ECSE 543 Fall 2017
Guanqing Hu 260556970

ECSE 543 Assignment 3

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

B (T)	H (A/m)
0.0	0.0
0.2	14.7
0.4	36.5
0.6	71.7
0.8	121.4
1.0	197.4
1.1	256.2
1.2	348.7
1.3	540.6
1.4	1062.8
1.5	2318.0
1.6	4781.9
1.7	8687.4
1.8	13924.3
1.9	22650.2

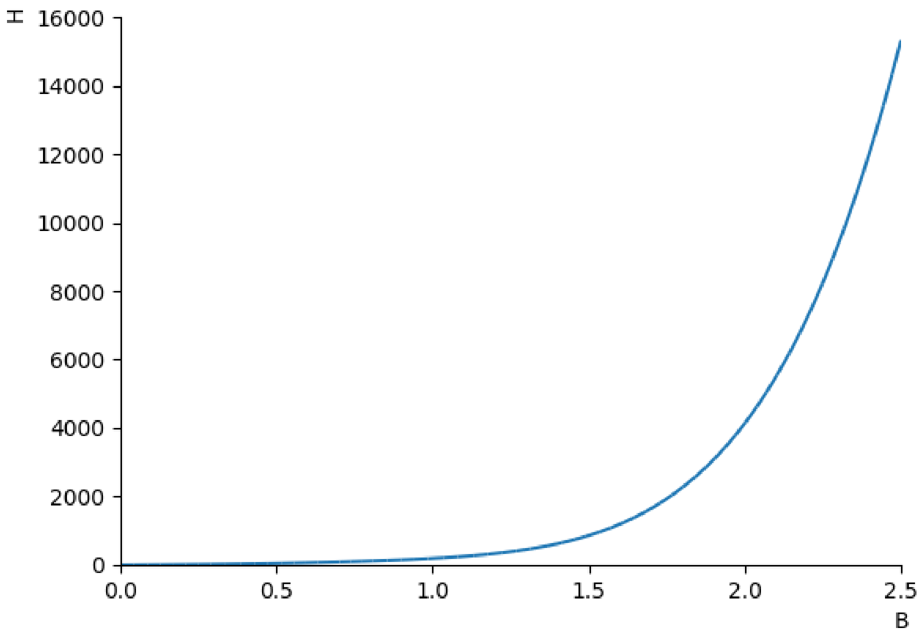
Table 1: BH Data for M19 Steel

(a) Interpolate the first 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?

A python script has been written for this question called interpolation.py. It mainly uses sympy python library for expression calculation and plotting.

The figure below shows the interpolation using full-domain Lagrange on the first 6 points. It lies close to the true B-H curve in this range. The curve equation can be seen in the screenshot.

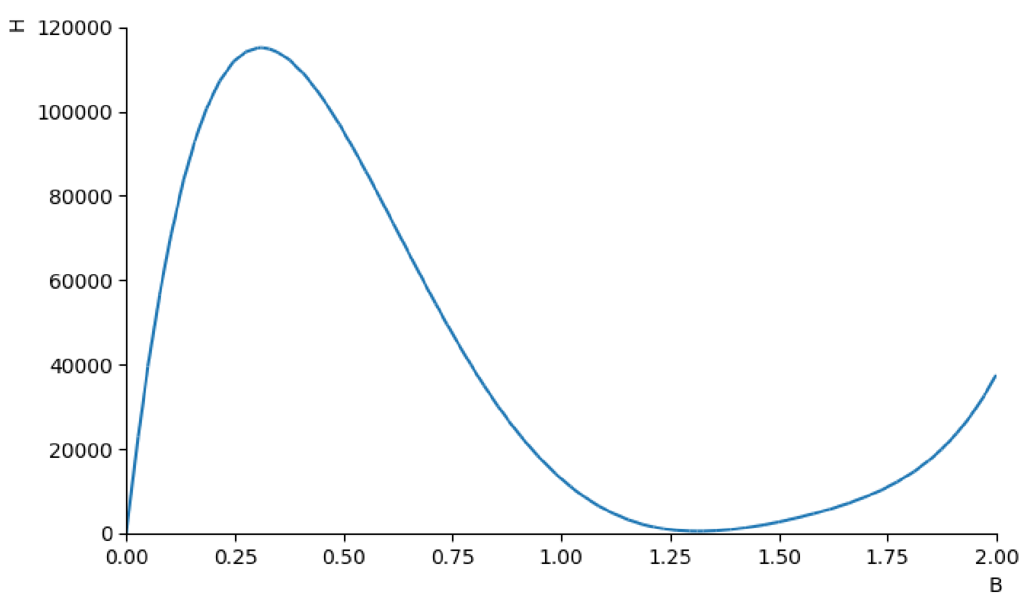
```
Full Lagrange Interpolation 1:  414.0625*x**5 - 963.541666666675*x**4 + 873.437500000004*x**3 - 215.208333333334*x**2
+ 88.6500000000002*x
```



