ECSE543 ASSIGNMENT 3 REPORT

Yi Zhu

260716006

All codes for this assignment are developed in Python language. Assignment has been discussed with Yuanzhe Gong.

Question 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.

(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?

For this question I developed a function *lagrange(X, Y)* in class *Interpolation()*. The basic idea of this function is to use the full-domain Lagrange polynomials to find the algebraic equation and curve between inputs X and Y, in our case H and B. This class imports *expand*, *symbols*, *lambdify* and *diff* methods from *sympy* library for an easier calculation. The formula for Lagrange polynomials coefficients learned in class is used:

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

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

Then, we will find our H by using formula:

$$H(x) = \sum_{j=1}^{n} B(x_j) * L_j(x) \text{ .... (2)}$$

In our case, the first 6 points for B in table 1 will be used. The curve equation derived by the function is shown in the screenshot below:

```
Full-domain Lagrange Interpolation:
414.0625*x**5 - 963.541666666672*x**4 + 873.437500000007*x**3 - 215.208333333334*x**2 + 88.6500000000001*x
```

Figure 1

Then, I plugged 20 points for x from 0 to 2 in excel and collected the output from the equation above to plot the curve. The result is shown in chart 1.

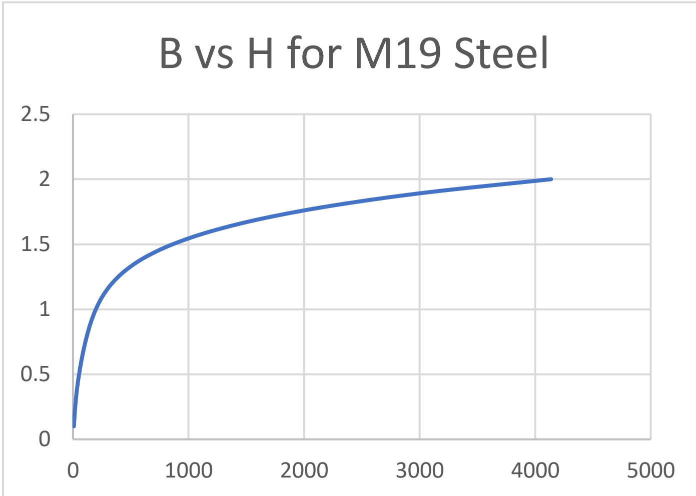

Chart 1

Since the plot looks smoothly and reflects the relation between B and H as shown in table 1, we conclude that the result is plausible.

(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?

In this section, we will use the full-domain Lagrange Polynomial that has the same formula as in the previous section to find the curve equation for B and H. The only difference is that instead of using the first six points for B, we will use certain 6 points that B = 0, 1.3, 1.4, 1.7, 1.8, 1.9.

The equation result derived by the function is shown in the screenshot below:

```
Full-domain Lagrange Interpolation with certain B:
156393.280524086*x**5 - 966235.572245102*x**4 + 2253820.22115057*x**3 - 2337828.82945772*x**2 + 906781.854422078*x
```

Figure 2

Same plotting method is used as for the previous section and the correspond plot is shown in chart 2.

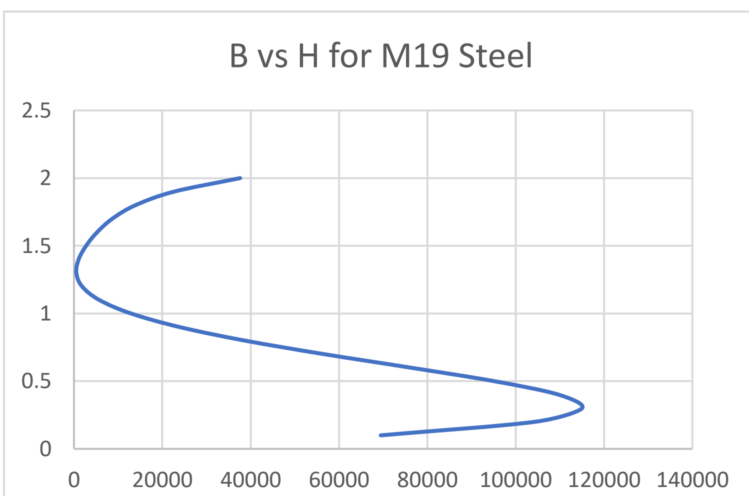

Chart 2

We can clearly see that the graph does not represent a proper relation for B and H. Thus, we conclude that the result of this interpolation is not plausible.

(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.

In this section, we will use the Cubic Hermite polynomial method to do the interpolation for six points illustrated in section (b). A new function named *cubicHermite(X, Y)* is developed, and the formulas used for this function will be:

$$L_j(x) = \prod_{\substack{r=1 \\ r\neq i}}^{n} \frac{x-x_r}{x_j - x_r} \text{...........} \quad (3)$$

$$U_j(x) = [1 - 2L_j'(x_j)(x - x_j)\; L_j{}^2(x) \text{ ....} \quad (4)$$

$$V_j(x) = (x - x_j)\; L_j{}^2(x) \text{ ....} \quad (5)$$

Then, in our case, we will find H by using the formula:

$$H(x) = \sum_{j=1}^{n} B(x_j)U_j(x) + B'(x_j)V_j(x) \text{...........} \quad (6)$$

The equation result derived by the new function is shown in the screenshot below:

```
Cubic Hermite Polynomials Interpolation with certain B:
1734143651.25292*x**11 - 25207926897.778*x**10 + 162399433111.171*x**9 - 608575443897.375*x**8 + 1461887595863.6*x**7 -
2334363871440.03*x**6 + 2477827584159.07*x**5 - 1685862719604.27*x**4 + 667146472719.455*x**3 - 116995129475.341*x**2 + 415.846153846154*x
```

Figure 3

This time, I plugged 20 points for x from 0 to 20 in excel and collected the output from the equation above to plot the curve. The plot result is shown in chart 3.

