ASSIGNMENT 3: ECSE 543

Jorawar Singh Dham
260899593

NOTE: functions to perform matrix operations are in Appendix E and are used in other programs when necessary.

Q1.

(a) The first six points were interpolated using Full Domain Lagrange Polynomials. The code implementing this interpolation can be found in the script **test_lagrange.m** *(Appendix A.2)* which uses the function **Lagrange.m***(Appendix A.1)* to interpolate using the first six points. Figure 1 shows the interpolated curve. This curve is plausible; however, the curve underestimates values of B for larger H.

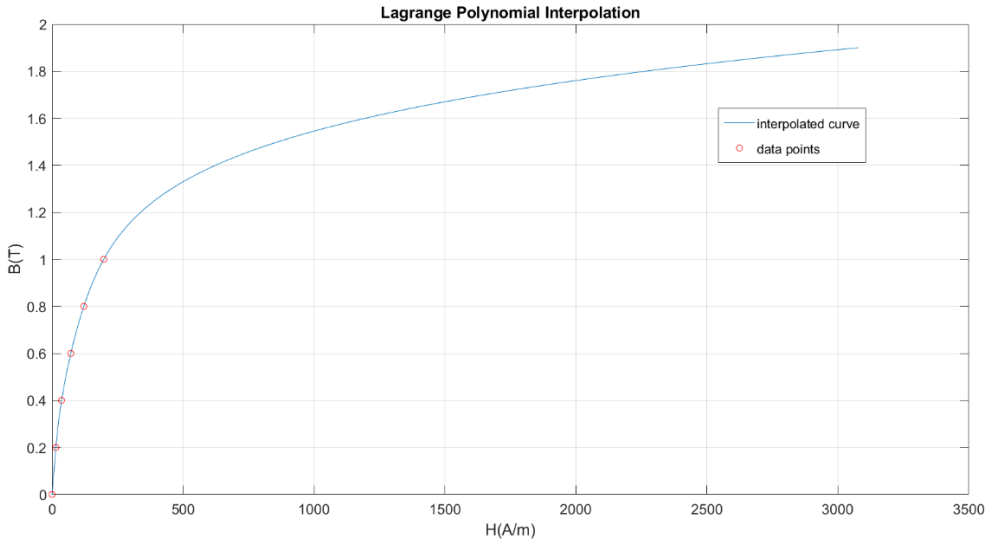


Figure 1:Full-domain Lagrange interpolation using the first six points

(b) The second part of the script **test_lagrange.m** *(Appendix A.2)* uses the six points at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9 in the full-domain Lagrange polynomial interpolation. The interpolated curve is shown in Figure 2. Clearly, when we compare the curve with the data provided we can see that the result is not plausible.

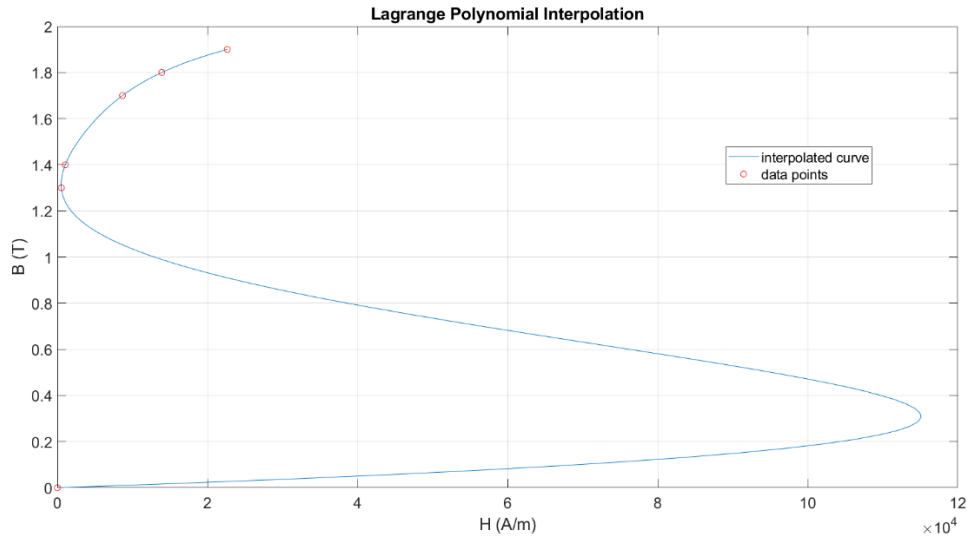


Figure 2: interpolation for the six points at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9

(c) The slopes at the six points can be fixed using Finite Difference formulas. For any point (B_i,H_i):

$$\frac{dB}{dH} = \frac{B_{i+1} - B_i}{H_{i+1} - H_i}$$

Q2. A= 1 cm^2 , L_c = 30 cm and L_a = 0.5 cm. The coil has N = 1000 turns and carries a current I = 8 A.

