ASSIGNMENT 1: ECSE 543

Jorawar Singh Dham
260899593

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

Q1.

(a) A program to solve the matrix equation Ax=b by Choleski decomposition is written in MATLAB as **Cholesky_lookahead.m** *(Appendix A.1)*. The program applies the look ahead modification on the inputs A and b to overwrite them with L and y respectively. It then does backward elimination to solve for and return the vector of unknowns' x (solve Ux=y).

(b) A program to create real, symmetric and positive definite matrices of size n=2,3,4,5 was written as **sym_posdef_gen.m** *(Appendix A.2)*. We know that if M is any n x n, real, non-singular matrix then, $A=MM^t$ is symmetric positive definite. The program takes the size n as input and creates a random non-singular lower triangular matrix L (of size n) and multiplies it with its transpose L^t to form a real, symmetric and positive definite matrix. The code makes sure the matrix L is non-singular by ensuring all the diagonal elements of L are non-zero. (A lower triangular matrix is non-singular iff all the elements on its diagonal are non-zero). The output matrix can be verified for the properties by passing it to the Cholesky decomposition program in Q1. (a).

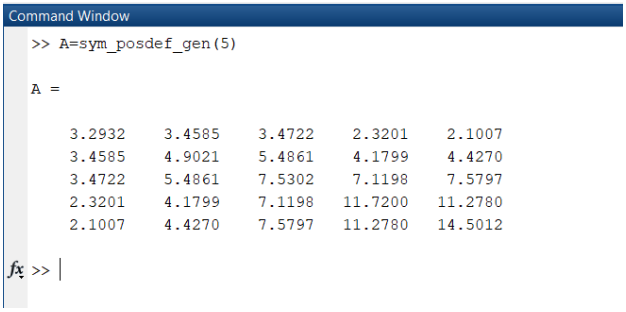


Figure 1: Example of an output of sym_posdef_gen.m for n = 5

(c) The script **test_c.m** *(Appendix A.3)* calls **sym_posdef_gen.m** *(Appendix A.2)* to create real, symmetric and positive definite matrices A of size n=2,3,4,5 as shown in Q1. (b) and figure 1. Random x vectors were invented and multiplied with matrix A to get b. Both of these were passed to **Cholesky_lookahead.m** *(Appendix A.1)* to verify that we get out original x back. Figure 2 shows an equivalent example for n=5.

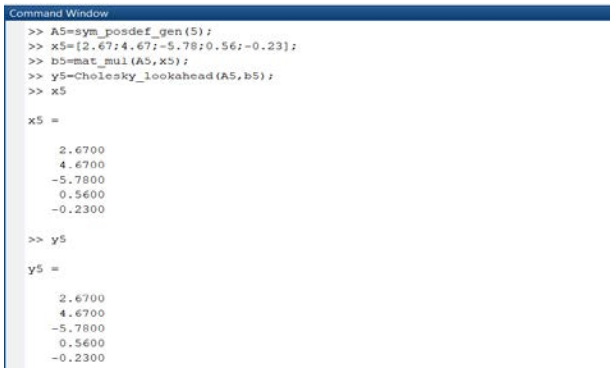
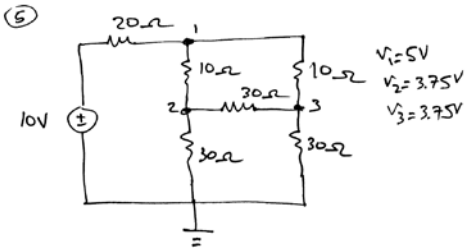


Figure 2:Example for n=5, we get y5=x5

(d) The program **read_data.m** *(Appendix A.4)* takes a filename and the number of nodes and branches in the associated circuit and reads a list of network branches (J, R, E) and a reduced incidence matrix from the file (.xlsx file) and creates the necessary A, Y, J and E matrices/vectors. This data is passed to another function **circuit_solve.m** (*Appendix A.5)* that forms the necessary equation: $(AYA^t)V=A(J-YE)$ and calls **Cholesky_lookahead.m** *(Appendix A.1)* to solve for voltage V at the nodes. This was tested for all the test circuits provided with the assignment and all the results match the analytical values. Figure 3 and 4 show the files for two of the given test circuits and the associated solution obtained from the code. As we can see the results are equal to the analytical results.