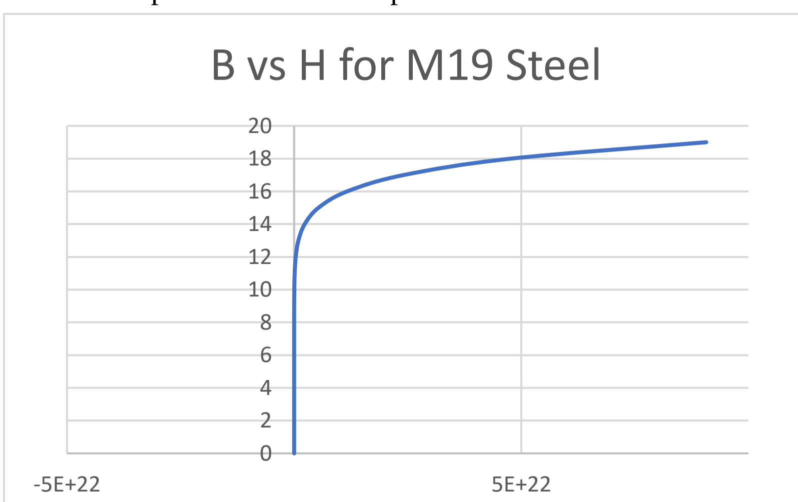

Chart 3

As we can see, the plot looks much more smoothly than the plot we found in section (b), and it reflects the relation between B and H shown in table 1 properly. Thus, we conclude the result is plausible.

To fix the 6 slopes, one possible way is that we can make the slopes of functions at each intersection of two subdomains to be the same. For example: $y_1'(1) = y_2'(1)$ and $y_2'(2) = y_3'(2)$. Then the slope will be continuous along the line.

Question 2

The magnetic circuit of Figure 4 has a core made of Ml9 steel, with a cross-sectional area 1 cm2. Lc = 30 cm and La = 0.5 cm. The coil has N = 1000 turns and carries a current 1 = 8 A.

(a) Derive a (nonlinear) equation for the flux Ψ in the core, of the form f(Ψ) = 0.

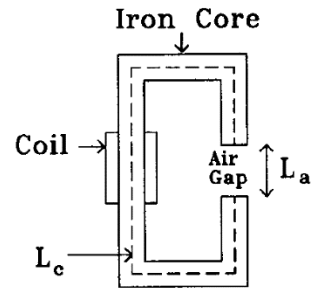

Figure 4

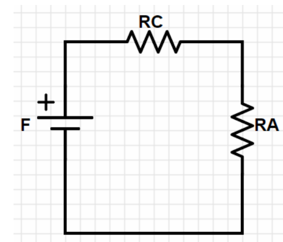

Figure 5

Based on the knowledge of magnetic equivalent circuit we know that a simple magnetic equivalent circuit consists of magnetomotive force F, reluctance of the magnetic path R_c, reluctance of the air gap R_a and flux Ψ. In our case, the equivalent circuit will be similar as in figure 5. Where:

$F = NI = 8000\ At$

$R_c = \frac{L_c}{\mu_c A}$ (μ_c is unknown)

$R_a = \frac{L_a}{\mu_0 A} = \frac{0.5*10^{-2}}{4*\pi*10^{-7}*1*10^{-4}} = 3.9788935 * 10^7 At/Wb$

According to KVL, this circuit satisfies:

$$NI = R_c\Psi + R_a\Psi \text{...........} (7)$$

or

$$NI = \frac{L_c}{\mu_c A}\Psi + \frac{L_a}{\mu_0 A}\Psi \text{...........} (8)$$

Since we know that:

$$\Psi = BA \text{...........} (9)$$

and

$$\mu = \frac{B}{H} \text{...........} (10)$$

Thus, we can derive:

$$\mu_c = \frac{B}{H} = \frac{\Psi}{HA} \text{...........} (11)$$

Plug equation (11) into equation (8) and we will have:

$$NI = \Psi\left(\frac{HL_C}{\Psi} + R_a\right) \text{...........} (12)$$

Then we simplify equation (12) and the nonlinear equation will be:

$$f(\Psi) = HL_C + R_a\Psi - NI = 0 \text{...........} (13)$$

Plug in umbers for L_C, R_a, N and I:

$$f(\Psi) = 0.3 * H + 3.9788935 * 10^7 * \Psi - 8000 = 0 \text{...........} (14)$$

The final nonlinear equation will be the one shown in equation (14).