(a) The magnetic circuit can be reperesented by the equation:

$(R_g + R_c)\psi = M = NI$

where $R_g = \frac{L_a}{A\mu_o}$ and $R_c = \frac{L_c}{A\mu}$

Now $\mu = \frac{B}{H}$, $and\ B = \frac{\psi}{A}$, Combining these two we have $R_c = \frac{HL_c}{\psi}$

$\Rightarrow \left(\frac{L_a}{A\mu_o} + \frac{HL_c}{\psi}\right)\psi = NI$

$\Rightarrow f(\psi): \frac{L_a}{A\mu_o}\psi + HL_c - NI = 0$

Substituting known values gives us the final non-linear function for flux:

$f(\psi): (3.978873577 \times 10^7)\psi + 0.3H - 8000 = 0$

(b) The script **Magnetic_newton.m** *(Appendix B.1)* solves the nonlinear equation using Newton-Raphson. It uses a piecewise linear interpolation of data in Q1 to calculate the value of H for each iteration. It incorporates and uses the following functions:

flux.m *(Appendix B.3)*: Evaluates the non-linear function
flux_der.m *(Appendix B.4))*: Evaluates the derivative of the non-linear function
h_val.m *(Appendix B.5):* calculates value of H using linear interpolation
h_derivative.m *(Appendix B.6):* calculates derivative of H using linear interpolation.

The final value of flux is found to be **1.6127e-04 Wb** in **3 iterations**. (figure 3)

Figure 3:Output for Newton Raphson

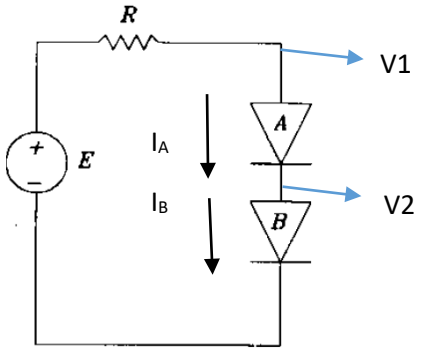
(c) The script **Magnetic_newton.m** *(Appendix B.2)* solves the nonlinear equation using successive substitution. It was observed that the method does not converge using the base successive substitution formula. To make it converge, the iterative update step was scaled by a factor $\propto =$

10^{-8} i.e. $\psi_{i+1} = \psi_i - \propto \times f(\psi_i)$. After the modification the method converges and the final value of flux is found to be **1.6127e-04 Wb** in **23 iterations**. (figure 4)

```
Command Window
>> Magnetic_succesive
>> i

i =

    23

>> x_guess

x_guess =

   1.6127e-04

fx >>
```

Figure 4: Output for successive substitution

Q3. E = 220 mV, R is 500 Ω, Is_A = 0.6 µA, Is_B = 1.2 µA, and kT/q = 25 mV.



(a) For the given circuit we have the equations

$$\frac{E - V1}{R} = Is_A\left(e^{\frac{q(V1-V2)}{KT}} - 1\right)$$

$$I_A = I_B \Longrightarrow Is_A\left(e^{\frac{q(V1-V2)}{KT}} - 1\right) = Is_B\left(e^{\frac{qV2}{KT}} - 1\right)$$

Hence the non-linear equations for the nodal voltages are:

$$f_1(V1,V2): V1 - E + RIs_A\left(e^{\frac{q(V1-V2}{KT}} - 1\right) = 0$$

$$f_2(V1,V2): Is_A\left(e^{\frac{q(V1-V2)}{KT}} - 1\right) - Is_B\left(e^{\frac{qV2}{KT}} - 1\right) = 0$$

