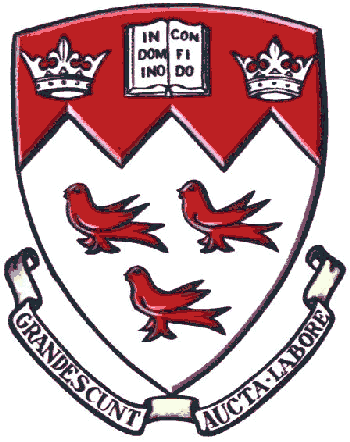
****

ECSE 543 – Numerical Methods

Assignment 3

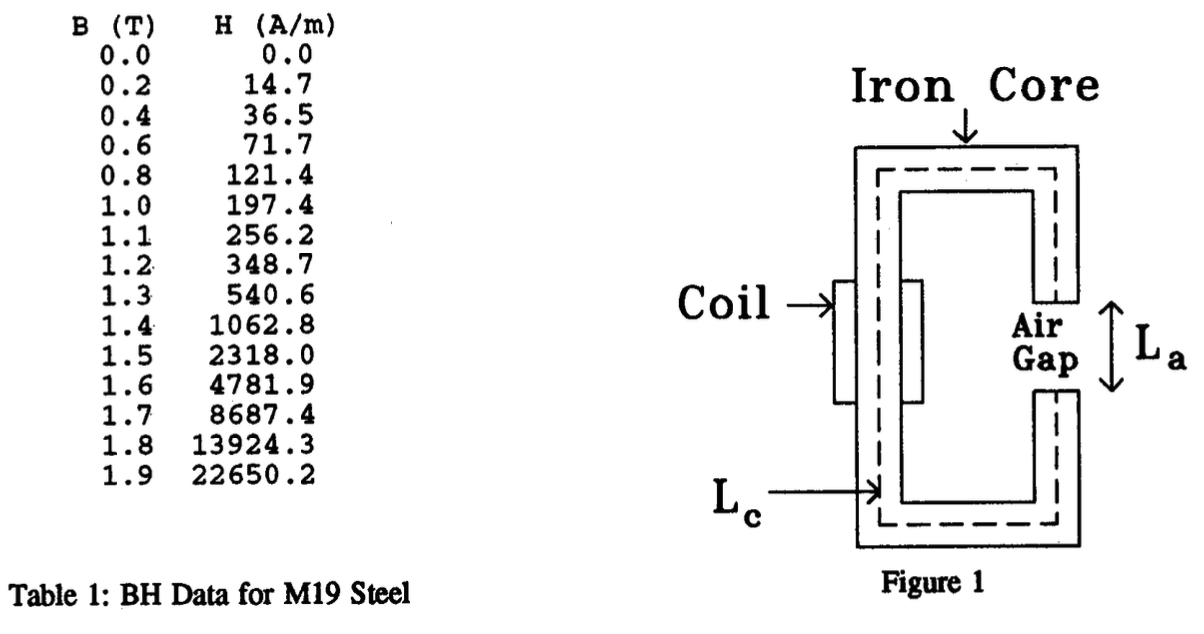
Sharhad Bashar

260519664

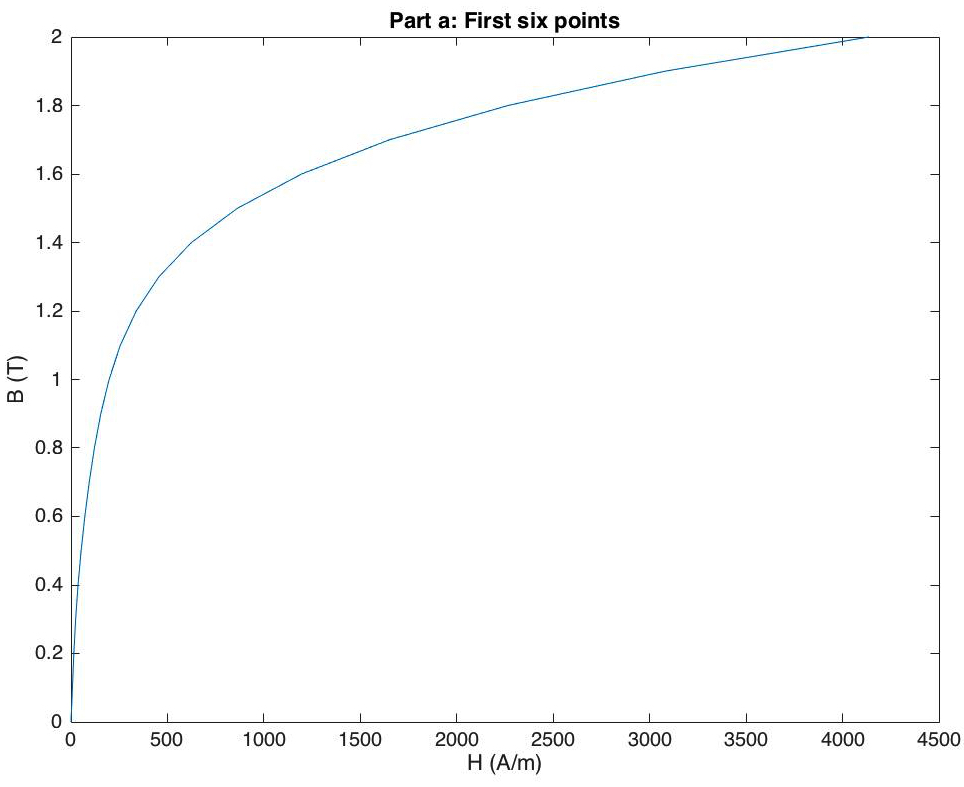
Dec 5th, 2016

**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.**

****

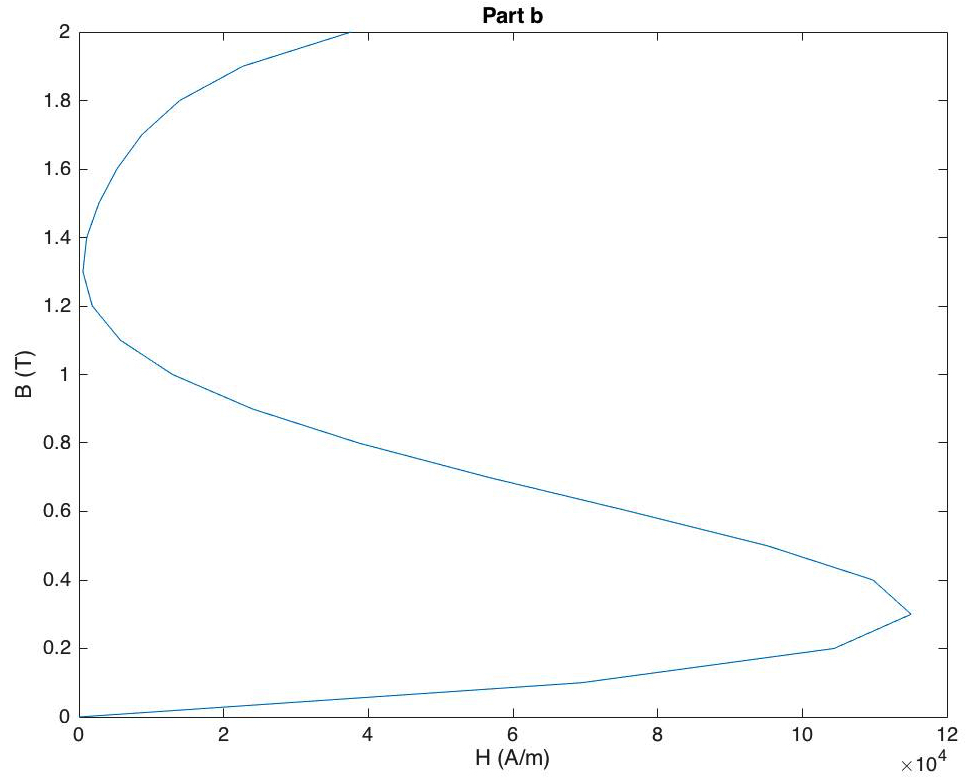
1. **Interpolate the first 6 points using full-domain Lagrange polynomials. Is the result plausible, i.e. do you think it lies close to the true B versus H graph over this range?**

****The code for this section is found in interpolate.py. A function was created that interpolates the given points and produces a graph of the points. For