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

The interpolation is not plausible as can be seen in the figure below. It fails to generalize the B-H relation. The curve function can be seen in the screenshot.

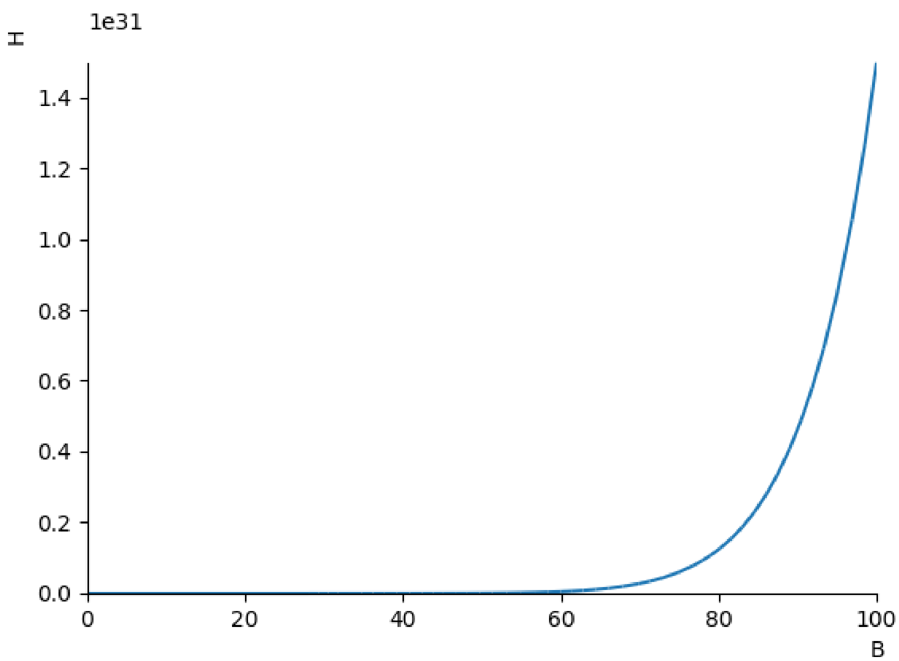
```
Full Lagrange Interpolation 2:  156393.280524088*x**5 - 966235.572245106*x**4 + 2253820.22115057*x**3 - 2337828.829457
73*x**2 + 906781.854422079*x
```



(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.

One way to estimate the slopes at the 6 points is using the slope of the line that connects the point itself and the closest point next to it. Particularly in the code for this question, the slope is calculated using point j and point j+1. The slope at the last point is calculated by using y/x directly. The interpolation can be seen in the figure below. It is able to convey a good B-H relation in a large range of B. The curve equation can be seen in the screenshot.

```
Cubic Hermite Interpolation:  1734143651.25328*x**11 - 25207926897.7831*x**10 + 162399433111.203*x**9 - 608575443897.4
97*x**8 + 1461887595863.9*x**7 - 2334363871440.49*x**6 + 2477827584159.55*x**5 - 1685862719604.6*x**4 + 667146472719.5
82*x**3 - 116995129475.364*x**2 + 415.846153846154*x
```



ECSE 543 Fall 2017
Guanqing Hu 260556970

Question2. The magnetic circuit of Figure 1 has a core made of Ml9 steel, with a cross-sectional area 1 cm^2. L_c = 30 cm and L_a = 0.5 cm. The coil has N = 1000 turns and carries a current 1 = 8 A.