(b) The script **Circuit.m** *(Appendix C.1)* solves the two nonlinear equations using Newton-Raphson and records the value of the two equations and the voltage across each diode. It incorporates and uses the following functions:

psi1.m *(Appendix C.2)*: Evaluates the first non-linear equation
psi2.m *(Appendix C.3)*: Evaluates the second non-linear equation
jacobian.m *(Appendix C.4):* Calculates the partial derivative matrix

The output of the program is shown in figure 5. The script also plots the 2norm of the functions vs the iteration number (figure 6). From the graph we can infer that the convergence is approximately quadratic.

```
Command Window
>> Circuit
>> psi_store

psi_store =

  -2.2000e-01   9.9056e-02   2.8369e-02   5.0999e-03   1.9434e-04   3.0876e-07
            0   1.8078e-04   5.5187e-05   8.5611e-06   3.3308e-07   5.0976e-10

>> Vstore

Vstore =

            0   2.1825e-01   2.0570e-01   2.0011e-01   1.9821e-01   1.9813e-01
            0   7.2751e-02   8.1581e-02   8.9250e-02   9.0516e-02   9.0571e-02

>> V_diode

V_diode =

            0   1.4550e-01   1.2411e-01   1.1086e-01   1.0770e-01   1.0756e-01
            0   7.2751e-02   8.1581e-02   8.9250e-02   9.0516e-02   9.0571e-02

>> i

i =

     5

fx >>
```

Iteration	$f_1(V1,V2)$	$f_2(V1,V2)$	V1(volts)	V2(volts)	Vd_1=V1-V2	Vd_2=V2
0	-0.22000	0	0	0	0	0
1	0.099056	1.8078e-04	0.2183	0.0728	0.1455	0.0728
2	0.028369	5.5187e-05	0.2057	0.0816	0.1241	0.0816
3	5.0999e-03	8.5611e-06	0.2001	0.0892	0.1109	0.0892
4	1.9434e-04	3.3308e-07	0.1982	0.0905	0.1077	0.0905
5	3.0876e-07	5.0976e-10	0.1981	0.0906	0.1076	0.0906

Figure 5: Output Of Circuit.m

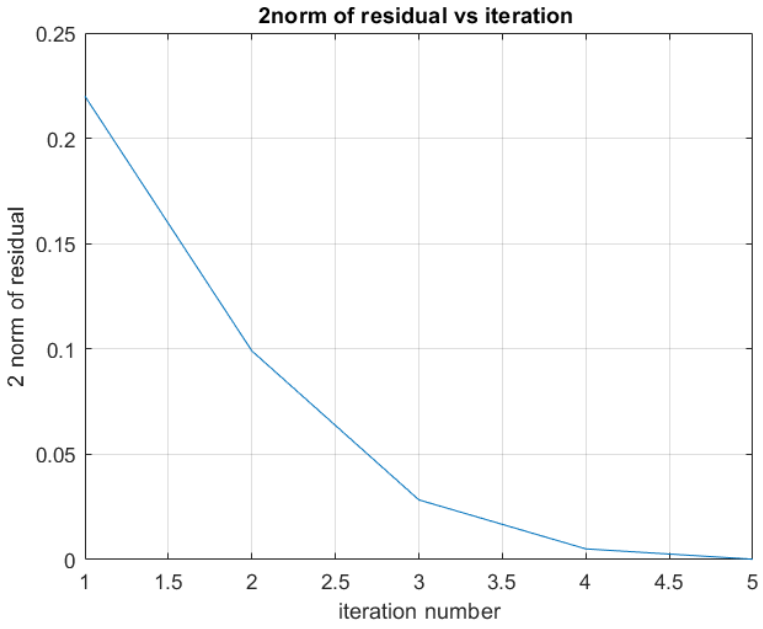


Figure 6: 2norm Vs Iteration number

Q4.

(a) The script **cos_legendre.m** *(Appendix D.1)* integrates the function cos(x) from x=0 to x=1 using one-point Gauss-Legendre integration for each segment. The integration is calculated for N= 1 to N=20 segments. The error is calculated for each integration corresponding to a N and log(E) vs log(N) is plotted as shown in figure 7. Since the log-log graph is a straight line with a negative slope we can infer that the E and N have a power law relationship with negative power, i.e. the relationship is of the form: $E = aN^{-k}$

As the number of segments increases the error reduces exponentially.

