Tiffany Wang
260684152

ECSE 543 – Assignment 3

TABLE OF CONTENT

Question 1

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. BH Data for M19 Steel

a) Interpolate the first six points using full-domain Lagrange polynomials.

The Lagrange polynomials coefficients are found using:

$$L_j(x) = \prod_{\substack{r=1 \\ r \neq i}}^{n} \frac{x - x_j}{x_j - x_r}$$

Then, the approximated value of H is found using

$$H(x) = \sum_{j=1}^{n} B(x_j) \times L_j(x)$$

The approximated H-field from B = 0.0, 0.01, 0.02 … 0.99, 100 is the following:

B: 0	H: 0.0	B: 26	H: 19.94126	B: 51	H: 54.19884	B: 76	H: 110.0159
B: 1	H: 0.86584	B: 27	H: 20.91216	B: 52	H: 56.01032	B: 77	H: 112.78011
B: 2	H: 1.69375	B: 28	H: 21.91352	B: 53	H: 57.85507	B: 78	H: 115.59734
B: 3	H: 2.48862	B: 29	H: 22.94609	B: 54	H: 59.73303	B: 79	H: 118.46984
B: 4	H: 3.25514	B: 30	H: 24.01055	B: 55	H: 61.64418	B: 80	H: 121.4
B: 5	H: 3.99777	B: 31	H: 25.10747	B: 56	H: 63.58854	B: 81	H: 124.39038
B: 6	H: 4.72075	B: 32	H: 26.23736	B: 57	H: 65.56617	B: 82	H: 127.44372
B: 7	H: 5.42813	B: 33	H: 27.40064	B: 58	H: 67.57717	B: 83	H: 130.56289
B: 8	H: 6.12376	B: 34	H: 28.59766	B: 59	H: 69.62171	B: 84	H: 133.75098
B: 9	H: 6.81128	B: 35	H: 29.8287	B: 60	H: 71.7	B: 85	H: 137.01124
B: 10	H: 7.49414	B: 36	H: 31.09398	B: 61	H: 73.81231	B: 86	H: 140.34711
B: 11	H: 8.17562	B: 37	H: 32.39366	B: 62	H: 75.959	B: 87	H: 143.76222
B: 12	H: 8.8588	B: 38	H: 33.72786	B: 63	H: 78.14047	B: 88	H: 147.2604
B: 13	H: 9.5466	B: 39	H: 35.09663	B: 64	H: 80.35722	B: 89	H: 150.84567
B: 14	H: 10.24174	B: 40	H: 36.5	B: 65	H: 82.60983	B: 90	H: 154.52227
B: 15	H: 10.94681	B: 41	H: 37.93795	B: 66	H: 84.89895	B: 91	H: 158.29464
B: 16	H: 11.66422	B: 42	H: 39.41043	B: 67	H: 87.22533	B: 92	H: 162.16744
B: 17	H: 12.39621	B: 43	H: 40.91737	B: 68	H: 89.58984	B: 93	H: 166.14556
B: 18	H: 13.14489	B: 44	H: 42.45866	B: 69	H: 91.99341	B: 94	H: 170.23411
B: 19	H: 13.91222	B: 45	H: 44.03419	B: 70	H: 94.43711	B: 95	H: 174.43842
B: 20	H: 14.7	B: 46	H: 45.64383	B: 71	H: 96.92211	B: 96	H: 178.76406
B: 21	H: 15.50992	B: 47	H: 47.28744	B: 72	H: 99.44968	B: 97	H: 183.21685
B: 22	H: 16.34352	B: 48	H: 48.96488	B: 73	H: 102.02125	B: 98	H: 187.80285
B: 23	H: 17.20221	B: 49	H: 50.67601	B: 74	H: 104.63834	B: 99	H: 192.52838
B: 24	H: 18.0873	B: 50	H: 52.4207	B: 75	H: 107.30262	B: 100	H: 197.4
B: 25	H: 18.99996						

Figure 1. Approximated H-field using Lagrange Polynomials Interpolation part a

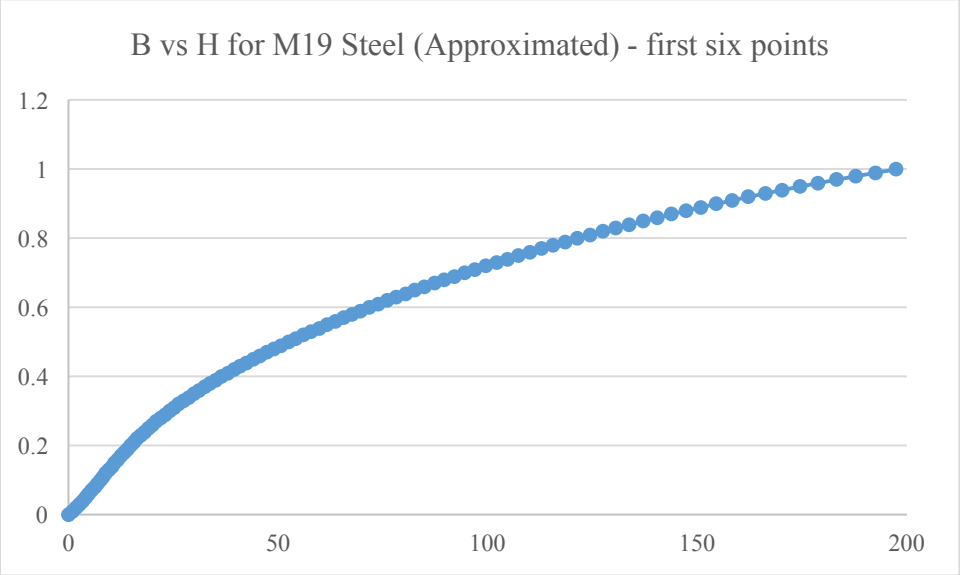


Figure 2. H vs B approximation using full-domain Lagrange Polynomial (first six points)

The graph is really smooth. There are no sharp turns, and the function's behavior is consistent (monotonically increasing). Therefore, we can assume that this interpolation is plausible.

b) Interpolate [0, 1.3, 1.4, 1.7, 1.7, 1.8] using full-domain Lagrange Polynomial

The Lagrange Polynomial coefficients and the final approximation are found using the same formula as in part a.