DATA ORGANISATION: The first n x m rows and columns of the xlsx file form the reduced incidence matrix where n= number of nodes and m=number of branches. This is followed by an empty column. The next three columns after the empty column contain values for the network branches J_k, R_k and E_k, in that order. By passing n and m to the program we can accurately read and create the necessary matrices for this data organisation scheme as shown in **read_data.m** *(Appendix A.4).*

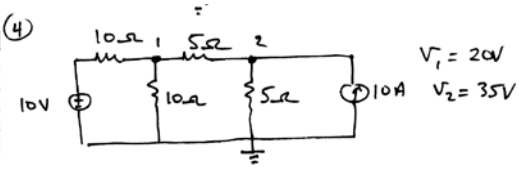
	A	B	C	D	E	F	G	H	I	J
1	-1	1	1	0	0	0		0	20	10
2	0	-1	0	1	1	0		0	10	0
3	0	0	-1	-1	0	1		0	10	0
4								0	30	0
5								0	30	0
6								0	30	0
7										
8										




```
Command Window
>> [A,Y,J,E]=read_data('c5.xlsx',3,6);
>> V=circuit_solve(A,Y,J,E)

V =

    5.0000
    3.7500
    3.7500

fx >>
```

Figure 3 the data organisation in the file(c5.xlsx) for the given circuit and the nodal voltages obtained from our program

	A	B	C	D	E	F	G	H
1	1	-1	1	0		0	10	0
2	0	0	-1	1		0	10	10
3						0	5	0
4						10	5	0
5								
6								

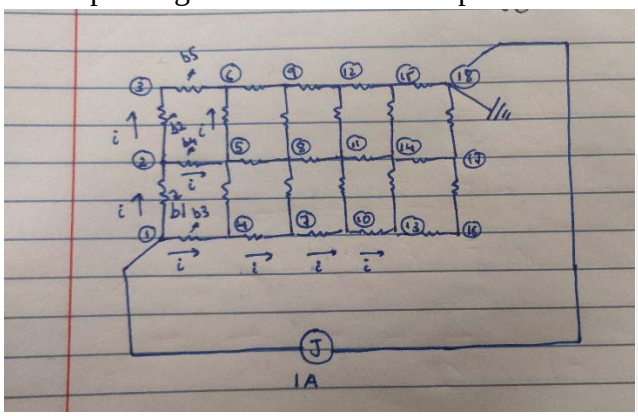



```
Command Window
>> [A,Y,J,E]=read_data('c4.xlsx',2,4);
>> V=circuit_solve(A,Y,J,E)

V =

   20.0000
   35.0000

fx >>
```

Figure 4: The data organisation in the file(c4.xlsx) for the given circuit and the nodal voltages obtained from our program

Q2.

(a) The program **mesh_final.m** *(Appendix B.1)* takes N as input and creates the reduced incidence matrix A, matrix Y and vectors J and E for a N by 2N resistor mesh with all R= 1k ohm. The code also adds a current source of 1A between the bottom left and top right nodes. This data is then passed to the function **solve.m** *(Appendix B.2)* which solves for voltage V at all the nodes using **Cholesky_lookahead.m** *(Appendix A.1)* or **Cholesky_sparse.m** *(Appendix B.3)* depending on user input. Figure 5 shows an example of this for N=3.




```
Command Window
>> [A,Y,J,E]=mesh_final(3);
>> V=solve(A,Y,J,E,'n');
>> R=V(1)
R =
    2.3795e+003
>>
```

Figure 5: Example input and output for calculating resistance of the mesh for n = 3

A part of the script **Q2_final.m** *(Appendix B.5)* calculates the resistance for n=2,3,4....,10 using the method described above using regular Cholesky decomposition from Q1. The results are tabulated in Table 1.

N	RESISTANCE (kilo ohms)	Computation time(s)
2	1.8750	1.4448e-4
3	2.3795	2.5817e-4
4	2.7410	7.8399e-4
5	3.0228	0.0025
6	3.2537	0.0067
7	3.4492	0.0165
8	3.6187	0.0366
9	3.7683	0.0819
10	3.9022	0.1622

Table 1 : Resistance values and associated computation time for N = 2 to 10

(b) From the results in Table 1, a graph is plotted between N and computation time (figure 6). Theoretically, the complexity of cholesky decomposition is $O(n^3)$. For our problem $n=2N^2$ and our complexity with respect to N becomes $O(8N^6) \approx O(N^6)$. Using polynomial fitting, the curve is found to fit accurately for a sixth degree polynomial (figure 6) given by:

$$y = -3.58e\text{-}06\, x^6 + 0.000135\, x^5 - 0.00191\, x^4 + 0.0135\, x^3 - 0.0498\, x^2 + 0.0917\, x - 0.0651$$