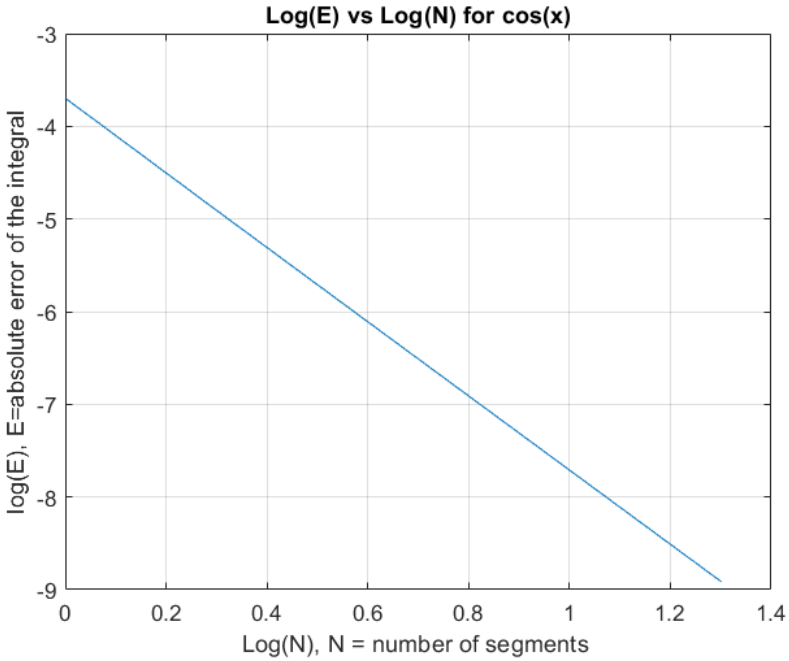


Figure 7: Log of error vs log of N(segments) for integration of Cos(x) from 0 to1

N	Error
1	2.0114e-04
5	3.1205e-07
10	1.9485e-08
15	5.0712e-09
20	1.2175e-09

Table 1: some selected output from cos_legendre.m

(b) The script **ln_legendre.m** *(Appendix D.2)* integrates the function ln(x) using one-point Gauss-Legendre integration for each segment. The integration is calculated for N= 10, 20, …., 200.
The error is calculated for each integration corresponding to a N and log(E) vs log(N) is plotted as shown in figure 8. We can see that the graph has a similar nature to the one in figure 7 and we can infer that the error and N have a similar power law relationship. The only difference is that the error in integrating ln(x) from 0 to 1 is much higher than integrating cos(x) even with a larger number of segments. This is because the value ln(x) changes very rapidly between 0 and 1.