The obtained results are

B: 0	H: 0.0	B: 26	H: 91582.0	B: 51	H: 11199.0	B: 76	H: 3182.0
B: 1	H: 17218.0	B: 27	H: 87868.0	B: 52	H: 9616.0	B: 77	H: 3650.0
B: 2	H: 32673.0	B: 28	H: 84050.0	B: 53	H: 8180.0	B: 78	H: 4147.0
B: 3	H: 46465.0	B: 29	H: 80156.0	B: 54	H: 6885.0	B: 79	H: 4676.0
B: 4	H: 58695.0	B: 30	H: 76213.0	B: 55	H: 5729.0	B: 80	H: 5238.0
B: 5	H: 69459.0	B: 31	H: 72245.0	B: 56	H: 4706.0	B: 81	H: 5836.0
B: 6	H: 78847.0	B: 32	H: 68276.0	B: 57	H: 3811.0	B: 82	H: 6474.0
B: 7	H: 86950.0	B: 33	H: 64327.0	B: 58	H: 3039.0	B: 83	H: 7157.0
B: 8	H: 93851.0	B: 34	H: 60417.0	B: 59	H: 2382.0	B: 84	H: 7892.0
B: 9	H: 99635.0	B: 35	H: 56563.0	B: 60	H: 1836.0	B: 85	H: 8687.0
B: 10	H: 104378.0	B: 36	H: 52782.0	B: 61	H: 1395.0	B: 86	H: 9551.0
B: 11	H: 108157.0	B: 37	H: 49088.0	B: 62	H: 1051.0	B: 87	H: 10494.0
B: 12	H: 111044.0	B: 38	H: 45494.0	B: 63	H: 798.0	B: 88	H: 11528.0
B: 13	H: 113110.0	B: 39	H: 42011.0	B: 64	H: 630.0	B: 89	H: 12666.0
B: 14	H: 114419.0	B: 40	H: 38648.0	B: 65	H: 541.0	B: 90	H: 13924.0
B: 15	H: 115037.0	B: 41	H: 35414.0	B: 66	H: 524.0	B: 91	H: 15318.0
B: 16	H: 115023.0	B: 42	H: 32318.0	B: 67	H: 573.0	B: 92	H: 16867.0
B: 17	H: 114435.0	B: 43	H: 29363.0	B: 68	H: 683.0	B: 93	H: 18590.0
B: 18	H: 113330.0	B: 44	H: 26555.0	B: 69	H: 848.0	B: 94	H: 20510.0
B: 19	H: 111758.0	B: 45	H: 23899.0	B: 70	H: 1063.0	B: 95	H: 22650.0
B: 20	H: 109770.0	B: 46	H: 21395.0	B: 71	H: 1323.0	B: 96	H: 25036.0
B: 21	H: 107414.0	B: 47	H: 19047.0	B: 72	H: 1625.0	B: 97	H: 27696.0
B: 22	H: 104734.0	B: 48	H: 16853.0	B: 73	H: 1964.0	B: 98	H: 30659.0
B: 23	H: 101771.0	B: 49	H: 14815.0	B: 74	H: 2339.0	B: 99	H: 33958.0
B: 24	H: 98567.0	B: 50	H: 12931.0	B: 75	H: 2745.0	B: 100	H: 37626.0
B: 25	H: 95159.0						

Figure 3. Approximated H-field using Lagrange Polynomials Interpolation part b

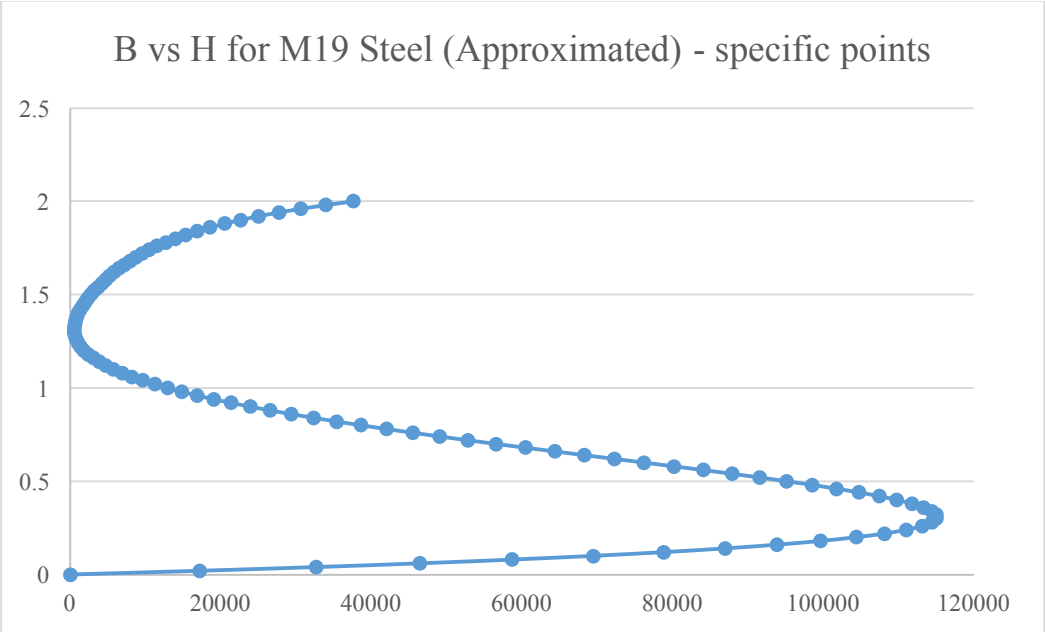


Figure 4. B vs H approximation using full-domain Lagrange Polynomial (specific points)

