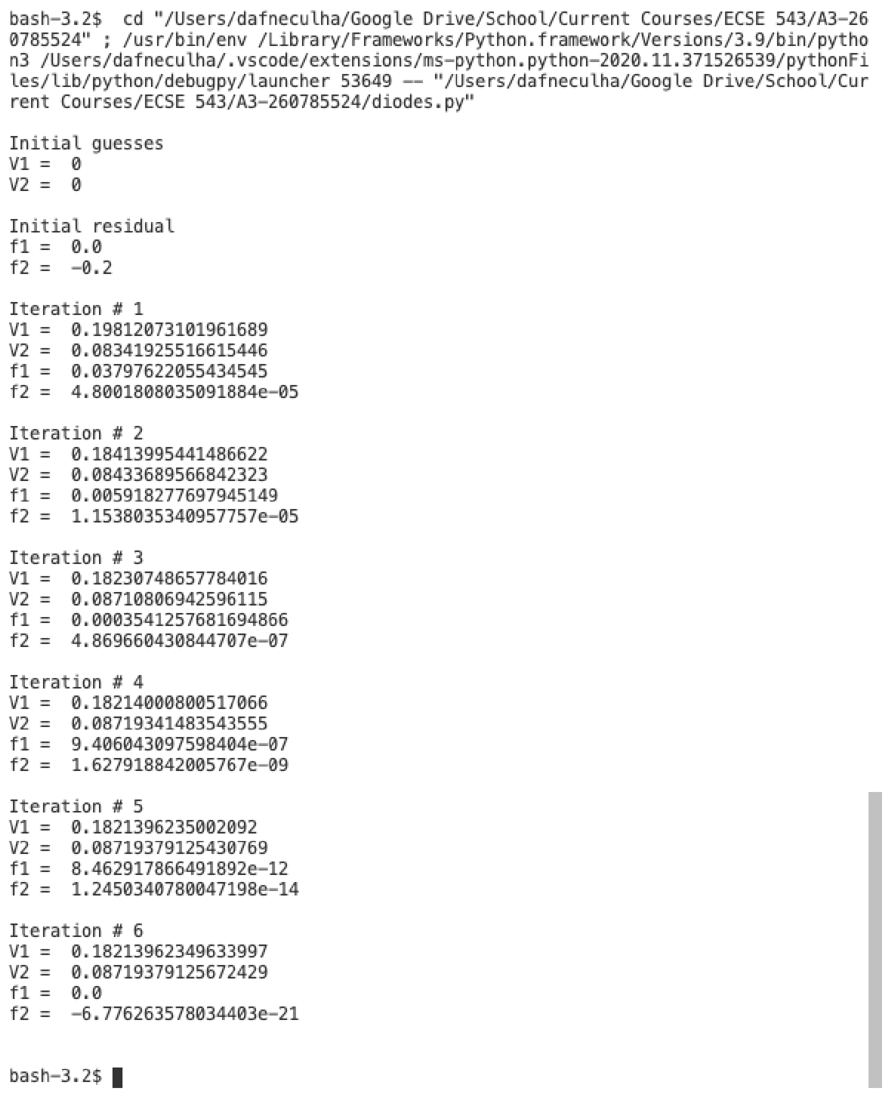
$$J = \begin{bmatrix} 1 + \mathrm{R} \times \mathrm{IsA} \times \frac{1}{0.025} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right)\right] & -\mathrm{R} \times \mathrm{IsA} \times \frac{1}{0.025} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right)\right] \\ \mathrm{IsA} \times \frac{1}{0.025} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right)\right] & -\mathrm{IsA} \times \frac{1}{0.025} \times \left[\exp\left(\frac{V1 - V2}{0.025}\right)\right] - \mathrm{IsB} \times \frac{1}{0.025} \times \left[\exp\left(\frac{V2}{0.025}\right)\right] \end{bmatrix}$$

And for each step $Vn = \begin{bmatrix} \mathrm{V1} \\ \mathrm{V2} \end{bmatrix}$ was updates as $V_{n+1} = \frac{-f + J \cdot V}{J} = -f * J^{-1} + V_n$

And $f(Vn)$ was updated with new Vn entries.

