ECSE 543
NUMERICAL METHODS IN ELECTRICAL ENGINEERING

Report 3

Zhiyu Chen
260605624

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: BH Data for M19 Steel

Table 1

(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?

The Lagrange polynomials coefficients are found using:

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

Then, the approximated value of H is found using

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

```
Interpolate the first 6 points with full-domain Lagrange polynomials result is:
H = 414.062500000011*B**5 - 963.541666666693*B**4 + 873.437500000015*B**3 - 215.208333333339*B**2 + 88.6500000000007*B
```

Figure 1a (1): interpolating function with full-domain Lagrange

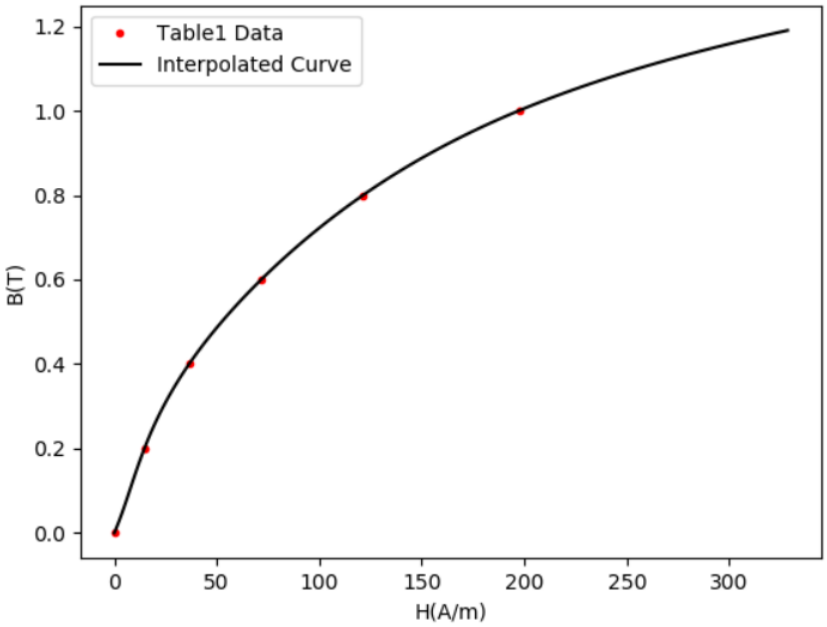


Figure 1a (2): interpolating with the first 6 points with full-domain Lagrange

The red dots are the points from the table 1 data, and the black curve is based on the result function of interpolation of the first 6 points using full-domain Lagrange polynomials. As we can see, this line is plausible, as the line is smooth with no sharp turns or squiggles, and it seems to fit the true B-H curve well.

(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?

```
Interpolate the selecting points with full-domain Lagrange polynomials result is:
H = 156393.280524088*B**5 - 966235.572245107*B**4 + 2253820.22115058*B**3 - 2337828.82945774*B**2 + 906781.85442208*B
```

Figure 1b (1): interpolating function with full-domain Lagrange

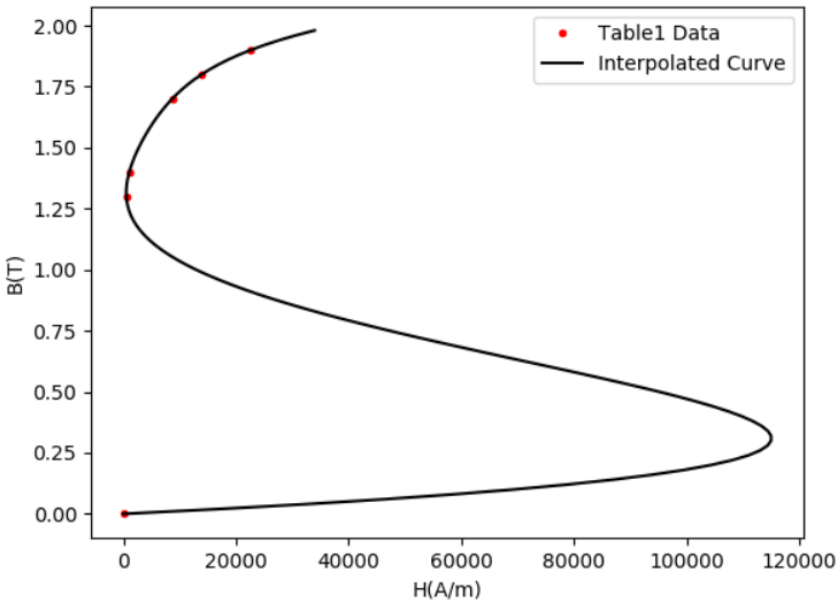


Figure 1b (2): interpolating with the given 6 points with full-domain Lagrange

The red dots are the points from the table 1 data, and the black curve is based on the result function of interpolation of the given 6 points using full-domain Lagrange polynomials. As we can see, this line is not plausible. It has some very strange U-turns, and as a result it seems not to fit the true B-H curve.

(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.

Using the cubic Hermite polynomial method:

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

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

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

Finally, the approximation answer is found using:

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

Where $a_j(x) = y(x_j)$, $b_j(x) = y'(x_j)$

Now we want to fix the remaining 6 degrees of freedom – the slopes at the 6 points.

In general case, to have a smooth curve, the slope has to be continuous through the result function. Consider two continuous subdomains -- subdomain 1 and subdomain 2, to describe subdomain 1, we must use both the y value and y' (first derivative) value of its start and end point, and the same for the subdomain 2. However, the y' of the end point of subdomain 1 must have the same value as the y' of the start point of subdomain 2, in order to make the slope continuous.

In the end, for this problem that has 6 points, the middle four points, apart from the two end points of the full domain, can all be constrained by this continuous slope condition. So, 4 degrees of freedom has been removed. To remove the last 2 degrees of freedom at the two end points of the full domain, we need to know their slopes by experiment.

In this problem, though, as there is no information about the derivative data, we have to approximate all the derivatives (except the derivative for the last end point x_6) using $y(x_{i+1}) - y(x_i) / (x_{i+1} - x_i)$, and for the last end point x_6, we approximate its derivative to be $y(x_i) - y(x_0) / (x_{i+1} - x_0)$. This means, if we define the 5 subdomains between the 6 points as A, B, C, D, E, respectively:

$$b_1^A(0) = (y(1.3) - y(0))/(1.3 - 0)$$
$$b_2^A(1.3) = b_1^B(1.3) = (y(1.4) - y(1.3))/(1.4 - 1.3)$$
$$b_2^B(1.4) = b_1^C(1.4) = (y(1.7) - y(1.4))/(1.7 - 1.4)$$
$$b_2^C(1.7) = b_1^D(1.7) = (y(1.8) - y(1.7))/(1.8 - 1.7)$$
$$b_2^D(1.8) = b_1^E(1.8) = (y(1.9) - y(1.8))/(1.9 - 1.8)$$
$$b_2^A(1.9) = (y(1.9) - y(0))/(1.9 - 0)$$

This finally cancels all degrees of freedom as all the $b_j(x) = y'(x_j)$ values are now fixed. Using the method discussed above, the test result is as below:

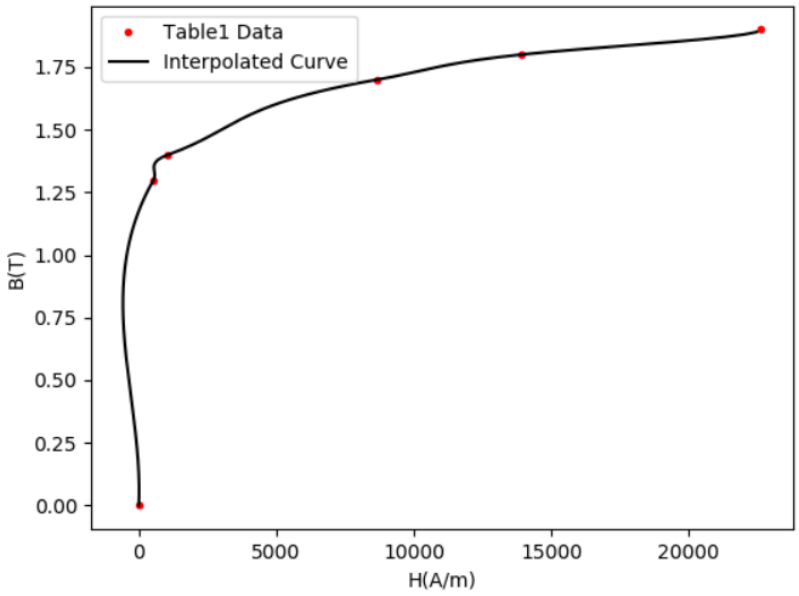


Figure 1c (1): interpolating with the given 6 given points with cubic Hermite

From Figure 1c (1), we can see that the result is much better than in the Figure 1b (2) and this curve seems plausible.

The drawback for this method is, since we use the subdomain with small ranges, and the result function is piecewise, as long as we go beyond B = 1.9T, H = 22650.2H/m threshold, the interpolate function predicts results very badly, so we have to stay in the region given in table 1.

(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 = 30 cm and L_a = 0.5 cm. The coil has N = 800 turns and carries a current I = 10 A. Derive a (nonlinear) equation for the flux ψ in the core, of the form f(ψ) = 0.

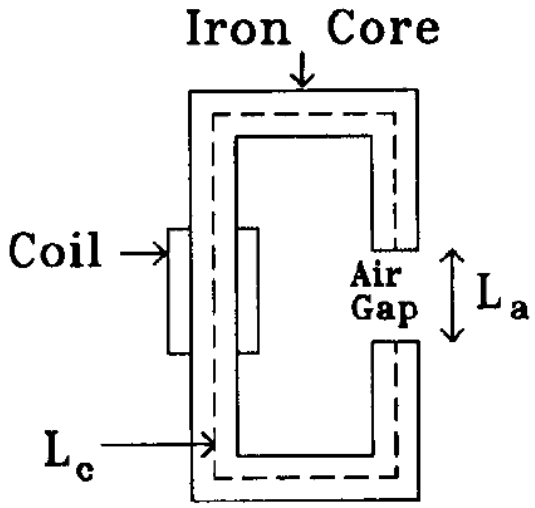


Figure 1

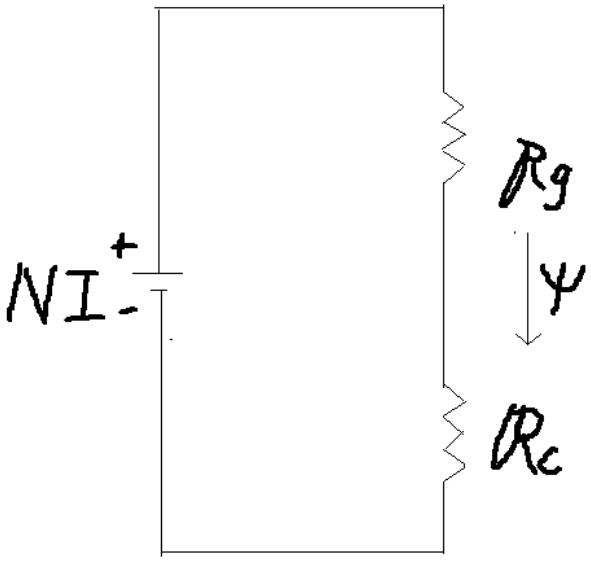


Figure 1d (1): equivalent magnetic circuit

As shown in figure 1d (1), I draw an equivalent magnetic circuit where:

R_G: reluctance of the air gap

R_C: reluctance of the steel core

NI: MMF produced by the coil winding

ψ: magnetic flux flowing through the circuit

By Ampere's Law, $NI = R_G * \psi + R_C * \psi$

Where $R_G = \frac{L_a}{\mu_G A}$, $R_C = \frac{L_C}{\mu_C A}$, where $\mu_G = \mu_0$, therefore:

$$NI = \frac{L_a}{\mu_0 A}\psi + \frac{L_C}{\mu_C A}\psi$$

Since by definition, $\psi = B*A = H*A*\mu_C$, we have then $\mu_C = \psi/(H*A)$

Then:

$$NI = \psi \frac{L_a}{\mu_0 A} + H * L_C$$

$$\psi \frac{L_a}{\mu_0 A} + H \cdot L_C - NI = 0$$

Divide both sides by $\frac{L_a}{\mu_0 A}$, we get

$$f(\psi) = \psi + (H \cdot L_C - NI) / \frac{L_a}{\mu_0 A} = 0$$

Plug in the values, we get $f(\psi) = \psi + (0.3H - 8000)/3.98 * 10^7 = 0$

(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.

The derivative df(ψ)/dψ = 1 + 0.3/3.98*10^7*dH/dψ, where, since dψ = dB*A, dH/dψ = dH/(dB*A) = (dH/dB)/A. And the cross-section area A is given and dH/dB can be computed from the Table 1 provided to us. (Considering we use a piecewise-linear interpolation, the slope at each interval is given by:
slope = (H(final) – H(start))/(B(final) – B(start))

Meanwhile, since H is a piecewise-linear function of B, as B can be computed with ψ/A, and dH/dB for each interval is known from the previous step, we can get H as well:
H = slope*(B - B(start)) + H(start)

Finally, to solve the nonlinear equation with Newton-Raphson, the original guess is ψ = 0.
As everything is set up, we solve for the

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

The result is as below, it takes 3 steps to get the final flux result: 1.6127*10^{-4} Wb.

```
Total iteration number: 3
The flux is: 0.00016126936944407854 Wb
```

Figure 1e (1): Newton-Raphson simulation result

(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, we have

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

Which means:

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

The method **does not converge**, and the loop never ends. A way to modify the method is to use the inverse of H(ψ):

Since: $R_G\psi + H(B_\psi) \cdot L_C - NI = 0$, We can get $H(B_\psi) = (NI - \frac{L_a}{\mu_G A} * \psi)/L_C$, which

means $B_\psi = H^{-1}((NI - \frac{L_a}{\mu_G A} * \psi)/L_C)$

Since $B_\psi = \psi/A$, we get finally $\psi - A * H^{-1}((NI - \frac{L_a}{\mu_G A} * \psi)/L_C) = 0$, where H^{-1} means the inverse function of H(B). As how to find H from B from table 1 has been discussed in part e, consider now the value of H is known to be $(NI - \frac{L_a}{\mu_G A} * \psi)/L_C$. Using the same strategy in part e, we can also find the B-H slope to be:
slope = (B(final) – B(start))/(H(final) – H(start)) for each interval that contains H, and with this information: B = slope*(H - H(start)) + B(start)

So up to now we have

$$f(\psi) = \psi - A * H^{-1}((NI - \frac{L_a}{\mu_G A} * \psi)/L_c) = 0$$ with H^{-1} known.