(b) 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) | < $\mathbf{10^{-6}}$

For this question, a class named *IronCore()* is developed. Inside this class, a function named *newtonRaphson(iguess, maxerror)* is developed. This method will take the initial guess and maxima error as inputs, and it will calculate the result by using the Newton-Raphson iterative method. The formula used for iteration is shown below:

$$f'^{(k)}\left(\mathrm{v}^{(k+1)} - \mathrm{v}^{(k)}\right) + f^{(k)} = 0 \ldots\ldots\ldots\ldots (15)$$

Where:

$$f'^{(k)} = \left.\frac{df}{dv}\right|_{v=v^{(k)}} \ldots\ldots\ldots\ldots (16)$$

In our case, f will be our equation (14) and v will be ψ. Plug them into equation (15) we will get our iteration formula:

$$f'(\psi)^k\,(\psi^{k+1} - \psi^k) + f(\psi)^k = 0 \ldots\ldots\ldots (17)$$

Take derivative of equation (14) and we will get our $f'(\psi)$:

$$f'(\psi) = 3.9788935 * 10^7 + 0.3 * H' \ldots\ldots\ldots\ldots (18)$$

Where H' can be obtained by using the piecewise-linear interpolation of the data in Table 1. To calculate the result using Newton-Raphson method, we will take $\psi = 0$ as our initial guess and we will stop our iteration when the maxima error $\left|\frac{f(\psi)}{f(0)}\right|$ is less than 10^{-6}.

The result calculated by the function derived is shown in the screenshot below, where the flux ψ is calculated to be about 0.00016127Wb.

```
Flux in the M19 iron core calculated by NR method is:  0.0001612693  Wb
Iteration:  3
```

Figure 6

(c) 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.

Record the final flux, and the number of steps taken.

In this question, we will use the Successive Substitution method to calculate the flux. Another function named *successiveSubstitution(iguess, maxerror)* is developed. Different from Newton-Raphson method, the Successive Substitution method will use formula:

$$\mathrm{v}^{(k+1)} - \mathrm{v}^{(k)} + f^{(k)} = 0 \ldots\ldots\ldots\ldots (19)$$

Where in our case f is equation (14) and v is ψ. However, after running this function, it seems that this method does not converge. The result is ‘nan’ as shown in the screenshot below:

```
Flux in the M19 iron core calculated by SS method is:  nan  Wb
Iteration:  40
```

Figure 7

After analysis, I found out that the step size of the Successive Substitution method: $f^{(k)}$ is much greater than the step size of Newton-Raphson method: $\frac{f^{(k)}}{f'^{(k)}}$. Thus, the result may be canceled out in one iteration directly. To solve this problem, I manually decreased the step size by multiplying $f^{(k)}$ by 10^{-8}. The new calculated result is shown in the screenshot below. As we can see, it is around 0.00016127Wb, which is same as we got from the Newton-Raphson method in section (b). The umber of iterations is 23.

```
Flux in the M19 iron core calculated by SS method is:  0.0001612693  Wb
Iteration:  23
```

Figure 8

Question 3

In the circuit shown below, the DC voltage E is 220 mV, the resistance R is 500 Ω, the diode A reverse saturation current IsA is 0.6μA, the diode B reverse saturation current IsB is 1.2μA, and assume kT/q to be 25 mV.

(a) Derive nonlinear equations for a vector of nodal voltages, vn, in the form f(vn) = 0. Give f explicitly in terms of the variables IsA, IsB, E, R and vn.

For the circuit in this question we have:

$E = 220mV$

$R = 500\Omega$

$I_{SA} = 0.6\mu A$

$I_{SB} = 1.2\mu A$

$\frac{kT}{q} = 25mV$

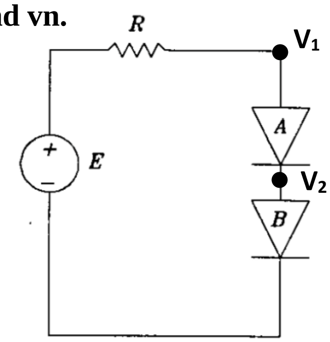

Figure 9

Tow create unknowns V_n for our nonlinear equations, I added two new labels $V_1$and V_2 in the circuit as shown in figure 9.

Now we start deriving our equations. We know for diode:

$$I = I_s * (e^{\frac{V}{V_T}} - 1) \dots\dots\dots\dots (20)$$

Where:

$$V_T = \frac{kT}{q} \dots\dots\dots\dots\dots\dots (21)$$

Thus, we can establish two current equations for two diodes:

$$I = I_{sA} * (e^{\frac{V_1 - V_2}{V_T}} - 1) \dots\dots\dots\dots (22)$$

$$I = I_{sB} * (e^{\frac{V_2}{V_T}} - 1) \dots\dots\dots\dots\dots (23)$$

And from the voltage source and resistor we know:

$$I = \frac{E - V_1}{R} \dots\dots\dots\dots\dots\dots (24)$$

According to KCL we know that the current flowing in the whole circuit is the same, we can then generate two nonlinear equations:

$$f_1 = \frac{E-V_1}{R} - I_{sA} * (e^{\frac{V_1 - V_2}{V_T}} - 1) \ldots\ldots\ldots\ldots (25)$$

$$f_2 = \frac{E-V_1}{R} - I_{sB} * (e^{\frac{V_2}{V_T}} - 1) \ldots\ldots\ldots\ldots\ldots (26)$$

Writing them in vector form and we will get our final answer:

$$f = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} \frac{E-V_1}{R} - I_{sA} * (e^{\frac{V_1 - V_2}{V_T}} - 1) \\ \frac{E-V_1}{R} - I_{sB} * (e^{\frac{V_2}{V_T}} - 1) \end{bmatrix} = 0 \ldots\ldots\ldots\ldots\ldots (27)$$

(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].