the first six points, the following chart was produced:

As we can see, the graph is really smooth, and there are no sharp turns or squiggles, and thus we can be pretty sure that the interpolation result is plausible and a good representation.

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?**

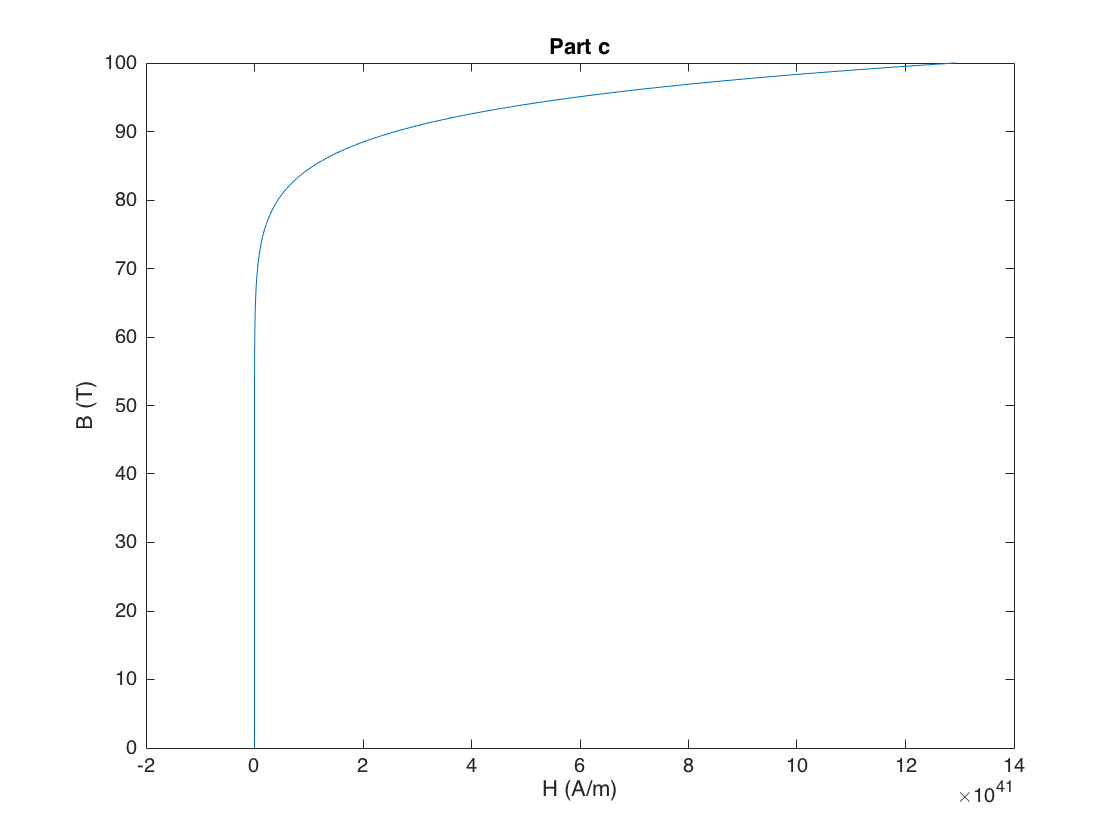
The code is found in interpolation.py. The graph that was generated is as follows:



The above graph was produced once the points were interpolated. The shape of the curve does not represent a B vs H graph, even though it passes through all the points provided. We can conclude that the result is NOT plausible.

