McGill University

Numerical Methods

Assignment 3

Student Name: Laith Mubaslat

I. PROBLEM 1

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

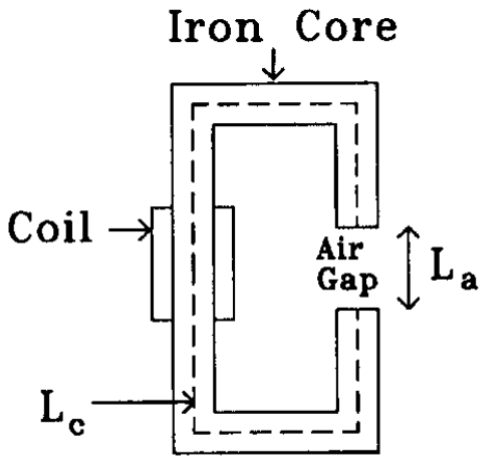

Figure 1

(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?

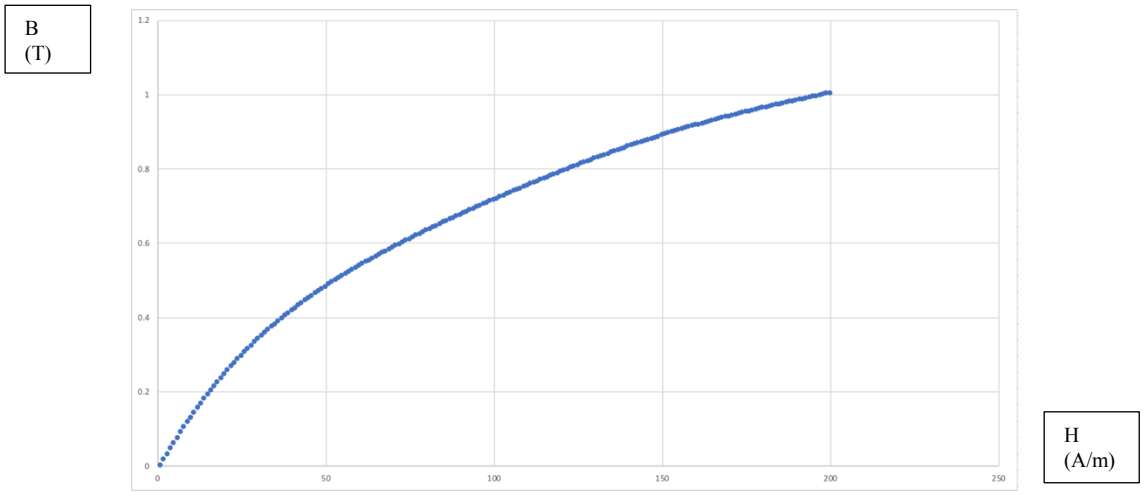

Figure 2:First 6 Points Full Domain Lagrange Polynomial Interpolation(over the range of the first 6 points)

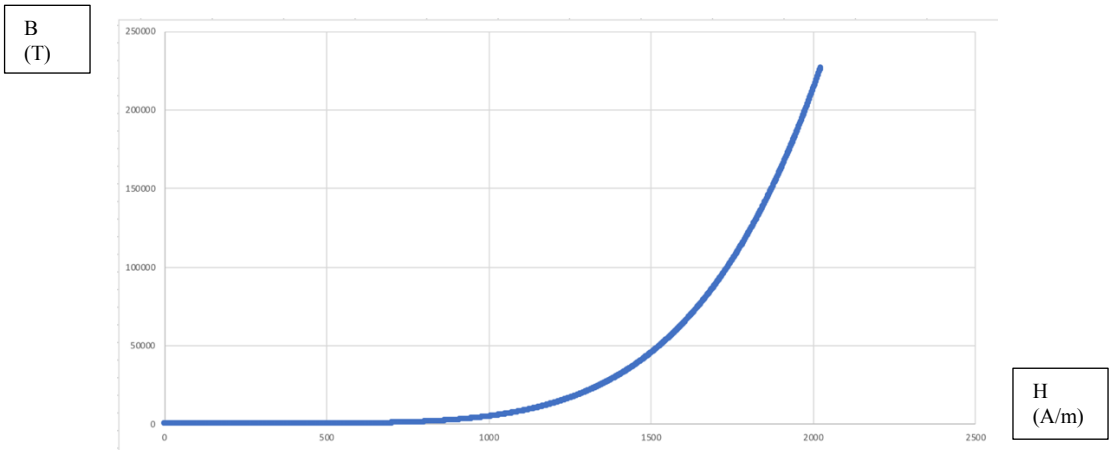

Figure 3: First 6 Points Full Domain Lagrange Polynomial (Extrapolation beyond the range of the first 6 points)

Comments:

It can be clearly seen how the Full Domain Lagrange Polynomial Interpolation which is of an order of 5 provides plausible results for the B versus H over the range of the first 6 points the Polynomial is defined on. That can be attributed to the fact that these points are taken at such short intervals that the wiggles of the 5^{th} order Polynomial do not show interpolating over this period. However, it should be noted that extrapolating for points beyond the range of the first 6 points the Lagrange Polynomial is defined on provides results that are not even remotely plausible for B versus H curve. Which is something that can be clearly seen in Figure 3 above.

(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?

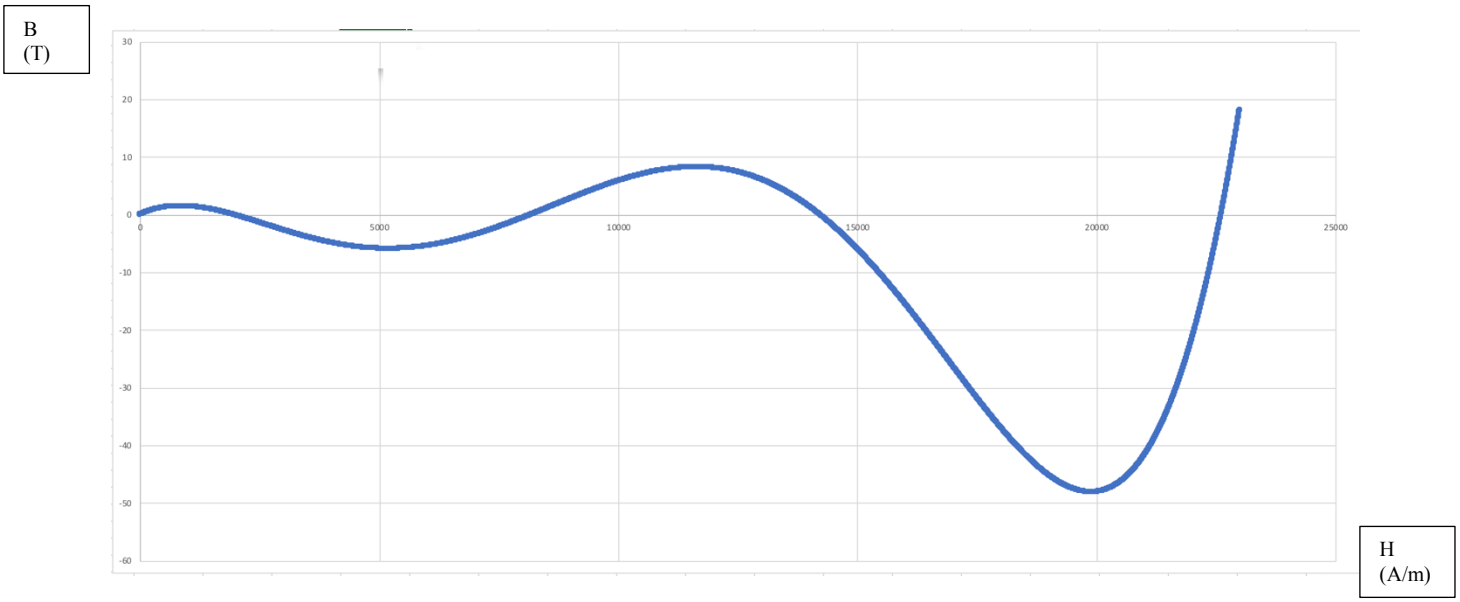

Figure 4: B=0, 1.3, 1.4 , 1.7,1.8,1.9 Full Domain Lagrange Polynomial Interpolation

Comments:

The full domain Lagrange Polynomial defined over these 6 points of part b which span the B versus H curve provides a non-plausible result. That can be attributed to the fact that we are attempting to force a 5^{th} order Polynomial (which has an extremely high potential for wiggles when of a high order) over a number of points which are spread so far apart that the Polynomial in questions has the ability to display these wiggles which are characteristic of Lagrange Polynomial Interpolation. From a practical point of view, this result can be also attributed to the fact that the order 5 Full Domain Lagrange Polynomial is being defined over such a wide range (e.g points of B= 0 and B=1.3 and points of B=1.4, B=1.7 and B=1.8) such that it is trying to represent the different dynamics (e.g. representing the linear or saturation behavior of the material) of the B versus H curve which are in fact of a varying nature over these intervals.

(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.