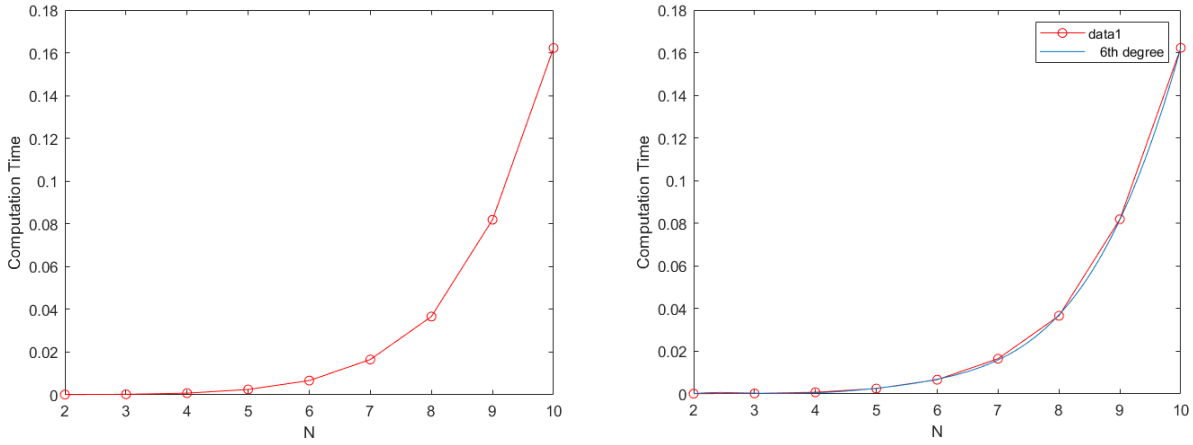


Figure 6 : Graph of Computation time(s) vs N and fit with sixth degree polynomial

(c) The program for the look ahead cholesky algorithm was modified to **Cholesky_sparse.m** *(Appendix B.3)* which uses the half bandwidth of the matrix AYA^t to reduce computation time. The program **find_band.m** *(Appendix B.4)* is used to find the half bandwidth of the matrix. The script **Q2_final.m** *(Appendix B.5)* calculates computation time for both the methods for n=2 to 10 and the results are tabulated in Table 2. From the table we can see that the half bandwidth is N+1 for our chosen node numbering scheme. Theoretically, the complexity of the banded problem is $O(b^2n)$. For our problem b=N+1 and $n=2N^2$. Therefore, the complexity is $O((N+1)^2\ 2N^2)$ or $O((N+1)^2\ 2N^2) \approx O(N^4)$ for large N. If we look at the plot of the computation times for optimised and unoptimised methods vs N (figure 7), we see that the reduction in computation time is significant as N increases. This is in line with our theoretical prediction.

N	Unoptimised Computation time(s)	Optimised Computation time (s)	Half Bandwidth *b*
2	1.4448e-4	1.2593e-4	3
3	2.5817e-4	2.2422e-4	4
4	7.8399e-4	7.4372e-4	5
5	0.0025	0.0022	6
6	0.0067	0.0065	7
7	0.0165	0.0163	8
8	0.0366	0.0361	9
9	0.0819	0.0767	10
10	0.1622	0.1545	11

Table 2: Computation time for Unoptimised and optimised methods. (includes half bandwidth)