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.**

The code for cubic hermite is in interpolate.py. The function cubicHer takes in the H and B points, and produces a polynomial for the function, following the steps outlined in the lecture CV. The following graph was produced.

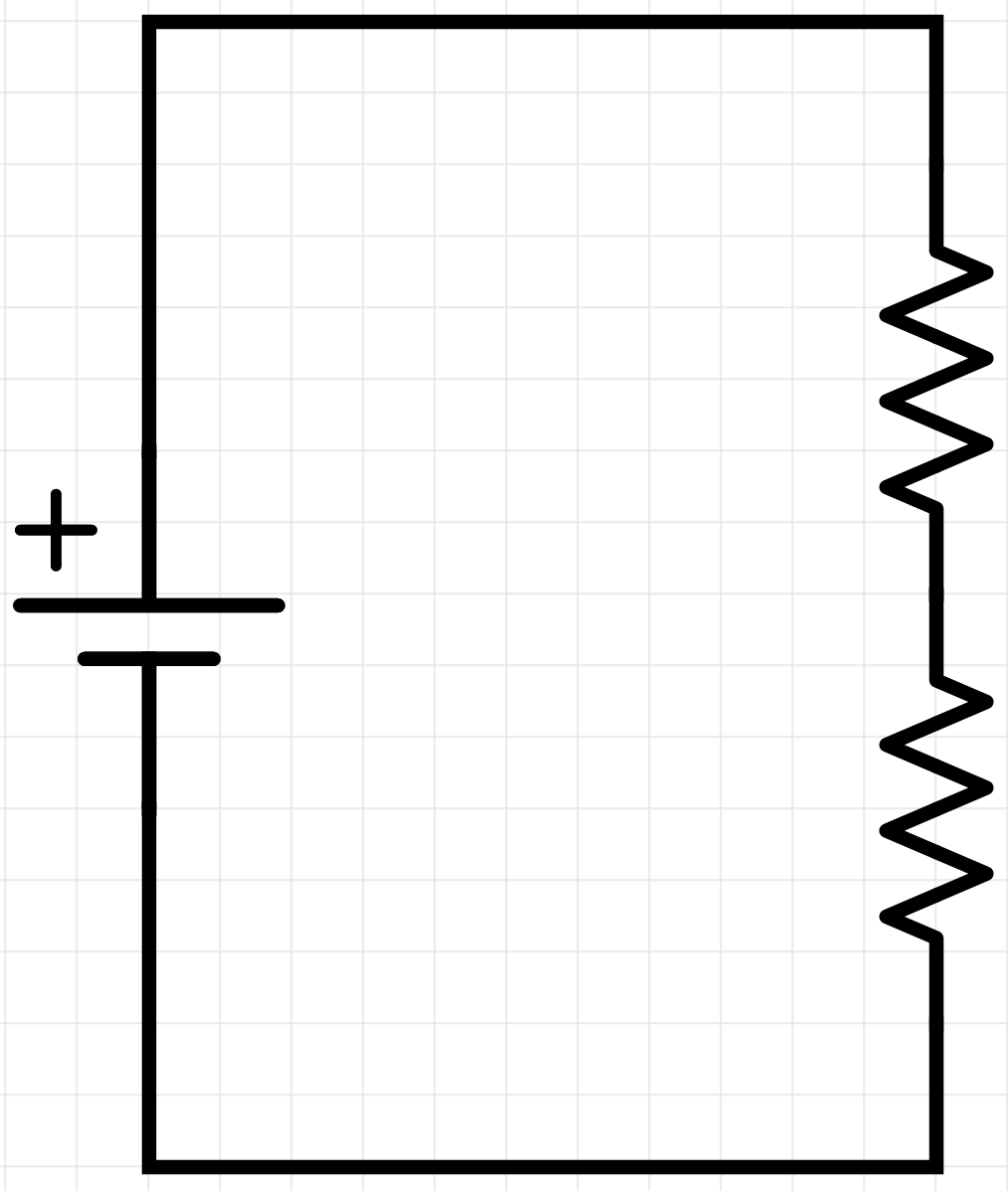
****

As we can see, the function is much smoother here, and we can conclude that cubic hermite produces a better curve, which matches the original function.

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 y in the core, of the form f(y) = 0.**

Using knowledge from ECSE 361 (Power engineering), the following equivalent circuit was produced. The steps for calculation of the equations and the values are outlined below:

RC



RG

NI

Substituting equations (4) and (5) into (3), gives us the following:

Expanding and simplifying the above equation gives us:

Plugging in the values provided to us, generates the following nonlinear equation for the flux in the core:

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(y)/f(0)|<10-6. Record the final flux, and the number of steps taken.**

The code for this can be found in M19circuit.py

We used the above equation and the steps outlined in lecture NL to solve for the flux.

Where H’ is the change in slope calculated from the values in the table provided to us.

Performing Netwon Raphson, the following result was achieved:

**Iterations: 3 Flux: 161.269369447\*10-6 Wb**

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.**

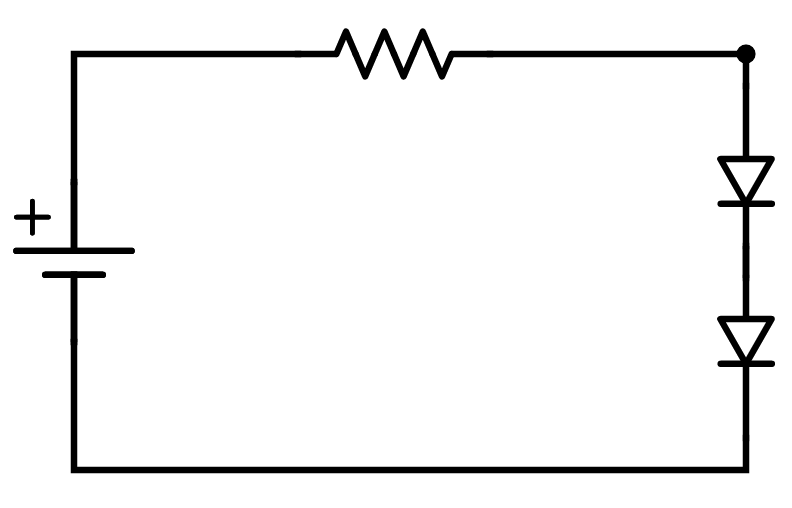
The code for this is also found in M19circuit.py

The steps for successive substitution is outlined in the lecture NL, and the function was created using that.