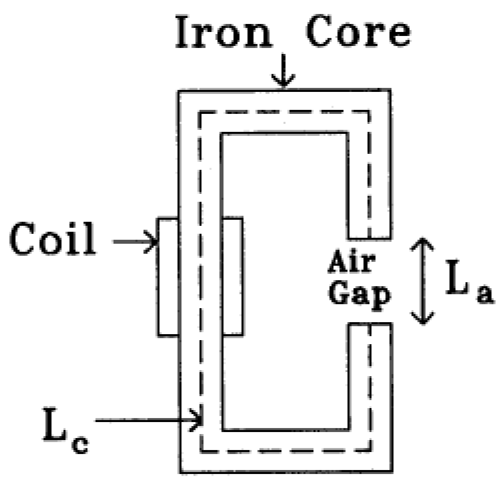


Figure 1

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

The function of $f(\varphi)$ can be derived following the equations below:

$$R_G = \frac{l_G}{\mu_0 A_G} = \frac{0.5cm}{4\pi \times 10^{-7} \times 1cm^2} = 39.788735 \times 10^6 \Omega$$

$$\mu = \frac{B}{H} = \frac{\varphi}{HA}$$

$$\varphi = A \times B$$

$$R_C = \frac{l_C}{\mu \ \ A_C} = \frac{l_C H}{A_C B} = \frac{l_C H}{\varphi}$$

$$NI = \varphi(R_G + R_C)$$

$$NI = \varphi(R_G + \frac{l_C H}{\varphi})$$

$$\boldsymbol{f(\varphi) = R_G \varphi + l_C H - NI = 0}$$

$$f(\varphi) = 39.788735 \times 10^6 \varphi + 0.3H - 8000 = 0$$

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

The python script q2.py contains all the methods for this question, including find_piecewise(), run_newton_raphson(), and run_substitution(). $f(\varphi)$ derived in part (a) of this question contains variable H which can be substituted with the equation below. Then $f(\varphi)$ only contains variable φ.

$$H = piecewise(B) = piecewise(\frac{\varphi}{A})$$

$$f(\varphi) = 39.788735 \times 10^6 \varphi + 0.3 \times piecewise(\frac{\varphi}{1cm^2}) - 8000 = 0$$

The results of Newton-Raphson can be seen in the screenshot below. It took three iterations to reach a satisfied error rate (in screenshot: |f/df| < 10^{-6}). The final flux equals to **161.2694*10^{-6}Wb**.

```
Interation:  0  Flux:  0  f:  -8000.00000000000  df:  40009235.0000000
Interation:  1  Flux:  0.000199953835658192  f:  9356.65541633201  df:  301565735.000000
Interation:  2  Flux:  0.000168926917376737  f:  1201.88062431530  df:  156953735.000000
Interation:  3  Flux:  0.000161269370238306  f:  -4.54747350886464e-13  df:  156953735.000000
```

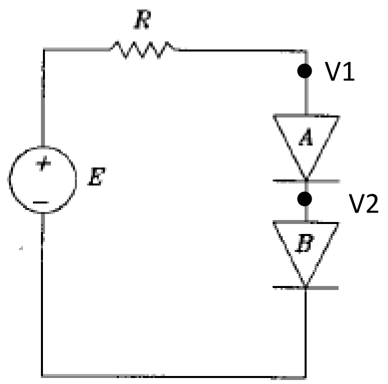
(c) Try solving the same problem with successive substitution. If the method does not converge, suggest and test a modification of the method that *does* converge.

When running $f(\varphi)$ using successive substitution, it diverges to infinity because it is using a step size that is too large compare to Newton-Raphson method where step size is f/f'. In the case of substitution, the step size becomes f which starts at more than 8000 when using $f(\varphi)$. The way to solve this problem is using a smaller step size by dividing the whole f equation by a large enough number, so that the search will not miss the answer point or end up explode or oscillate. After several trying, dividing the whole equation by a factor of 10^8 works the best. The results can be seen below calculated using $f2(\varphi)$. The final flux is **161.7058*10^{-6}Wb**, which is similar to what Newton-Raphson produced.

$$f2(\varphi) = (39.788735 \times 10^6 \varphi + 0.3H - 8000) \times 10^{-8} = 0$$

```
Interation:  0  Flux:  0  f:  -8.00000000000000e-5
Interation:  1  Flux:  8.00000000000000e-5  f:  -4.78048120000000e-5
Interation:  2  Flux:  0.000127804812000000  f:  -2.76526590092318e-5
Interation:  3  Flux:  0.000155457471009232  f:  -7.15743997654106e-6
Interation:  4  Flux:  0.000162614910985773  f:  2.11187645909585e-6
Interation:  5  Flux:  0.000160503034526677  f:  -1.20279252204084e-6
Interation:  6  Flux:  0.000161705827048718  f:  6.85035265602949e-7
```

Question 3. In the circuit shown below, the DC voltage *E* is 220 mV, the resistance *R* is 500 Ω, the diode *A* reverse saturation current I_{sA} is 0.6 μA, the diode *B* reverse saturation current I_{sB} is 1.2 μA, and assume *kT/q* to be 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 $\mathbf{v}_n$.