In this question, a class named *Diode()* is developed. In this class, a function named *newtonRaphson(iguess, maxerror)* specially designed for solving equations we generated in section (a) will be implemented. Since we have developed two nonlinear equations, we will use the Jacobian matrix in our Newton-Raphson formula. The formula used for Newton-Raphson function is:

$$J^{(k)}\left(\mathrm{v}^{(k+1)} - \mathrm{v}^{(k)}\right) + f^{(k)} = 0 \ldots\ldots\ldots\ldots (28)$$

Where the Jacobian matrix J is:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial V_1} & \frac{\partial f_1}{\partial V_2} \\ \frac{\partial f_2}{\partial V_1} & \frac{\partial f_2}{\partial V_2} \end{bmatrix} \ldots\ldots\ldots\ldots (29)$$

The error I chose for the iteration is ${J^{(k)}}^{-1} * f^{(k)}$and the maxima error is 10^{-5} . Thus, after the error got less than 10^{-5}, the iteration will stop. The result I got is shown in the screenshot below:

```
Iteration: 0 v1: 0.21825396825396826v v2: 0.07275132275132368v f: [0.00044, 0.00044]
Iteration: 1 v1: 0.2056950901096827v v2: 0.0815810263104612v f: [-0.00019811256486904915, -1.733726033021313e-05]
Iteration: 2 v1: 0.20010958229029016v v2: 0.08924973648098972v f: [-5.673770458336547e-05, -1.5511026475102825e-06]
Iteration: 3 v1: 0.1982110618828134v v2: 0.09051583274769154v f: [-1.0199769893483941e-05, -1.638627118623343e-06]
Iteration: 4 v1: 0.1981341341917673v v2: 0.09057062760850919v f: [-3.886721613901022e-07, -5.5589641187179834e-08]
Iteration: 5 v1: 0.198134008229098v v2: 0.09057070781748133v f: [-6.17528266423726e-10, -1.0776787251351666e-10]
```

Figure 10

As we can see, it took 6 iterations to converge. The final value for V_1 is 0.19813v and for V_2 is 0.09057v. Since we can see from the screenshot that the value of $V_1$and V_2 is changing around 2 digits better after each iteration, we can conclude that the convergence is quadratic.

Question 4

(a) Integrate the function cos(x) on the interval x=0 to x=1, by dividing the interval into N equal segments and using one-point Gauss-Legendre integration for each segment. Plot log10(E) versus log10(N) for N=1, 2, …20, where E is the absolute error in the computed integral. Comment on the result.

For this question, a class named *Integration()* is developed. I developed a function named *gaussLegendreUni(f, n ,a b)* to integrate a function by using the Gauss-Legendre integration method for even segments. Where the input f is the function we will integrate, n is the number of segments, a is the lower bond and b is the upper bond. The formula we will use for one-point Gauss-Legendre integration in this function is:

$$\int_a^b f(x)\ dx = \sum_{i=0}^{n} (b-a) * f(\frac{a+b}{2}) \ldots\ldots\ldots\ldots (30)$$

The result derived by this function for $N = 1, 2, \ldots 20$ is shown in the screenshot below:

```
N =  1    Result =  0.8775825618903728   Absolute Error =  0.036111577082476254
N =  2    Result =  0.8503006452922328   Absolute Error =  0.00882966048433631
N =  3    Result =  0.8453793458454515   Absolute Error =  0.003908361037554986
N =  4    Result =  0.8436663167025465   Absolute Error =  0.0021953318946500433
N =  5    Result =  0.8428750743698314   Absolute Error =  0.0014040895619349403
N =  6    Result =  0.8424456991964261   Absolute Error =  0.000974714388529585
N =  7    Result =  0.842186947503467   Absolute Error =  0.0007159626955705045
N =  8    Result =  0.8420190672464982   Absolute Error =  0.0005480824386017158
N =  9    Result =  0.8419039961670828   Absolute Error =  0.000433011359186275
N =  10    Result =  0.8418217000072956   Absolute Error =  0.0003507151993991098
N =  11    Result =  0.8417608174053209   Absolute Error =  0.0002898325974244331
N =  12    Result =  0.8417145153208724   Absolute Error =  0.0002435305129758758
N =  13    Result =  0.8416784838788396   Absolute Error =  0.00020749907094308462
N =  14    Result =  0.8416498955690671   Absolute Error =  0.00017891076117060312
N =  15    Result =  0.8416268329703337   Absolute Error =  0.0001558481624371888
N =  16    Result =  0.8416079585815617   Absolute Error =  0.00013697377366517216
N =  17    Result =  0.8415923163990293   Absolute Error =  0.00012133159113281167
N =  18    Result =  0.8415792084113783   Absolute Error =  0.00010822360348183846
N =  19    Result =  0.8415681153452524   Absolute Error =  9.713053735593835e-05
N =  20    Result =  0.8415586444272835   Absolute Error =  8.765961938694833e-05
```