N	Error
10	1.0447e-02
50	2.0895e-03
100	1.0447e-03
150	6.9649e-04
200	5.2237e-04

Table 2: some selected output from ln_legendre.m

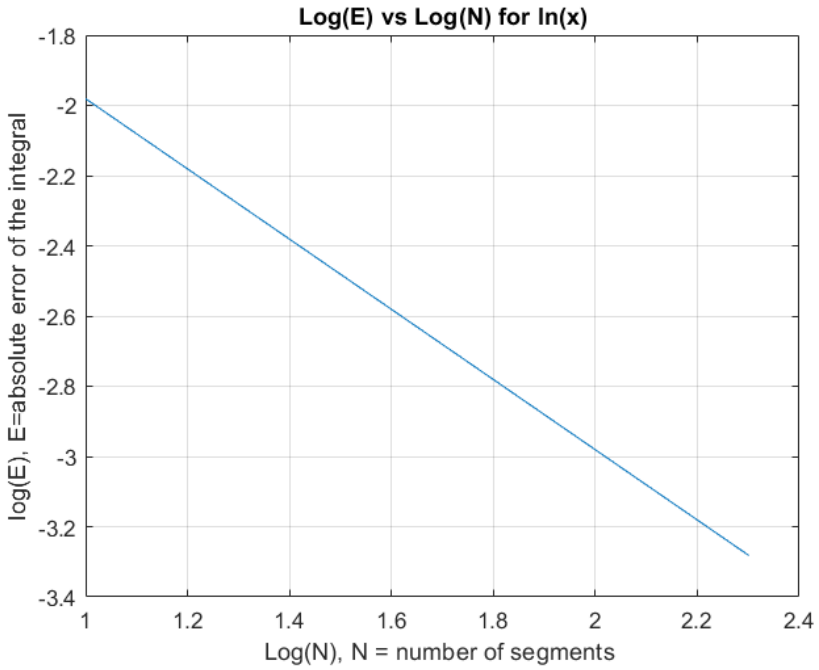


Figure 8: Log of error vs log of N(segments) for integration of ln(x) from 0 to1

(c) The script **ln_nonUniform_legendre.m** *(Appendix D.3)* integrates the function ln(x) using one-point Gauss-Legendre integration by dividing the interval into non uniform segments for N=10. Smaller segments are used near x = 0 which is the more difficult part of the interval as ln(x) changes very rapidly near x = 0. The output is shown in figure 9. We can see that the **error = 4.5185e-03** is much lower than for integration using equally spaced segments in (b) where it was 1.0447e-02.

```
Command Window
>> ln_nonUniform_legendre
>> error

error =

   4.5185e-03

>> integral

integral =

  -9.9548e-01

fx >>
```

Figure 9: Output of non-uniform legendre integration of ln(x) from 0 to 1 for N=10

APPENDIX

A. Code for Question 1

A.1 Lagrange.m

```
%Function to interpolate using lagrange Polyomials
function L=Lagrange(x,H,B,n)
L=0;
for i=1:n
    L=L+H(i)*lagrange_f(x,i,B,n);
end
%Function to calculate lagrange Polyomials
function L=lagrange_f(x,i,B,n)
L=1;
for k=1:n
    if k==i
       continue;
    end
    L=((x-B(k))/(B(i)-B(k)))*L;
end
```

A.2 test_lagrange.m

```
%Script to call the lagrange function and plot the interpolated data
B1=[0;0.2;0.4;0.6;0.8;1.0];
H1=[0;14.7;36.5;71.7;121.4;197.4];
n=6;
i=1;
for b=0:(1.9/100):1.9
    h(i,1)=Lagrange(b,H1,B1,n);
    i=i+1;
end
figure(1)
    plot(h,0:(1.9/100):1.9);
    legend("interpolated curve")
    hold on
    plot(H1, B1, 'or')
    legend("interpolated curve", "data points")
    title('Lagrange Polynomial Interpolation')
    hold off
    xlabel('H(A/m)')
    ylabel('B(T)')
    grid

B2=[0;1.3;1.4;1.7;1.8;1.9];
H2=[0;540.6;1062.8;8687.4;13924.3;22650.2];
n=6;
i=1;
for b=0:(1.9/100):1.9
    h(i,1)=Lagrange(b,H2,B2,n);
    i=i+1;
end
figure(2)
    plot(h,0:(1.9/100):1.9);
    legend("interpolated curve")
    hold on
    plot(H2, B2, 'or')
```

```
    legend("interpolated curve", "data points")
    title('Lagrange Polynomial Interpolation')
    hold off
    xlabel('H (A/m)')
    ylabel('B (T)')
    grid
```

B. Code for Question 2

B.1 Magnetic_newton.m

```
%function to calculate the flux using the Newton Raphson method
global B H
B=[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=[0,14.7,36.5,71.7,121.4,197.4,256.2,348.7,540.6,1062.8,2318,4781.9,8687.4,13924.3,22650.2];
x_guess=0;
err=1e-6;
i=0;
while abs(flux(x_guess)/flux(0)) > err
       i = i + 1;
       x_guess = x_guess - (flux(x_guess) / flux_der(x_guess));
end
```

B.2 Magnetic_succesive.m

```
%function to calculate the flux using succesive substitution
global B H
B=[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=[0,14.7,36.5,71.7,121.4,197.4,256.2,348.7,540.6,1062.8,2318,4781.9,8687.4,13924.3,22650.2];
x_guess=0;
err=1e-6;
i=0;
while abs(flux(x_guess)/flux(0)) > err
    i = i + 1;
    x_guess = x_guess - 1e-8*flux(x_guess) ;
end
```

B.3 flux.m

```
function[phi] = flux(x_guess)
        % Evaluates the flux
               phi = 3.978873577e7 * x_guess + 0.3 * h_val(x_guess) - 8000;
end
```

B.4 flux_der.m

```
function[phi_der] = flux_der(x_guess)
        % Evaluates the flux derivative
               phi_der = 3.978873577e7 + 0.3 * h_derivative(x_guess)*1e4;
end
```