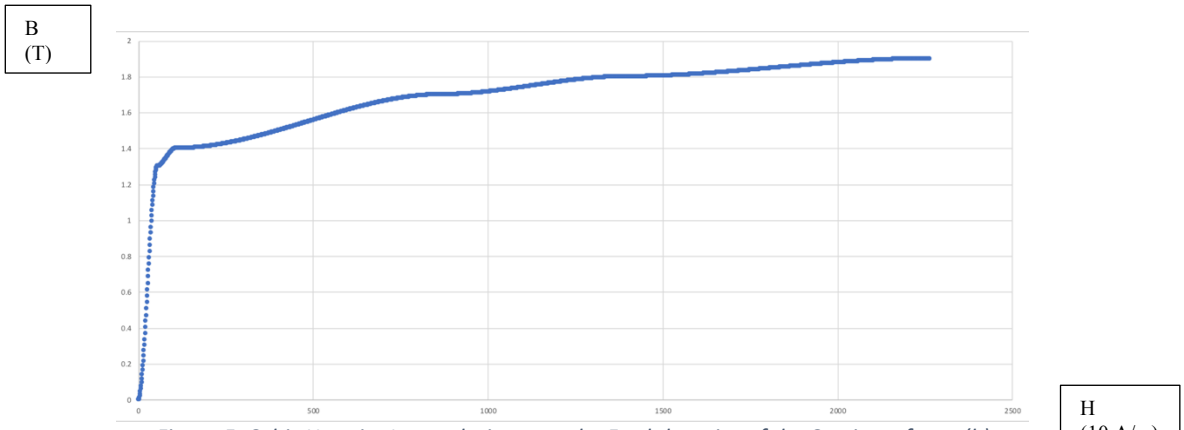

Figure 5: Cubic Hermite Interpolation over the 5 subdomains of the 6 points of part (b) Slopes at 6 points set at zero

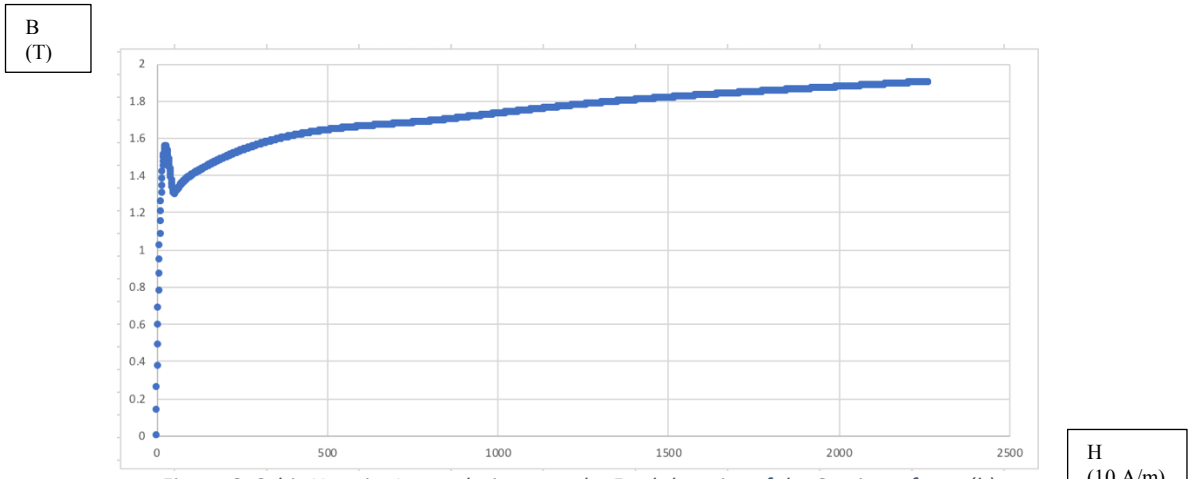

Figure 6: Cubic Hermite Interpolation over the 5 subdomains of the 6 points of part (b) First set of Slopes

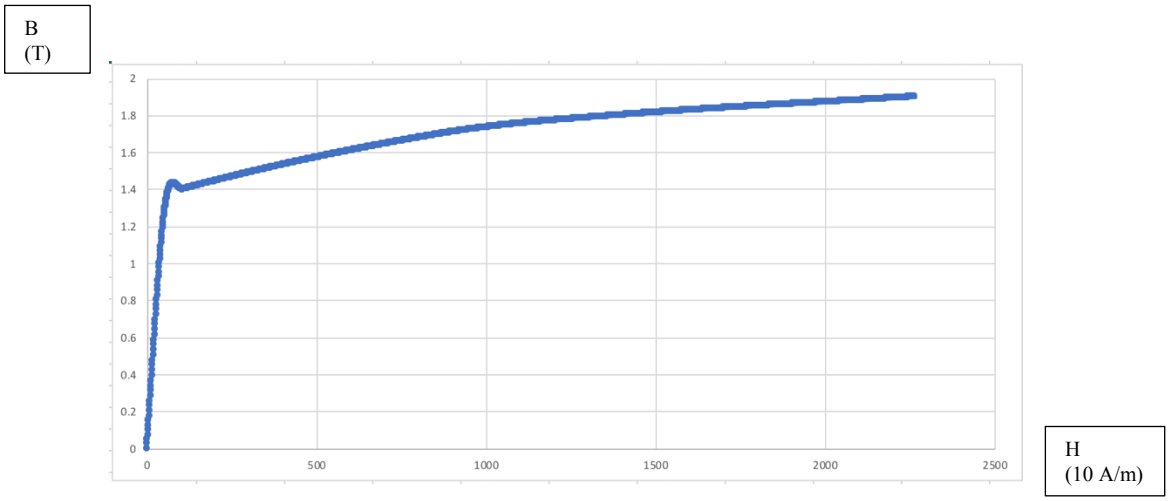

Figure 7: Cubic Hermite Interpolation over the 5 subdomains of the 6 points of part (b) Second set of Slopes

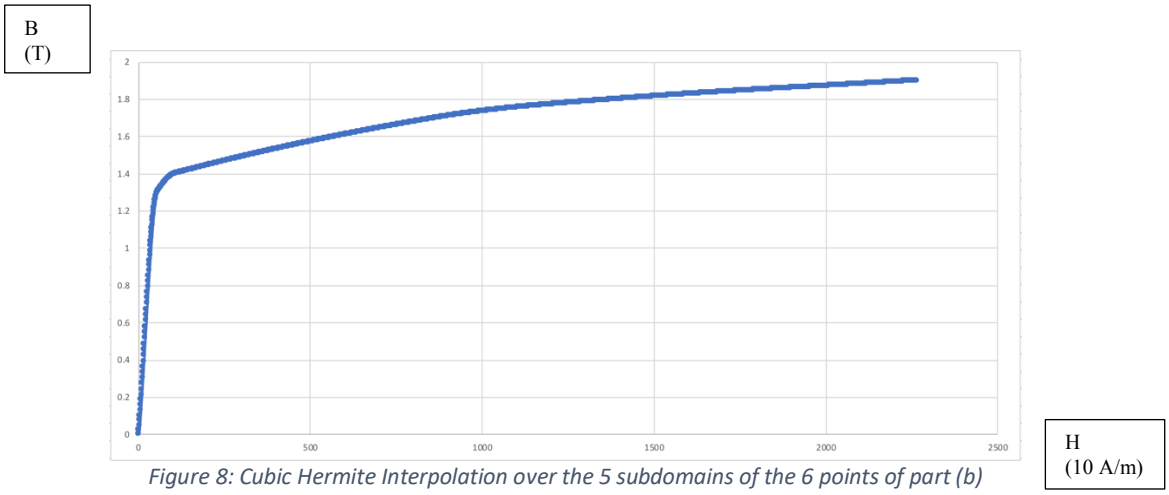

Figure 8: Cubic Hermite Interpolation over the 5 subdomains of the 6 points of part (b) Third set of Slopes

Comments:

A characteristic of Interpolation using Cubic Hermite Polynomials is the utilization of slope data of the points being used. That could come as both an advantage and a disadvantage which is determined based on the availability (or the ability to obtain a valid approximation) for the slopes at those aforementioned Points. Figure 5 demonstrated the curve obtained using this method when the influence of slope data is ignored (i.e. assumed to be zero). It can be clearly seen how several anomalies exist for this iteration of the process which make the curve not accurately representative of the B versus H curve.

The slopes at the 6 points were obtained by computing the finite difference derivative between the points preceding and following each of the 6 points. For that purpose the point preceding the first point was extrapolated using a first order Lagrange polynomial defined between the points (0,0) and (0.2,14.7). Additionally, the point following the last point was extrapolated using a first order Lagrange polynomial defined between the points (1.8, 13924.3) and (1.9, 22650.2).

Figure 6 displays the result for slopes obtained when the points preceding and following are determined according to the points closest to the 6 points used in our interpolation. The effect of the wrong determination of the slope values is most apparent in the spike that can be seen in the first portion of the curve. Additionally, a slight ripple can be observed in the following portion of the curve.