Figure 11

The plot of log10(E) versus log10(N) for cos(x) where $N = 1, 2, \ldots 20$ is shown in the chart below:

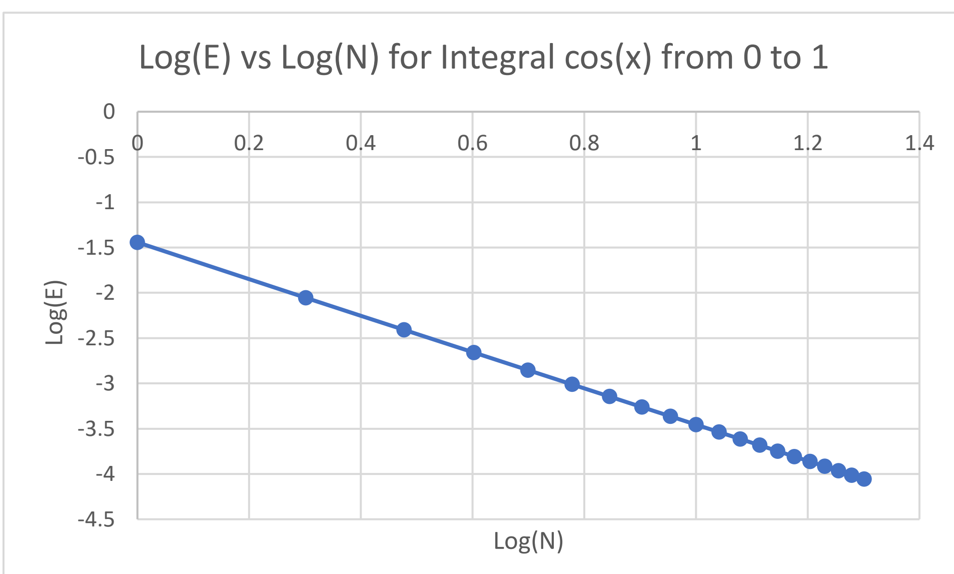

Chart 4

From chart 4 we can see that the relation between Log(E) and Log(N) is almost linear. The logarithm of absolute error is decreasing while the logarithm of number of segments is increasing. We can then conclude that the relation between E and N is monomial.

(b) Repeat part (a) for the function loge(x), only this time plot for N=10, 20, …200. Comment on the result.

For this question, we will use the same formula but integrate function ln(x). The N will be raging from 10 to 200. The result is shown in the screenshot below:

```
N =  10   Result =  -0.9657590653461393   Absolute Error =  0.03424093465386069
N =  20   Result =  -0.982775471973686   Absolute Error =  0.017224528026314023
N =  30   Result =  -0.9884938402873318   Absolute Error =  0.011506159712668218
N =  40   Result =  -0.9913617009604189   Absolute Error =  0.008638299039581132
N =  50   Result =  -0.9930851944722272   Absolute Error =  0.006914805527772794
N =  60   Result =  -0.994235347381881   Absolute Error =  0.005764652618118982
N =  70   Result =  -0.9950574520104222   Absolute Error =  0.004942547989577828
N =  80   Result =  -0.9956743404788297   Absolute Error =  0.004325659521170255
N =  90   Result =  -0.9961543263261001   Absolute Error =  0.0038456736738998742
N =  100   Result =  -0.9965384307395624   Absolute Error =  0.0034615692604376136
N =  110   Result =  -0.9968527745070248   Absolute Error =  0.003147225492975192
N =  120   Result =  -0.997114780254464   Absolute Error =  0.002885219745535572
N =  130   Result =  -0.9973365147802633   Absolute Error =  0.0026634852197366943
N =  140   Result =  -0.9975266001991566   Absolute Error =  0.002473399800843379
N =  150   Result =  -0.9976913612451839   Absolute Error =  0.002308638754816128
N =  160   Result =  -0.9978355426612079   Absolute Error =  0.002164457338792114
N =  170   Result =  -0.9979627735721436   Absolute Error =  0.00203722642785642
N =  180   Result =  -0.9980758771710266   Absolute Error =  0.0019241228289733625
N =  190   Result =  -0.9981770826716382   Absolute Error =  0.0018229173283618172
N =  200   Result =  -0.9982681737137477   Absolute Error =  0.001731826286252347
```

Figure 12

The plot of log10(E) versus log10(N) for ln(x) where $N = 10, 20, \ldots 200$ is shown in the chart below:

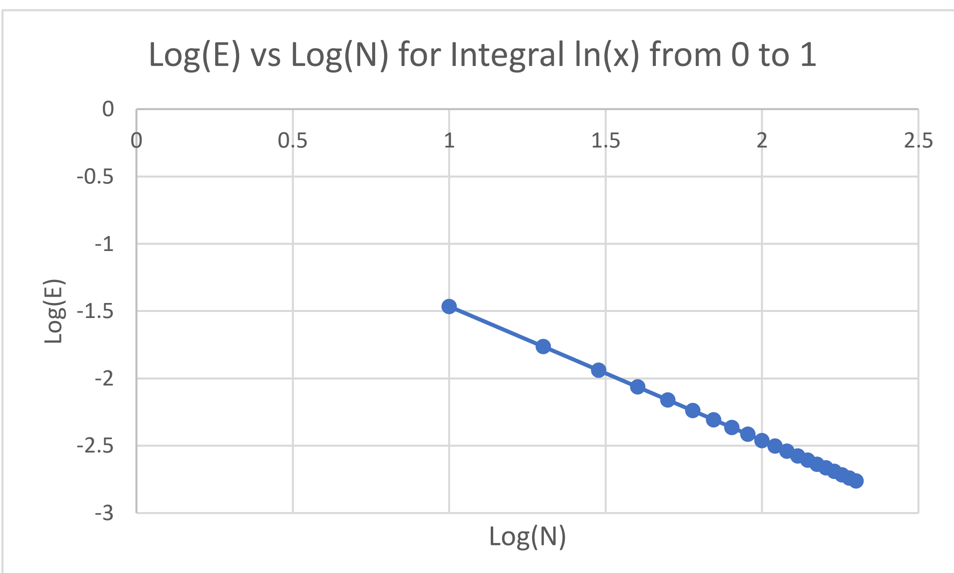

Chart 5

From chart 5 we can see that the relation between Log(E) and Log(N) is still linear. The logarithm of absolute error is decreasing while the logarithm of number of segments is increasing. The change of N is 10 times greater than in section (a), but the relation of E and N is still monomial.

(c) An alternative to dividing the interval into equal segments is to use smaller segments in more difficult parts of the interval. Experiment with a scheme of this kind and see how accurately you can integrate loge(x) using only 10 segments.

For this question, a function named *gaussLegendreNonUni(f, segs)* is developed. In this function, instead of using equal segments, a smaller segment in non-uniform segmentation of the interval will be implemented. Thus, we will have the input *segs*, which is an array of segmentations bonds we chose for our integration. For my experiment, I will choose 10 segments between 0 and 1 and the array *segs* to be:

$$segs \ = [0.0, 0.03, 0.04, 0.08, 0.13, 0.22, 0.45, 0.68, 0.73, 0.91, 1]$$

The calculated result is shown in the screenshot below:

```
Result = -0.9817991221704806  Absolute Error = 0.018200877829519402
```

Figure 13

Compared with the result we calculated by using uniform 10 segments in figure 12, section (b), where the error is 0.034241, the result calculated in this section by using non-uniform segments is 47.7% more accurate.

Appendix

(See following pages of codes)

```python
from sympy import symbols, expand, lambdify, diff

# Author@Yi Zhu
# ID@260716006

class Interpolation(object):

    # Function operates interpolation using full-domain Lagrange polynomials
    def lagrange(sefl, X, Y):
        x = symbols('x')
        res = 0
        n = len(X)

        def l(i, n):
            lx = 1
            for j in range(n):
                if i == j:
                    continue
                xj = X[j]
                lx *= (x - xj) / (xi - xj)
            return lx

        for i in range(n):
            xi = X[i]
            yi = Y[i]
            res += yi * l(i, n)

        return expand(res)

    # Function operates interpolation using Cubic Hermite polynomials
    def cubicHermite(self, X, Y):
        x = symbols('x')
        n = len(X)
        res = 0
        Uj = []
        Vj = []

        def l(i, n):
            lx = 1
            for j in range(n):
                if i == j:
                    continue
                xj = X[j]
                lx *= (x - xj) / (xi - xj)
            return lx

        for i in range(n):
            xi = X[i]
            lx = l(i, n)
            ld = lambdify(x, diff(lx))
            U = (1 - 2 * ld(X[i]) * (x - X[i])) * (lx ** 2)
            V = (x - X[i]) * (lx ** 2)
            Uj.append(U)
            Vj.append(V)

        Yprime = []
        for i in range(n - 1):
            Yprime.append((Y[i + 1] - Y[i]) / (X[i + 1] - X[i]))

        Yprime.append(Y[-1] / X[-1])

        for j in range(n):
            res += Y[j] * Uj[j] + Yprime[j] * Vj[j]

        return expand(res)

```

```python
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
Lc = 0.3
La = 0.5
Ra = 3.9788935e7
NI = 8000

# Author@Yi Zhu
# ID@260716006

class IronCore(object):

    # Function solves nonlinear equation using Newton-Raphson iteration method
    def newtonRaphson(self, iguess, maxerror):
        count = 0
        psi = iguess
        H = self.H(psi)
        fi = Lc * H - NI
        while (abs(self.f(psi) / fi) > maxerror):
            psi -= self.f(psi) / (Ra + Lc * self.Hprime(psi) / A)
            count += 1
        return psi, count

    # Nonlinear function f
    def f(self, psi):
        f = Ra * psi + Lc * self.H(psi) - NI
        return f

    # Obtain h by using piecewise-linear interpolation
    def H(self, psi):
        b = psi / A
        i = 0
        for i in range(len(B) - 1):
            if b <= B[i + 1]:
                break
        x0 = B[i]
        x1 = B[i + 1]
        y0 = H[i]
        y1 = H[i + 1]
        m = (y1 - y0) / (x1 - x0)
        y = m * (b - x0) + y0
        return y

    # Calculate H prime
    def Hprime(self, psi):
        b = psi / A
        i = 0
        for i in range(len(B) - 1):
            if b <= B[i + 1]:
                break
        x0 = B[i]
        x1 = B[i + 1]
        y0 = H[i]
        y1 = H[i + 1]
        m = (y1 - y0) / (x1 - x0)
        return m

    # Function solves nonlinear equation using Successive Substitution method
    def successiveSubstitution(self, iguess, maxerror):
        count = 0
        psi = iguess
        fi = Ra * psi + Lc * self.H(iguess) - NI
        while (abs(self.f(psi) / fi) > maxerror):
            psi -= (Ra * psi + Lc * self.H(psi) - NI)*10**(-8)
            count += 1
```

```
        return psi, count

