**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.**



**Table 1**

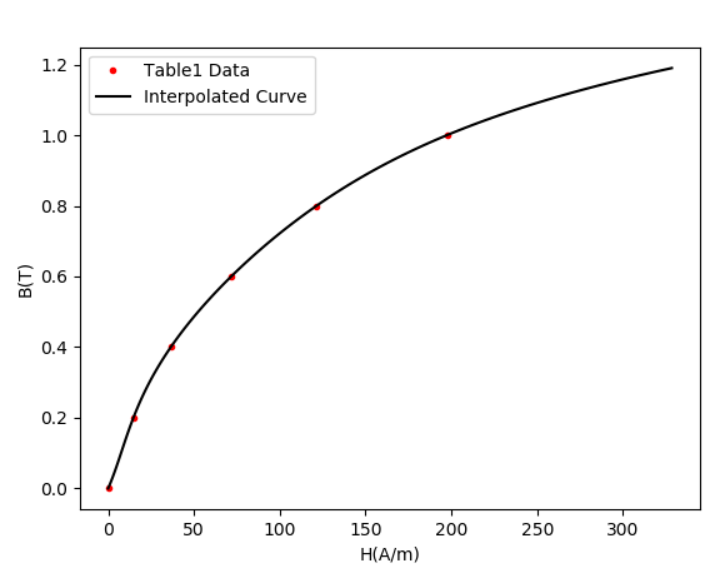
1. **Interpolate the first 6 points using full-domain Lagrange polynomials. Is the result plausible, i.e. do you think it lies close to the true B versus H graph over this range?**

The Lagrange polynomials coefficients are found using:

Then, the approximated value of H is found using



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



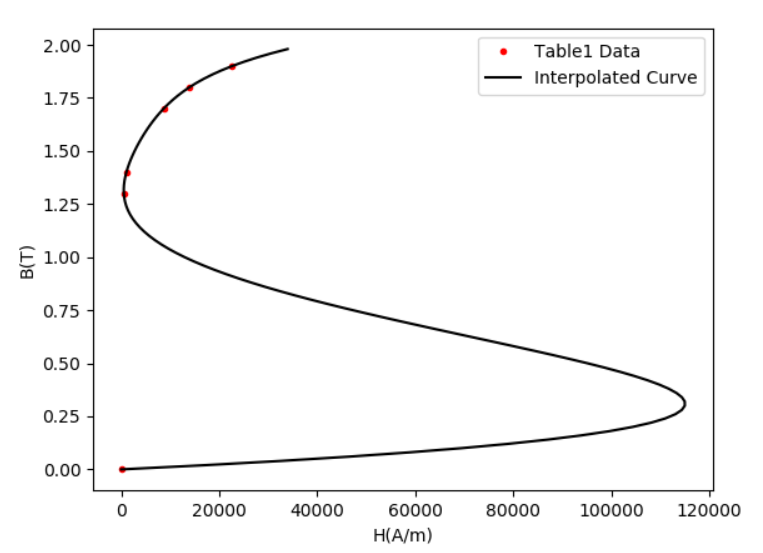
**Figure 1a (2): interpolating with the first 6 points with full-domain Lagrange**

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.

1. **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 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.

1. **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:

Finally, the approximation answer is found using:

Where aj(x) = y(xj), bj(x) = y’(xj)