Figure 7 displays the results for slopes obtained when the points preceding and following are determined according to the points following and proceeding a given point amongst the 6 points chosen for interpolation (and the points earlier extrapolated as far as the first and second point are concerned). It should be noted however that though less severe an anomalous spike could still be observed at the earlier portion of the curve.

To tackle the aforementioned anomaly, the slope at the second point of (1.3, 540.6) was taken using the nearest preceding and following points. The result for that modification, which displays a far superior outcome, is displayed in Figure 8.

The slopes used here are between the follow pairs:

[extrapolated value,0.2] for B=0
[1.2,1.4] for B= 1.3
[1.3,1.5] for B= 1.4
[1.6,1.8] for B= 1.7
[1.7,1.9] for B= 1.8
[1.8, extrapolated value] for B= 1.9

The formula used is : slope = [B(point2)-B(point1)] / [H(point2) – H(point1)]

The experiments done above serve to demonstrate the huge influence of the N (number of points) degree of freedom the Hermite Polynomial Interpolation. The results provide a measure of the error that a wrong estimate for the slopes could incur (as shown in Figures 5, 6 and 7) in addition to the improvement a good estimate could provide (as shown in Figure 8).

(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 $f(\psi) = 0$.

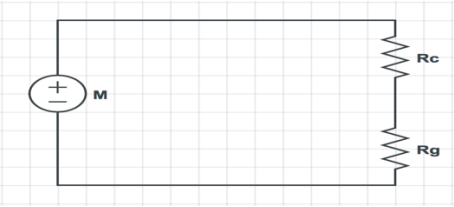

Figure 9: Magnetic Circuit

$$R_c = \frac{L_c}{A * \mu} \qquad equation\ (1-1)$$

$$R_g = \frac{L_g}{A * \mu_o} \qquad equation\ (1-2)$$

$$M = IN \qquad equation\ (1-3)$$

$$B = \varphi/A \qquad equation\ (1-4)$$

$$M = \varphi * (R_g + R_c) \qquad equation\ (1-5)$$

From equations (1-2) and (1-3) Rg and M are shown to be constant and Equation (1-6) bellow is obtained by Substituting for equations (1-1),(1-3) and (1-4) in equation (1-5).

$$M = \varphi * (R_g + R_c) = \varphi * \left(R_g + \frac{L_c}{A * \mu(B)}\right) = \varphi * \left(R_g + \frac{L_c}{A * \mu(\varphi/A)}\right)$$

$$M = \varphi * \left(R_g + \frac{L_c}{A * \mu\left(\frac{\varphi}{A}\right)}\right) \qquad equation\ (1-6)$$

Subtracting M from both sides in equation (1-6) we obtain the following non-linear equation for the flux $\varphi\ in\ the\ form\ \ f(\varphi) = 0$.

$$f(\varphi) = \varphi * \left(R_g + \frac{L_c}{A * \mu\left(\frac{\varphi}{A}\right)}\right) - M = 0 \qquad equation\ (1-7)$$

To find $\mu(B) = \mu\left(\frac{\varphi}{A}\right)$ as well as its derivative (with respect to φ); values for μ are computed from the data points according to the relation $\mu = \frac{B}{H}$ and then interpolated as a function of B using first order Lagrange Polynomials.

M: *Magnetomotive Force* $(Ampere - turn)$
φ: *flux* (T/m^2)
B: *Magnetic Flux Density* (T)
H: *Magnetic Flux Intensity* (A/m)
R_g: *Airgap magnetic Reluctance* $(1/H)$
R_c: *Airgap magnetic Reluctance* $(1/H)$
L_g: *Airgap Length* (m)
L_c: *Core Length* (m)
A: Area (m^2)
N: Number of turns
I: Current (A)
μ: *Magnetic Permiability* (H/m)
μ_o: *Magnetic Permeability of Vacuum; equal to* $4\pi * 10^{-7}\ H/m$

(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 $|f(\psi) / f(0)| < 10^{-6}$. Record the final flux, and the number of steps taken.

Table 2:Question 1 (e) results

Final Flux (T/m^2)	Number of Steps Taken
0.000161752387399445	11

```
In [201]: countersteps
Out[201]: 11

In [202]: fluxnew
Out[202]: 0.000161752387399445

In [203]: f
Out[203]: -0.006167197293507343

In [204]: ref
Out[204]: 7.708996616884178e-07
```

Figure 10: Question 1 (e) output

(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.

The attempt of solving the problem using successive substitution does not converge. To solve the problem a modification was suggested. The modification suggested takes advantage of the fact that at early iterations the flux fluctuates before diverging into larger and larger values for later iterations(which comes as a result of the relatively large "M" term) . The equation is modified such that the old value is multiplied by a constant (a factor of 0.0000001) which prevents the method from diverging into large values by neutralizing the effect of the aforementioned "M" term, thus allowing the method to converge.

Table 3:Question 1 (f) results

Final Flux (T/m^2)	Number of Steps Taken
0.00016175241062935976	11

```
In [211]: countersteps
Out[211]: 11

In [212]: fluxnew
Out[212]: 0.00016175241062935976

In [213]: f
Out[213]: -0.0033915379326572292

In [214]: ref
Out[214]: 4.239422415821537e-07
```

Figure 11: Question 1 (f) output

II. PROBLEM 2

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

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

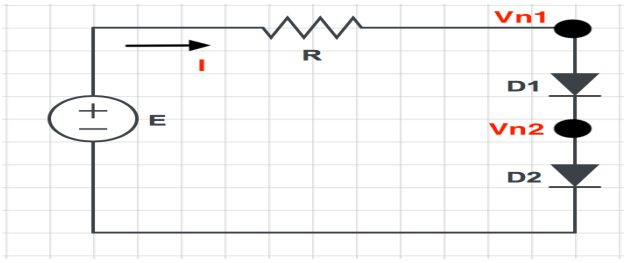

Figure 12: Question 2 Circuit Diagram

D1 is used for Diode A and D2 is used for Diode B.

$$V_{D1} = V_{n1} - V_{n2} \qquad equation\ (2-1)$$

$$V_{D2} = V_{n2} \qquad equation\ (2-2)$$

$$I_{D2} = I_{D1} = I \qquad equation\ (2-3)$$

$$I = \frac{E - V_{n1}}{R} \qquad equation\ (2-4)$$

$$I_{D1} = I_{sA}\ (e^{V_{D1}\, q/kT} - 1) \qquad equation\ (2-5)$$

$$I_{D2} = I_{sB}\ (e^{V_{D2}\, q/kT} - 1) \qquad equation\ (2-6)$$

Substitution equations (2-1) and (2-3) in equation (2-5) the first non linear equation is obtained as follows:

$$I_{D1} = I = \frac{E - V_{n1}}{R} = I_{sA}\left(e^{V_{D1}\, q/kT} - 1\right) = I_{sA}\left(e^{(V_{n1} - V_{n2})\, q/kT} - 1\right)$$

$$\frac{E - V_{n1}}{R} = I_{sA}\left(e^{(V_{n1} - V_{n2})\, q/kT} - 1\right)$$

$$E - V_{n1} = R\, I_{sA}\left(e^{(V_{n1} - V_{n2})\, q/kT} - 1\right)$$

$$V_{n1} = E - R\, I_{sA}\left(e^{(V_{n1} - V_{n2})\, q/kT} - 1\right) \qquad equation\ (2-7)$$

Substitution equations (2-2) and (2-3) in equation (2-6) the second non linear equation is obtained as follows:

$$I_{D2} = I = \frac{E - V_{n1}}{R} = I_{sB}\left(e^{V_{D2}\, q/kT} - 1\right) = I_{sB}\left(e^{(V_{n2})\, q/kT} - 1\right)$$

$$\frac{E - V_{n1}}{R} = I_{sB}\left(e^{(V_{n2})\, q/kT} - 1\right)$$

$$E - V_{n1} = R\, I_{sB}\left(e^{(V_{n2})\, q/kT} - 1\right)$$

$$V_{n1} = E - R\, I_{sB}\left(e^{(V_{n2})\, q/kT} - 1\right) \qquad equation\ (2-8)$$

Accordingly, $f(\boldsymbol{v_n}) = \mathbf{0}$ given explicitly in terms of the variables $I_{sA}, I_{sB}, E, R \; and \; \boldsymbol{v_n}$ can be obtained from equations (2-7) and (2-8) by subtracting v_{n1} from both sides of the two equations as follows :

$$0 = E - R\ I_{sA}\ \left(e^{(V_{n1}-V_{n2})\ q/kT} - 1\right) - V_{n1}$$
$$0 = E - R\ I_{sB}\ \left(e^{(V_{n2})\ q/kT} - 1\right) - V_{n1}$$

$$f(\boldsymbol{v_n}) = \begin{bmatrix} E - R\ I_{sA}\ \left(e^{(V_{n1}-V_{n2})\ q/kT} - 1\right) - V_{n1} \\ E - R\ I_{sB}\ \left(e^{(V_{n2})\ q/kT} - 1\right) - V_{n1} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \qquad equation\ (2-9)$$

(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].