Initial guess for V1 and V2 were set as 0 volts. Since f1 and f2 are supposed to be 0, absolute value of them give the error. Error measure was set as 10^{-15} so that the program would keep iterating until f1 and f2 both become as small as 10^{-15}. Using these, f1, f2, V1 and V2 was computed after each iteration. The full implementation can be seen in Appendix at diodes.py. Results obtained are presented in Figure 2.2.1. It takes 6 iterations for the program to reach a tolerable result. V1 was found as 0.18213962349633997 volts and V2 was found as 0.08719379125672429 volts.

To see if the convergence is quadratic, f1 and f2 are observed. Looking at Figure 2.2.1, it can be observed that the number of zeros after decimal point in f1 is 1, 2, 3, 7, 12, 'inf' as iterations increment from 1 to 6. f2 operates in a similar fashion where the number of zeros after decimal point in f1 is 0, 5, 7, 9, 14, 21 as iterations increment from 0 to 6. This confirms the expectation that the convergence is quadratic.



```
bash-3.2$  cd "/Users/dafneculha/Google Drive/School/Current Courses/ECSE 543/A3-26
0785524" ; /usr/bin/env /Library/Frameworks/Python.framework/Versions/3.9/bin/pytho
n3 /Users/dafneculha/.vscode/extensions/ms-python.python-2020.11.371526539/pythonFi
les/lib/python/debugpy/launcher 53649 -- "/Users/dafneculha/Google Drive/School/Cur
rent Courses/ECSE 543/A3-260785524/diodes.py"

Initial guesses
V1 =  0
V2 =  0

Initial residual
f1 =  0.0
f2 =  -0.2

Iteration # 1
V1 =  0.19812073101961689
V2 =  0.08341925516615446
f1 =  0.03797622055434545
f2 =  4.8001808035091884e-05

Iteration # 2
V1 =  0.18413995441486622
V2 =  0.08433689566842323
f1 =  0.005918277697945149
f2 =  1.1538035340957757e-05

Iteration # 3
V1 =  0.18230748657784016
V2 =  0.08710806942596115
f1 =  0.0003541257681694866
f2 =  4.869660430844707e-07

Iteration # 4
V1 =  0.18214000800517066
V2 =  0.08719341483543555
f1 =  9.406043097598404e-07
f2 =  1.627918842005767e-09

Iteration # 5
V1 =  0.1821396235002092
V2 =  0.08719379125430769
f1 =  8.462917866491892e-12
f2 =  1.2450340780047198e-14

Iteration # 6
V1 =  0.18213962349633997
V2 =  0.08719379125672429
f1 =  0.0
f2 =  -6.776263578034403e-21

bash-3.2$ 
```