This function converges and finally, we get a very similar result as in part e: 15 steps to $1.6127*10^{-4}$ Wb.

```
Total iteration number: 15
The flux is: 0.00016126939623324393 Wb
```

Figure 1f (1): successive substitution simulation result

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

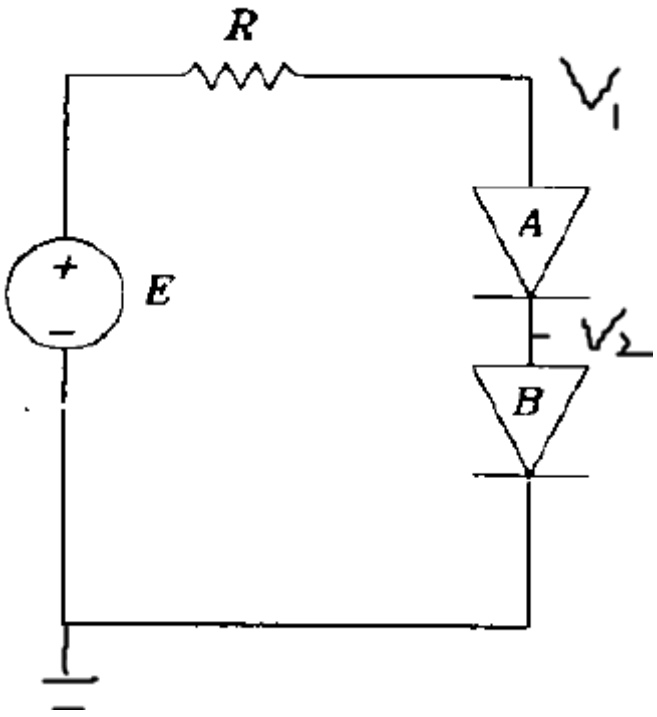


Figure 2

(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.

For a diode, the current flows through it is given by:

$$I = I_s\left(e^{\frac{qV}{kT}} - 1\right)$$

By KCL, the current flows through diode A and B should be the same, so:

$$f_1(v_n) = I_{sA}\left(e^{\frac{q(V_1 - V_2)}{kT}} - 1\right) - I_{sB}\left(e^{\frac{qV_2}{kT}} - 1\right) = 0$$

By KVL we have

$$E - RI - V_1 = 0$$

$$f_2(v_n) = V_1 - E + R \cdot I_{sB}\left(e^{\frac{qV_2}{kT}} - 1\right) = 0$$

(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 ε_k]

Combine f_1 and f_2 we get a matrix:

$$\bar{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} I_{sA}\left(e^{\frac{q(V_1 - V_2)}{kT}} - 1\right) - I_{sB}\left(e^{\frac{qV_2}{kT}} - 1\right) \\ V_1 - E + R \cdot I_{sB}\left(e^{\frac{qV_2}{kT}} - 1\right) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

And the Jacobian matrix is given by:

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

Then we can compute:

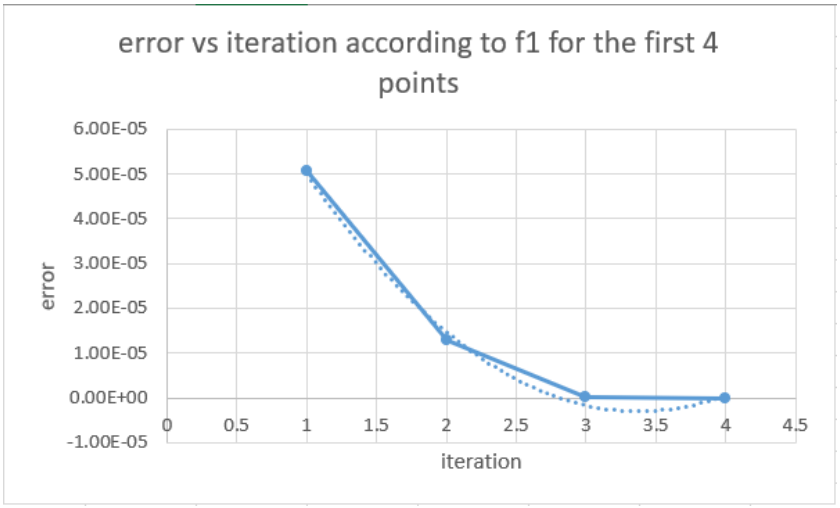
$F * \left[V_n^{(k+1)} - V_n^k\right] + f^k = 0$, where $v_n = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$.

Setting the ε_k **to be** $(f_1(V_1, V_2) / f_1(0, 0)) < 10^{-8}$ for the simulation break condition (if we use $(f_2(V_1, V_2) / f_2(0, 0))$, since $f_2(0, 0) = 0$, it will return an error since the denominator cannot be 0), and the result of f and v is:

```
number of iteration: 0
f1 = 0.0; f2 = 0.2
V1 = 0V; V2 = 0V
number of iteration: 1
f1 = 5.050998579338411e-05; f2 = 0.01579271414916271
V1 = 0.20000370533180406V; V2 = 0.08421208645549644V
number of iteration: 2
f1 = 1.2956737685586772e-05; f2 = 0.00019767709661978536
V1 = 0.18421060350836177V; V2 = 0.08390862709816298V
number of iteration: 3
f1 = 7.06393142093373le-08; f2 = 0.0031097955346614213
V1 = 0.18441382636976542V; V2 = 0.08830269354304884V
number of iteration: 4
f1 = 4.68879962359567e-09; f2 = 0.0011439816695473816
V1 = 0.18130173215889248V; V2 = 0.08677495082187685V
number of iteration: 5
f1 = 2.999696490008953e-11; f2 = 0.00041986385101220117
V1 = 0.18244652297263533V; V2 = 0.08734661101822984V
number of iteration: 6
f1 = 1.0283112482725211e-11; f2 = 0.00015478405334843834
V1 = 0.18202635613650694V; V2 = 0.0871373913200817V
number of iteration: 7
f1 = 1.4105409819765297e-12; f2 = 5.690777604385952e-05
V1 = 0.18218125095049956V; V2 = 0.08721451740171722V
number of iteration: 8
f1 = 1.9604234685263733e-13; f2 = 2.0942139370396656e-05
V1 = 0.1821243023284623V; V2 = 0.08718616267666024V
number of iteration: 9
f1 = 2.6276872930157402e-14; f2 = 7.70410322689652e-06
V1 = 0.18214525948244997V; V2 = 0.08719659744509944V
number of iteration: 10
f1 = 3.5699524933430204e-15; f2 = 2.834507959265098e-06
V1 = 0.18213754985344305V; V2 = 0.08719275877427728V
number of iteration: 11
f1 = 4.8257161070972e-16; f2 = 1.0428292294174801e-06
V1 = 0.18214038639415053V; V2 = 0.08719417110872012V
```

Figure 2b (1): Newton-Raphson result

Now since f_1 and f_2 is supposed to be 0, their recorded values are actually the errors, and if we plot the f_1 and f_2 from iteration n = 1 we get the following results:



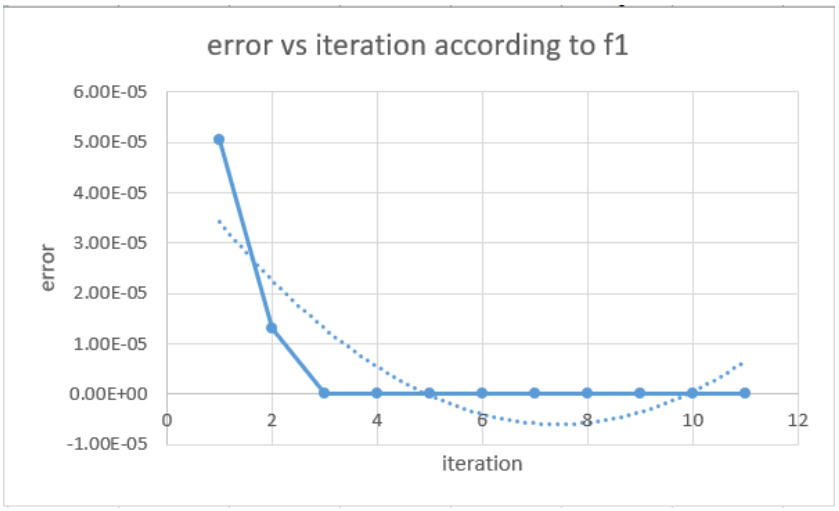
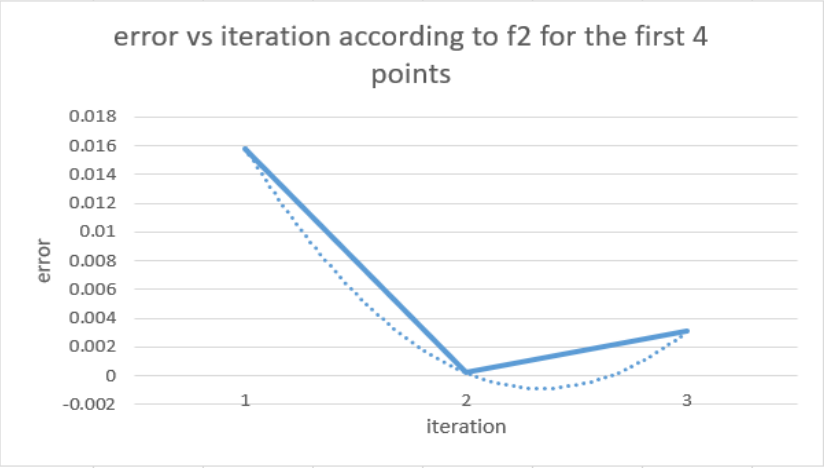


Figure 2b (2): error according to f_1 with ① 4 points ② the whole iteration

For these plots I added a quadratic trend line and we can see from the error according to f_1 that the first several iterations give us a **quadratic convergence**, and after that the function still converges to 0, but in a much slower speed.



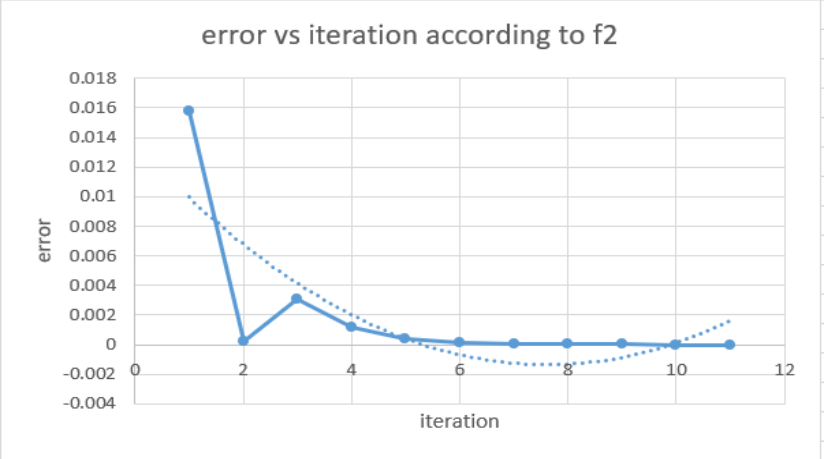


Figure 2b (3): error according to f_2 with ① 4 points ② the whole iteration

The arguments are the same as above, we can see from the error according to f_2 that the first several iterations seem a **quadratic convergence**, and after that the function converges to 0, but in a much slower speed.

3. **Write a program that accepts as input the values for the parameters x_0, x_N, and N and integrates a function f(x) on the interval $x = x_0$ to $x = x_N$ by dividing the interval into N equal segments and using one-point Gauss-Legendre integration for each segment.**

For an integral $\int_a^b f(x)\ dx$, in Gauss-Legendre integration method, we can write it as

$$\int_a^b f(x)\ dx\ =\ \sum_{i=0}^{n} w_i * \mathrm{f}(x_i)$$

Where w_i is the weight, and the $f(x_i)$ is the value of the function evaluated at x_i. In the case that we divide the interval into N equal segments, each region has a length:
h = (b - a)/N, therefore for any given region:
$[x_{start}, x_{start} + h]$, we can have x_i to be $x_{start} + 1/2*h$, which means it is the mid-point of the region. Meanwhile, as the interval is equally distributed, we should have:
$w_i = h$.

(a) Use your program to integrate the function f(x) = sin(x) on the interval $x_0 = 0$ to $x_N = 1$ for N = 1, 2, …, 20. Plot $\log_{10}(E)$ versus $\log_{10}(N)$ for N=1,2, …,20, where E is the absolute error in the computed integral. Comment on the result.

First of all, the actual integral is:

$$\int_0^1 \sin(x)\,dx = -\cos(1) - (-\cos(0)) = 0.45969769413$$

Then using the one-point Gauss-Legendre integration to compute:

```
integral of sin(x), where the actual value is 0.45969769413:

N = 1  the Gauss-Legendre solution: 0.479425538604203  the error is: 0.019727844474203005
N = 2  the Gauss-Legendre solution: 0.46452135963892854  the error is: 0.004823665508928543
N = 3  the Gauss-Legendre solution: 0.46183284149788495  the error is: 0.0021351473678849486
N = 4  the Gauss-Legendre solution: 0.46089700941194117  the error is: 0.0011993152819411712
N = 5  the Gauss-Legendre solution: 0.46046475175550916  the error is: 0.0007670576255091599
N = 6  the Gauss-Legendre solution: 0.4602301830292006  the error is: 0.0005324888992006005
N = 7  the Gauss-Legendre solution: 0.4600888263350858  the error is: 0.00039113220508579793
N = 8  the Gauss-Legendre solution: 0.459997112932708  the error is: 0.00029941880270800 69
N = 9  the Gauss-Legendre solution: 0.4599342493155144  the error is: 0.0002365551855144088
N = 10  the Gauss-Legendre solution: 0.45988929071851814  the error is: 0.0001915965885181392
N = 11  the Gauss-Legendre solution: 0.4598560304014713  the error is: 0.00015833627147132656
N = 12  the Gauss-Legendre solution: 0.45983073545745196  the error is: 0.00013304132745195485
N = 13  the Gauss-Legendre solution: 0.4598110513909568  the error is: 0.00011335726095679233
N = 14  the Gauss-Legendre solution: 0.4597954335261477  the error is: 9.773939614771132e-05
N = 15  the Gauss-Legendre solution: 0.4597828343710374  the error is: 8.514024103739581e-05
N = 16  the Gauss-Legendre solution: 0.45977252324545687  the error is: 7.482911545686477e-05
N = 17  the Gauss-Legendre solution: 0.45976397788219286  the error is: 6.628375219286387e-05
N = 18  the Gauss-Legendre solution: 0.4597568169559022  the error is: 5.912282590220252 5e-05
N = 19  the Gauss-Legendre solution: 0.4597507567862578  the error is: 5.306265627818475e-05
N = 20  the Gauss-Legendre solution: 0.45974558280018996  the error is: 4.788867018995502e-05
```

Figure 3a (1): Gauss-Legendre evaluation for sin(x)