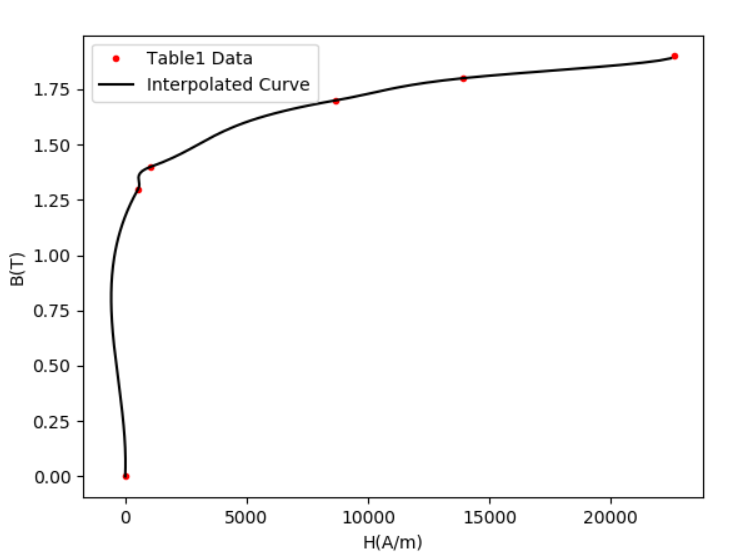
Now we want to fix the remaining 6 degrees of freedom – the slopes at the 6 points.

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 x6) using y(xi+1) – y(xi) / (xi+1 – xi), and for the last end point x6, we approximate its derivative to be y(xi) – y(x0) / (xi+1 – x0). This means, if we define the 5 subdomains between the 6 points as A, B, C, D, E, respectively:

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

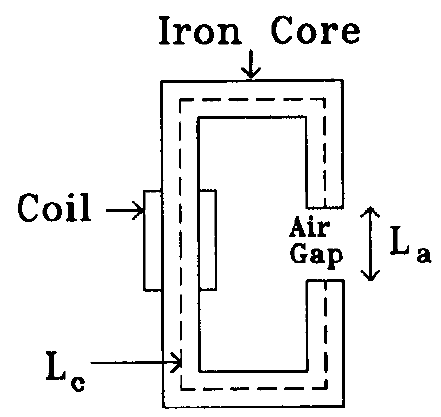


**Figure 1c (1): interpolating with the given 6 given points with cubic Hermite**

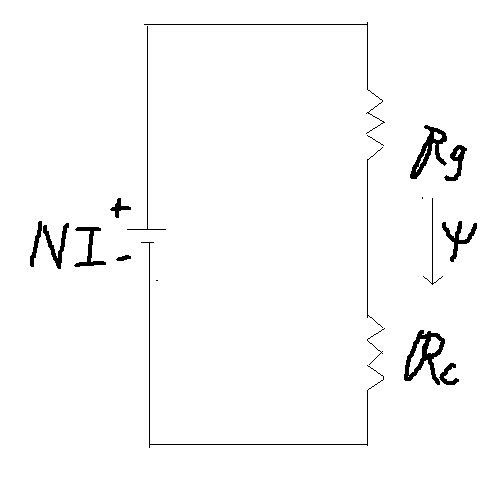
From Figure 1c (1), we can see that the result is much better than in the Figure 1b (2) and this curve seems plausible.

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.

1. **The magnetic circuit of Figure 1 has a core made of Ml9 steel, with a cross-sectional area of 1 cm2. Lc = 30 cm and La = 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:

RG: reluctance of the air gap

RC: reluctance of the steel core

NI: MMF produced by the coil winding

ψ: magnetic flux flowing through the circuit

By Ampere’s Law, NI = RG \* ψ + RC \* ψ

Where , , where µG = µ0, therefore:

Since by definition, ψ = B\*A = H\*A\*µC, we have then µC = ψ/(H\*A)

Then:

Divide both sides by , we get

Plug in the values, we get

1. **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\*107\*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

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



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

1. **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

Which means:

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: ， We can get H(Bψ) = (NI - \*ψ)/LC, which means Bψ = H-1((NI - \*ψ)/LC)

Since Bψ = ψ/A, we get finally ψ - A\* H-1((NI - \*ψ)/LC) = 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 - \*ψ)/LC. 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)

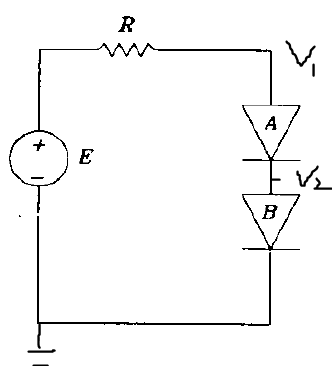
So up to now we have with H-1 known.