Figure 2.2.1: V1, V2, f1 and f2 in each iteration

Appendix

1 - interpolation.py

```
from sympy import *


def lagrange_interpolation (points_x, points_y):
    param = symbols('B')
    n = len(points_x)
    yx = 0
    Ljx = [0.0 for j in range (n)]
    # For j = 0,1,2,....,n
    for j in range (n):
        Fjx = 1
        Fjxj = 1
        for r in range (n):
            if (r != j):
                # Fjx = PRODUCT (x-xr)
                Fjx = Fjx * (param - points_x[r])
                Fjxj = Fjxj * (points_x[j] - points_x[r])
        #Ljx = Fjx / Fjxj
        Ljx[j] = Fjx.expand()/Fjxj
    # yx = SUM (aj*Ljx)
    for j in range (n):
        yx = yx + points_y[j] * Ljx[j]
    return yx

def cubic_hermite(points_x,points_y):
    n = len(points_x)
    param = symbols('B')
    Ujx = []
    Vjx = []
    yx = 0
    # Lj(x)
    Ljx = [None for j in range (n)]
    #L'j(x)
    Ljx_prime = [None for j in range (n)]
    b = [None for j in range (n)]
    a = [None for j in range (n)]

    for j in range(n):
        Fjx = 1
        Fjxj = 1
        for r in range (n):
            if (r != j):
                # Fjx = PRODUCT (x-xr)
                Fjx = Fjx * (param - points_x[r])
                Fjxj = Fjxj * (points_x[j] - points_x[r])
        # To obtain Lj(x)
        # Lj(x) = Fjx / Fjxj
        Ljx = Fjx.expand() / Fjxj

        #To obtain L'j(x), diff Lj(x) / dx
        Ljx_prime = lambdify(param, diff(Ljx))

        U = (1 - 2 * Ljx_prime(points_x[j]) * (param - points_x[j]))*(Ljx**2)
        Ujx.append(U)
        V = (param - points_x[j])*(Ljx * Ljx)
        Vjx.append(V)

    # a[j] = y(j)
    for j in range (n):
        a[j] = points_y[j]

    # b[j] = y'(j)
    for j in range (n):
        if (j < (n-1)):
            b[j] = (points_y[j + 1] - points_y[j]) / (points_x[j + 1] - points_x[j])
        elif (j == (n-1)):
            b[j] = points_y[j]/points_x[j]
        #a[j] = points_y[j]

    # y(x) = SUM (a*Ujx + b*Vjx)
    for j in range(n):
        yx = yx + a[j]*Ujx[j] + b[j]*Vjx[j]
    yx = expand(yx)
    return yx



if __name__ == "__main__":

    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, 2318.0, 4781.9, 8687.4, 13924.3, 22650.2]
    print ('\nPart A\n')
    points_x = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
    print ('B = ', points_x)
    points_y = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4]
    print ('H = ', points_y)
    a = lagrange_interpolation(points_x, points_y)
    print ('H = ',a)

    print ('\n')

    print ('Part B\n')
    points_x = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
    print ('B = ', points_x)
    points_y = [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]
    print ('H = ', points_y)
    b = lagrange_interpolation(points_x, points_y)
    print ('H = ',b)

    print ('\n')

    print ('Part C\n')
    points_x = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
    print ('B = ', points_x)
    points_y = [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]
    print ('H = ', points_y)
    c = cubic_hermite(points_x, points_y)
    print ('H = ',c)

    print ('\n')
```