Table 4: Question 2(b) function values and voltages across each diode for every iteration

k	Vn1	Vn2	f1	f2	VD1=Vn1-Vn2	VD2=Vn2
0	0	0	-2.00E-01	-2.00E-01	0	0
1	0.19812073	0.08341926	3.80E-02	1.34E-02	0.114701476	0.08341926
2	0.18413995	0.0843369	5.92E-03	1.08E-05	0.099803059	0.0843369
3	0.18230749	0.08710807	3.54E-04	1.05E-04	0.095199417	0.08710807
4	0.18214001	0.08719341	9.41E-07	1.07E-07	0.094946593	0.08719341
5	0.18213962	0.08719379	8.46E-12	2.09E-12	0.094945832	0.08719379
6	0.18213962	0.08719379	1.73E-17	-3.47E-18	0.094945832	0.08719379

The error measure defined for the purposes of this experiment is the Euclidian norm of the difference between the final solution vector $\boldsymbol{v}^f$ and the solution obtained at each iteration. $\boldsymbol{v}^f$ is the value at k for which $\boldsymbol{v}^k = \boldsymbol{v}^{k+1}$.

Definition: A sequence $\{x^k\}$ in $\mathcal{R}^n$ that converges to x^* is said to converge quadratically if $\exists\ c \in (0, \infty)$ s.t :

$$||x^* - x^{k+1}|| \leq c\,||x^* - x^k||^2 \quad \text{for all k sufficiently large}$$

This condition is equivalent to $\frac{||x^* - x^{k+1}||}{||x^* - x^k||^2} \leq c$ for all k sufficiently large.

Accordingly, the following ratios can be found from Table 5:

For k = 1 $\frac{||\boldsymbol{v}^f - \boldsymbol{v}^2||}{||\boldsymbol{v}^f - \boldsymbol{v}^1||^2} = 12.934$

For k=2 $\frac{||\boldsymbol{v}^f - \boldsymbol{v}^3||}{||\boldsymbol{v}^f - \boldsymbol{v}^2||^2} = 15.4963$

For k=3 $\frac{||\boldsymbol{v}^f - \boldsymbol{v}^4||}{||\boldsymbol{v}^f - \boldsymbol{v}^3||^2} = 15.146$

For k = 4 $\frac{||\boldsymbol{v}^f - \boldsymbol{v}^5||}{||\boldsymbol{v}^f - \boldsymbol{v}^4||^2} = 15.758$

Table 5 : Error at each iteration ($\varepsilon_k = ||\boldsymbol{v}^f - \boldsymbol{v}^k||$)

k	Vn1	Vn2	Euclidian Norm
0	0.182139623	0.08719379	0.201934642
1	-0.015981108	0.00377454	0.016420808
2	-0.002000331	0.0028569	0.003487575
3	-0.000167863	8.5722E-05	0.000188484
4	-3.84509E-07	3.7642E-07	5.38089E-07

The persistence of the pattern can not be verified for k>4 considering the error drops below machine accuracy and convergence is achieved at the 6th iteration. Accordingly, from that, the ratios and the definition displayed above; it can be concluded that the convergence is quadratic.

III. PROBLEM 3

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.

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

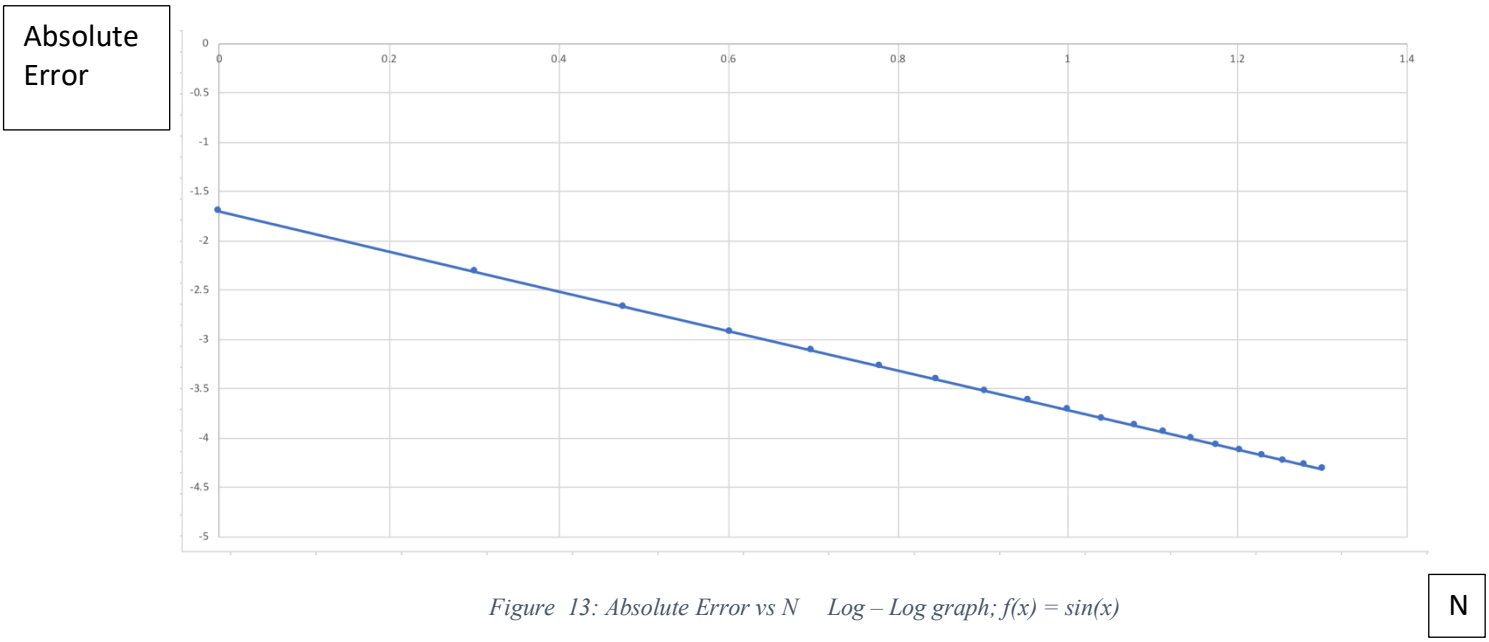

Figure 13: Absolute Error vs N Log – Log graph; f(x) = sin(x)

Comment:
A Gauss-legendre order n integration integrates exactly all polynomials of order n. Accordingly, in theory the error should be proportional to h^{n+2}. Since $h = \frac{b-a}{N}$, with b=1 and a=0; the error should be proportional to $\frac{1}{N}^{n+2} = N^{-(n+2)}$. Since we are using the one point scheme; n=0 and thus it is reasonable to assume that the error should be of order $O(N^{-(2)})$. The slope of the curve obtained for the absolute error is -2.0006443 which agrees with the aforementioned assumption (considering it's a log-log curve) .

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

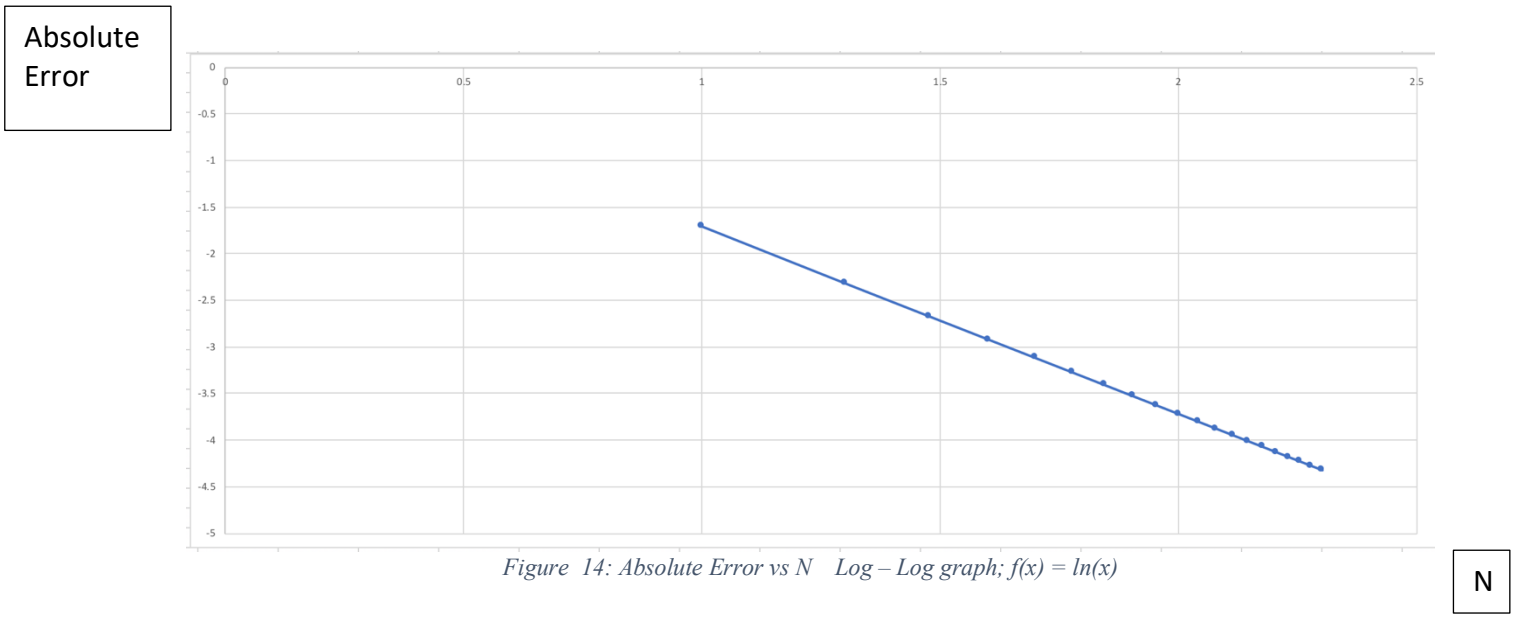

Figure 14: Absolute Error vs N Log – Log graph; f(x) = ln(x)

Comment:

The slope of the curve is -2.0003393 which agrees with the assumption made in part a which is valid considering the same 1 point scheme is used in both parts (a) and (b).