```

```python
import math
from Matrix import matrix

E = 0.22
R = 500
Isa = 0.6e-6
Isb = 1.2e-6
ktq = 25e-3

m = matrix()

# Author@Yi Zhu
# ID@260716006

class Diode(object):

    # Function solves nonlinear equation using Newton-Raphson iteration method
    def newtonRaphson(self, vn, maxerror):
        count = 0
        v1 = vn[0]
        v2 = vn[1]

        f1 = (E - v1) / R - Isa * (math.exp((v1 - v2) / ktq) - 1.0)
        f2 = (E - v1) / R - Isb * (math.exp(v2 / ktq) - 1.0)
        f = [f1, f2]

        J = [[0 for x in range(2)] for y in range(2)]
        J[0][0] = (-1 / R) - (Isa / ktq) * (math.exp((v1 - v2) / ktq))
        J[0][1] = (Isa / ktq) * (math.exp((v1 - v2) / ktq))
        J[1][0] = (-1 / R)
        J[1][1] = -1 * (Isb / ktq) * (math.exp(v2 / ktq))

        Jinv = m.matrixInverse(J)
        Jinvf = m.matrixVectorMultiplication(Jinv, f)
        vn = m.vectorSubtraction(vn, Jinvf)
        error = [abs(z) for z in Jinvf]
        print("Iteration: " + str(count) + " v1: " + str(vn[0]) + "v v2: " + str(vn[1]) + "v f: " + str(f))

        while (abs(max(error)) > maxerror):
            count += 1
            v1 = vn[0]
            v2 = vn[1]

            f1 = (E - v1) / R - Isa * (math.exp((v1 - v2) / ktq) - 1.0)
            f2 = (E - v1) / R - Isb * (math.exp(v2 / ktq) - 1.0)
            f = [f1, f2]

            J = [[0 for x in range(2)] for y in range(2)]
            J[0][0] = (-1 / R) - (Isa / ktq) * (math.exp((v1 - v2) / ktq))
            J[0][1] = (Isa / ktq) * (math.exp((v1 - v2) / ktq))
            J[1][0] = (-1 / R)
            J[1][1] = -1 * (Isb / ktq) * (math.exp(v2 / ktq))

            Jinv = m.matrixInverse(J)
            Jinvf = m.matrixVectorMultiplication(Jinv, f)
            vn = m.vectorSubtraction(vn, Jinvf)
            error = [abs(z) for z in Jinvf]
            print("Iteration: " + str(count) + " v1: " + str(vn[0]) + "v v2: " + str(vn[1]) + "v f: " + str(f))

        return f, Jinvf, vn, count

```

```python
# Author@Yi Zhu
# ID@260716006

class Integral:

    # Function calculate integral by using Gauss-Legendre method with uniform segments
    def gaussLegendreUni(self, f, n, a, b):
        n, a, b = float(n), float(a), float(b)
        sum = 0
        w = (b - a) / n
        h = [w] * int(n)
        for w in h:
            low = a
            a += w
            up = a
            sum += (up - low) * f((low + up) / 2)
        return sum

    # Function calculate integral by using Gauss-Legendre method with non-uniform segments
    def gaussLegendreNonUni(self, f, segs):
        sum = 0
        h = len(segs)
        for i in range(1, h):
            b = segs[i]
            a = segs[i - 1]
            sum += (b - a) * f((a + b) / 2)
        return sum