2 - matrices.py

```
import math
import numpy as np
import time
import random

# returns True if matrix A is symmetric
def is_symmetric(A):
    n = len(A)
    if (n==1):
        return True
    for i in range(n):
        for j in range(i + 1, n):
            return A[i][j] == A[j][i]
#A = [[0.2]]
#print(is_symmetric(A))


# transforms col vectors to right format.
# [x,y,z] into [[x],[y],[z]]
def transform(A):
    try:
        cols = len(A[0])
        return A
    except TypeError:
        result = [[0] for a in range(len(A))]
        for i in range(0, len(A)):
            result[i][0] = A[i]
        return result
#A = [2,3,4]
#print(A)
#print(transform(A))

#returns the transpose of matrix A
def transpose(A):
    A = transform (A)
    A_transpose = [[0 for i in A] for j in A[0]]
    for i in range(len(A)):
        for j in range(len(A[0])):
            A_transpose[j][i] = A[i][j]
    return A_transpose
#A = [[2,3,0]]
#print(transpose(A))
#print(np.transpose(A))

#returns A+B
def add(A, B):
    A = transform(A)
    B = transform(B)
    if len(A)!= len(B) or len(A[0]) != len(B[0]):
        exit('addition not successful because of matrix size error')
    result = []
    for i in range(len(A)):
        result.append([A[i][k]+B[i][k] for k in range(len(A[0]))])
    return result
#A = [2,3,0]
#B = [1,3,0]
#print(add(A,B))

#returns A-B
def subtract(A, B):
    A = transform(A)
    B = transform(B)
    if len(A)!= len(B) or len(A[0]) != len(B[0]):
        exit('subtration not successful because of matrix size error')
    result = []
    for i in range(len(A)):
        result.append([A[i][k]-B[i][k] for k in range(len(A[0]))])
    return result
#A = [[2,3,0], [2,2,1]]
#B = [[1,3,0], [5,1,1]]
#print(subtract(A,B))

#returns A*B
def dot_product(A, B):
    A = transform(A)
    B = transform(B)
    if len(A[0])!= len(B):
        exit('dot product not successful because of matrix size error')
    result = []
    result = [[0 for i in range(len(B[0]))]for k in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(A[0])):
                result[i][j] += A[i][k]*B[k][j]
    return result
#A = [[1,3], [0,1]]
#B = [3,1]
#print(dot_product(A,B))

#returns scalar*A
def scalar_product(scalar, A):
    A = transform(A)
    result = [[0 for i in range(len(A[0]))]for k in range(len(A))]
    for i in range(len(A)):
        for j in range(len(A[0])):
            result[i][j] = scalar*A[i][j]
    return result
#A = [2,6]
#print(scalar_product(5,A))

#returns determinant of a 2x2 matrix
def determinant(A):
    det = A[0][0] * A[1][1] - A[0][1] * A[1][0]
    return det

#returns product inverse of A
def inverse (A):
    det_A = determinant(A)
    inv_A = [[None for x in range (len(A))] for y in range(len(A))]
    inv_A[0][0] = A[1][1] / det_A
    inv_A[0][1] = -1 * A[0][1] / det_A
    inv_A[1][0] = -1 * A[1][0] / det_A
    inv_A[1][1] = A[0][0] / det_A
    return inv_A
#A = [[2,6],[3,1]]
#print(determinant(A))
#print (inverse(A))

#returns a random symmetric positive definite square matrix of size n
def random_symmetric_positive_definite_matrix(n):
    A = np.random.randint(-10,10, size=(n,n))
    return dot_product(A,transpose(A))

#returns a random symmetric positive definite vector of size n
def random_vector(n):
    return np.random.randint(-10,10, size=(n))
```

3 - diodes.py

```
from matrices import *
import numpy as np
import math

# constants in SI units
E = 0.2
R = 512
IsA = 0.8 * 10**(-6)
IsB = 1.1 * 10**(-6)
Vt = 0.025
tolerance = 10**(-15)

def newton_raphson (V1, V2, k):

    #initial residuals
    f1 = V1 - E + R * IsA * (math.exp((V1 - V2) / Vt) - 1)
    f2 = IsA * ((math.exp((V1 - V2) / Vt) - 1)) - IsB * (math.exp(V2 / Vt) - 1)

    print ('\nIteration #', k)
    #Partial Derivatives
    d_f1_V1 = 1.0 + (R * IsA / Vt) * (math.exp((V1 - V2) / Vt))
    d_f1_V2 = -1.0 * (R * IsA / Vt) * (math.exp((V1 - V2) / Vt))
    d_f2_V1 = (IsA / Vt) * (math.exp((V1 - V2) / Vt))
    d_f2_V2 = -1.0 * (IsA / Vt) * (math.exp((V1 - V2) / Vt)) - (IsB / Vt) * (math.exp(V2 / Vt))

    # Jacobian Matrix
    J = [[d_f1_V1, d_f1_V2], [d_f2_V1, d_f2_V2]]

    V = [V1, V2]
    f = [f1, f2]

    #Update voltages
    V = add(scalar_product(-1, dot_product(inverse(J), f)), V)
    V1 = V[0][0]
    V2 = V[1][0]
    print ('V1 = ', V1)
    print ('V2 = ', V2)

    #Update residuals
    f1 = V1 - E + R * IsA * (math.exp((V1 - V2) / Vt) - 1)
    f2 = IsA * ((math.exp((V1 - V2) / Vt) - 1)) - IsB * (math.exp(V2 / Vt) - 1)

    print ('f1 = ', f1)
    print ('f2 = ', f2)

    return V1, V2, f1, f2


if __name__ == "__main__":
    # initial guesses
    V1 = 0
    V2 = 0
    #initial residuals
    f1 = IsA * (math.exp((V1 - V2) / Vt) - 1) - IsB * (math.exp(V2 / Vt) - 1)
    f2 = V1- E + R * IsB * (math.exp(V2 / Vt) - 1)
    print ('\nInitial guesses')
    print('V1 = ', V1)
    print('V2 = ', V2)
    print ('\nInitial residual')
    print('f1 = ', f1)
    print('f2 = ', f2)

    #counter
    k = 1
    while (abs(f1) >=tolerance or abs(f2)>=tolerance):
        V1, V2, f1, f2 = newton_raphson(V1, V2,k)
        k = k + 1
    print ('\n')
```