(c) Repeat part (b) for the function $f(x) = ln(0.2\,|\sin(x)|)$. Comment on the result.

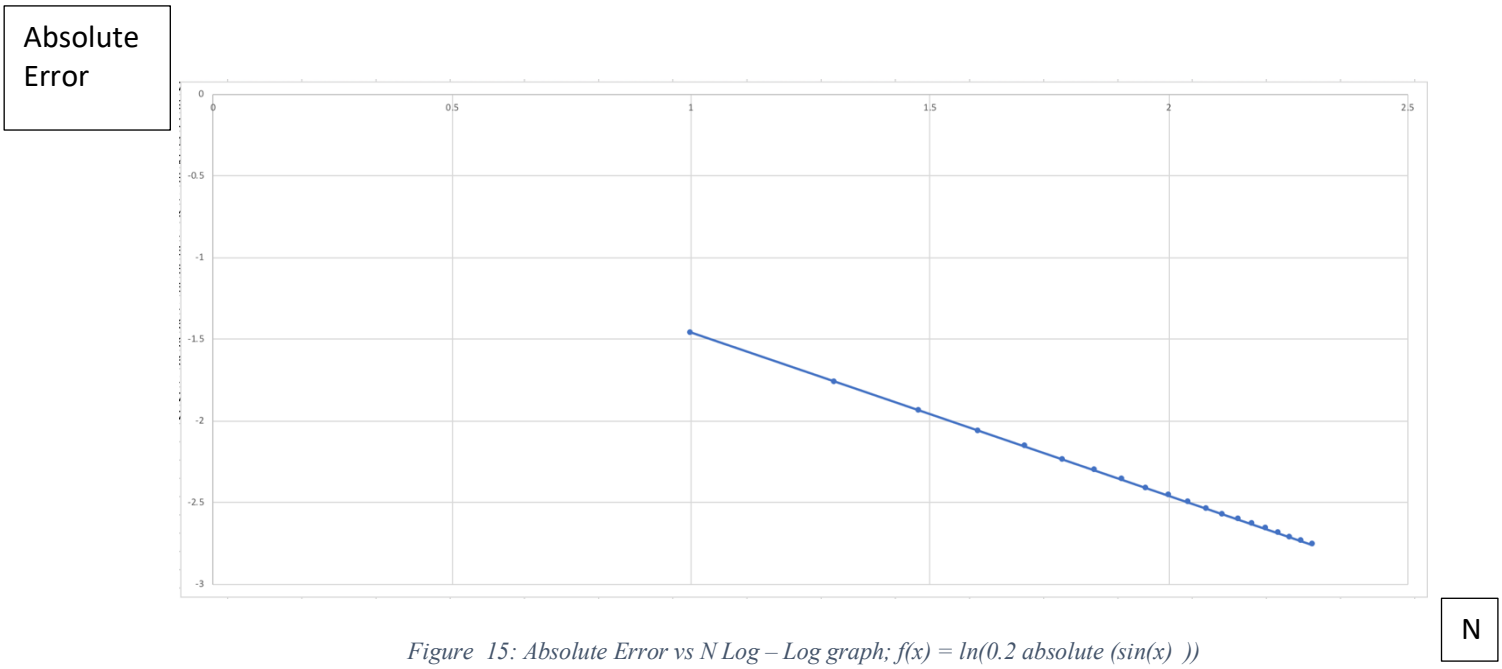

Figure 15: Absolute Error vs N Log – Log graph; f(x) = ln(0.2 absolute (sin(x)))

The curve for this part has a slope of -1. This comes as a result of the curve being harder to integrate due to the waveform shape the curve takes. This difficulty could be explained in one of two ways. The first, is that the mid point of equal spacings of x are taken which are mapped to their corresponding values in the domain of sin(x). Those values are then themselves mapped by the ln function. As far as sin(x) is concerned the points it uses are equally spaced but for the ln they are not as these point come from the output of sin(x). The effect of using the wrong spacing scheme is argued for in the following part.

Another explanation is that the scheme we use integrates polynomial terms up to order one. The Taylor series expansion equivalent function in this case is the taylor series expansion of a taylor series expansion (i.e. the approximation of an approximation) which thus makes the integration less reliable.

(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.

Tables 6 and 7 demonstrate a preliminary study into the effect of changing the location of smallest spacing in order to get a sense of how the 10 segments spacing should be varied for each function.

$$f(x) = sin(x)$$

Table 6: Comparing the effect of location of smallest spacing relative to the absolute error

Segment Location of smallest spacing	(Largest Abs E - AbsError(i)) * 100000000
1	0
2	133.0679865
3	264.3875277
4	392.4253386
5	515.6864518
6	632.7316724
7	742.1943821
8	842.7964962
9	933.3633858
10	1012.837593

$$f(x) = ln(x)$$

Table 7: Comparing the effect of location of smallest spacing relative to the absolute error

Segment Location of smallest spacing	(Largest Abs E - AbsError(i))* 100000000
1	93675078.04
2	93675078.04
3	93675078.04
4	93675078.04
5	93675078.04
6	93675078.04
7	93675078.04
8	93675078.04
9	93675078.04
10	0

Suggested Scheme and Results

Spacings from right to left [f(x) = sin(x)] := [1,1,1,0.9,0.8,0.75,0.7,0.7,0.65,0.64]
Spacings from right to left [f(x) = ln(x)] := [0.3,0.5,0.7,1,2,2,3,3,4,6]

Table 8: Results

Function	N	Abs Error Uniform	Abs Error Non Uniform	% improvement
Sin	10	0.000191597	0.000173629	9.377590198
Ln	10	0.034240935	0.010442024	69.50426661

Table 9: f=sin(x) ; putting the improved result into perspective

N	10	10 (non uniform)	11
Abs Error	0.000191597	0.000173629	0.000158336

Table 10: f=ln(x) ; putting the improved result into perspective

N	10	20	30	10 (non uniform)	40
Abs Error	0.034240935	0.017224528	0.01150616	0.010442024	0.00691481

Comment:

Table 9 shows the result for each of the two schemes separately while tables 10 and 11 put these results into perspective by comparing them to the previously obtained error values for the same functions in part a) and b). It can be seen how varying the spacing by just the right amount could improve the integration accuracy. It can also be seen how the level of improvement is different based on the function we are trying to integrate which does make sense considering how some functions would leave a larger space for improvement than others (i.e. larger error values because the taylor series powers integrated make up less of the function than their high power counterparts) . Furthermore, it should be noted how varying the spacing randomly or inconsistently would be counterproductive.