```

```python
import math, copy
#Author@Yi Zhu
#ID@260716006

class matrix(object):

    # Method to solve Ax=b using Choleski and fwd/bwd elimination
    def solveMatrix(self, matrix, b):
        self.checkSym(matrix)
        self.checkDet(matrix)
        L = self.choleskiDecompose(matrix)
        Y = self.forwardElm(L, b)
        Lt = self.matrixTranspose(L)
        X = self.backElm(Lt, Y)
        return X

    # Method to solve Ax=b using Choleski and fwd/bwd elimination and sparse matrix
    def sparseSolveMatrix(self, matrix, b, band):
        self.checkSym(matrix)
        self.checkDet(matrix)
        L = self.sparseCholeskiDecompose(matrix, band)
        Y = self.forwardElm(L, b)
        Lt = self.matrixTranspose(L)
        X = self.backElm(Lt, Y)
        return X

    # Choleski decomposition using look ahead method return L
    def choleskiDecompose(self, matrix):
        A = copy.deepcopy(matrix)
        n = len(A)
        L = [[0.0] * n for i in range(n)]
        for j in range(n):
            if A[j][j] <0:
                exit('Input matrix must be a positive-definite matrix!')
            L[j][j] = math.sqrt(A[j][j])
            for i in range(j + 1, n):
                L[i][j] = A[i][j] / L[j][j]
                for k in range(j + 1, i + 1):
                    A[i][k] = A[i][k] - L[i][j] * L[k][j]
        return L

    # Choleski decomposition using look ahead method and sparse matrix return L
    def sparseCholeskiDecompose(self, matrix, b):
        A = copy.deepcopy(matrix)
        n = len(A)
        L = [[0.0] * n for i in range(n)]
        for j in range(n):
            if A[j][j] < 0:
                exit('Input matrix must be a positive-definite matrix!')
            L[j][j] = math.sqrt(A[j][j])
            for i in range(j + 1, min(j + 1 + b, n)):
                L[i][j] = A[i][j] / L[j][j]
                for k in range(j + 1, min(j + 1 + b ,i + 1)):
                    A[i][k] = A[i][k] - L[i][j] * L[k][j]
        return L

    # Forward elimination return Y
    def forwardElm(self, lMatrix, bVector):
        L = copy.deepcopy(lMatrix)
        b = copy.deepcopy(bVector)
        n = len(b)
        Y = []
        for j in range(n):
            b[j] = b[j]/L[j][j]
            Y.append(b[j])
            for i in range(j+1, n):
                b[i] = b[i] - L[i][j]*b[j]
```

```python
        return Y

    # Backward elimination return X
    def backElm(self, LtMatrix, yVector):
        Lt = copy.deepcopy(LtMatrix)
        Y = copy.deepcopy(yVector)
        n = len(Y)
        X = []
        for i in range(n)[::-1]:
            sum = 0
            for j in range(n)[:i:-1]:
                sum += X[n - j - 1] * Lt[i][j]
            X.append((Y[i] - sum) / Lt[i][i])
        return X[::-1]

    # Check if the matrix is symmetric
    def checkSym(self, matrix):
        n = len(matrix)
        for i in range(n):
            for j in range(i + 1, n):
                if matrix[i][j] != matrix[j][i]:
                    exit('Input matrix must be a symmetric matrix!')

    # Check if the determinant of the matrix is zero
    def checkDet(self, matrix):
        n = len(matrix)
        det = 1
        for i in range(n):
            det *= matrix[i][i]
        if det <= 0:
            exit('Input matrix must be a positive definite matrix!')

    # Check if A * x equals b
    def checkSol(self, A, X, b):
        n = len(A)
        for i in range(n):
            res = self.matrixVectorMultiplication(A,X)
            if abs(b[i] - res[i]) > 0.01:
                return False
        return True

    # Matrix transpose return the transpose of a matrix
    def matrixTranspose(self, matrix):
        A = copy.deepcopy(matrix)
        nRow = len(A)
        nCol = len(A[0])
        res = []
        for i in range(nCol):
            row = []
            for j in range(nRow):
                row.append(A[j][i])
            res.append(row)
        return res

    # Matrix multiply by a vector return the result vector
    def matrixVectorMultiplication(self, A, b):
        res = []
        nRow = len(A)
        nCol = len(b)
        for i in range(nRow):
            sum = 0
            for j in range(nCol):
                sum += A[i][j] * b[j]
            res.append(sum)
        return res

    # Matrix multiply by another matrix return the result matrix
```

```
    def matrixMultiplication(self, A, B):
        nRow = len(A)
        nCol = len(B[0])
        nB = len(B)
        res = [0.0] * nRow
        for i in range(nRow):
            res[i] = [0] * nCol
        for i in range(nRow):
            for j in range(nCol):
                for k in range(nB):
                    res[i][j] += A[i][k] * B[k][j]
        return res

    # Vector multiply by another matrix return the result vector
    def vectorMatrixMultiplication(self, b, A):
        res = []
        nRow = len(A)
        nCol = len(b)
        for i in range(nRow):
            sum = 0
            for j in range(nCol):
                sum += b[j] * A[i][j]
            res.append(sum)
        return res

    # Two vectors adding up return the result vector
    def vectorAddition(self, a, b):
        n = len(a)
        res = []
        for i in range(n):
            res.append(a[i] + b[i])
        return res

    # Two vectors subtracting return result vector
    def vectorSubtraction(self, a, b):
        n = len(a)
        res = []
        for i in range(n):
            res.append(a[i] - b[i])
        return res

    # Two vectors multiplying return result value
    def vectorMultiplication(self, a, b):
        nb = len(b)
        res = 0
        for i in range(nb):
            res += a[i] * b[i]
        return res

    # Two matrix subtraction return result matrix
    def matrixSubstract(self, A, B):
        res = []
        nRow = len(A)
        nCol = len(A[0])
        for i in range(nRow):
            row = []
            for j in range(nCol):
                row.append(A[i][j]-B[i][j])
            res.append(row)
        return res

    # 2x2 matrix inverse return the inverse of a 2x2 matrix A
    def matrixInverse(self, A):
        det = 1.0 / (A[0][0] * A[1][1] - A[1][0] * A[0][1])
        temp = A[0][0]
        A[0][0] = A[1][1] * det
        A[1][1] = temp * det
```

```
        A[1][0] *= -1.0 * det
        A[0][1] *= -1.0 * det

        return A
```