The following equations explain how vector f was derived from the circuit.

$$I = \frac{E - V1}{R} \quad (1)$$

$$I = I_{sA} \times \left(e^{\frac{(V1-V2)}{V_T}} - 1\right) \quad (2)$$

$$I = I_{sB} \times \left(e^{\frac{V2}{V_T}} - 1\right) \quad (3)$$

$$V_T = \frac{kT}{q}$$

$$f_1 = (1) - (2) = \boldsymbol{\frac{E - V1}{R} - I_{sA} \times \left(e^{\frac{(V1-V2)}{V_T}} - 1\right) = 0}$$

$$f_2 = (1) - (3) = \boldsymbol{\frac{E - V1}{R} - I_{sB} \times \left(e^{\frac{V2}{V_T}} - 1\right) = 0}$$

$$\boldsymbol{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 44.06 \times 10^{-5} - 0.002V1 - 0.6 \times 10^{-6} e^{40(V1-V2)} \\ 44.12 \times 10^{-5} - 0.002V1 - 1.2 \times 10^{-6} e^{40V2} \end{bmatrix} = \mathbf{0}$$

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

A matrix of partial derivatives of f1 and f2 corresponding to V1 and V2 can be calculated as shown in the first equation below. An updating function can be derived from the second equation and as a result shown in the last equation. The product of f' inverse and f is a length-two vector that defines the updating step for V1 and V2. The script for this question can be seen in q3.py which contains two methods: run_newton_raphson() and inverse2b2(). The first method considers the maximum error rate and keeps updating V1 and V2 until converge. The output for this part can be seen in the screenshot below where the maximum error in (f'^{-1}*f) should be smaller than 10^{-5}. The method converges after 5 iterations with V1 = 0.198V and V2 = 0.091V. The convergence is quadratic because as can be seen from the value change of V1 and V2, after each iteration, the precision gets around 2 digits better.

$$\boldsymbol{f'} = \begin{bmatrix} \frac{df_1}{dV1} & \frac{df_1}{dV2} \\ \frac{df_2}{dV1} & \frac{df_2}{dV2} \end{bmatrix}$$

$$\boldsymbol{f'} \cdot \left(\begin{bmatrix} V1_{t+1} \\ V2_{t+1} \end{bmatrix} - \begin{bmatrix} V1_t \\ V2_t \end{bmatrix} \right) + \boldsymbol{f} = \boldsymbol{0}$$

$$\begin{bmatrix} V1_{t+1} \\ V2_{t+1} \end{bmatrix} = \begin{bmatrix} V1_t \\ V2_t \end{bmatrix} - \boldsymbol{f'^{-1}} \cdot \boldsymbol{f}$$

```
Interation:  0  v1:  0  v2:  0  f'*f:  [-0.218253968253968, -0.0727513227513237]
Interation:  1  v1:  0.218253968253968  v2:  0.0727513227513237  f'*f:  [0.0125588781442855, -0.00882970355913752]
Interation:  2  v1:  0.205695090109683  v2:  0.0815810263104612  f'*f:  [0.00558550781939255, -0.00766871017052853]
Interation:  3  v1:  0.200109582290290  v2:  0.0892497364809897  f'*f:  [0.00189852040747677, -0.00126609626670183]
Interation:  4  v1:  0.198211061882813  v2:  0.0905158327476915  f'*f:  [7.69276910460991e-5, -5.47948608176555e-5]
Interation:  5  v1:  0.198134134191767  v2:  0.0905706276085092  f'*f:  [1.25962669349302e-7, -8.02089721338843e-8]
```

Question 4.

(a) Integrate the function cos(x) on the interval x=0 to x=1, by dividing the interval into N equal segments and using one-point Gauss-Legendre integration for each segment. Plot $log_{10}(E)$ versus $log_{10}(N)$ for N=1, 2,..., 20, where E is the absolute error in the computed integral. Comment on the result.

A script q4.py was created for this question. There is only one method integral() which is capable of estimating both even and uneven segmented integrals using one-point Gauss-Legendre method.