The shape of the graph does not represent a standard BH graph (i.e. the one found in part a). The result of this interpolation is not plausible.

c) Cubic Hermite polynomial interpolation

We are going to interpolation the six points provided in part b, however using the cubic hermite polynomial method.

$$L_j(x) = \prod_{\substack{r=1 \\ r \neq i}}^{n} \frac{x - x_j}{x_j - x_r}$$

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

$$V_j(x) = (x - x_j)\, {L_j}^2(x)$$

Finally, the approximation answer is found using:

$$H(x) = \sum_{j=1}^{n} y(x_j)U_j(x) + y'(x_j)V_j(x)$$

```
B:  0    H:  0.0    B:  26    H:  216.0  B:  51    H:  424.0  B:  76    H:  4113.0
B:  1    H:  8.0    B:  27    H:  225.0  B:  52    H:  432.0  B:  77    H:  4621.0
B:  2    H:  17.0   B:  28    H:  233.0  B:  53    H:  441.0  B:  78    H:  5129.0
B:  3    H:  25.0   B:  29    H:  241.0  B:  54    H:  449.0  B:  79    H:  5638.0
B:  4    H:  33.0   B:  30    H:  250.0  B:  55    H:  457.0  B:  80    H:  6146.0
B:  5    H:  42.0   B:  31    H:  258.0  B:  56    H:  466.0  B:  81    H:  6654.0
B:  6    H:  50.0   B:  32    H:  266.0  B:  57    H:  474.0  B:  82    H:  7162.0
B:  7    H:  58.0   B:  33    H:  274.0  B:  58    H:  482.0  B:  83    H:  7671.0
B:  8    H:  67.0   B:  34    H:  283.0  B:  59    H:  491.0  B:  84    H:  8179.0
B:  9    H:  75.0   B:  35    H:  291.0  B:  60    H:  499.0  B:  85    H:  8687.0
B:  10    H:  83.0  B:  36    H:  299.0  B:  61    H:  507.0  B:  86    H:  9735.0
B:  11    H:  91.0  B:  37    H:  308.0  B:  62    H:  516.0  B:  87    H:  10782.0
B:  12    H:  100.0 B:  38    H:  316.0  B:  63    H:  524.0  B:  88    H:  11830.0
B:  13    H:  108.0 B:  39    H:  324.0  B:  64    H:  532.0  B:  89    H:  12877.0
B:  14    H:  116.0 B:  40    H:  333.0  B:  65    H:  541.0  B:  90    H:  13924.0
B:  15    H:  125.0 B:  41    H:  341.0  B:  66    H:  645.0  B:  91    H:  15669.0
B:  16    H:  133.0 B:  42    H:  349.0  B:  67    H:  749.0  B:  92    H:  17415.0
B:  17    H:  141.0 B:  43    H:  358.0  B:  68    H:  854.0  B:  93    H:  19160.0
B:  18    H:  150.0 B:  44    H:  366.0  B:  69    H:  958.0  B:  94    H:  20905.0
B:  19    H:  158.0 B:  45    H:  374.0  B:  70    H:  1063.0 B:  95    H:  22650.0
B:  20    H:  166.0 B:  46    H:  383.0  B:  71    H:  1571.0 B:  96    H:  24395.0
B:  21    H:  175.0 B:  47    H:  391.0  B:  72    H:  2079.0 B:  97    H:  26141.0
B:  22    H:  183.0 B:  48    H:  399.0  B:  73    H:  2588.0 B:  98    H:  27886.0
B:  23    H:  191.0 B:  49    H:  408.0  B:  74    H:  3096.0 B:  99    H:  29631.0
B:  24    H:  200.0 B:  50    H:  416.0  B:  75    H:  3604.0 B:  100    H:  31376.0
B:  25    H:  208.0
```

Figure 5. Results from the Approximation using cubic Hermite polynomials

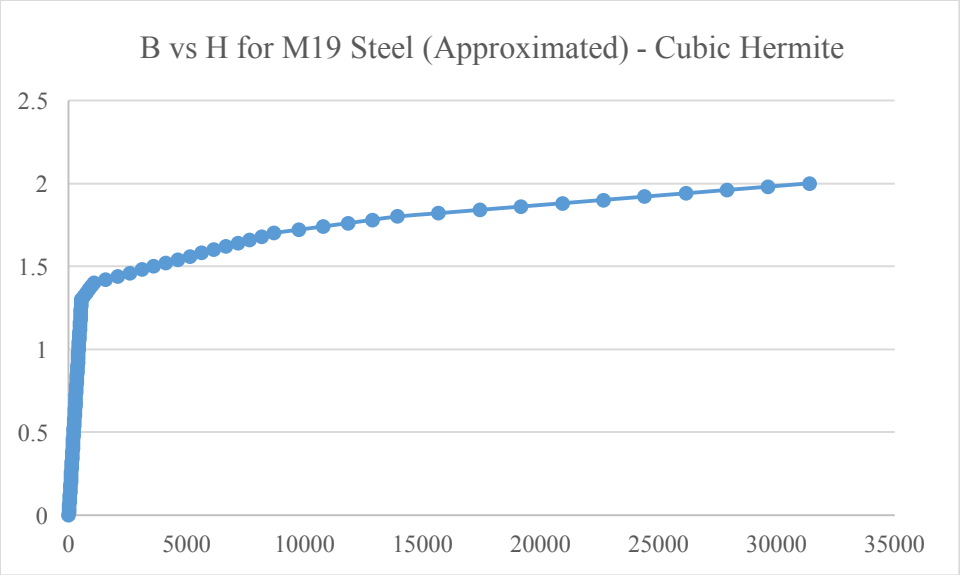


Figure 6. B vs H approximation using Cubic Hermite Polynomials

The curve is a lot smoother, and there is not squiggling. Therefore, we can say that the cubic Hermite's approximation result is plausible and it matches the original function.