B.5 h_val.m

```
%function to calculate value of H using linear interpolation
function h = h_val(x_guess)
    global B  H
               b = x_guess*1e4;
```

```
                hb_rows = size(H,2);
        % Linear Interpolate
        if b > B(15)
                slope = (H(15) - H(14)) / (B(15) - B(14));
                h = (b - B(15)) * slope + H(15);
                return
                end

        for i = 1:hb_rows
                if B(i) > b
                slope = (H(i) - H(i-1)) / (B(i) - B(i-1));
                h = (b - B(i-1)) * slope + H(i-1);
                return
            end
         end
end
```

B.6 h_derivative.m

```
%function to calculate derivative of H using linear interpolation
function h_der = h_derivative(x_guess)
    global B  H
                b = x_guess*1e4;
                hb_rows = size(H,2);
        % Interpolate
        if b > B(15)
             h_der = (H(15) - H(14)) / (B(15) - B(14));
                return
         end

        for i = 1:hb_rows
                if B(i) > b
                h_der = (H(i) - H(i-1)) / (B(i) - B(i-1));
                return
            end
         end
end
```

C. Code for Question 3

C.1 Circuit.m

```
%script to solve the non linear equations for the circuit
global E R Isa Isb kt
V1 =  0;
V2 = 0;
E = 220e-3;
R = 500;
Isa =0.6e-6;
Isb = 1.2e-6;
kt = 25e-3;
i=0;
err=1e-6;
V=[V1;V2];
V_d=[V1-V2;V2];
V_diode=V_d;%variable to store value of voltage accross each diode in each iteration
Vstore=V;%variable to store value of V1 and V2 in each iteration
p1=psi1(V1,V2);
p2=psi2(V1,V2);
psi=[p1;p2];
psi_store=[p1;p2];%variable to store value of NL functions in each iteration
while sqrt(p1^2 + p2^2)> err
   %(abs(p1) + abs(p2)) / ( abs(psi1(0,0)) + abs(psi2(0,0)) ) > err
    i = i + 1;
    itr(i,1)=i;
    e(i,1)=sqrt(p1^2 + p2^2);
    V = mat_sub(V,mat_mul(mat_inv(jacobian(V1,V2)),psi));
    V1=V(1);
    V2=V(2);
    V_d(1)=V1-V2;
    V_d(2)=V2;
    Vstore=[Vstore V];
    V_diode=[V_diode V_d];
    p1=psi1(V1,V2);
    p2=psi2(V1,V2);
    psi=[p1;p2];
    psi_store= [psi_store psi];
end
plot(itr,e)
title('2norm of residual vs iteration ')
xlabel('iteration number')
ylabel('2 norm of residual')
grid
```

C.2 psi1.m

```
function psi = psi1(V1,V2)
%evaluate first non linear equation
global E R Isa kt
psi=V1-E + R*Isa*(exp((V1-V2)/kt)-1);
end
```

C.3 psi2.m

```
function psi = psi2(V1,V2)
%evaluate second non linear equation
global  Isa Isb kt
psi= Isa*(exp((V1-V2)/kt)-1) - Isb*(exp(V2/kt)-1);
end
```

C.4 jacobian.m

```
function J = jacobian(V1,V2)
global  R Isa Isb kt
%function to calculate the jacobian
  J(1,1) = 1.0 + (R * Isa / kt) * (exp((V1 - V2) / kt));
  J(1,2) = -1.0 * (R * Isa / kt) * (exp((V1 - V2) / kt));
  J(2,1) = (Isa / kt) * (exp((V1 - V2) / kt));
  J(2,2) = -1.0 * (Isa / kt) * (exp((V1 - V2) / kt)) - (Isb / kt) * (exp(V2 / kt));
end
```