One-point Gauss-Legendre is very straightforward to implement. The way it works is to simplify the area under the curve as many rectangle areas whose height can be found by plugging in the middle point value (i.e. h = f((x1+x2)/2)). The error is calculated by setting up a ground truth value first and subtract it from the result. The results for integral of cos(x) between 0 and 1 can be seen in the screenshot below. When number of segments are 20, this method is able to achieve a small error below 10^{-4}.

ECSE 543 Fall 2017
Guanqing Hu 260556970

```
Integral of cos(x):  0.8414709848078965
i =  1  Integral =  0.8775825618903728  Error:  0.036111577082476254
i =  2  Integral =  0.8503006452922328  Error:  0.00882966048433631
i =  3  Integral =  0.8453793458454515  Error:  0.003908361037554986
i =  4  Integral =  0.8436663167025465  Error:  0.0021953318946500433
i =  5  Integral =  0.8428750743698314  Error:  0.0014040895619349403
i =  6  Integral =  0.8424456991964261  Error:  0.0009747143885295585
i =  7  Integral =  0.842186947503467  Error:  0.0007159626955705045
i =  8  Integral =  0.8420190672464982  Error:  0.0005480824386017158
i =  9  Integral =  0.8419039961670828  Error:  0.000433011359186275
i =  10  Integral =  0.8418217000072956  Error:  0.0003507151993991098
i =  11  Integral =  0.841760817405321  Error:  0.00028983259742454415
i =  12  Integral =  0.8417145153208724  Error:  0.0002435305129758758
i =  13  Integral =  0.8416784838788396  Error:  0.00020749907094308462
i =  14  Integral =  0.8416498955690671  Error:  0.00017891076117060312
i =  15  Integral =  0.8416268329703337  Error:  0.0001558481624371888
i =  16  Integral =  0.8416079585815617  Error:  0.00013697377366517216
i =  17  Integral =  0.8415923163990293  Error:  0.00012133159113281167
i =  18  Integral =  0.8415792084113783  Error:  0.00010822360348183846
i =  19  Integral =  0.8415681153452524  Error:  9.713053735593835e-05
i =  20  Integral =  0.8415586444272835  Error:  8.765961938694833e-05
```

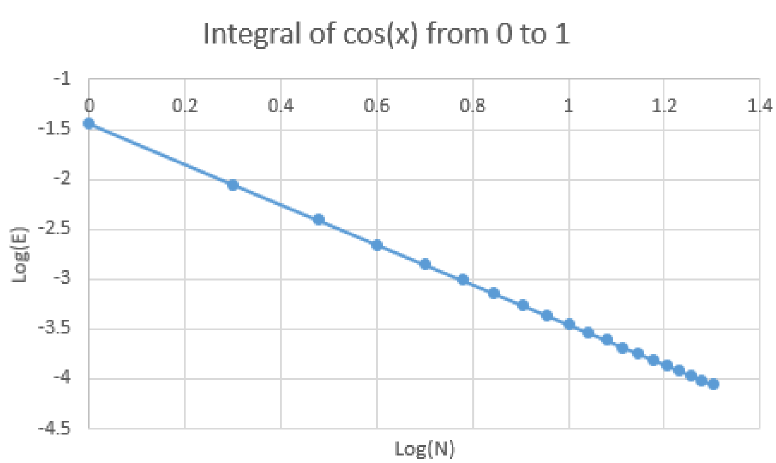
The plotting of log(E)-log(N) is linear, meaning that relation between the number of segments and the integral error is monomial ($y = k*x^b$). The value of the power b can be derived using the equation below.

$$b = slope = \frac{\log(E2) - \log(E1)}{\log(N2) - \log(N1)} = \frac{\log\left(\frac{E2}{E1}\right)}{\log\left(\frac{N2}{N1}\right)} = log_{\frac{N2}{N1}} \frac{E2}{E1}$$