In order to obtain a smoother curve, hence a better interpolation of the points, we can force the slopes of the functions to be continuous at the intersection of the subdomains.
We would have:

$$y_1'(1.3) = y_2'(1.3)$$
$$y_2'(1.4) = y_3'(1.4)$$
$$y_3'(1.7) = y_4'(1.7)$$
$$y_4'(1.8) = y_5'(1.8)$$
$$y_5'(1.9) = y_6'(1.3)$$

Question 2

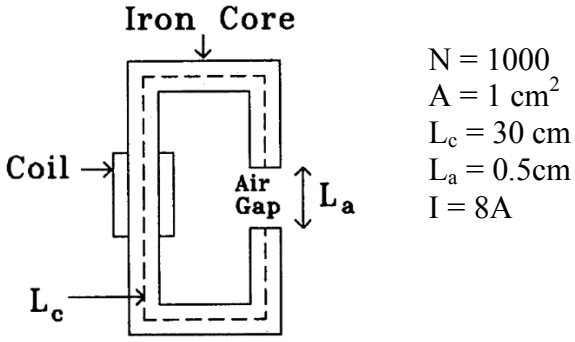


Figure 7. Icon core with magnetic flux

a) Derive a (nonlinear) equation for the flux in the core

The equivalent impedance from a magnetic flux travelling through is $R = \frac{L}{\mu A}\psi$

The iron core is equivalent to the following circuit:

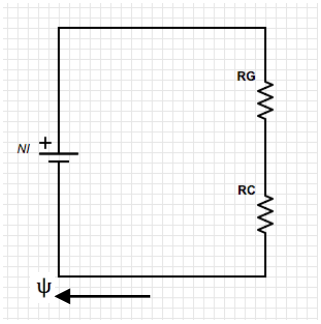


Figure 8. Equivalent circuit to the iron core

Where

R_G: equivalent impedance in the air gap
R_C: equivalent impedance in the core
NI: equivalent voltage source produced by the coil winding
ψ: equivalent current flowing through the circuit

By KVL, this circuit must satisfy:

$$NI = R_G\psi + R_C\psi$$

$$NI = \frac{L_G}{\mu_G A}\psi + \frac{L}{\mu_C A}\psi$$

$$\psi = B \cdot A$$

$$\mu_C = \frac{\psi}{H\,A}$$

$$NI = \psi\,(R_G + \frac{H \cdot L_C}{\psi})$$

$$\boxed{f(\psi) = R_G\psi + H \cdot L_C - NI = 0}$$

b) Solve the flux nonlinear equation using Newton-Raphson

With the original guess of ψ = 0V and $f'(\psi) = R_G + H' \cdot L_C$, where H' is the derivative obtained from Table 1, we solve for ψ^{k+1} in

$$f'(\psi)^k (\psi^{k+1} - \psi^k) + f(\psi)^k = 0$$

until

$$\left|\frac{f(\psi)}{f(0)}\right| < 10^{-6}$$

```
Approximated flux in the steel core is:  0.000160661  Wb
Iteration count:  89
```

Figure 9. Results from nonlinear Newton-Raphson approximation

c) Solve the problem with successive substitution.

$$f(\psi) = R_G\psi + H \cdot L_C - NI = 0$$

H is a function of ψ, since $H = \frac{B}{\mu} = \frac{\psi}{\mu A} = H(\mu)$

Therefore, the equation can be rewritten as

$$R_G\psi + H(\psi) \cdot L_C - NI = 0$$

$$\psi = f(\psi)$$

$$\psi = \frac{NI}{R_G + \frac{H(\psi)}{\psi} \cdot L_C} = f(\psi)$$

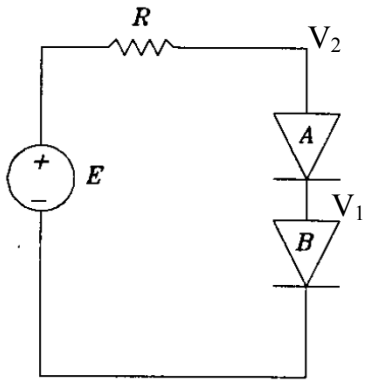
Our goal is to solve for $\psi^{k+1} = \frac{NI}{R_G + \frac{H(\psi)}{\psi} \cdot L_C}$ until $f(\psi^k) < 10^{-6}$

At first the method throws a Divide by Zero error. Yet if you change the initial guess to a very small number, the method converges.
With $\psi_0 = 10^{-6}$

```
Approximated flux in the steel core is:  0.00016066  Wb
```

Figure 10. Results from nonlinear Successive Substitution approximation

Question 3



I_{SA} = 0.6μA
I_{SB} = 1.2μA
kT/q = 25mV
E= 220mW
R=500Ω

Figure 11. Circuit of the analyzed nonlinear circuit

a) Derivation of the nonlinear equations for the nodal voltages [V_2, V_1]

By KVL we have

$$E - RI - V2 = 0$$
$$f_1(\bar{V}_n) = E - R \cdot I_{SB}\left(e^{\frac{qV_1}{kT}} - 1\right) - V_2 = 0$$