IV. APPENDIX

```
#####Q1A


xstartingpoint=0
xendpoint=200 #i.e 199




x1=0
x2=14.7
x3=36.5
x4=71.7
x5=121.4
x6= 197.4


y1=0
y2=0.2
y3=0.4
y4=0.6
y5=0.8
y6= 1




n=6

y=[0,y1,y2,y3,y4,y5,y6]




xr=[0,x1,x2,x3,x4,x5,x6] #in order to start from x1 at r=1


numberofvaluestobemeasure=xendpoint-xstartingpoint
#used to hold the values we compute for L at an x=value
L1int=[0]*(numberofvaluestobemeasure+1)
L2int=[0]*(numberofvaluestobemeasure+1)
L3int=[0]*(numberofvaluestobemeasure+1)
L4int=[0]*(numberofvaluestobemeasure+1)
L5int=[0]*(numberofvaluestobemeasure+1)
L6int=[0]*(numberofvaluestobemeasure+1)


def Fj(j,xr,x):
    F=1
    for r in range (1,7):
        if (r!=j):
```

```
            F=F*(x-xr[r])

    return F

def Lj(j,xr,x):
    xj=xr[j]
    L=Fj(j,xr,x)/Fj(j,xr,xj)
    return L



#n=6 > order 5 lag poly


#j=1
#L1=Lj(j,xr,x)
#j=2
#L2=Lj(j,xr,x)
#j=3
#L3=Lj(j,xr,x)
#j=4
#L4=Lj(j,xr,x)
#j=5
#L5=Lj(j,xr,x)
#j=6
#L6=Lj(j,xr,x)




#main starts here

totalcurve=[0]*(numberofvaluestobemeasure+1)



a=[0]*(n+1)

for i in range (1,7):
    a[i]=y[i]


for i in range (xstartingpoint,xendpoint):
    valuestobecomputedat=i
    x=valuestobecomputedat

    j=1
    L1int[i+1]=Lj(j,xr,x)
```

```
    j=2
    L2int[i+1]=Lj(j,xr,x)


    j=3
    L3int[i+1]=Lj(j,xr,x)


    j=4
    L4int[i+1]=Lj(j,xr,x)

    j=5
    L5int[i+1]=Lj(j,xr,x)


    j=6
    L6int[i+1]=Lj(j,xr,x)


    totalcurve[i+1]=a[1]*L1int[i+1]+a[2]* L2int[i+1]+a[3]*L3int[i+1]+a[4]



#first output value starts at 1>total curve

#
```

```
#Q1B


xstartingpoint=0
xendpoint=23000 #i.e 199



x1=0
x2=540.6
x3=1062.8
x4=8687.4
x5=13924.3
x6= 22650.2


y1=0
y2=1.3
y3=1.4
y4=1.7
y5=1.8
y6= 1.9



n=6

y=[0,y1,y2,y3,y4,y5,y6]



xr=[0,x1,x2,x3,x4,x5,x6] #in order to start from x1 at r=1


numberofvaluestobemeasure=xendpoint-xstartingpoint
#used to hold the values we compute for L at an x=value
L1int=[0]*(numberofvaluestobemeasure+1)
L2int=[0]*(numberofvaluestobemeasure+1)
L3int=[0]*(numberofvaluestobemeasure+1)
L4int=[0]*(numberofvaluestobemeasure+1)
L5int=[0]*(numberofvaluestobemeasure+1)
L6int=[0]*(numberofvaluestobemeasure+1)


def Fj(j,xr,x):
    F=1
    for r in range (1,7):
        if (r!=j):
```

```
            F=F*(x-xr[r])

    return F

def Lj(j,xr,x):
    xj=xr[j]
    L=Fj(j,xr,x)/Fj(j,xr,xj)
    return L



#n=6 > order 5 lag poly


#j=1
#L1=Lj(j,xr,x)
#j=2
#L2=Lj(j,xr,x)
#j=3
#L3=Lj(j,xr,x)
#j=4
#L4=Lj(j,xr,x)
#j=5
#L5=Lj(j,xr,x)
#j=6
#L6=Lj(j,xr,x)




#main starts here

totalcurve=[0]*(numberofvaluestobemeasure+1)



a=[0]*(n+1)

for i in range (1,7):
    a[i]=y[i]


for i in range (xstartingpoint,xendpoint):
    valuestobecomputedat=i
    x=valuestobecomputedat

    j=1
    L1int[i+1]=Lj(j,xr,x)
```

```
    j=2
    L2int[i+1]=Lj(j,xr,x)


    j=3
    L3int[i+1]=Lj(j,xr,x)


    j=4
    L4int[i+1]=Lj(j,xr,x)

    j=5
    L5int[i+1]=Lj(j,xr,x)


    j=6
    L6int[i+1]=Lj(j,xr,x)


    totalcurve[i+1]=a[1]*L1int[i+1]+a[2]* L2int[i+1]+a[3]*L3int[i+1]+a[4]



#first output value starts at 1>total curve

#
```

```
#Q1C first and last point slope
def Fj(j,xr,x):
    F=1
    for r in range (1,3):
        if (r!=j):
            F=F*(x-xr[r])



    return F

def Lj(j,xr,x):
    xj=xr[j]
    L=Fj(j,xr,x)/Fj(j,xr,xj)
    return L



#point 1

x1=0
y1=0

x2=14.7
y2=0.2







#define the function (1st order lagrange poly.)
x=-14.7
xr=[0,x1,x2]
L1=Lj(1,xr,x)
L2=Lj(2,xr,x)

y=y1*L1+y2*L2


xextrapolated1=x
yextrapolated1=y



slope1=(y2-yextrapolated1)/(x2-xextrapolated1)
```

```
#point 6

x1=13924.3
y1=1.8

x2=22650.2
y2=1.9



x=23000
xr=[0,x1,x2]
L1=Lj(1,xr,x)
L2=Lj(2,xr,x)

y=y1*L1+y2*L2



xextrapolated2=x
yextrapolated2=y

slope6=(y1-yextrapolated2)/(x1-xextrapolated2)



####slope1 fixed
#x2 y2 fixed


x1=0
y1=0

x2=540.6
y2=1.3



#define the function (1st order lagrange poly.)
```

```
x=-14.7
xr=[0,x1,x2]
L1=Lj(1,xr,x)
L2=Lj(2,xr,x)

y=y1*L1+y2*L2


xextrapolated1=x
yextrapolated1=y



slope1=(y2-yextrapolated1)/(x2-xextrapolated1)
```

```
#Q1C slope ALL POINTS


x1=0
x2=540.6
x3=1062.8
x4=8687.4
x5=13924.3
x6= 22650.2


y1=0
y2=1.3
y3=1.4
y4=1.7
y5=1.8
y6= 1.9




# the slope is needed for each of these points
slope=[0]*7


slope[2]=(1.4-1.2)/(1062.8-348.7)
slope[3]=(1.5-1.3)/(2318-540.6)
slope[4]=(1.8-1.6)/(13924-4781.9)
slope[5]=(1.9-1.7)/(22650.2-8687.4)


#computed using Q1 part A program
yinterpolatedbefore=-14.7
xinterpolatedbefore= -0.280482169122

#computed using Q1 part A program for the last 6 points



slope[1]= (0.2-yinterpolatedbefore)/(14.7-xinterpolatedbefore)
slope[6]= (yinterpolatedafter-1.8)/(xinterpolatedafter-13924)


slope[1]=slope1
slope[6]=slope6
```

```
def Fj(j,j1,j2,xr,x):
    F=1
    for r in range (j1,j2+1):
        if (r!=j):
            F=F*(x-xr[r])



    return F

def Lj(j,j1,j2,xr,x):
    xj=xr[j]
    L=Fj(j,j1,j2,xr,x)/Fj(j,j1,j2,xr,xj)
    return L




def derLj(j,j1,j2,xr,x):
    xj=xr[j]
    derL=1/Fj(j,j1,j2,xr,xj)
    return derL




################################################################################
################################################################################

#main starts here



#6degrees of freedom
b=[0]*(n+1)

#for i in range (1,7):
    #a[i]=y[i]
```

```
start=0
end=22650
reductionfactor=10
numberofvaluestobemeasure=int((end-start)/reductionfactor)

V1int=[0]*(numberofvaluestobemeasure+1)
V2int=[0]*(numberofvaluestobemeasure+1)
V3int=[0]*(numberofvaluestobemeasure+1)
V4int=[0]*(numberofvaluestobemeasure+1)
V5int=[0]*(numberofvaluestobemeasure+1)
V6int=[0]*(numberofvaluestobemeasure+1)


U1int=[0]*(numberofvaluestobemeasure+1)
U2int=[0]*(numberofvaluestobemeasure+1)
U3int=[0]*(numberofvaluestobemeasure+1)
U4int=[0]*(numberofvaluestobemeasure+1)
U5int=[0]*(numberofvaluestobemeasure+1)
U6int=[0]*(numberofvaluestobemeasure+1)


totalcurve=[0]*(numberofvaluestobemeasure+1)


end=int(end/reductionfactor)




x1=0
x2=540.6
x3=1062.8
x4=8687.4
x5=13924.3
x6= 22650.2


y1=0
y2=1.3
y3=1.4
y4=1.7
y5=1.8
```

```
y6= 1.9



n=6

y=[0,y1,y2,y3,y4,y5,y6]



xr=[0,x1,x2,x3,x4,x5,x6] #in order to start from x1 at r=1



a=[0]*(n+1)

for i in range (1,7):
    a[i]=y[i]



#start should be zero
for i in range (start,end):

    x= int(i*reductionfactor)




    if (x>=x1 and x<=x2):

        j1=1

        j2=2

    if (x>=x2 and x<=x3):

        j1=2
        j2=3

    if (x>=x3 and x<=x4):

        j1=3
```

```
    j2=4

if (x>=x4 and x<=x5):

    j1=4
    j2=5

if (x>=x5 and x<=x6):

    j1=5
    j2=6


L=Lj(j1,j1,j2,xr,x)
derL=derLj(j1,j2,j2,xr,x)
xj=xr[j1]
U1 = (1 - 2 * derL * (x-xj) )* L * L

V1 = (x-xj) * L * L


L=Lj(j2,j1,j2,xr,x)
derL=derLj(j2,j1,j2,xr,x)
xj=xr[j2]
U2 = (1 - 2 * derL * (x-xj) )* L * L

V2 = (x-xj) * L * L









b=slope
#try only Lag and see what happens

#ouput au +bv

totalcurve[i+1]=a[j1]*U1+a[j2]* U2
totalcurve[i+1]=totalcurve[i+1]+ b[j1]*V1+b[j2]* V2