$$\frac{E2}{E1} = \left(\frac{N2}{N1}\right)^b$$

$$b = \frac{-4 + 1.5}{1.3 - 0} = -1.923$$

$$E = k * N^{-1.923}$$



ECSE 543 Fall 2017
Guanqing Hu 260556970

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

The results for ln(x) with required number of segments can be seen in the screenshot below. The error rate decreases much slower than cos(x) because the part of the curve that is closer to 0 changes much more dramatically than the part close to 1.

```
Integral of ln(x):  -1
i =  10  Integral =  -0.9657590653461393  Error:  0.03424093465386069
i =  20  Integral =  -0.982775471973686  Error:  0.017224528026314023
i =  30  Integral =  -0.9884938402873318  Error:  0.011506159712668218
i =  40  Integral =  -0.9913617009604189  Error:  0.008638299039581132
i =  50  Integral =  -0.9930851944722272  Error:  0.006914805527772794
i =  60  Integral =  -0.994235347381881  Error:  0.005764652618118982
i =  70  Integral =  -0.9950574520104222  Error:  0.004942547989577828
i =  80  Integral =  -0.9956743404788297  Error:  0.004325659521170255
i =  90  Integral =  -0.9961543263261001  Error:  0.0038456736738998742
i =  100  Integral =  -0.9965384307395624  Error:  0.0034615692604376136
i =  110  Integral =  -0.996852774507024  Error:  0.003147225492975192
i =  120  Integral =  -0.9971147802544644  Error:  0.002885219745535572
i =  130  Integral =  -0.9973365147802633  Error:  0.0026634852197366943
i =  140  Integral =  -0.9975266001991566  Error:  0.002473399800843379
i =  150  Integral =  -0.9976913612451839  Error:  0.002308638754816128
i =  160  Integral =  -0.9978355426612079  Error:  0.0021644573387792114
i =  170  Integral =  -0.9979627735721436  Error:  0.002037226427856642
i =  180  Integral =  -0.9980758771710266  Error:  0.0019241228289733625
i =  190  Integral =  -0.9981770826716382  Error:  0.0018229173283618172
i =  200  Integral =  -0.9982681737137477  Error:  0.001731826286252347
```

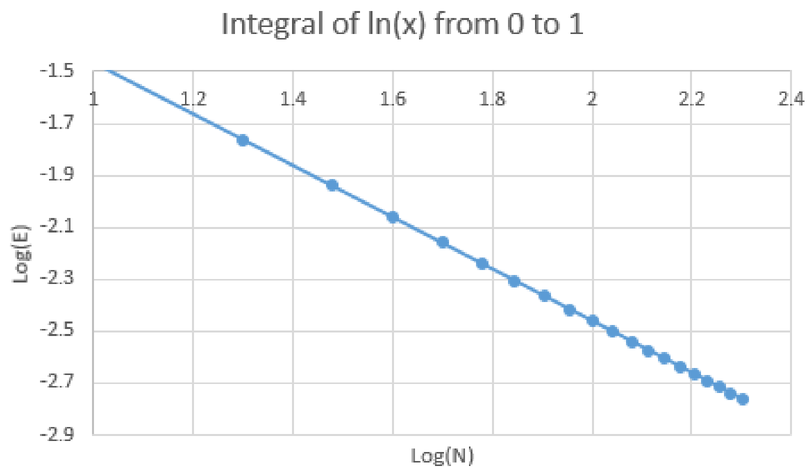
The plotting of log(E)-log(N) is still linear, but N is 10 times larger than the N in cos(x) case. It means that ln(x) is harder to converge than cos(x) using this method. The relation between E and N is still monomial ($y=k*x^b$). Using similar equations, the power b can be derived from the curve.

$$b = slope = \frac{\log(E2) - \log(E1)}{\log(N2) - \log(N1)} = \frac{\log\left(\frac{E2}{E1}\right)}{\log\left(\frac{N2}{N1}\right)} = log_{\frac{N2}{N1}} \frac{E2}{E1}$$

$$\frac{E2}{E1} = \left(\frac{N2}{N1}\right)^b$$

$$b = \frac{-2.8 + 2.1}{2.3 - 1.6} = -1$$

$$E = k * N^{-1}$$



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

A hyper parameter width factor is used to describe the relation between the uneven segments. Particularly, for ln(x), the width of each segment grows exponentially when it approaches 1. For example, when width factor is 2, the width for each segment is 2^i, where i goes from 0 to 9. The width parameter has been tuned to get optimized results. As can be seen in the screenshot below, 1.45 gives the best result with error equals to 0.0106 which is better than the performance of using 30 even segments.

```
Uneven integral
Integral of ln(x):
Width factor =  1  Integral =  -0.9657590653461393  Error:  0.03424093465386069
Width factor =  1.2  Integral =  -0.9839342905695285  Error:  0.016065709430471475
Width factor =  1.3  Integral =  -0.9876810320433765  Error:  0.012318967956623461
Width factor =  1.4  Integral =  -0.9892127563552104  Error:  0.010787243644789557
Width factor =  1.45  Integral =  -0.9893885632877436  Error:  0.010611436712256395
Width factor =  1.5  Integral =  -0.9892686631035535  Error:  0.010731336896446453
Width factor =  1.6  Integral =  -0.9883585018915269  Error:  0.011641498108473147
Width factor =  1.7  Integral =  -0.9868194958070355  Error:  0.01318050419296446
Width factor =  1.8  Integral =  -0.9848715109595413  Error:  0.015128489040458715
Width factor =  1.9  Integral =  -0.9826580718449364  Error:  0.017341928155063635
Width factor =  2  Integral =  -0.9802738362302624  Error:  0.01972616376973757
```