We know that the current following through both diodes are the same. Therefore, we can set our second equation to be

$$f_2(\bar{V}_n) = I_{SA}\left(e^{\frac{q(V_2 - V_1)}{kT}} - 1\right) - I_{SB}\left(e^{\frac{qV_1}{kT}} - 1\right) = 0$$

b) Solve nodal voltage equations using Newton-Raphson

$$\bar{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} E - R \cdot I_{SB}\left(e^{\frac{qV_1}{kT}} - 1\right) - V_2 \\ I_{SA}\left(e^{\frac{q(V_2 - V_1)}{kT}} - 1\right) - I_{SB}\left(e^{\frac{qV_1}{kT}} - 1\right) \end{bmatrix} = \bar{0}$$

We need the Jacobian matrix for Newton-Raphson

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial V_1} & \frac{\partial f_1}{\partial V_2} \\ \frac{\partial f_2}{\partial V_1} & \frac{\partial f_2}{\partial V_2} \end{bmatrix}$$

We want to use a similar approach as for Question 2, part b to solve for the nodal voltages of this problem.

$$J^k \cdot (\bar{V}_n^{k+1} - \bar{V}_n^k) + \bar{f}^k = 0$$

$$\bar{V}_n^{k+1} = \bar{V}_n^k - J^{k^{-1}} \bar{f}^k$$

The results of the Newton-Raphson approximation are

```
iteration  0 :  VA =  0.1455   VB =  0.0728
f:  [0.22, 0.0]    vn:  [0.0728, 0.2183]

iteration  1 :  VA =  0.1241   VB =  0.0816
f:  [-0.0087, 0.0002]    vn:  [0.0816, 0.2057]

iteration  2 :  VA =  0.1109   VB =  0.0892
f:  [-0.0008, 0.0001]    vn:  [0.0892, 0.2001]

iteration  3 :  VA =  0.1077   VB =  0.0905
f:  [-0.0008, 0.0]    vn:  [0.0905, 0.1982]

iteration  4 :  VA =  0.1076   VB =  0.0906
f:  [-0.0, 0.0]    vn:  [0.0906, 0.1981]

iteration  5 :  VA =  0.1076   VB =  0.0906
f:  [-0.0, 0.0]    vn:  [0.0906, 0.1981]

iteration  6 :  VA =  0.1076   VB =  0.0906
f:  [-0.0, 0.0]    vn:  [0.0906, 0.1981]
```

Figure 12. Records of f and V_n during Newton-Raphson

The final voltage across Diode A is 107.6 mV and Diode B is 90.6mV

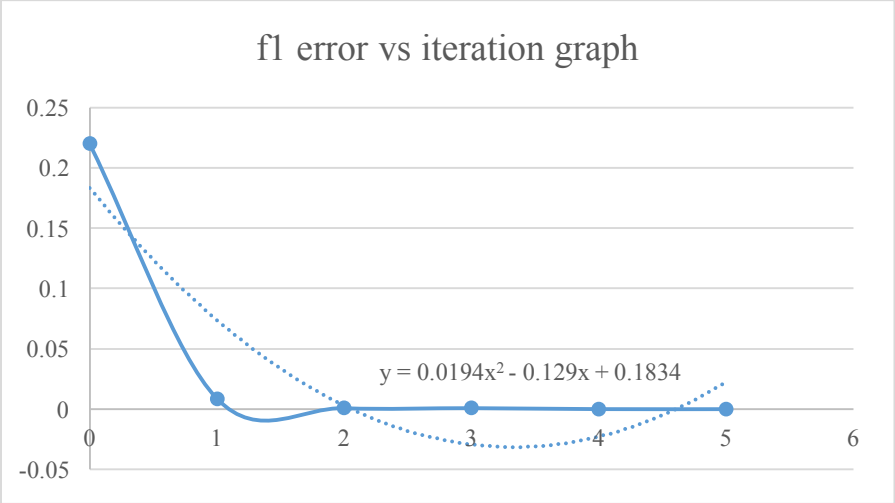


Figure 13. Error trend in the Newton-Raphson approximation

The best fitting trend line to this curve is a quadratic function. Therefore, we can conclude that the convergence is quadratic.

Question 4

a) Integrate cos(x) over x = 0 to x = 1 using one-point Gauss-Legendre integration

The answer of $\int_0^1 \cos(x)\,dx = \sin(1) = 0.0.8414$

We know that for one-point Gauss-Legendre, we have constants:

$$w_i = 2$$
$$x_i = 0$$

We are going to use these constants to solves the integration.

$$\int_a^b f(x)\ dx = w_i \frac{(b-a)}{2} \cdot \int_1^{-1} f\left(\frac{b-a}{2} x_i + \frac{b+a}{2}\right) dx$$

```
N =  1   res =  0.87758256189   error =  0.0361115770825
N =  2   res =  0.850300645292   error =  0.00882966048434
N =  3   res =  0.845379345845   error =  0.00390836103755
N =  4   res =  0.843666316703   error =  0.00219533189465
N =  5   res =  0.8428750743   error =  0.00140408956193
N =  6   res =  0.920510014492   error =  0.0790390296841
N =  7   res =  0.91059703264   error =  0.0691260478319
N =  8   res =  0.842019067246   error =  0.000548082438602
N =  9   res =  0.841903996167   error =  0.000433011359186
N =  10   res =  0.891578804796   error =  0.0501078199886
N =  11   res =  0.841760817405   error =  0.000289832597425
N =  12   res =  0.841714515321   error =  0.000243530512976
N =  13   res =  0.880720517215   error =  0.0392495324068
N =  14   res =  0.878072153789   error =  0.0366011689812
N =  15   res =  0.875757388095   error =  0.0342864032866
N =  16   res =  0.841607958582   error =  0.000136973773665
N =  17   res =  0.841592316399   error =  0.000121331591133
N =  18   res =  0.841579208411   error =  0.000108223603482
N =  19   res =  0.868829895022   error =  0.027358902145
N =  20   res =  0.841558644427   error =  8.76596193869e-05
```

Figure 14. Integration result and error of cos(x) using N equal segments with one-point Gauss-Legendre