D. Code for Question 4

D.1 cos_legendre.m

```
%script to integrate cos(x) between 0 and 1
integral(1:20,1)=0;
error(1:20,1)=0;
logN(1:20,1)=0;
logE(1:20,1)=0;
for i=1:20
    pts=linspace(0,1,i+1);
    for j=1:i
        integral(i)= integral(i)+ ((pts(j+1)-pts(j))/2)*cos( (pts(j)+pts(j+1))/2 + (pts(j+1)-
pts(j))/(2*sqrt(3))) + ((pts(j+1)-pts(j))/2)*cos( (pts(j)+pts(j+1))/2 + (pts(j+1)-pts(j))/(2*-sqrt(3)));
    end
    error(i)= abs(sin(1)-integral(i));
    logN(i)=log10(i);
    logE(i)=log10(error(i));
end
plot (logN,logE)
title('Log(E) vs Log(N) for cos(x)')
hold off
xlabel('Log(N), N = number of segments ')
ylabel('log(E), E=absolute error of the integral')
grid
```

D.2 ln_legendre.m

```
%script to integrate ln(x) between 0 and 1
integral(1:20,1)=0;
error(1:20,1)=0;
k=1;
logN(1:20,1)=0;
logE(1:20,1)=0;
for i=10:10:200
    pts=linspace(0,1,i+1);
    for j=1:i
        integral(k)= integral(k)+ ((pts(j+1)-pts(j))/2)*log( (pts(j)+pts(j+1))/2 + (pts(j+1)-
pts(j))/(2*sqrt(3))) + ((pts(j+1)-pts(j))/2)*log( (pts(j)+pts(j+1))/2 + (pts(j+1)-pts(j))/(2*-sqrt(3)));
    end
    error(k)=abs(-1 - integral(k));
    logN(k)=log10(i);
    logE(k)=log10(error(k));
    k=k+1;
end
plot (logN,logE)
title('Log(E) vs Log(N) for ln(x)')
hold off
xlabel('Log(N), N = number of segments ')
ylabel('log(E), E=absolute error of the integral')
grid
```

D.3 ln_nonUniform_legendre.m

```
%script to integrate ln(x) for N=10 with non uniform segments
integral =0;
error =0;
i=7;
pts=linspace(0,0.3,i+1);
for j=1:i
    integral = integral + ((pts(j+1)-pts(j))/2)*log( (pts(j)+pts(j+1))/2 + (pts(j+1)-pts(j))/(2*sqrt(3))) +
((pts(j+1)-pts(j))/2)*log( (pts(j)+pts(j+1))/2 + (pts(j+1)-pts(j))/(2*-sqrt(3)));
end
i=3;
pts=linspace(0.3,1,i+1);
for j=1:i
    integral = integral + ((pts(j+1)-pts(j))/2)*log( (pts(j)+pts(j+1))/2 + (pts(j+1)-pts(j))/(2*sqrt(3))) +
((pts(j+1)-pts(j))/2)*log( (pts(j)+pts(j+1))/2 + (pts(j+1)-pts(j))/(2*-sqrt(3)));
end
error= abs(-1-integral);
```

E. General Matrix Operations

E.1 mat_sum.m

```
function A = mat_sum(B,C)%function to add two matrices
[n1,m1]=size(B);
[n2,m2]=size(C);
if m1==m2 && n1==n2
    for i=1:n1
        for j=1:m1
            A(i,j)=B(i,j)+C(i,j);
        end
    end
else
    disp('matrices are not the same size');
    A=nan;
    return;
end
end
```

E.2 mat_sub.m

```
function A = mat_sub(B,C)%function to subtract two matrices
[n1,m1]=size(B);
[n2,m2]=size(C);
if m1==m2 && n1==n2
    for i=1:n1
        for j=1:m1
            A(i,j)=B(i,j)-C(i,j);
        end
    end
else
    disp('matrices are not the same size');
    A=nan;
    return;
end
end
```

E.3 mat_mul.m

```
function M = mat_mul(A,B)%function to multiply two matrices
[n1,m1]=size(A);
[n2,m2]=size(B);
    if m1~=n2
        disp('size mismatch');
        M=nan;
        return;
    else
        for i=1:n1
            for j=1:m2
                M(i,j)=0;
                for t=1:m1
                    M(i,j)=M(i,j)+ A(i,t)*B(t,j);
                end
            end
```

```
        end
    end
end
```

E.4 mat_inv.m

```
function M = mat_inv(A)
%function to calculate the inverse of a 2*2 matrix
det = A(1,1) * A(2,2) - A(1,2) * A(2,1);
M(1,1) = A(2,2) / det;
M(2,2) = A(1,1) / det;
M(1,2) = -1 * A(1,2) / det;
M(2,1) = -1 * A(2,1) / det;
end
```