ECSE 543 Fall 2017
Guanqing Hu 260556970

Appendix

interpolation.py

```
from sympy import symbols, expand, diff, lambdify
from sympy.plotting import plot

def full_lagrange_interpolate(X, Y):
    # lagrange dimension decided by length of X and Y
    n = len(X)
    x = symbols('x')
    Ls = []
    # find Fj for each j in n
    for j in range(n):
        X_sub = X[:j]+X[j+1:]
        F = 1
        for xr in X_sub:
            F *= x-xr
        F_f = lambdify(x, F)
        L = expand(F/F_f(X[j]))
        Ls.append(L)
    y = 0
```

```
    for j, L in enumerate(Ls):
        y += Y[j]*L
    return expand(y)


def cubic_hermite(X,Y):
    n = len(X)
    x = symbols('x')
    Us = []
    Vs = []
    for j in range(n):
        X_sub = X[:j] + X[j + 1:]
        F = 1
        for xr in X_sub:
            F *= x - xr
        F_f = lambdify(x, F)
        L = F / F_f(X[j])
        L_d = lambdify(x, diff(L))
        U = (1 - 2 * L_d(X[j]) * (x - X[j]))*(L**2)
        V = (x - X[j])*(L**2)
        Us.append(U)
        Vs.append(V)
    y = 0
    # initial b[j] = y'(j)
    b = [(Y[j+1]-Y[j])/(X[j+1]-X[j]) for j in range(n-1)]
    b.append(Y[-1]/X[-1])
    print(b)
    for j in range(n):
        y += Y[j]*Us[j] + b[j]*Vs[j]
    return expand(y)


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]

x = symbols('x')
lag = full_lagrange_interpolate(B[:6],H[:6])
p1 = plot(lag,(x,0,2.5),xlabel='B',ylabel='H')
print('Full Lagrange Interpolation 1: ',lag)

lag2 = full_lagrange_interpolate(B[:1]+B[8:10]+B[12:],H[:1]+H[8:10]+H[12:])
p2 = plot(lag2, (x,0,2),xlabel='B',ylabel='H')
print('Full Lagrange Interpolation 2: ',lag2)

cub = cubic_hermite(B[:1]+B[8:10]+B[12:],H[:1]+H[8:10]+H[12:])
p2 = plot(cub, (x,0,100),xlabel='B',ylabel='H')
print('Cubic Hermite Interpolation: ',cub)
```

ECSE 543 Fall 2017
Guanqing Hu 260556970

q2.py

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

x = symbols('x')
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]


def find_piecewise(X,Y):
    f = 0
    for j in range(len(X)-1):
        x1 = X[j]
        x2 = X[j+1]
        y1 = Y[j]
        y2 = Y[j+1]
        k = (y2-y1)/(x2-x1)
        b = y2 - k*x2
        if j==0:
            g = Piecewise((0, x >= x2), (k * x + b, True))
        elif j==len(X)-2:
            g = Piecewise((0, x < x1), (k * x + b, True))
        else:
            g = Piecewise((0, x<x1),(0, x>=x2),(k*x+b, True))
        #print x1,x2,y1,y2
        #print g.subs(x,x2)
        f += g
    #f = lambdify(x, f)
    return f

def run_newton_raphson(B,H,err):
    #find piecewise curve of B and H
    h = find_piecewise(B, H)
    print('Piecewise linear function: ', h)
    #initial flux and iterration counter
    flux = 0
    i = 0
    #initial reduction function F and its derivative
    F = 39.788735e6 * x + 0.3 * h.subs(x, x / 1e-4) - 8000
    dF = diff(F)
    while True:
        f = F.subs(x,flux)
        df = dF.subs(x,flux)
        #print(i, flux, f, df)
        print('Interation: ',i, ' Flux: ',flux, ' f: ',f, ' df: ',df)
        if (abs(f/df)<err):
            break
```