```

```
#Q1E


# we are curve fitting the permiability


#function
X=[0]*17
Y=[0]*17
perm=[0]*17
X[1]=0
X[2]=14.7
X[3]=36.5
X[4]=71.7
X[5]=121.4
X[6]= 197.4
X[7]= 256.2
X[8]= 348.7
X[9]= 540.6
X[10]=1062.8
X[11]= 2318.0
X[12]= 4781.9
X[13]=8687.4
X[14]=13924.3
X[15]=22650.2



Y[1]=0
Y[2]=0.2
Y[3]=0.4
Y[4]=0.6
Y[5]=0.8
Y[6]= 1
Y[7]=1.1
Y[8]=1.2
Y[9]=1.3
Y[10]=1.4
Y[11]=1.5
Y[12]=1.6
Y[13]=1.7
Y[14]=1.8
Y[15]=1.9


X[16]=30000 # value to be found  from part c
Y[16]=2
```

```
for i in range (2,17):
    perm[i]=Y[i]/X[i]
X=Y
Y=perm



# X axis is B
# Y axis is perm

#first order lagrange
#j1 and j2 are x1 and x2 for each interval
#j is which lag poly it is

def Fj(j,j1,j2,xr,x):
    F=1
    for r in range (j1,j2+1):
        if (r!=j):
            F=F*(x-xr[r])

    return F

def Lj(j,j1,j2,xr,x):
    xj=xr[j]
    L=Fj(j,j1,j2,xr,x)/Fj(j,j1,j2,xr,xj)
    return L


def function(Y,X,x):

    # we have 14 intervals > each interval has it's own linear function
    interval=0

    for i in range (1,15):  #because the last interval is between i=14 an
        if (x>=X[i] and x<=X[i+1]):
            interval = i
            break



    #FIRST L IS THAT OF i

    #SECOND L IS THAT OF i+1
    xr=X

    j1=i
    j2=i+1
```

```
    j=j1
    L1=Lj(j,j1,j2,xr,x)


    j=j2
    L2=Lj(j,j1,j2,xr,x)




    f= Y[j1]*L1 + Y[j2]* L2

    return f
```

```
def derfunction(Y,X,x):
     # we have 14 intervals > each interval has it's own linear function
    interval=0

    for i in range (1,15):  #because the last interval is between i=14 an
        if (x>=X[i] and x<=X[i+1]):
            interval = i
            break



    #FIRST L IS THAT OF i

    #SECOND L IS THAT OF i+1
    xr=X

    j1=i
    j2=i+1


    j=i
    xj=xr[j]
    Fj1=Fj(j,j1,j2,xr,xj)


    j=i+1
    xj=xr[j]
```

```
    Fj2=Fj(j,j1,j2,xr,xj)



    derf= Y[j1]/Fj1 + Y[j2]/Fj2

    return derf



output=[0]*200
deto=[0]*200
for i in range (200):
    x=i/100
    output[i]=function (Y,X,x)
    deto[i]=derfunction(Y,X,x)




function (Y,X,1)
```

```
#constants
Lc= 30 * 10**(-2)
Lg= 0.5 * 10**(-2)
N= 800
I= 10
A= 1 * 10**(-4)
perm0=1.256637 * 10**(-6)
M= I * N


Rg=Lg/(A*perm0)

#reference for error
refflux=0
refB = refflux/A
refcoreperm=function(Y,X,refB)
#refRc=Lc/(A*refcoreperm)

#fref=(Rg + refRc) * refflux - M
fref=-M




fluxold =0

countersteps=0


flux=[0]*100000

for i in range (100000):

    countersteps=countersteps+1

    B=fluxold/A
    coreperm=function(Y,X,B)
    if (coreperm==0):
        coreperm=7
    Rc=Lc/(A*coreperm)

    if (fluxold==0):
        Rc=0

    derterm= Rg - fluxold*(Lc/ (A*A * coreperm*coreperm)) * derfunction(Y
    #if we remove Rg it will converge in 7 steps


    f= (Rg + Rc) * fluxold - M
```

```
derf= Rg + Rc + derterm


fluxnew = - (f/derf) + fluxold

#compute f (fluxnew)

B=fluxnew/A
coreperm=function(Y,X,B)
Rc=Lc/(A*coreperm)

f= (Rg + Rc) * fluxnew - M

ref= f/fref

if (ref < 0 ):
    ref=-ref


if (ref < 10**(-6)):
    break

fluxold=fluxnew
flux[i]=fluxnew
```

```
#Q1F
#constants

Lc= 30 * 10**(-2)
Lg= 0.5 * 10**(-2)
N= 800
I= 10
A= 1 * 10**(-4)
perm0=1.256637 * 10**(-6)
M= I * N


Rg=Lg/(A*perm0)

#reference for error
refflux=0
refB = refflux/A
refcoreperm=function(Y,X,refB)
#refRc=Lc/(A*refcoreperm)

#fref=(Rg + refRc) * refflux - M
fref=-M




fluxold =0

countersteps=0


flux=[0]*100000



#####addition to make it converge


fold=fref


fold=(Rg + Rc) * fluxold - M
for i in range (100000):

    countersteps=countersteps+1

    B=fluxold/A
    coreperm=function(Y,X,B)
    if (fluxold==0):
        coreperm=7
    Rc=Lc/(A*coreperm)
```

```
#derterm= - (Lc/ (A*A * coreperm*coreperm)) * derfunction(Y,X,B)


fnew= (Rg + Rc) * fluxold - M


#derf= Rg + Rc + derterm


fluxnew = - fnew*0.00000001+ fluxold




fold=fnew
#compute f (fluxnew)

B=fluxnew/A
coreperm=function(Y,X,B)
Rc=Lc/(A*coreperm)

f= (Rg + Rc) * fluxnew - M

ref= f/fref

if (ref < 0 ):
    ref=-ref


if (ref < 10**(-6)):
    break

fluxold=fluxnew
flux[i]=fluxnew
```

```
#Q2B

from math import exp

def Jacobian (v1,v2):
    KTq=25 / 1000
    qKT= 1/KTq

    E=200/1000
    R= 512
    Isa= 0.8 * ( (10) ** (-6)    )
    Isb=1.1 * ( (10) ** (-6)    )



    df1v1=1 + R * Isa * qKT * exp (    (v1-v2) * qKT )
    df1v2= - R * Isa * qKT * exp (    (v1-v2) * qKT )
    df2v1=1
    df2v2= R * Isb * qKT * exp (    (v2) * qKT )


    Jvec=[df1v1, df1v2,df2v1,df2v2]

    J=Matrix(Jvec,2,2)


    return J



def EuclidianNorm(V):
    for i in range (2):
        norm=V[0]*V[0]+V[1]*V[1]
        norm=sqrt(norm)


    return norm



#main newton raphson
numberofmeasuments=1000
initialv1=0
initialv2=0




counter=0
v=[0]*numberofmeasuments
```

```
KTq=25 / 1000
qKT= 1/KTq
E=200/1000
R= 512
Isa= 0.8 * ( (10) ** (-6)    )
Isb=1.1 * ( (10) ** (-6)    )




v1=initialv1
v2=initialv2


#saved voltage values
Vector=[0]* (2* numberofmeasuments)



IterationsVoltageOutput=Matrix(Vector,numberofmeasuments,2)
IterationsFOutput=Matrix(Vector,numberofmeasuments,2)

for i in range (numberofmeasuments):




    counter=counter+ 1

    J= Jacobian(v1,v2)

    A= J[0][0]
    B= J[0][1]
    C= J[1][0]
    D= J[1][1]



    f1old= v1-E+R*Isa* ( exp ( (v1-v2) * qKT        )    - 1
    f2old= v1-E+R*Isb* ( exp ( (v2) * qKT        )    - 1             )
```

```
foldvector= [f1old,f2old]
fold= Matrix(foldvector, 2, 1)

vvectorold=[v1,v2]
vold= Matrix(vvectorold,2,1)



R1=Multiplication (J,vold)
Rightside=[0]*2
Rightside=Matrix(Rightside,2,1)

for j in range (2):
    Rightside[j][0]= - fold[j][0]+R1[j][0]


X=Rightside[0][0]
Y=Rightside[1][0]


v2new= (Y- C*X/A) / (D-C*B/A)


v1new= (X-B*v2new)/A







IterationsVoltageOutput[i][0]=v1
IterationsVoltageOutput[i][1]=v2


IterationsFOutput[i][0]=f1old
IterationsFOutput[i][1]=f2old


Vitekminus1=IterationsVoltageOutput[i-1]
Vitek=[v1new,v2new]

if (EuclidianNorm(Vitekminus1 )==EuclidianNorm(Vitek ) ):
    break




# add stop check here
```

```
    v1=v1new
    v2=v2new


    xfinal=IterationsVoltageOutput[8]




#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###



xxfinalminus1=[0]*2
xxfinalminus2=[0]*2
xxfinalminus3=[0]*2
xxfinalminus4=[0]*2
xxfinalminus5=[0]*2


for i in range (2):
    xxfinalminus1[i]=xfinal[i]-IterationsVoltageOutput[1][i]
    xxfinalminus2[i]=xfinal[i]-IterationsVoltageOutput[2][i]
    xxfinalminus3[i]=xfinal[i]-IterationsVoltageOutput[3][i]
    xxfinalminus4[i]=xfinal[i]-IterationsVoltageOutput[4][i]
    xxfinalminus5[i]=xfinal[i]-IterationsVoltageOutput[5][i]



finalminus1norm=EuclidianNorm(xxfinalminus1)
finalminus2norm=EuclidianNorm(xxfinalminus2)
finalminus3norm=EuclidianNorm(xxfinalminus3)
finalminus4norm=EuclidianNorm(xxfinalminus4)
finalminus5norm=EuclidianNorm(xxfinalminus5)


term1=finalminus2norm/(finalminus1norm*finalminus1norm)
term2=finalminus3norm/(finalminus2norm*finalminus2norm)