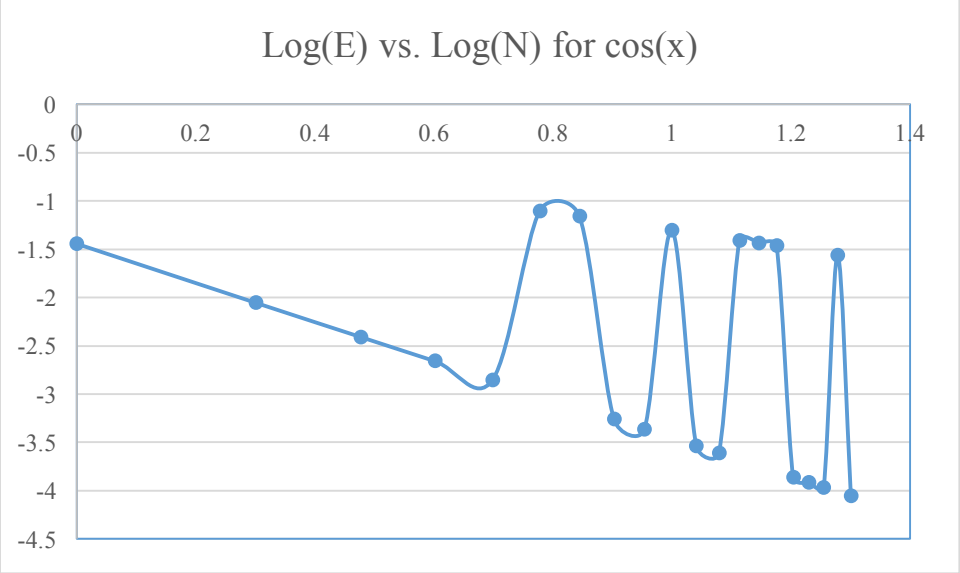


Figure 15. Graph Log(E) vs. Log(N) of the one-point Gauss-Legendre for cos(x)

The logarithmic error decreases in a linearly manner, with a few misplaced points. This means that while increasing the number of segments help with reducing the error, it is not always the case. Certain partitions are not beneficial than others.

b) Integrate ln(x) over x = 0 to x = 1 using one-point Gauss-Legendre integration

The answer of $\int_0^1 \ln(x)\,dx = x \cdot \ln(x) - x = -1$
We know that for one-point Gauss-Legendre, we have constants:

$$w_i = 2$$
$$x_i = 0$$

We are going to use these constants to solves the integration.

```
N =  10   res =  -0.960880048929   error =  0.0391199510708
N =  20   res =  -0.982775471974   error =  0.0172245280263
N =  30   res =  -0.987942863556   error =  0.0120571364444
N =  40   res =  -0.99136170096   error =  0.00863829903958
N =  50   res =  -0.993085194472   error =  0.00691480552777
N =  60   res =  -0.994235347382   error =  0.00576465261812
N =  70   res =  -0.994955773899   error =  0.00504422610056
N =  80   res =  -0.995596458607   error =  0.00440354139305
N =  90   res =  -0.996092768766   error =  0.00390723123363
N =  100   res =  -0.99653843074   error =  0.00346156926044
N =  110   res =  -0.996811545824   error =  0.0031884541763
N =  120   res =  -0.99708013017   error =  0.00291986983011
N =  130   res =  -0.997306985732   error =  0.00269301426825
N =  140   res =  -0.997501135441   error =  0.00249886455916
N =  150   res =  -0.997691361245   error =  0.00230863875482
N =  160   res =  -0.997816041865   error =  0.00218395813464
N =  170   res =  -0.997945497927   error =  0.00205450207299
N =  180   res =  -0.998060466466   error =  0.00193953353388
N =  190   res =  -0.998177082672   error =  0.00182291732836
N =  200   res =  -0.998268173714   error =  0.00173182628625
```

Figure 16. Integration result and error of ln(x) using N equal segments with one-point Gauss-Legendre

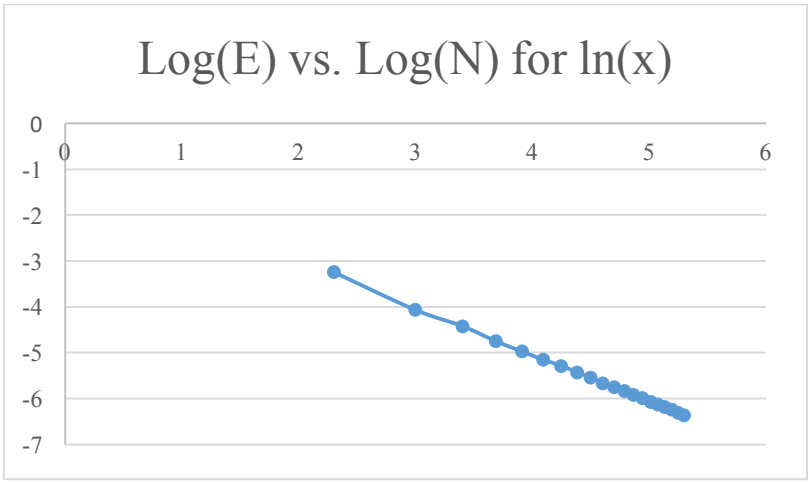


Figure 17. Graph Log(E) vs. Log(N) of the one-point Gauss-Legendre for ln(x)

c) Non-Uniform segmentation for one-point Gauss-Legendre integration

The slope of the ln(x) is steeper as you get closer to 0, which is the area of interest of more difficult integration.