At first it was not converging, because of a division by zero, so I changed the initial conditions, so the initial flux is really small instead of zero.

And by doing so, I did get the same answer as part e.

**Question 2**

**For the circuit shown below, the DC voltage E is 200 mV, the resistance R is 512** Ω**, the reverse saturation current for diode A is IsA = 0.8 μA, the reverse saturation current for diode B is IsB = 1.1 μA, and assume kT/q = 25 mV.**

R

DA

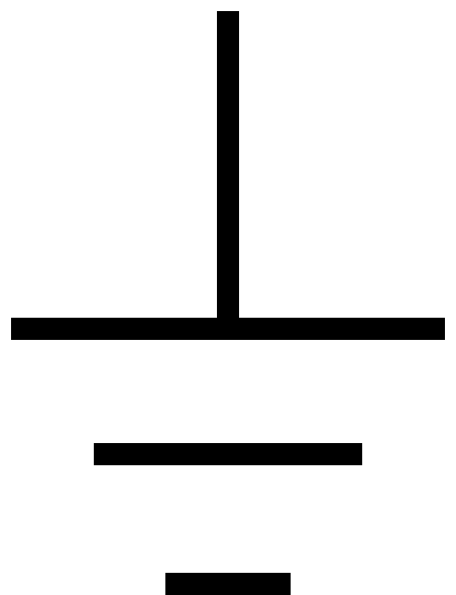
DB

E

I

V1

V2



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.**

First, we know the following relation for the current I:

And since all the elements are connected in series, the same current flows through them all. So the current I through the resistance and the diodes are the same.

Therefore, we get the following equations:

Diode A :

Diode B :

Equating equations (1) and (2), we get:

Solving for V1, we get:

And we know that the current through the diodes are also equal, giving us the following equation:

I = IA = IB

Rearranging equation (5), we get:

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 ]**

The code for solving the function is provided in newtonRaphson.py. To perform basic matrix operations such as inverse, matrix multiplication and addition are imported from basicDefinitions.py, which was created for assignment 1. f1, f2 and the jacobian was calculated by hand, and inputted into the code. The jacobian is shown below:

The equation we have to solve to get the voltages at the nodes are:

Rearranging the above equation (1), we get:

The following table shows the voltage and f values at each step:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **k** | **f(vn)** | | **Vn** | |
| **f1** | **f2** | **V1** | **V2** |
| 0 | -0.2 | 0 | 0 | 0 |
| 1 | 0.058998671 | 0.000102706 | 0.198374913 | 0.066124971 |
| 2 | 0.015365126 | 2.84E-05 | 0.188435007 | 0.076313899 |
| 3 | 0.002043571 | 3.12E-06 | 0.184348331 | 0.082579242 |
| 4 | 3.55E-05 | 5.69E-08 | 0.183446499 | 0.083262944 |
| 5 | 1.16E-08 | 1.76E-11 | 0.183430837 | 0.083276563 |
| 6 | 1.17E-15 | 1.87E-18 | 0.183430832 | 0.083276567 |
| 7 | -1.73E-17 | -1.36E-20 | 0.183430832 | 0.083276567 |

Overall, the voltage across Diode A: V1 – V2 = 100.15 mV

Diode B: V2 – 0 = 83.28 mV

The error function looks as follows:

And as we can see, the curve is indeed quadratic

**Question 3**

**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.**

The code for the Integrations is found in gaussLegendreOnePoint.py. There are two functions.

The first, “integral()” calculates the integral using N equal segments. Its inputs are the functions that we are going to calculate the integral for, the limits a and b, and finally, n, which is the number of segments.

The second, “integralUneven()”, takes the functions as its inputs, as well as the limits a and b.

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.**

The table below shows a few things. First it shows the value of i, which is the number of segments. Secondly, it shows the value it calculated after computing the integral for the given number of segments i, and finally, it shows error, which is how far off the computed value is to the actual value. The actual value is given in the first line, beside the function.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Integral of sin(x) = 0.45970** | | | | |
| **N** | **Integral** | **Error** | **log(N)** | **log(E)** |
| 1 | 0.4794255 | 0.0197255 | 0 | -1.7049711 |
| 2 | 0.4645214 | 0.0048214 | 0.301029996 | -2.3168305 |
| 3 | 0.4618328 | 0.0021328 | 0.477121255 | -2.6710414 |
| 4 | 0.4608970 | 0.0011970 | 0.602059991 | -2.9219024 |
| 5 | 0.4604648 | 0.0007648 | 0.698970004 | -3.1164795 |
| 6 | 0.4607484 | 0.0004478 | 0.77815125 | -3.3489313 |
| 7 | 0.4595501 | 0.0003258 | 0.84509804 | -3.4870475 |
| 8 | 0.4599971 | 0.0002971 | 0.903089987 | -3.5270784 |
| 9 | 0.4599342 | 0.0002342 | 0.954242509 | -3.6303217 |
| 10 | 0.4549631 | 0.0001869 | 1 | -3.7283172 |
| 11 | 0.4598560 | 0.0001560 | 1.041392685 | -3.8067908 |
| 12 | 0.4598307 | 0.0001307 | 1.079181246 | -3.8836066 |
| 13 | 0. 4597123 | 0.0001025 | 1.113943352 | -3.9891777 |
| 14 | 0. 4597307 | 0.0000864 | 1.146128036 | -4.0635373 |
| 15 | 0.4597002 | 0.0000640 | 1.176091259 | -4.1939701 |
| 16 | 0.4597725 | 0.0000615 | 1.204119983 | -4.2108411 |
| 17 | 0.4597640 | 0.0000574 | 1.230448921 | -4.2414652 |
| 18 | 0.4597568 | 0.0000508 | 1.255272505 | -4.2939914 |
| 19 | 0.5047716 | 0.0000456 | 1.278753601 | -4.3411990 |
| 20 | 0.4597456 | 0.0000457 | 1.301029996 | -4.3403535 |