4 – non_linear.py

```
import math
import numpy as np

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, 2318.0, 4781.9, 8687.4, 13924.3, 22650.2]

Area = 10**(-4)
Lc = 0.3
La = 0.005
N = 800
I = 10
perm0 = 4*math.pi*10**(-7)

Ra = La / (perm0 * Area)
#print (Ra)
MMF = N * I

tolerance = 10**(-6)

# B = flux / area
```

```
def get_B (flux):
    B = flux / Area
    return B
#print(get_B(0.3))

# B = mu * H
# mu = B / H
# H = B / mu
# H_prime = 1/mu
def get_H_and_H_prime(flux):
    Bc = flux/Area
    if Bc > B[-1]:
        mu = (B[-1] - B[-2]) / (H[-1] - H[-2])
        Hc = (Bc - B[-1]) / mu + H[-1]
        H_prime = 1/mu
        return Hc, H_prime

    for i in range(len(B)):
        if B[i] > Bc:
            mu = (B[i] - B[i-1]) / (H[i] - H[i-1])
            Hc = (Bc - B[i-1]) / mu + H[i-1]
            H_prime = 1/mu
            return Hc, H_prime

def get_f(flux):
    Hc, H_prime = get_H_and_H_prime(flux)
    f = Hc * Lc + Ra * flux - MMF
    return f

def get_f_prime(flux):
    Hc, H_prime = get_H_and_H_prime(flux)
    f_prime = H_prime * Lc / Area + Ra
    return f_prime

def newton_raphson(flux):
    print ('Initial flux = ', flux)
    i = 1
    while abs(get_f(flux)/get_f(0)) > tolerance:
        print ('\nIteration #', i)
        flux = flux - get_f(flux)/get_f_prime(flux)
        print ('Flux = ', flux)
        i += 1
    return flux

def successive_sub_xxx(flux):
    i = 1
    while abs(get_f(flux)/get_f(0)) >= tolerance:
        print ('\nIteration #', i)
        flux = get_f(flux)
        print ('Flux = ', flux)
        i += 1
    return flux

def get_Bc(flux):
    Hc = (MMF - Ra * flux) / Lc
    for i in range(len(H) - 1):
        if H[i] <= Hc < H[i + 1]:
            mu = ((B[i + 1] - B[i])/(H[i + 1] - H[i]))
            Bc = B[i] + ((Hc) - H[i]) * mu
            return Bc
        elif Hc > H[- 1]:
            mu = ((B[- 1] - B[- 2])/(H[- 1] - H[- 2]))
            Bc = B[- 1] + (Hc - H[- 1]) * mu
            return Bc

def successive_sub(flux):
    i = 1
    print ('Initial flux = ', flux)
    while abs(get_f(flux)/get_f(0)) >= tolerance:
        print ('\nIteration #', i)
        flux = Area * get_Bc(flux)
        print ('Flux = ', flux)
        i += 1
    return flux
```

```python
if __name__ == "__main__":
    print ('\nSolving with Newton-Raphson')
    newton_raphson(0)
    print ('\n')

    print ('\nSolving with Successive Substitution')
    successive_sub (0)
    print('\n')
```