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



**Figure 1f (1): successive substitution simulation result**

1. **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 IsA = 0.8 uA, the reverse saturation current for diode B is IsB = 1.1 uA, and assume kT/q = 25 mV.**



**Figure 2**

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

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

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

By KVL we have

1. **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 ]**

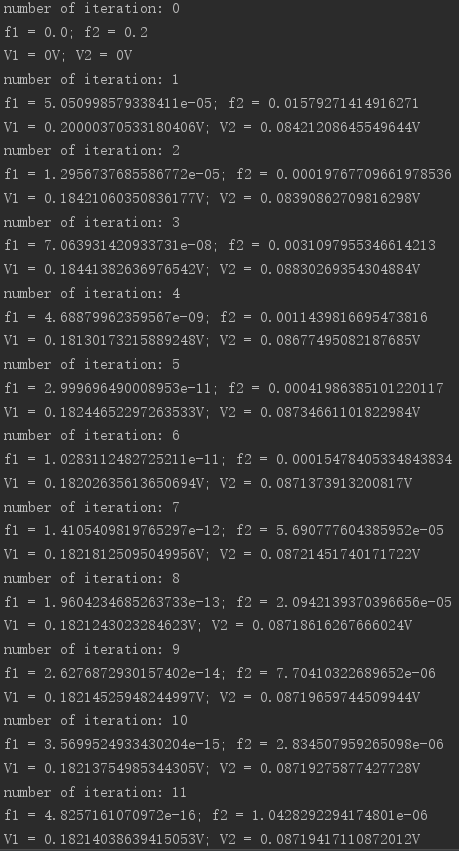
Combine f1 and f2 we get a matrix:

And the Jacobian matrix is given by:

Then we can compute:

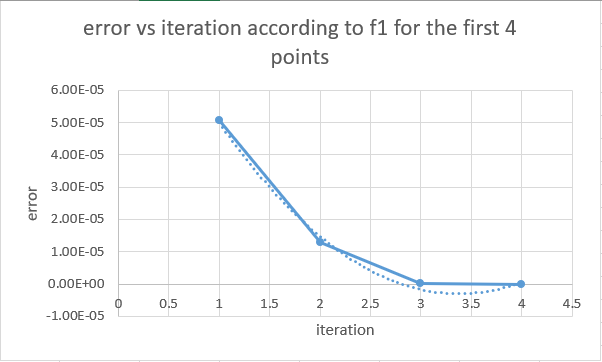
, where .

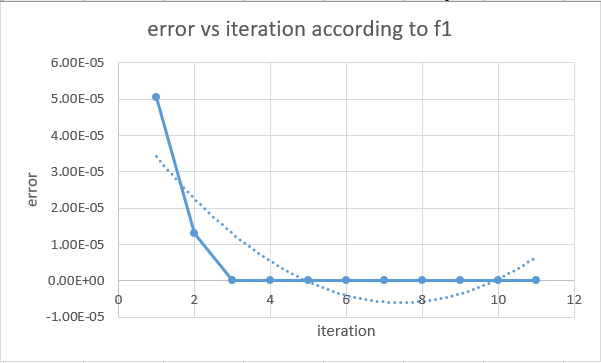
Setting the  **to be (f1(V1, V2) / f1(0, 0)) < 10-8** for the simulation break condition (if we use (f2(V1, V2) / f2(0, 0)), since f2(0, 0) = 0, it will return an error since the denominator cannot be 0), and the result of f and v is:



**Figure 2b (1): Newton-Raphson result**

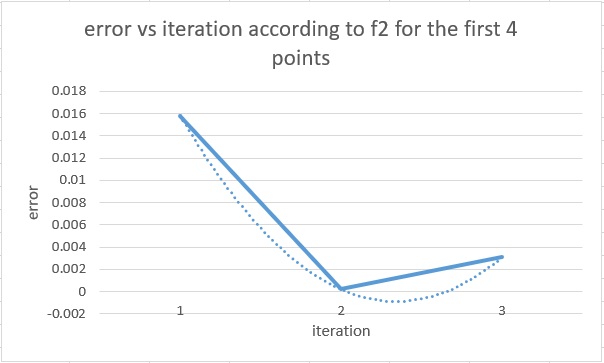
Now since f1 and f2 is supposed to be 0, their recorded values are actually the errors, and if we plot the f1 and f2 from iteration n = 1 we get the following results:

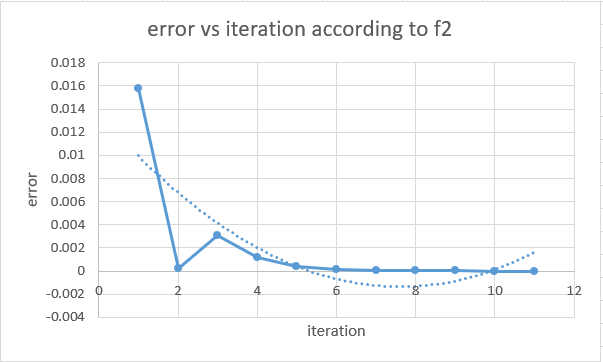




**Figure 2b (2): error according to f1 with ① 4 points ② the whole iteration**

For these plots I added a quadratic trend line and we can see from the error according to f1 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 f2 with ① 4 points ② the whole iteration**

The arguments are the same as above, we can see from the error according to f2 that the first several iterations seem a **quadratic convergence**, and after that the function converges to 0, but in a much slower speed.

1. **Write a program that accepts as input the values for the parameters x0, xN, and N and integrates a function f(x) on the interval x = x0 to x = xN by dividing the interval into N equal segments and using one-point Gauss-Legendre integration for each segment.**

For an integral , in Gauss-Legendre integration method, we can write it as

=

Where wi is the weight, and the f(xi) is the value of the function evaluated at xi. 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:

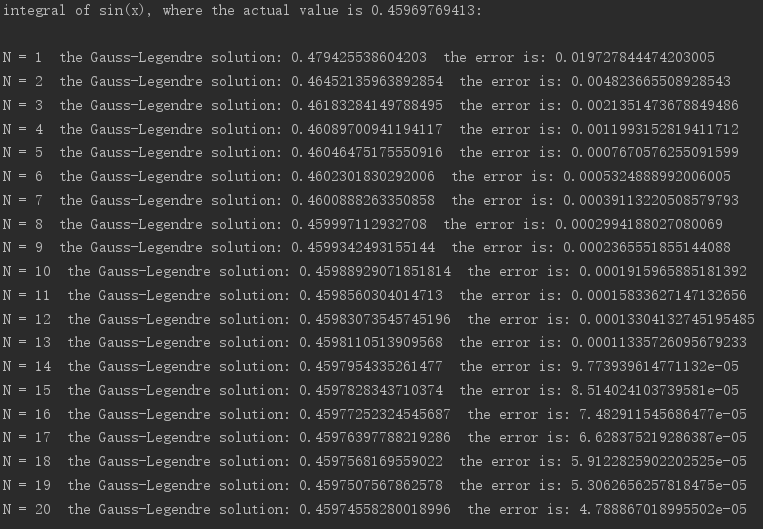
[xstart, xstart + h], we can have xi to be xstart + 1/2\*h, which means it is the mid-point of the region. Meanwhile, as the interval is equally distributed, we should have:

wi = h.