The values of log(N) and log(E), which is the log value of the error are computed in the columns beside. The graph below shows log(E) vs log(N) :

The error obviously decreases with increasing N, but when the axes are scaled logarithmically, the error becomes more or less a linear function. The linearity means that increasing the number of segments N, does not reduce the absolute error of the integral.

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

The table below shows the same items as for part a:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Integral of ln(x) = -1** | | | | |
| **N** | **Integral** | **Error** | **log(N)** | **log(E)** |
| 10 | -0.9608800 | 0.0391200 | 1 | -1.4076017 |
| 20 | -0.9827755 | 0.0172245 | 1.301029996 | -1.7638527 |
| 30 | -0.9879429 | 0.0120571 | 1.477121255 | -1.9187558 |
| 40 | -0.9913617 | 0.0086383 | 1.602059991 | -2.0635718 |
| 50 | -0.9930852 | 0.0069148 | 1.698970004 | -2.1602200 |
| 60 | -0.9942353 | 0.0057647 | 1.77815125 | -2.2392269 |
| 70 | -0.9949558 | 0.0050442 | 1.84509804 | -2.2972055 |
| 80 | -0.9955965 | 0.0044035 | 1.903089987 | -2.3561979 |
| 90 | -0.9960928 | 0.0039072 | 1.954242509 | -2.4081309 |
| 100 | -0.9965384 | 0.0034616 | 2 | -2.4607270 |
| 110 | -0.9968115 | 0.0031885 | 2.041392685 | -2.4964198 |
| 120 | -0.9970801 | 0.0029199 | 2.079181246 | -2.5346365 |
| 130 | -0.9973070 | 0.0026930 | 2.113943352 | -2.5697613 |
| 140 | -0.9975011 | 0.0024989 | 2.146128036 | -2.6022573 |
| 150 | -0.9976914 | 0.0023086 | 2.176091259 | -2.6366440 |
| 160 | -0.9978160 | 0.0021840 | 2.204119983 | -2.6607557 |
| 170 | -0.9979455 | 0.0020545 | 2.230448921 | -2.6872934 |
| 180 | -0.9980605 | 0.0019395 | 2.255272505 | -2.7123027 |
| 190 | -0.9981771 | 0.0018229 | 2.278753601 | -2.7392330 |
| 200 | -0.9982682 | 0.0017318 | 2.301029996 | -2.7614957 |

The log of N and errors were graphed, and as we can see bellow, produces a similar linear relation as part a.

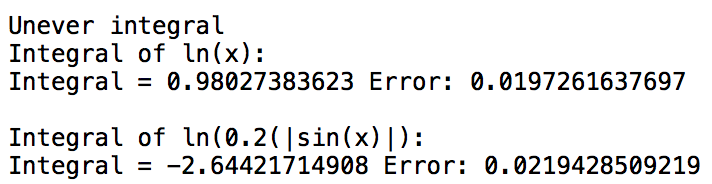
1. **Repeat part (b) for the function f(x) = ln(0.2|sin(x)1|). Comment on the result.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Integral of ln(0.2(|sin(x)|) : -2.66616** | | | | |
| **N** | **Integral** | **Error:** | **log(N)** | **log(E)** |
| 10 | -2.3446046 | 0.3215554 | 1 | -0.4927442 |
| 20 | -2.4994320 | 0.1667280 | 1.301029996 | -0.7779916 |
| 30 | -2.5535800 | 0.1125800 | 1.477121255 | -0.9485387 |
| 40 | -2.5811784 | 0.0849816 | 1.602059991 | -1.0706751 |
| 50 | -2.5979101 | 0.0682499 | 1.698970004 | -1.1658978 |
| 60 | -2.6100165 | 0.0561435 | 1.77815125 | -1.2507002 |
| 70 | -2.6157180 | 0.0504420 | 1.84509804 | -1.2972075 |
| 80 | -2.6232539 | 0.0429061 | 1.903089987 | -1.3674811 |
| 90 | -2.6279347 | 0.0382253 | 1.954242509 | -1.4176491 |
| 100 | -2.6289799 | 0.0371801 | 2 | -1.4296895 |
| 110 | -2.6348724 | 0.0312876 | 2.041392685 | -1.5046275 |
| 120 | -2.6349164 | 0.0312436 | 2.079181246 | -1.5052384 |
| 130 | -2.6374627 | 0.0286973 | 2.113943352 | -1.5421585 |
| 140 | -2.6372534 | 0.0289066 | 2.146128036 | -1.5390027 |
| 150 | -2.6395996 | 0.0265604 | 2.176091259 | -1.5757648 |
| 160 | -2.6409193 | 0.0252407 | 2.204119983 | -1.5978985 |
| 170 | -2.6446017 | 0.0215583 | 2.230448921 | -1.6663863 |
| 180 | -2.6458640 | 0.0202960 | 2.255272505 | -1.6925892 |
| 190 | -2.6469866 | 0.0191734 | 2.278753601 | -1.7173005 |
| 200 | -2.6488963 | 0.0172637 | 2.301029996 | -1.7628664 |

Similarly, the graph produces a linear relationship. This shows, that even if we integrate complicated functions, the resulting graph produced from log(N) vs log(E) remains the same

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.**

The segment width was decreased, as we got closer to the result. The values computed from 10 segments are as follows:

****

As we can see, for ln(x), using 10 segments, we get an error of 0.0197, where as using equal widths for segments, as done in part b) the error was 0.039. This shows that using this method gives more accurate results with less segments. The same thing is observed when we try to compute the integral of ln(0.2|sin(x)|). The error is only 0.0219 with uneven segment width, where as in part c) the error for 10 segments was 0.321

Appendix

#!/usr/bin/env python2

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

ECSE 543

Assignment 3

interpolate.py

Contains the functions to perform Lagrange Interpolation and Cubic Hermite

"""

from sympy import \*

def diff(f,x):

n = len(x)

y = []

for i in range(n):

if (i == n - 1):

y.append((f[i]-f[i-1])/(x[2]-x[1]))

else:

y.append((f[i+1]-f[i]/(x[2]-x[1])))

return y

def interLag(pointsX, pointsY):

x = symbols('x')

n = len(pointsX)

Ljx = [None for j in range (n)]

for j in range (n):

Fjx = 1

Fjxj = 1

yx = 0

for k in range (n):

if (k != j):

Fjx \*= (x - pointsX[k])

Fjxj \*= (pointsX[j] - pointsX[k])

Ljx[j] = Fjx.expand()/Fjxj

# Now we have al the Lj(x)

for j in range (n):

yx += pointsY[j] \* Ljx[j]

return yx

def cubicHer(pointsX, pointsY):

x = symbols('x')

n = len(pointsX)

Ljx = [None for j in range (n)]

LjxPrime = [None for j in range (n)]

Ujx = [None for j in range (n)]

Vjx = [None for j in range (n)]

b = [None for j in range (n)]

yx = 0

for j in range (n):

Fjx = 1

Fjxj = 1

for k in range (n):

if (k != j):

Fjx \*= (x - pointsX[k])

Fjxj \*= (pointsX[j] - pointsX[k])

Ljx[j] = Fjx.expand()/Fjxj

#now we have Lj(x)

for j in range (n):

LjxPrime[j] = Ljx[j].diff(x)

#now we have Lj'(x)

for j in range(n):

Ujx[j] = ((1 - 2 \* (LjxPrime[j] \* (x - pointsX[j]))) \* (Ljx[j] \* Ljx[j])).expand()

#now we have Uj(x)

Vjx[j] = ((x - pointsX[j]) \* (Ljx[j] \* Ljx[j])).expand()

#now we have Uj(x)

if (j < 5):

b[j] = (pointsY[j + 1] - pointsY[j]) / (pointsX[j + 1] - pointsX[j])

elif (j == 5):

b[j] = pointsY[j]/pointsX[j]

#now we have the bj

for j in range (n):

yx += pointsY[j] \* Ujx[j] + b[j] \* Vjx[j]

return yx

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]

a = interLag (B[:6],H[:6])

b = interLag (Bb,Hb)

print ("First Six points: " + str(a))

print ('\n')

print ("Part b: " + str(b))

cubic = (cubicHer(Bb, Hb))

print cubic

%Plot the interpolated points

%First six points

a = @(x) 414.062500000009\*x.^5 - 963.54166666669\*x.^4 + 873.437500000015\*x.^3 - 215.208333333338\*x.^2 + 88.6500000000005\*x;

%The six points given in part b

b = @(x) 156393.280524088\*x.^5 - 966235.57224511\*x.^4 + 2253820.22115058\*x.^3 - 2337828.82945774\*x.^2 + 906781.854422079\*x;

%Cubic Hermite

c= @(x)1568971513379.42\*x.^15 - 30309713506701.9\*x.^14 + 268567396219957.0\*x.^13 - 1.44432330293827e+15\*x.^12 + 5.25560062780712e+15\*x.^11 - 1.36509778845371e+16\*x.^10 + 2.60061934819287e+16\*x.^9 - 3.67319109930033e+16\*x.^8 + 3.83801274237492e+16\*x.^7 - 2.9210617380455e+16\*x.^6 + 1.56706833559662e+16\*x.^5 - 5.57371221336095e+15\*x.^4 + 1.16503319968491e+15\*x.^3 - 105922354752559.0\*x.^2 + 415.846153846154\*x;

%%%%%%%%%%%%%%%%%%%%%%%%%%%

x\_1 = 0:0.1:2;

x\_2 = 0:0.1:100;

figure;

plot(a(x\_1),x\_1);

title('Part a: First six points');

xlabel('H (A/m)');

ylabel('B (T)');

figure;

plot(b(x\_1),x\_1);

title('Part b');

xlabel('H (A/m)');

ylabel('B (T)');

figure();

plot(c(x\_2),x\_2);

title('Part c');

xlabel('H (A/m)');

ylabel('B (T)');

%%%%%%%%%%%%%%%%%%%%%%%%%%%

#!/usr/bin/env python2

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

ECSE 543

Assignment 3

M19circuit.py

Contains the functions to calculate the flux of the M19 circuit

"""

HvB = [[0.0, 0.0],[0.2, 14.7],[0.4,36.5],[0.6,71.7],[0.8,121.4],[1,197.4],[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]]

def Hval(flux):

B = flux/(1.0/pow(100,2))

# interpolate for values outside domain

if B > HvB[-1][0]:

slope = (HvB[-1][1] - HvB[-2][1]) / (HvB[-1][0] - HvB[-2][0])

return (B - HvB[-1][0]) \* slope + HvB[-1][1]

for i in range(len(HvB)):

if HvB[i][0] > B:

slope = (HvB[i][1] - HvB[i-1][1]) / (HvB[i][0] - HvB[i-1][0])

return (B - HvB[i-1][0]) \* slope + HvB[i-1][1]

else: # must be smaller

slope = (HvB[1][1] - HvB[0][1]) / (HvB[1][0] - HvB[0][0])

return (B - HvB[0][0]) \* slope + HvB[0][1]

def Hder(flux):