Using the following segmentation coordinates:

$$coord = [\,0.0\,,0.025\,,0.086\,,0.15\,,0.28\,,0.37\,,0.5\,,0.640\,,0.75\,,0.85\,,1\,]$$

The integration result is:

```
Non uniform segments: res =  -0.984617061613   error =  0.0153829383869
```

Figure 18. Result of Non Uniform segmenting one-point Gauss-Legendre integration

Comparison of 10 segmentations one-point Gauss-Legendre integration of ln(x)	
Uniform	Non-Uniform
Error = 0.03912	Error = 0.01538

Table 2. Comparison between uniform and non-uniform segmenting integration

The Non-Uniform segmenting integrating is 60.6% more accurate than the uniform spacing.

APPENDIX

curvefitting.py

```
#interpolate function using lagrange polynomials (whole domain)
def lagrange(self, pos, x, y):
    result = 0
    for i in range(len(x)):
        result += y[i] * self.lagrangePolyWhole(pos, x, i)

    return result



#generate lagrange polynomial coefficients given a set of coordinates of degree index (whole domain)
def lagrangePolyWhole(self, pos, x, index):
    #the polynomials start counting at 1 (not 0)
    numerator = 1.0
    denominator = 1.0
    for i in range(len(x)):
        if(index != i):
            numerator *= (pos - x[i])
            denominator *= (x[index] - x[i])
    return numerator / denominator



#generate lagrange polynomial coefficients given a set of coordinates of degree index (subdomain)
def lagrangePolySub(self, pos, x, n1, n2):
    numerator = float((pos - x[n2]))
    denominator = float(x[n1] - x[n2])
    return numerator / denominator



#generate dL/dx at degree n
def lagrangeDerivativeGenerator(self, x, n1, n2):
    return 1.0 / (x[n1] - x[n2])



#generate Hermite's U polynomial (subdomain)
#U_1 ==> (1, 2)
#U_2 ==> (2, 1)
def hermiteU(self, pos, x, n1, n2):
    L = self.lagrangePolySub(pos, x, n1, n2)
    DL = self.lagrangeDerivativeGenerator(x, n1, n2)
    return (1 - 2 * DL * (pos - x[n1])) * L**2



#generate Hermite's V polynomial (subdomain)
#V_1 ==> (1, 2)
#V_2 ==> (2, 1)
def hermiteV(self, pos, x, n1, n2):
    L = self.lagrangePolySub(pos, x, n1, n2)
    return (pos - x[n1]) * L**2


#approximate result at x in subdomain n
def hermiteSub(self, pos, x, y, n):
    result = 0
    b = (y[n] - y[n-1])/(x[n] - x[n-1])

    #j=n-1
    U = self.hermiteU(pos, x, n - 1, n)
    V = self.hermiteV(pos, x, n - 1, n)
    a = y[n - 1]
    result += a * U + b * V


    #j=n
    U = self.hermiteU(pos, x, n, n-1)
    V = self.hermiteV(pos, x, n, n-1)
    a = y[n]
    result += a * U + b * V

    return result
```

circuit.py (for Question 3)

```
    def newtonRaphson(self, vn, threshold):
        #f1 = E - RIs(e^(q*V1/kt) - 1) - V2
        flog = []
        vlog = []

        while True:
            v2 = vn[1]
            v1 = vn[0]
            f1 = E - R * Isb * math.expm1(v1 / ktq) - v2
            f2 = Isa*(math.exp((v2 - v1) / ktq) - 1.0) - Isb*math.expm1(v1 / ktq)

            f = [f1, f2]

            J = [[None for x in range(2)] for y in range(2)]
            J[0][0] = f1der1(v1, v2)
            J[0][1] = f1der2(v1, v2)
            J[1][0] = f2der1(v1, v2)
            J[1][1] = f2der2(v1, v2)

            Jinv = inverse(J)

            Jinvf = m.matrixVectorMultiplication(Jinv, f)
            vn = m.vectorDifference(vn, Jinvf)

            flog.append(f)
            vlog.append(vn)

            if abs(f1) < threshold: break

        return (flog, vlog)

def f1der1(v1, v2):
    return - R * Isb * math.exp(v1 / ktq) / ktq

def f1der2(v1, v2):
    return -1.0


def f2der1(v1, v2):
    return - Isa / ktq * math.exp((v2-v1)/ktq) - Isb/ktq * math.exp(v1/ktq)

def f2der2(v1, v2):
    return Isa / ktq * math.exp((v2-v1)/ktq)

def inverse(matrix):
    det = 1.0/(matrix[0][0] * matrix[1][1] - matrix[1][0] * matrix[0][1])
    temp = matrix[0][0]
    matrix[0][0] = matrix[1][1] * det
    matrix[1][1] = temp * det

    matrix[1][0] *= -1.0 * det
    matrix[0][1] *= -1.0 * det

    return matrix
```

integration.py

```
class Integration:
    def __init__(self):
        pass

    def GL1P(self, f1, n, s, e):
        n, s, e = float(n), float(s), float(e)
        w = 2
        x = 0
        n = (e - s) / n
        sum = 0
        while(s < e):
            e2 = s + n
            sum += (e2 - s)* f1((e2 - s) / 2.0 * x + (s + e2) / 2.0)
            s += n
        return sum


    def Gl1PNonUni(self, f1, coord):
        w = 2
        x = 0
        sum = 0
        for i in range(1, len(coord)):
            e = coord[i]
            s = coord[i - 1]
            n = e - s
            # print (e - s) * f1((e - s) / 2.0 * x + (s + e) / 2.0)
            sum += (e - s) * f1((e - s) / 2.0 * x + (s + e) / 2.0)

        return sum
```