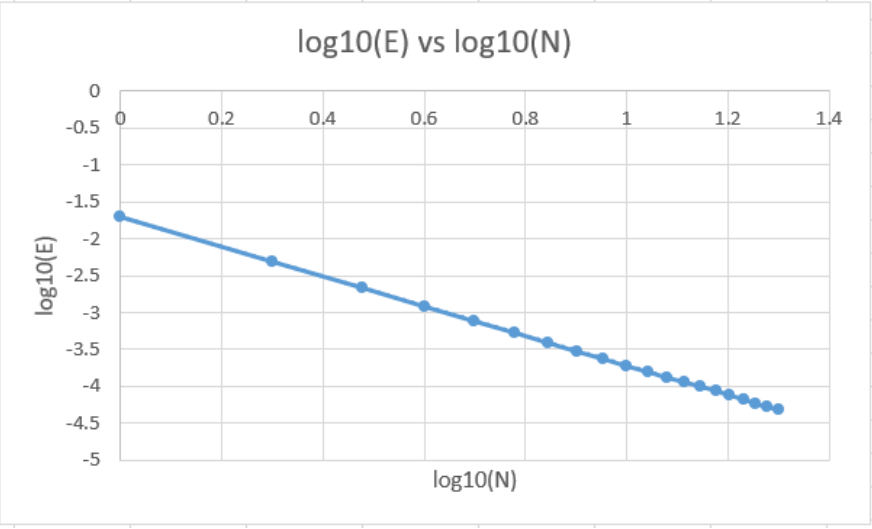


Figure 3a (2): $\log_{10}(E)$ versus $\log_{10}(N)$

We can see that, not only the error decreases as N increases, but also when taking 1og on both E and N with base 10, the $\log_{10}(E)$ decreases **linearly** with respect to $\log_{10}(N)$. This can mean that there are diminishing returns when using a very high value of N

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

First of all, the actual integral is:

$$\int_0^1 \ln(x)\,dx = x \cdot \ln(x) - x = -1$$

Then using the one-point Gauss-Legendre integration to compute:

```
integral of ln(x), where the actual value is -1:

N = 10  the Gauss-Legendre solution: -0.9657590653461392  the error is: 0.0342409346538608
N = 20  the Gauss-Legendre solution: -0.9827754719736861  the error is: 0.017224528026313912
N = 30  the Gauss-Legendre solution: -0.9884938402873319  the error is: 0.011506159712668107
N = 40  the Gauss-Legendre solution: -0.9913617009604189  the error is: 0.008638299039581132
N = 50  the Gauss-Legendre solution: -0.9930851944722273  the error is: 0.006914805527772683
N = 60  the Gauss-Legendre solution: -0.994235347381881  the error is: 0.005764652618118982
N = 70  the Gauss-Legendre solution: -0.9950574520104228  the error is: 0.004942547989577162
N = 80  the Gauss-Legendre solution: -0.9956743404788306  the error is: 0.004325659521169367
N = 90  the Gauss-Legendre solution: -0.9961543263261007  the error is: 0.003845673673899319
N = 100  the Gauss-Legendre solution: -0.9965384307395617  the error is: 0.0034615692604382797
N = 110  the Gauss-Legendre solution: -0.9968527745070253  the error is: 0.003147225492974748
N = 120  the Gauss-Legendre solution: -0.9971147802544645  the error is: 0.002885219745535461
N = 130  the Gauss-Legendre solution: -0.9973365147802647  the error is: 0.002663485219735251
N = 140  the Gauss-Legendre solution: -0.9975266001991578  the error is: 0.0024733998008421576
N = 150  the Gauss-Legendre solution: -0.9976913612451838  the error is: 0.002308638754816239
N = 160  the Gauss-Legendre solution: -0.9978355426612094  the error is: 0.0021644573387905597
N = 170  the Gauss-Legendre solution: -0.9979627735721452  the error is: 0.0020372264278547547
N = 180  the Gauss-Legendre solution: -0.9980758771710281  the error is: 0.0019241228289719192
N = 190  the Gauss-Legendre solution: -0.9981770826716362  the error is: 0.0018229173283638156
N = 200  the Gauss-Legendre solution: -0.9982681737137472  the error is: 0.0017318262862527911
```

Figure 3b (1): Gauss-Legendre evaluation for ln(x)

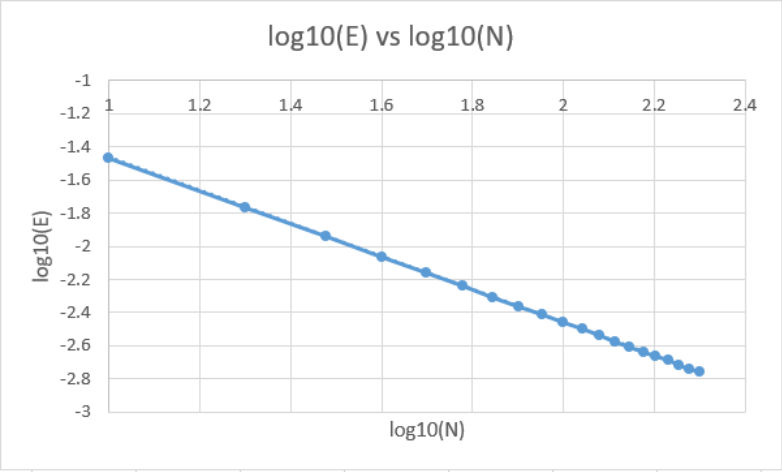


Figure 3b (2): $\log_{10}$(E) versus $\log_{10}$(N)

The same as part a), we can see that, not only the error decreases as N increases, but also when taking 1og on both E and N with base 10, the $\log_{10}$(E) decreases **linearly** with respect to $\log_{10}$(N) (This can mean that there are diminishing returns when using a very high value of N).

(c) Repeat part (b) for the function f(x) = ln(0.2|sin(x)|). Comment on the result.
First of all, the actual integral is:

$$\int_0^1 \ln(0.2|\sin(x)|)\,dx = -2.66616$$

Then using the one-point Gauss-Legendre integration to compute:

```
integral of ln(0.2|sin(x)|, where the actual value is -2.66616:

N = 10  the Gauss-Legendre solution: -2.6317680783273416  the error is: 0.034391921672658476
N = 20  the Gauss-Legendre solution: -2.6488963097954876  the error is: 0.01726369020451246
N = 30  the Gauss-Legendre solution: -2.6546353892063084  the error is: 0.011524610793691714
N = 40  the Gauss-Legendre solution: -2.6575104989697795  the error is: 0.008649501030220552
N = 50  the Gauss-Legendre solution: -2.659237347811081  the error is: 0.006922652188919187
N = 60  the Gauss-Legendre solution: -2.6603893233784652  the error is: 0.005770676621534854
N = 70  the Gauss-Legendre solution: -2.6612125270154032  the error is: 0.004947472984596857
N = 80  the Gauss-Legendre solution: -2.6618301287838477  the error is: 0.004329871216152359
N = 90  the Gauss-Legendre solution: -2.662310603667872  the error is: 0.0038493963321282187
N = 100  the Gauss-Legendre solution: -2.662695057886746  the error is: 0.00346494211325421
N = 110  the Gauss-Legendre solution: -2.6630096604708147  the error is: 0.0031503395291854197
N = 120  the Gauss-Legendre solution: -2.6632718630695975  the error is: 0.002888136930402574
N = 130  the Gauss-Legendre solution: -2.663493750792088  the error is: 0.002666249207912141 4
N = 140  the Gauss-Legendre solution: -2.6636839577678995  the error is: 0.002476042232100628
N = 150  the Gauss-Legendre solution: -2.6638488168798635  the error is: 0.002311183120136562
N = 160  the Gauss-Legendre solution: -2.663993078555689  the error is: 0.0021669214443109652
N = 170  the Gauss-Legendre solution: -2.6641203759839933  the error is: 0.002039624016006769
N = 180  the Gauss-Legendre solution: -2.664233535325086  the error is: 0.0019264646749141967
N = 190  the Gauss-Legendre solution: -2.6643347880003305  the error is: 0.0018252119996695626
N = 200  the Gauss-Legendre solution: -2.6644259193193833  the error is: 0.0017340806806167564
```