B = flux/(1.0/pow(100,2))

if B > HvB[-1][0]:

return (HvB[-1][1] - HvB[-2][1]) / (HvB[-1][0] - HvB[-2][0])

for i in range(len(HvB)):

if HvB[i][0] > B:

slope = (HvB[i][1] - HvB[i-1][1]) / (HvB[i][0] - HvB[i-1][0])

return slope

else: # must be smaller

slope = (HvB[1][1] - HvB[0][1]) / (HvB[1][0] - HvB[0][0])

return slope

def fFlux(flux):

return 3.978873577e7 \* flux + 0.3 \* Hval(flux) - 8000

def fFluxDer(flux):

return 3.978873577e7 + 0.3 \* Hderivative(flux)/(1.0/pow(100,2))

def newRap(x, tolerance):

i = 0

while abs(fFlux(x)/fFlux(0)) > tolerance:

i += 1

x -= fFlux(x)/fFluxDer(x)

print('Iterations: ' + str(i) + ' Flux: ' + str(x))

return x

def fSubstitution(flux):

return 8000/(39.78873577e6 + 0.3 \* Hval(flux)/flux)

def succesSub (x,tolerance):

i = 0

while abs(fFlux(x)/fFlux(0)) > tolerance:

i += 1

x = fSubstitution(x)

print('Iterations: ' + str(i) + ' Flux: ' + str(x))

return x

NR = newtonRaph(0.0, 1e-6)

#!/usr/bin/env python2

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

260519664

ECSE 543

Assignment 3

newtonRaphson.py

Computing the voltages using Newton Raphson

Dec 5th, 2016

"""

import math

from basicDefinitions import matrixAddorSub, matrixMult, matInverse,is1Dor2D,matTranspose, scalarMult

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

def newRap (E,R,V1,V2,Isa,Isb,k):

f1 = V1 - E + R \* Isa \* (math.exp((V1 - V2) / 0.025) - 1.0)

f2 = Isa \* ((math.exp((V1 - V2) / 0.025) - 1.0)) - Isb \* (math.exp(V2 / 0.025) - 1.0)

print ('f1 : ' + str(f1))

print ('f2 : ' + str(f2))

print ('Number of iterations : ' + str(k))

print('')

f1V1Prime = 1.0 + (R \* Isa / 0.025) \* (math.exp((V1 - V2) / 0.025))

f1V2Prime = -1.0 \* (R \* Isa / 0.025) \* (math.exp((V1 - V2) / 0.025))

f2V1Prime = (Isa / 0.025) \* (math.exp((V1 - V2) / 0.025))

f2V2Prime = -1.0 \* (Isa / 0.025) \* (math.exp((V1 - V2) / 0.025)) - (Isb / 0.025) \* (math.exp(V2 / 0.025))

V = [[V1],[V2]]

f = [[f1],[f2]]

J = [[None for x in range (2)] for y in range(2)]

J[0][0] = f1V1Prime

J[0][1] = f1V2Prime

J[1][0] = f2V1Prime

J[1][1] = f2V2Prime

invJ = matInverse(J)

V = matrixAddorSub(matTranspose(scalarMult(-1.0,matrixMult(invJ,f))),V,'a')

print ('V1 : ' + str(V[0][0])+ ' V')

print ('V2 : ' + str(V[1][0])+ ' V')

return V

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

E = 0.2

R = 512.0

Isa = 0.0000006

Isb = 0.0000012

V1 = 0.0

V2 = 0.0

k = 0

print ('V1 : ' + str(V1)+ ' V')

print ('V2 : ' + str(V2)+ ' V')

V = newRap(E,R,V1,V2,Isa,Isb,k)

f1 = V[0][0] - E + R \* Isa \*(math.exp((V[0][0] - V[1][0])/0.025) - 1.0)

while (f1 > 0):

k += 1

V = newRap(E,R,V[0][0],V[1][0],Isa,Isb,k)

f1 = V[0][0] - E + R \* Isa \*(math.exp((V[0][0] - V[1][0])/0.025) - 1.0)

f1 = V[0][0] - E + R \* Isa \* (math.exp((V[0][0] - V[1][0]) / 0.025) - 1.0)

f2 = Isa \* ((math.exp((V[0][0] - V[1][0]) / 0.025) - 1.0)) - Isb \* (math.exp(V[1][0] / 0.025) - 1.0)

print ('f1 : ' + str(f1))

print ('f2 : ' + str(f2))

print ('Number of iterations : ' + str(k + 1))

#!/usr/bin/env python2

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

260519664

ECSE 543

Assignment 3

gaussLegendreOnePoint.py

Computing the Integral using Gauss Legendre One Point method

Dec 5th, 2016

"""

import math

def integral(func1,func2,n,a,b):

segments = n

n,a,b = float(n),float(a),float(b)

wi = 2.0

xi = 0.0

summation = 0.0

segSize = (b - a) / float(segments)

if (func2 == 1):

while (a < b):

lowLim = a

a += segSize

highLim = a

summation += (highLim - lowLim) \* func1((highLim + lowLim)/2.0 \* xi + (highLim + lowLim)/2.0)

else:

while (a < b):

lowLim = a

a += segSize

highLim = a

summation += (highLim - lowLim) \* func2 (0.2 \* func1((highLim + lowLim)/2.0 \* xi + (highLim + lowLim)/2.0))

return summation

def integralUneven(func1,func2, a, b):

relativeWidths = [2\*\*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]

runningWidth = 0

runningSum = 0

if (func2 == 1):

for w in widths:

runningSum += func1(w/2 + runningWidth) \* w

runningWidth += w

else:

for w in widths:

runningSum += func2(0.2\*abs(func1(w/2 + runningWidth))) \* w

runningWidth += w

return runningSum

print ("Integral of sin(x): 0.45970")

for i in range (1,21):

error = 0.45970

integ = integral (math.sin,1,i,0,1)