nonlinear.py

```
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, 4781.8, 8687.4, 13924.3, 22650.2]
A = 0.0001
Rg = 3.978873577e7
NI = 8000


cv = CurveFitting()

class NonLinear(object):
    def __init__(self):
        pass

    #this newtonRaphson method is for the assignment problem only
    #return approximated flux in the steel core
    def newtonRaphson(self, guess, threshold):
        flux = guess
        count = 0
        H_cur = self.Hcur(flux)
        f_0 = 0.3 * H_cur - NI
        while(abs(self.fcur(flux) / f_0) > 1e-6):
            flux = flux - self.fcur(flux) / self.fder(flux)
            count += 1
        return (flux, count)

    def fder(self, flux):
        return Rg + 0.3 * self.Hder(flux) / A

    def fcur(self, flux):
        return Rg * flux + 0.3 * self.Hcur(flux)- NI

    #consider piecewise linear functions when finding H and H_derivative
    def Hder(self, flux):
        b = flux / A
        if b > B[-1] :
            return (H[-1] - H[-2]) / (B[-1] - B[-2])

        for i in range(len(B)):
            if(B[i] > b):
                return (H[i] - H[i - 1]) / (B[i] - B[i - 1])

        return (H[1] - H[0]) / (B[1] - B[0])


    def Hcur(self, flux):
        b = flux / A
        m = self.Hder(b)
        if b > B[-1] :
            return H[-1] + (b - B[-1]) * m

        for i in range(len(B)):
            if(B[i] > b):
                return H[i - 1] + (b - B[i - 1]) * m


        return H[0] + (b - B[0]) * m


    def sucSub(self, guess, threshold):
        f_0 = self.fcur(0)
        flux = guess
        count = 0
        while(abs(self.fcur(flux) / f_0) > threshold):
            flux = self.fsub(flux)
        return flux

    def fsub(self, flux):
        return NI / (Rg + 0.3 * self.Hcur(flux)/flux)
```

Test file for Question 1

```
#====================================Question 1====================================

# a)
B_1 = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
H_1 = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4]

#get H value at every 0.01
x = [0.01 * i for i in range(101)]
result = []
for i in range(len(x)):
    print "B: ", i, "   H: ", round(cv.lagrange(x[i], B_1, H_1), 5)
    result.append(cv.lagrange(x[i], B_1, H_1))


file = open("magnetic1.csv", "w")
for i in range(len(x)):
    string = str(x[i]) + ", " + str(result[i]) + "\n"
    file.write(string)


#b)
B_2 = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
H_2 = [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]

get H value at every 0.01
x = [0.02 * i for i in range(101)]
result = []
for i in range(len(x)):
    print "B: ", i, "   H: ", round(cv.lagrange(x[i], B_2, H_2))
    result.append(cv.lagrange(x[i], B_2, H_2))

file = open("magnetic2.csv", "w")
for i in range(len(x)):
    string = str(x[i]) + ", " + str(result[i]) + "\n"
    file.write(string)

# c)
a = [3, 12]
b = [5, 6]
print cv.lagrangeDerivativeGenerator(a, 1)
print cv.lagrangeDerivativeGenerator(a, 3)
print cv.lagrangeDerivativeGenerator(a, 2)

print cv.hermiteSub(3.5, a, b, 1)
B_3 = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
H_3 = [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]

#get H value at every 0.01
x = [0.02 * i for i in range(101)]
result = []
n = 1;
for i in range(len(x)):
    if (x[i] > B_3[n] and n < len(B_3) - 1):
        n += 1
    print "B: ", i, "   H: ", round(cv.hermiteSub(x[i], B_3, H_3, n))
    result.append(cv.hermiteSub(x[i], B_3, H_3, n))

file = open("magnetic3.csv", "w")
for i in range(len(x)):
    string = str(x[i]) + ", " + str(result[i]) + "\n"
    file.write(string)
```

Test file for Question 2

```
#=========================================Question 2=========================================
result = nl.newtonRaphson(0.0, 1e-6)
flux = result[0]
count = result[1]

print "Approximated flux in the steel core is: ", round(flux, 9), " Wb"
print "Iteration count: ", count

result = nl.sucSub(0.00009, 1e-6)
print round(result, 9)
```

Test file for Question 3

```
#=====================================Question 3=====================================

v = [0, 0]
result = c.newtonRaphson(v, 5e-10)
vlog = result[1]
flog = result[0]

for row in vlog: print "vn: ", [round(elem, 10) for elem in row]
for row in flog: print "fn: ", [round(elem, 10) for elem in row]

for i in range(len(vlog)):
    print "iteration ", i, ": ", "VA = ", str(round(vlog[i][1] - vlog[i][0], 4)), "  VB = ", str(round(vlog[i][0], 4))
    print "f: ", [round(elem, 6) for elem in flog[i]], "    vn: ", [round(elem, 4) for elem in vlog[i]], "\n"

matrix =  [[4, 7], [2, 6]]
```

Test file for Question 4

```
#=========================================Question 4=========================================
# the answer of Int(cos(x)) from x = 0 to x = 1 is sin(1)
answer = math.sin(1)
file = open("cos.csv", "w")

# print result and parse it into a csv file for graphing purposes for cos
for n in range(1, 21):
    res = i.GL1P(math.cos, n, 0, 1)
    error = abs(answer - res)
    print "N = ", n, "  res = ", res, "  error = ", error
    file.write(str(math.log10(n)) + ", " + str(math.log10(error)) + "\n")

print "\n"

# the answer of Int(log(x)) from x = 0 to x = 1 is x*lnx - x = -1
answer = -1
file = open("log.csv", "w")

#print result and parse it into a csv file for graphing purposes for log
for m in [(10 * d) for d in range(1, 21)]:
    res = i.GL1P(math.log, m, 0, 1)
    error = abs(answer - res)
    print "N = ", m, " res = ", res, "  error = ", error
    file.write(str(math.log(m)) + ", " + str(math.log(error)) + "\n")

coord = [0.0, 0.025, 0.086, 0.15, 0.28, 0.37, 0.5, 0.640, 0.75, 0.85, 1]


res = i.Gl1PNonUni(math.log, coord)
error = abs(answer - res)
print "Non uniform segments: res = ", res, "  error = ", error
```