Figure 3c (1): Gauss-Legendre evaluation for ln(0.2|sin(x)|)

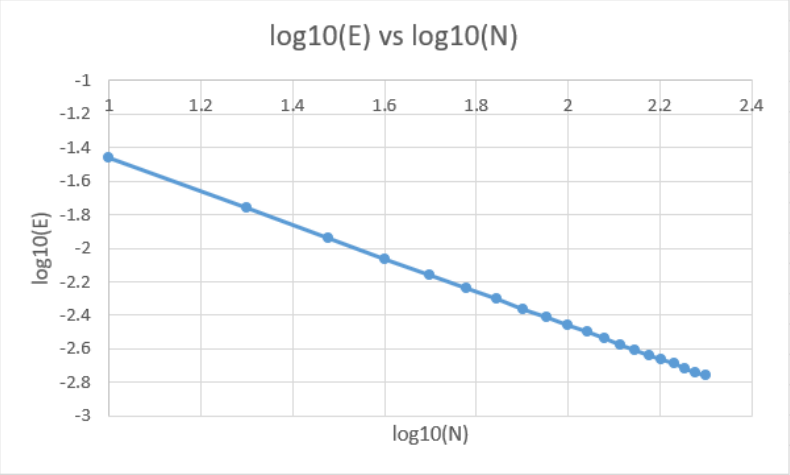


Figure 3c (2): $\log_{10}$(E) versus $\log_{10}$(N)

The same as part a) and b), we can see that, not only the error decreases as N increases, but also when taking 1og on both E and N with base 10, the $\log_{10}$(E) decreases **linearly** with respect to $\log_{10}$(N) (This can mean that there are diminishing returns when using a very high value of N).

(d) 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 f(x) in part (b) and (c) using only 10 segments. Comment on the results.

Since for b) and c), ln is a function that has a derivative of 1/x, which means its slope decreases when x increases(|sin(x)| increases with x when x is in region [0, 1] so ln(x) and ln(|sin(x)|) should have the same behavior in region [0, 1]). So, the 'difficult' part should lie in the region that x is close to 0, where there is a sharp increase of ln function and therefore we needs more point to sample.

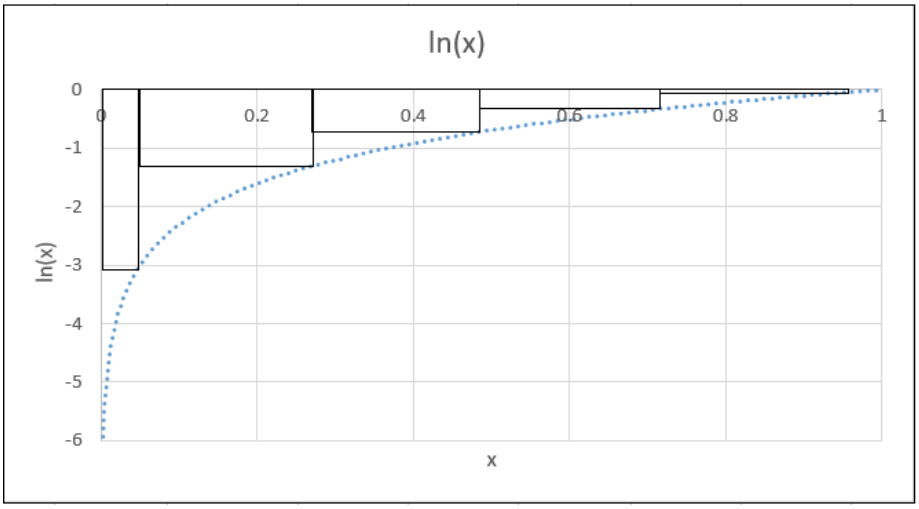


Figure 4d (1): an example of how the intervals can be arranged (take N = 5 as an example), we can see this allies with the slope change of ln function and should give a more accurate result

So I generates some intervals with increasing interval sizes as x increases, and the code can be found in **gaussLegendre.py**(see Appendix). The basic idea is to have the interval size h = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] *h_s, where h_s is the smallest interval size.

The result is as below:

```
integral of ln(x), where the actual value is -1:

N = 10  the Gauss-Legendre solution: -0.9883774366311462  the error is: 0.011622563368853811

integral of ln(0.2|sin(x)|, where the actual value is -2.66616:

N = 10  the Gauss-Legendre solution: -2.6542551627711006  the error is: 0.011904837228899456
```

Figure 4d (2): Gauss-Legendre evaluation for ln(x) and ln(0.2|sin(x)|) with unequal interval

As we can see, the error of ln(x) integral from 0 to 1 reduces from 0.03424 **(from Figure 3b (1))** to 0.01162, while the error of ln(0.2|sin(x)|) integral from 0 to 1 reduces from 0.03439**(from Figure 3c (1))** to 0.0119, both decreases dramatically.

Appendix：

Question 1:

1. question1.py:

```
from sympy import *
import numpy
from interpolate import *
from M19 import *
import matplotlib.pyplot as plt


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]
Bb = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
Hb = [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]

b_a = numpy.arange(0, 1.2, 0.01)
h_a = [0 for i in range(len(b_a))]

b_b = numpy.arange(0, 2.0, 0.02)
h_b = [0 for i in range(len(b_b))]

b_c = numpy.arange(0, 1.9, 0.01)
h_c = [0 for i in range(len(b_c))]

variable = Symbol('B')
problem_a = Lagrange(variable, B[:6], H[:6])
problem_b = Lagrange(variable, Bb, Hb)
problem_c = cubicHermite(variable, Bb, Hb)

print("Interpolate the first 6 points with full-domain Lagrange polynomials result is: \n"
+ "H = " + str(problem_a) + "\n")
print("Interpolate the selecting points with full-domain Lagrange polynomials result is:
\n" + "H = " + str(problem_b) + "\n")

#plotting block
for i in range(len(b_a)):
      h_a[i] = problem_a.subs({variable: b_a[i]})
pa = plt.figure(1)
plot(b_a, B[:6], h_a, H[:6])

for i in range(len(b_b)):
```

```
        h_b[i] = problem_b.subs({variable: b_b[i]})
pb = plt.figure(2)
plot(b_b, Bb, h_b, Hb)

for i in range(len(b_c)):
        for j in range(5):
                if Bb[j] <= b_c[i] < Bb[j + 1]:
                        h_c[i] = problem_c[j].subs({variable: b_c[i]})
                        break
pb = plt.figure(3)
plot(b_c, Bb, h_c, Hb)

plt.show()

(phi, counter) = NewtonRaphson(B, H)
print("Total iteration number: " + str(counter))
print("The flux is: " + str(phi) + " Wb")

(phi2, counter2) = successivesub(B, H)
print("Total iteration number: " + str(counter2))
print("The flux is: " + str(phi2) + " Wb")
```