```
        else:
            i += 1
            flux += -f/df


def run_substitution(B,H,err):
    #find piecewise curve of B and H
    h = find_piecewise(B, H)
    print('Piecewise linear function: ', h)
    #initial flux and iterration counter
    flux = 0
    i = 0
    #initial reduction function F and its derivative
    F = 39.788735e6 * x + 0.3 * h.subs(x, x / 1e-4) - 8000
    F /= 1e8
    while True:
        f = F.subs(x,flux)
        #print(i, flux, f, df)
        print('Interation: ',i, ' Flux: ',flux, ' f: ',f)
        if (abs(f)<err):
            break
        else:
            i += 1
            flux += -f

err = 1e-6

run_newton_raphson(B,H,err)

run_substitution(B,H,err)
```

q3.py

```
from sympy import symbols, expand, diff, lambdify
from sympy.functions import exp

v1,v2 = symbols('v1 v2')


def inverse2b2(a,b,c,d):
    det = a*d-b*c
    #check if possible
    if det==0:
        print('Matrix not invertible.')
        return [None,None,None,None]
```

```
    else:
        return [d/det,-1*b/det,-1*c/det, a/det]


def run_newton_raphson(err):
    #initial node voltages and iterration counter
    v = [0,0]
    i = 0
    #initial reduction function F and its derivative
    f1 = 44.06e-5 - 0.002 * v1 - 6e-7 * exp(40 * (v1 - v2))
    f2 = 44.12e-5 - 0.002 * v1 - 12e-7 * exp(40 * v2)
    df11 = diff(f1,v1)
    df12 = diff(f1,v2)
    df21 = diff(f2,v1)
    df22 = diff(f2,v2)
    while True:
        f1t = f1.subs([(v1,v[0]),(v2,v[1])])
        f2t = f2.subs([(v1, v[0]), (v2, v[1])])
        df11t = df11.subs([(v1, v[0]), (v2, v[1])])
        df12t = df12.subs([(v1, v[0]), (v2, v[1])])
        df21t = df21.subs([(v1, v[0]), (v2, v[1])])
        df22t = df22.subs([(v1, v[0]), (v2, v[1])])
        inv = inverse2b2(df11t, df12t, df21t, df22t)
        r = [inv[0]*f1t+inv[1]*f2t, inv[2]*f1t+inv[3]*f2t]
        #print('Interation: ',i, ' v1: ',v[0], ' v2: ',v[1], ' f\'*f: ',r)
        print(i, ',', v[0], ',', v[1], ',', r[0], ',', r[1])
        if (abs(max(r))<err):
            break
        else:
            i += 1
            v[0] += -r[0]
            v[1] += -r[1]

run_newton_raphson(1e-15)
```

q4.py

```
import math

def integral(func1,a, b, even=True, width_factor=10):
    if even == True:
        w = float(b - a) / float(width_factor)
        widths = [w]*int(width_factor)
    else:
        relativeWidths = [width_factor**i for i in range (0,10)]
```

```
        a,b = float(a),float(b)
        scale = (b - a) / sum(relativeWidths)
        widths = [width * scale for width in relativeWidths]
    summation = 0
    for w in widths:
        lowLim = a
        a += w
        highLim = a
        #try:
        #    height = (func1(lowLim) + func1(highLim)) / 2
        #except:
        height = func1((lowLim + highLim) / 2)
        summation += (highLim - lowLim) * height
    return summation

ground_truth = math.sin(1) + math.sin(0)
print ("Integral of cos(x): ", ground_truth)
for i in range (1,21):
    result =  integral (math.cos,0,1,True,i)
    #print ("i = ", i,  " Integral = ", result, " Error: ", result-ground_truth)
    print(result - ground_truth)

ground_truth = -1
print ("Integral of ln(x): ", ground_truth)
for i in range (10, 210, 10):
    result =  integral (math.log,0,1,True,i)
    #print ("i = " ,i, " Integral = ",result, " Error: " , result-ground_truth)
    print(result - ground_truth)

print ('Uneven integral')
print ("Integral of ln(x): ")
for width_factor in [1,1.2,1.3,1.4,1.45,1.5,1.6,1.7,1.8,1.9,2]:
    result =  integral (math.log,0,1,False,width_factor)
    print ("Width factor = ", width_factor," Integral = ",result," Error: ",result - ground_truth)
```