print ("i = " + str(i) + " Integral = " + str(integ) + " Error: " + str(integ - error))

print ('')

print ("Integral of ln(x): -1")

for i in range (1,21):

error = -1

integ = integral (math.log,1,i \* 10,0,1)

print ("i = " + str(i\*10) + " Integral = " + str(integ) + " Error: " + str(integ - error))

print ('')

print ("Integral of ln(0.2(|sin(x)|): -2.66616")

for i in range (1,21):

error = -2.66616

integ = integral(math.sin,math.log,i,0,1)

print ("i = " + str(i\*10) + " Integral = " + str(integ) + " Error: " + str(integ - error))

print ('')

print ('Unever integral')

print ("Integral of ln(x): ")

integ = integralUneven (math.log,1,0,1)

error = -1

print ("Integral = " + str(abs(integ)) + " Error: " + str(integ - error))

print ('')

print ("Integral of ln(0.2(|sin(x)|): ")

integ = integralUneven(math.sin,math.log,0,1)

error = -2.66616

print ("Integral = " + str(integ) + " Error: " + str(integ - error))

print ('')

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

260519664

ECSE 543

Assignment 3

basicDefinitions.py

A list of functions, used for this assignment

Dec 5th, 2016

"""

import math

from scipy import random

import numpy as np

import csv

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

#Function that checks if a matrix is 1D or 2D

def is1Dor2D (A):

while True:

try:

length = len(A[0]) #If true, A is 2D

return A

break

except TypeError: #else A is 1D

return [A]

break

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

#function that creates floats from lists

def list2float(A):

length = len(A)

floatA = [0 for x in range(length)]

for i in range (length):

numVal = ''

stringVal = list(str(A[i]))

for j in range (1,len(stringVal)-1,1):

numVal = numVal + stringVal[j]

floatVal = float(numVal)

floatA [i] = floatVal

return floatA

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

#Function that transposes a Matrix

def matTranspose(A):

A = is1Dor2D(A)

rowsA = len(A)

colsA = len(A[0])

C = [[0 for rows in range (rowsA)] for cols in range(colsA)]

for i in range (colsA):

for j in range (rowsA):

C[i][j] = A[j][i]

return C

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

#Matrix that performs Inverse for a 2D matrix

def matInverse(A):

detA = A[0][0] \* A[1][1] - A[0][1] \* A[1][0]

invA = [[None for x in range (len(A))] for y in range(len(A))]

invA[0][0] = A[1][1] / detA

invA[1][1] = A[0][0] / detA

invA[0][1] = -1 \* A[0][1] / detA

invA[1][0] = -1 \* A[1][0] / detA

return invA

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

#Function that adds or subtracts two matricies

def matrixAddorSub(A,B,operation):

A = is1Dor2D(A)

B = is1Dor2D(B)

rowsA = len(A)

colsA = len(A[0])

rowsB = len(B)

colsB = len(B[0])

if (rowsA == rowsB and colsA == colsB):

C = [[0 for row in range(colsA)] for col in range(rowsA)]

if (operation == 'a'):

for i in range (rowsA):

for j in range (colsA):

C[i][j] = A[i][j]+B[i][j]

elif (operation == 's'):

for i in range (rowsA):

for j in range (colsA):

C[i][j] = A[i][j]-B[i][j]

return C

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

#function for scalar matrix multiplication

def scalarMult(scalar, A):

newMat = [0 for x in range (len(A))]

for i in range (len(A)):

newMat[i] = scalar\*A[i][0]

return newMat

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

#Function that multiplies two matricies

def matrixMult (A, B):

A = is1Dor2D(A)

B = is1Dor2D(B)

rowsA = len(A)

colsA = len(A[0])

rowsB = len(B)

colsB = len(B[0])

if (rowsA == colsB or colsA == rowsB):

C = [[0 for row in range(colsB)] for col in range(rowsA)]

for i in range(rowsA):

for j in range(colsB):

for k in range(colsA):

# Create the result matrix

# Dimensions would be rows\_A x cols\_B

C[i][j] += A[i][k] \* B[k][j]

else:

print ("Cannot multiply the two matrices. Incorrect dimensions.")

return

return C

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

#Function that creates a diagonal matrix

def diogMat (A):

length = len(A)

floatA = [0 for x in range(length)]

for i in range (length):

numVal = ''

stringVal = list(str(A[i]))

for j in range (1,len(stringVal)-1,1):

numVal = numVal + stringVal[j]

floatVal = float(numVal)

floatA [i] = floatVal

diognalMatrix = [[0 for x in range(length)] for y in range(length)]

for i in range (length):

diognalMatrix[i][i] = 1/floatA[i]

return diognalMatrix

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

#Function that creates random A, given a length input

def randomSPD(length):

A = [[0 for x in range(length)] for y in range(length)]

L = random.rand(length,length)

A = np.dot(L,L.T)

return A

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

#Function that performs the Cholesky decomposition and returns L

def cholesky(A,length):

global sum

L = [[0 for x in range(length)] for y in range(length)]

for i in range(length):

for k in range(i + 1):

sum = 0

for j in range(k):

sum += L[i][j] \* L[k][j]

if (i == k):

L[i][k] = math.sqrt(abs(A[i][i] - sum))

else:

L[i][k] = (A[i][k] - sum) / L[k][k]

return L

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

#Function that solves y in Ly = b

def forwElim (L,b,length):

b = list2float(b)

global sum

y = [0 for x in range (length)]

for i in range (length):

sum = 0.00

for j in range (i):

sum += L[i][j] \* y[j]

y[i] = (b[i]-sum)/L[i][i]

return y

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

#Function that solves for x in L^Tx = y

def backSub (L,y,length):

global sum

X = [0 for x in range(length)]

for i in range (length - 1, -1, -1):

sum = 0

for j in range (i + 1, length, 1):

sum += L[j][i] \* X[j]

X[i] = (y[i] - sum) / L[i][i]

return X

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