2. interpolate.py:

```
import matplotlib.pyplot as plt

def Lagrange(variable, xPoints, yPoints):
        l = len(xPoints)
        L = [None for i in range(l)]
        yf = 0
        for i in range(l):
                F1 = 1
                F2 = 1
                for j in range(l):
                        if j != i:
                                F1 *= (variable - xPoints[j])
                                F2 *= (xPoints[i] - xPoints[j])
                L[i] = F1/F2
                yf += yPoints[i] * L[i]
        yf = yf.expand()
        return yf

def cubicHermite(variable, xPoints, yPoints):
        l = len(xPoints)
        yf = [0 for i in range(l - 1)]
```

```
    for i in range(l - 1):
        ydiff1 = (yPoints[i + 1] - yPoints[i])/(xPoints[i + 1] - xPoints[i])
        if i == l - 2:
            ydiff2 = yPoints[i + 1] / xPoints[i]
        else:
            ydiff2 = (yPoints[i + 2] - yPoints[i + 1]) / (xPoints[i + 2] - xPoints[i +
            1])
        U1 = (1 - 2 * (variable - xPoints[i])/(xPoints[i] - xPoints[i + 1]))*((variable -
        xPoints[i + 1])/(xPoints[i] - xPoints[i + 1]))**2
        U2 = (1 - 2 * (variable - xPoints[i + 1])/(xPoints[i + 1] -
        xPoints[i]))*((variable - xPoints[i])/(xPoints[i + 1] - xPoints[i]))**2
        V1 = (variable - xPoints[i])*((variable - xPoints[i + 1])/(xPoints[i] - xPoints[i
        + 1]))**2
        V2 = (variable - xPoints[i + 1])*((variable - xPoints[i])/(xPoints[i + 1] -
        xPoints[i]))**2
        yf[i] = (yPoints[i]*U1 + yPoints[i + 1]*U2 + ydiff1*V1 +
        ydiff2*V2).expand()

    return yf


def plot(b, B, h, H):
    plt.plot(H, B, "r.", label="Table1 Data")
    plt.plot(h, b, "k", label="Interpolated Curve")
    plt.xlabel("H(A/m)")
    plt.ylabel("B(T)")
    plt.legend()
```

3. M19.py:

```
import math
A = 1/100**2
Lc = 30/100
La = 0.5/100
u0 = 4*math.pi/10**7
N = 800
I = 10
mmf = N*I
k = La/(u0*A)
def getbandh(phi, B, H):
    b = phi/A
    h = 0
    h_derivative = 0
    for i in range(len(B) - 1):
        if B[i] <= b < B[i + 1]:
```

```
                h_derivative = (H[i + 1] - H[i])/((B[i + 1] - B[i]) * A)
                h = H[i] + (b - B[i]) * ((H[i + 1] - H[i])/(B[i + 1] - B[i]))
                break
        elif b > B[len(B) - 1]:
                h_derivative = (H[len(B) - 1] - H[len(B) - 2]) / ((B[len(B) - 1] - B[len(B)
                - 2]) * A)
                h = H[len(B) - 1] + (b - B[len(B) - 1]) * ((H[len(B) - 1] - H[len(B) - 2])
                / (B[len(B) - 1] - B[len(B) - 2]))
                break
    return h, h_derivative
def getInverseH(phi, B, H):
    h = (mmf - k*phi)/Lc
    b = 0
    for i in range(len(H) - 1):
        if H[i] <= h < H[i + 1]:
            b = B[i] + (h - H[i]) * ((B[i + 1] - B[i])/(H[i + 1] - H[i]))
            break
        elif h > H[len(B) - 1]:
            b = B[len(B) - 1] + (h - H[len(B) - 1]) * ((B[len(B) - 1] - B[len(B) -
            2])/(H[len(B) - 1] - H[len(B) - 2]))
            break
    return b
def fphi(phi, h):
    return phi + (Lc * h - mmf)/k
def fphiderivative(h_derivative):
    return 1 + Lc * h_derivative/k
def NewtonRaphson(B, H):
    phi = 0
    (h0, h_derivative0) = getbandh(0, B, H)
    counter = 0
    h = h0
    h_derivative = h_derivative0
    while abs(fphi(phi, h)/fphi(0, h0)) >= 10**(-6):
        phi = -(fphi(phi, h)/fphiderivative(h_derivative)) + phi
        (h, h_derivative) = getbandh(phi, B, H)
        counter = counter + 1
    return phi, counter

#this method does not converge
def successivesub_initial(B, H):
    phi = 0
    (h0, h_derivative0) = getbandh(0, B, H)
    counter = 0
    h = h0
```

```
    while abs(fphi(phi, h)/fphi(0, h0)) >= 10**(-6):
        phi = -fphi(phi, h) + phi
        (h, h_derivative) = getbandh(phi, B, H)
        counter = counter + 1
    return phi, counter

def successivesub(B, H):
    phi = 0
    (h0, h_derivative0) = getbandh(0, B, H)
    counter = 0
    h = h0
    while abs(fphi(phi, h)/fphi(0, h0)) >= 10**(-6):
        phi = A*getInverseH(phi, B, H)
        (h, h_derivative) = getbandh(phi, B, H)
        counter = counter + 1
    return phi, counter
```

Question 2:

1. NewtonRaphson.py:

```
from methods import *
import numpy as np
import math
E = 0.2
R = 512
Vt = 25 * 10**(-3)
Isa = 0.8 * 10**(-6)
Isb = 1.1 * 10**(-6)
k = 0
counter = 0


def newtonRap (V1, V2):
    f1 = Isa * (math.exp((V1 - V2) / Vt) - 1) - Isb * (math.exp(V2 / Vt) - 1)
    f2 = V1- E + R * Isb * (math.exp(V2 / Vt) - 1)
    f1V1Partial = Isa/Vt * math.exp((V1 - V2) / Vt)
    f1V2Partial = -Isa/Vt * math.exp((V1 - V2) / Vt) - Isb/Vt * math.exp(V2 / Vt)
    f2V1Partial = 1
    f2V2Partial = -Isb/Vt * math.exp(V2 / Vt)
    V = [V1, V2]
    f = [f1, f2]
    # F is the Jacobian Matrix
    F = [[f1V1Partial, f1V2Partial], [f2V1Partial, f2V2Partial]]
    invF = np.linalg.inv(F)
```

```
    V = matrixAddOrSub(scalarmultiplier(-1, multiplyMatrix(invF, f)), V,'add')
    f1_final = Isa * (math.exp((V[0][0] - V[1][0]) / Vt) - 1) - Isb * (math.exp(V[1][0]
/ Vt) - 1)
    f2_final = V[0][0] - E + R * Isb * (math.exp(V[1][0] / Vt) - 1)
    return f1_final, f2_final, V

# initial guess

V1 = 0
V2 = 0
f1_initial = Isa * (math.exp((V1 - V2) / Vt) - 1) - Isb * (math.exp(V2 / Vt) - 1)
f2_initial = V1 - E + R * Isb * (math.exp(V2 / Vt) - 1)
print("number of iteration: " + str(counter))
print("f1 = " + str(abs(f1_initial)) + "; " + "f2 = " + str(abs(f2_initial)))
print("V1 = " + str(V1) + "V" + "; " + "V2 = " + str(V2) + "V")
while abs((V1 - E + R * Isb * (math.exp(V2 / Vt) - 1))/(0 - E + R * Isb * (math.exp(0 /
Vt) - 1))) >= 10**(-5):
    counter = counter + 1
    (f1, f2, V) = newtonRap(V1, V2)
    V1 = V[0][0]
    V2 = V[1][0]
    print("number of iteration: " + str(counter))
    print("f1 = " + str(abs(f1)) + "; " + "f2 = " + str(abs(f2)))
    print("V1 = " + str(V1) + "V" + "; " + "V2 = " + str(V2) + "V")
```