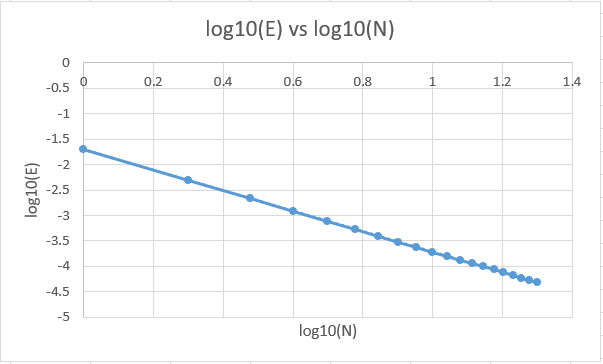
1. **Use your program to integrate the function f(x) = sin(x) on the interval x0 = 0 to xN = 1 for N = 1, 2, …, 20. Plot log10(E) versus log10(N) for N=1,2, …,20, where E is the absolute error in the computed integral. Comment on the result.**

First of all, the actual integral is:

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



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



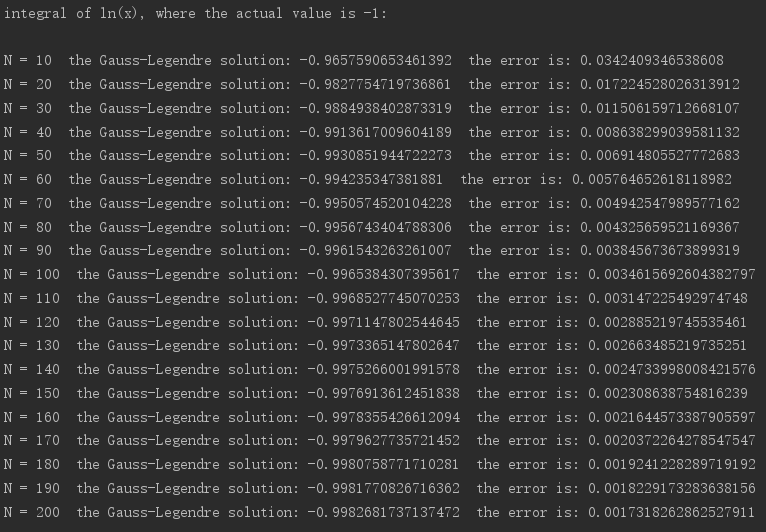
**Figure 3a (2): log10(E) versus log10(N)**

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 log10(E) decreases **linearly** with respect to log10(N). This can mean that there are diminishing returns when using a very high value of N

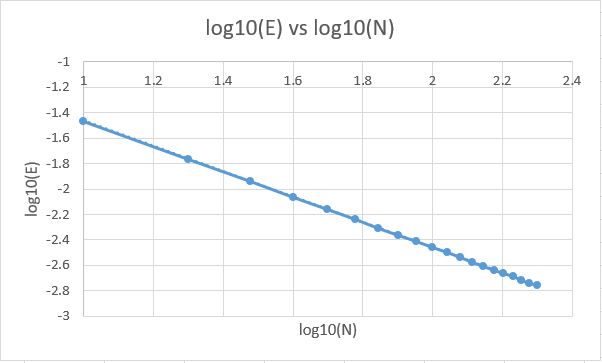
1. **Repeat part (a) for the function f(x) =** **ln(x), only this time for N = 10, 20, …, 200. Comment on the result.**

First of all, the actual integral is:

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



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



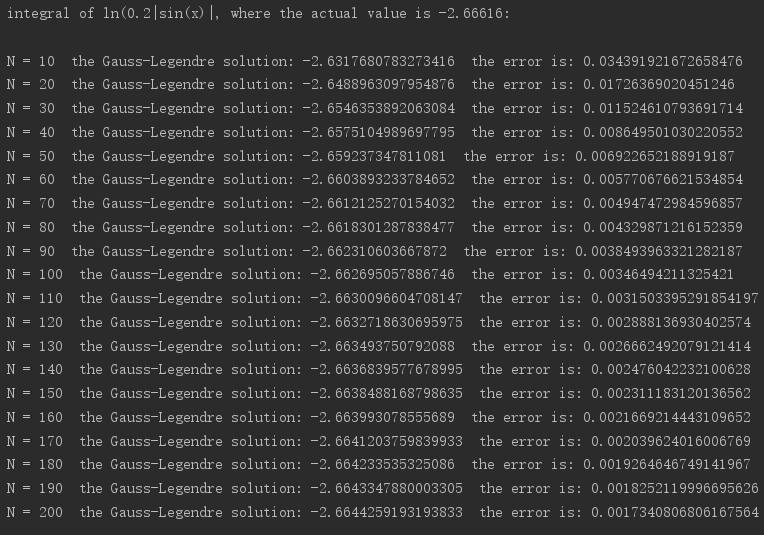
**Figure 3b (2): log10(E) versus log10(N)**

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 log10(E) decreases **linearly** with respect to log10(N) (This can mean that there are diminishing returns when using a very high value of N).

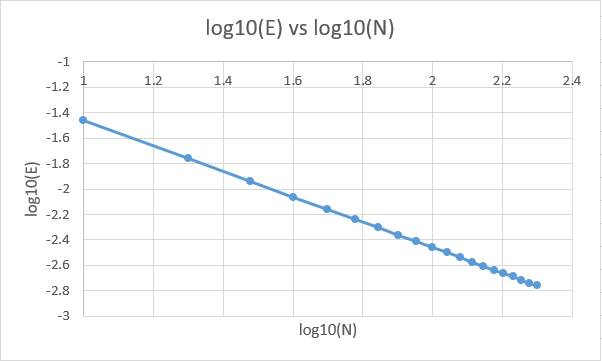
1. **Repeat part (b) for the function f(x) = ln(0.2|sin(x)|). Comment on the result.**

First of all, the actual integral is:

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



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

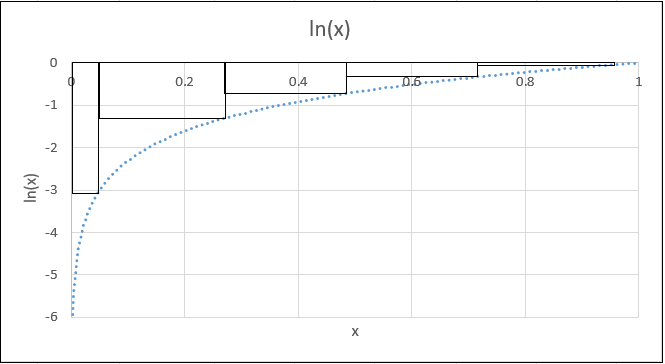


**Figure 3c (2): log10(E) versus log10(N)**

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 log10(E) decreases **linearly** with respect to log10(N) (This can mean that there are diminishing returns when using a very high value of N).

1. **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.**

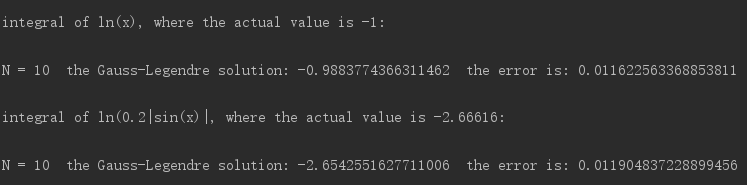
Since for b) and c), ln is a function that has a derivative of 1/x, which means its slope decreases when x increases(|sin(x)| increases with x when x is in region [0, 1] so ln(x) and ln(|sin(x)|) should have the same behavior in region [0, 1]). So, the ‘difficult’ part should lie in the region that x 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] \*hs, where hs is the smallest interval size.

The result is as below:



**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")

1. **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()

1. **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")

1. **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)))