term3=finalminus4norm/(finalminus3norm*finalminus3norm)
term4=finalminus5norm/(finalminus4norm*finalminus4norm)
```

```
#term2 term3 term4

#as can be clearly seen as x is close enough to x norm[1]/norm[2]^2 = C
```

```
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###

#QUESTION 3

def integrate(x0,xN,N,function):

    h=(xN-x0)/N
    # since it's the one point scheme
    wi=2
    zetai=0









    sum=0

    for i in range (N):
        sum = sum + wi*function(zetai+h*(i+1/2) ) *h/2



    return sum


#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#PARTA

from math import sin
from math import log
from math import cos


#integration limits
x0=0
xN=1


function=sin
#Reference value
exactvaluesSINintegral = -cos(xN)- (-cos(x0))
```

```
log10N=[0]*21
for i in range (1,21):
    log10N[i]= log(i,10)



log10NB=[0]*21
for i in range (1,21):
    x=i*10
    log10NB[i]= log(x,10)

abserrorPARTA = [0]*21
log10abserrorPARTA = [0]*21

for i in range (1,21):
    N=i

    abserrorPARTA[i]=exactvaluesSINintegral-integrate(x0,xN,N,function)

    if (abserrorPARTA[i]<0):
        abserrorPARTA[i]=-abserrorPARTA[i]


    log10abserrorPARTA[i]= log (abserrorPARTA[i],10)




#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#PARTB
function=log
#reference
exactvaluesLnintegral = -1


abserrorPARTB = [0]*21
log10abserrorPARTB = [0]*21

for i in range (1,21):
```

```
    N=i*10

    abserrorPARTB[i]=exactvaluesLnintegral-integrate(x0,xN,N,function)

    if (abserrorPARTB[i]<0):
        abserrorPARTB[i]=-abserrorPARTB[i]


    log10abserrorPARTB[i]= log (abserrorPARTB[i],10)



#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#PARTC
def integrateLNABS02SIN(x0,xN,N):

    h=(xN-x0)/N
    # since it's the one point scheme
    wi=2
    zetai=0









    sum=0

    for i in range (N):
        sum = sum + wi*            log(0.2*(abs(sin(zetai+h*(i+1/2)))))



    return sum
```

```
exactvaluesLnABSintegral = -2.66616

abserrorPARTC = [0]*21
log10abserrorPARTC = [0]*21

for i in range (1,21):
    N=i*10

    abserrorPARTC[i]=exactvaluesLnABSintegral-integrateLNABS02SIN(x0,xN,N

    if (abserrorPARTC[i]<0):
        abserrorPARTC[i]=-abserrorPARTC[i]


    log10abserrorPARTC[i]= log (abserrorPARTC[i],10)


#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###




#the following programs serve to evaluate the effect of decreasing h at d
#the integration is divided into 10 intervals
#the smallest interval is chosen to be each of them seperatly (with numer
#the values of the absolute error for each is then compared against the i
# a pattern can be clearly seen for both integral where the closer the sm
# due to the consistancy of the results for different normal/less ratios
# it can be concluded that the higher accuracy is consistent with choosin


#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#unequal spacing SIN


function=sin
```

```
N=10
hequalspacing=(xN-x0)/N
normal=2
less=0.5

normalizingfactor= normal*9+less

h=[0]*N
a=[normal]*N
abserrorsin=[0]*N

for j in range (N):
    a[j]=less
    for i in range(N):


        h[i]=(hequalspacing*a[i]/normalizingfactor)*10

    a[j]=normal
    wi=2
    zetai=0
    sum=0

    sumh=0
    for i in range (N):
        sumh=sumh+h[i]/2
        sum = sum + wi*function(zetai+sumh) *h[i]/2
        sumh=sumh+h[i]/2


    abserrorsin[j]=sum-exactvaluesSINintegral



largest=0
for i in range (N):
    if (abserrorsin[i]<0):
        abserrorsin[i]=-abserrorsin[j]

    if (abserrorsin[i]>largest):
        largest=abserrorsin[i]


for i in range (N):
    abserrorsin[i]=(largest-abserrorsin[i])*100000000


#largest error is when the smallest interval is near zero which means tha
```

```
#better the integration accuracy is



#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###


#unequal spacing ln


function=log



N=10
hequalspacing=(xN-x0)/N
normal=2
less=0.001

normalizingfactor= normal*9+less

h=[0]*N
a=[normal]*N
abserrorln=[0]*N

for j in range (N):
    a[j]=less
    for i in range(N):


        h[i]=(hequalspacing*a[i]/normalizingfactor)*10

    a[j]=normal
    wi=2
    zetai=0
    sum=0
    sumh=0

    for i in range (N):
        sumh=sumh+h[i]/2
        sum = sum + wi*function(zetai+(i+1)*h[i]/2) *h[i]/2
        sumh=sumh+h[i]/2


    abserrorln[j]=sum-exactvaluesLnintegral
```

```
for i in range (N):
    if (abserrorln[j]<0):
        abserrorln[j]=-abserrorln[j]


largest=0
for i in range (N):
    if (abserrorln[i]<0):
        abserrorln[i]=-abserrorln[j]

    if (abserrorln[i]>largest):
        largest=abserrorln[i]


for i in range (N):
    abserrorln[i]=(largest-abserrorln[i])*100000000


#largest error is when the smallest interval is near 1 which means that t
#worst the integration accuracy is


#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#final integration scheme (PARTD-PARTA)


N10abserrorSIN=abserrorPARTA[10]
N10abserrorLN=abserrorPARTB[1]







function=sin



N=10
hequalspacing=(xN-x0)/N

normalizingfactor= 0

h=[0]*N
```

```
a=[1,1,1,0.9,0.8,0.75,0.7,0.7,0.65,0.64]

for i in range (N):
    normalizingfactor=normalizingfactor+a[i]


for j in range (1):

    for i in range(N):


        h[i]=(hequalspacing*a[i]/normalizingfactor)*10


    wi=2
    zetai=0
    sum=0

    sumh=0
    for i in range (N):
        sumh=sumh+h[i]/2
        sum = sum + wi*function(zetai+sumh) *h[i]/2
        sumh=sumh+h[i]/2


    abserrorsinimproved=sum-exactvaluesSINintegral



largest=0
for i in range (1):
    if (abserrorsinimproved<0):
        abserrorsinimproved=-abserrorsinimproved

N10abserrorSIN
abserrorsinimproved


#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#### #### #### #### #### #### #### #### #### #### #### #### #### #### ###
#final integration scheme (PARTD-PARTB)
```

```
function=log



N=10
hequalspacing=(xN-x0)/N

normalizingfactor= 0

h=[0]*N
a=[0.3,0.5,0.7,1,2,2,3,3,4,6]

for i in range (N):
    normalizingfactor=normalizingfactor+a[i]


for j in range (1):

    for i in range(N):


        h[i]=(hequalspacing*a[i]/normalizingfactor)*10


    wi=2
    zetai=0
    sum=0

    sumh=0
    for i in range (N):
        sumh=sumh+h[i]/2
        sum = sum + wi*function(zetai+sumh) *h[i]/2
        sumh=sumh+h[i]/2


    abserrorlnimproved=sum-exactvaluesLnintegral




for i in range (1):
    if (abserrorlnimproved<0):
        abserrorlnimproved=-abserrorlnimproved

N10abserrorSIN
abserrorlnimproved


#ln improves more by the distribution of h which can be attributed to it'
#considering the curve going to infinity as x>0 which also agrees with ho
```

```
#choosing a spacing scheme which focuses more on that portion of the curv
#curve characteristic leaves room for improvement which this scheme explo
#comparing the abs error
#sin gives an accuracy half way between 10 and 11
#ln gives an accuracy better than 30
```