2. methods.py:

```
import math
from scipy import random
import csv
# Function to check the number of columns of a matrix
def numColumnCheck (A):
    numOfColumuns = 0
    try:
        numOfColumuns = len(A[0])
        return A
    except TypeError:
        B = [[0] for a in range(len(A))]
        for i in range(0, len(A)):
            B[i][0] = A[i]
        return B

# Function to multiply a scalar and a matrix
def scalarmultiplier(a, A):
    A = numColumnCheck(A)
```

```
    B = [[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])):
            B[i][j] = a*A[i][j]
    return B

# Function to multiply two matrices
def multiplyMatrix (A, B):
    A = numColumnCheck(A)
    B = numColumnCheck(B)
    if len(A[0]) == len(B):
        C = [[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])):
                    C[i][j] += A[i][k]*B[k][j]
        return C
    else:
        print('cannot multiply this two matrices, incorrect dimensions')


# Function to transpose a matrix
def transposeMatrix (A):
    numOfRows = len(A)
    numOfColumns = len(A[0])
    C = [[0 for i in range(numOfRows)]for k in range(numOfColumns)]
    for i in range(numOfRows):
        for j in range(numOfColumns):
            C[j][i] = A[i][j]
    return C

# Function to create a symmetric matrix
def symmetricMatrix(size, n):
    A = [[0 for i in range(size)] for k in range(size)]
    # assign the lower part of A a value
    for i in range(len(A)):
        for j in range(0, i + 1):
            A[i][j] = n * random.random() - n
    B = transposeMatrix (A)
    C = multiplyMatrix (A, B)
    return C


# Function to use the choleski decomposition to find L and y
```

```
def choleski(A, b, halfBandwidth=None):
    A = numColumnCheck(A)
    b = numColumnCheck(b)
    if len(b[0])!= 1:
        print('invalid b input')
        return
    try:
        numOfColumuns = len(A[0])
    except TypeError:
        print('A only has one column')
        return
    if len(A) != len(A[0]):
        print('A is not a nxn matrix')
        return
    size = len(A)
    for j in range (size):
        if A[j][j] < 0:
            print("the matrix A is not positive definite")
            return
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0]/A[j][j]
        for i in range (j+1, size):
            if halfBandwidth and i >= j + halfBandwidth:
                break
            A[i][j] = A[i][j]/A[j][j]
            b[i][0] = b[i][0]-A[i][j]*b[j][0]
            for k in range (j+1, i+1):
                if halfBandwidth and k >= j + halfBandwidth:
                    break
                A[i][k] = A[i][k]-A[i][j]*A[k][j]
    return [b,A]

# Function to find the solution through backward elimination, notice here L should be
a lower matrix
def backwardElim(y, L):
    y = numColumnCheck(y)
    L = numColumnCheck(L)
    x = [0 for a in range(len(y))]
    for i in range(len(L)-1, -1, -1):
        for j in range(len(L)-1, i, -1):
            y[i][0] = y[i][0] - L[j][i]*x[j]
        x[i] = y[i][0] / L[i][i]
    return x
```

```
def matrixAddOrSub(A, B, option):
    A = numColumnCheck(A)
    B = numColumnCheck(B)
    if len(A)!= len(B) or len(A[0])!= len(B[0]):
        print('cannot add or subtract two matrices with different sizes!')
        return
    C = [[0 for a in range(len(A[0]))] for b in range(len(A))]
    if option == 'add':
        for i in range(0, len(A)):
            for j in range(0, len(A[0])):
                C[i][j] = A[i][j] + B[i][j]
    elif option == 'sub':
        for i in range(0, len(A)):
            for j in range(0, len(A[0])):
                C[i][j] = A[i][j] - B[i][j]
    return C


def getCircuit(r):
    with open('test_circuit.csv')as circuitData:
        reader = csv.reader(circuitData)
        for n in reader:
            if (n[0].startswith('#')):
                cirNumber = int(n[0].replace('#', ''))
                if cirNumber == r:
                    A_pre = n[1].split(';')
                    J_pre = n[2].split(';')
                    R_pre = n[3].split(';')
                    E_pre = n[4].split(';')
                    A = [0 for i in range(len(A_pre))]
                    for i in range(len(A)):
                        rowA_pre = A_pre[i].split(',')
                        rowA = []
                        for j in range(len(rowA_pre)):
                            rowA.append(int(rowA_pre[j]))
                        A[i] = rowA
                    J, E = [], []
                    y = [[0 for a in range(len(R_pre))] for b in range(len(R_pre))]
                    for i in range(len(J_pre)):
                        J.append(int(J_pre[i]))
                        E.append(int(E_pre[i]))
                        y[i][i] = 1/int(R_pre[i])
                    return [A, J, y, E]
```

```
def solveCircuitProblem(A,J, y, E, halfBandwidth=None):
    A = numColumnCheck(A)
    J = numColumnCheck(J)
    y = numColumnCheck(y)
    E = numColumnCheck(E)
    A_final = multiplyMatrix(A, multiplyMatrix (y, transposeMatrix(A)))
    b_final = multiplyMatrix(A,    matrixAddOrSub(J, multiplyMatrix(y, E), 'sub'))
    choleskiOutput = choleski(A_final, b_final, halfBandwidth)
    voltage = backwardElim(choleskiOutput[0], choleskiOutput[1])
    return voltage
```

Question 3:

1. gaussLegendre.py:

```
import math


def gaussLegendre(start, end, function, N, h = None):
    if h == None:
        h = [(end - start)/N for i in range(N)]
    integral_sum = 0
    xstart = start
    xend = start
    for i in range(0, N):
        xend = xend + h[i]
        xi = (xstart + xend)/2
        integral_sum += h[i] * function(xi)
        xstart = xstart + h[i]
    return integral_sum


start = 0
end = 1
# integral of sin(x)
actual1 = 0.45969769413
print("integral of sin(x), where the actual value is " + str(actual1) + ":\n")
function1 = math.sin
for N in range(1, 21):
    gaussLegendreResult = gaussLegendre(start, end, function1, N)
    print("N = " + str(N) + "    the Gauss-Legendre solution: " +
str(gaussLegendreResult) + "    the error is: " + str(abs(gaussLegendreResult - actual1)))
print("\n")
# integral of ln(x)
actual2 = -1
```

```
print("integral of ln(x), where the actual value is " + str(actual2) + ":\n")
function2 = math.log
for N in range(10, 210, 10):
    gaussLegendreResult = gaussLegendre(start, end, function2, N)
    print("N = " + str(N) + "    the Gauss-Legendre solution: " + str(gaussLegendreResult) + "    the error is: " + str(abs(gaussLegendreResult - actual2)))
print("\n")
# integral of ln(0.2|sin(x)|)
actual3 = -2.66616
print("integral of ln(0.2|sin(x)|, where the actual value is " + str(actual3) + ":\n")


def function3(x):
    result = math.log(0.2*abs(math.sin(x)))
    return result


for N in range(10, 210, 10):
    gaussLegendreResult = gaussLegendre(start, end, function3, N)
    print("N = " + str(N) + "    the Gauss-Legendre solution: " + str(gaussLegendreResult) + "    the error is: " + str(abs(gaussLegendreResult - actual3)))
print("\n")
# integral of of ln(x) and ln(0.2|sin(x)|) with inequal interval
Ntest = 10
h = [0 for i in range(Ntest)]
smallest_interval = 2*(end - start)/(Ntest*(Ntest + 1))
for i in range(Ntest):
    h[i] = smallest_interval * (i + 1)
print("integral of ln(x), where the actual value is " + str(actual2) + ":\n")
gaussLegendreResult = gaussLegendre(start, end, function2, Ntest, h)
print("N = " + str(Ntest) + "    the Gauss-Legendre solution: " + str(gaussLegendreResult) + "    the error is: " + str(abs(gaussLegendreResult - actual2)) + "\n")
print("integral of ln(0.2|sin(x)|, where the actual value is " + str(actual3) + ":\n")
gaussLegendreResult = gaussLegendre(start, end, function3, Ntest, h)
print("N = " + str(Ntest) + "    the Gauss-Legendre solution: " + str(gaussLegendreResult) + "    the error is: " + str(abs(gaussLegendreResult - actual3)))
```