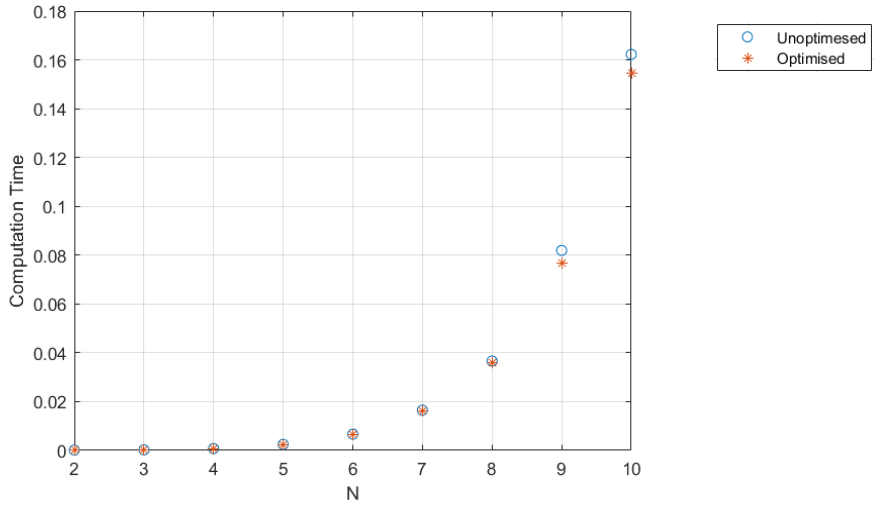


Figure 7 : Computation time Vs N for optimised and unoptimised methods

(d) Using the data in table 1, the graph for R Vs N is plotted as shown in figure 8. If we observe the data and the nature of the graph we can see that R increases as N increases but the rate of increase (slope) decreases as N increases. On this basis we can infer that the nature of the curve is logarithmic and $R(N) \approx Log(N)$ is asymptotically correct as $N \rightarrow \infty$.

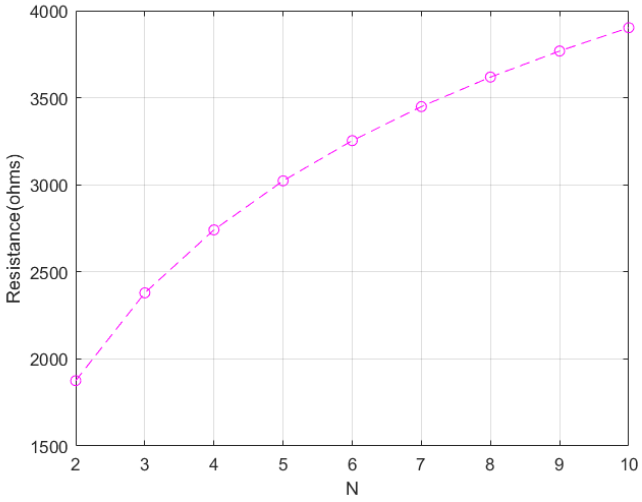


Figure 8 : graph of R vs N

Q3

(a) Using the planes of symmetry only ¼ of the problem is considered and all calculations are performed in the bottom left corner (x<=0.1 and y<=0.1). The program **mesh_cable_final.m** *(Appendix C.1)* forms the grids of nodes for uniform node spacing h and applies the Dirichlet boundary conditions and initializes all free nodes with $\Phi=0$. The matrix formed is then passed to the function **SOR_final.m** *(Appendix C.3)* with h and ω. The function applies the SOR formula to the matrix until the residual at all the nodes is $< 10^{-5}$. The program **max_residual_final.m** *(Appendix C.4)* calculates the maximum residual amongst all the free nodes and is called during each iteration of the program for SOR (or Jacobi). The program iterates until the residual of all

nodes < 10^{-5}. Finally, the function **get_potential.m** *(Appendix C.5)* returns the value of scaler potential at any node across the entire cable using symmetry.

(b) Part of the script **Q3_uniform_node.m** *(Appendix C.9)* calculates the potential for h=0.02 for ten values of ω between 1 and 2 and obtains potential at (x, y) = (0.06, 0.04) for each case. The results are shown in figure 9 and tabulated in Table 3. The graph of number of iterations versus ω is shown in figure 10. We can see that there is an optimal value of ω around ω ≈ 1.35 which gives minimum iterations.

```
Command Window
>> Q3_uniform_node
>> V

V =

  Columns 1 through 5

   5.526331389187691   5.526332641968612   5.526331221702579   5.526340275930925   5.526341015336559

  Columns 6 through 10

   5.526341240856198   5.526341885240776   5.526340813656259   5.526341747865994   5.526342696414132

>> itr1

itr1 =

  Columns 1 through 9

          36          30          24          22          28          36          49          70         128

  Column 10

        1145
```

Figure 9: Output of SOR method showing potential at (0.06,0.04) for h=0.02 and different values of ω

ω	Iterations	V(volts) at (x, y) = (0.06, 0.04)
1.1	36	5.526331389187691
1.1889	30	5.526332641968612
1.2778	24	5.526331221702579
1.3667	22	5.526340275930925
1.4556	28	5.526341015336559
1.5444	36	5.526341240856198
1.6333	49	5.526341885240776
1.7222	70	5.526340813656259
1.8111	128	5.526341747865994
1.9	1145	5.526342696414132

Table 3: Iterations and potential at (0.06, 0.04) for varying omega

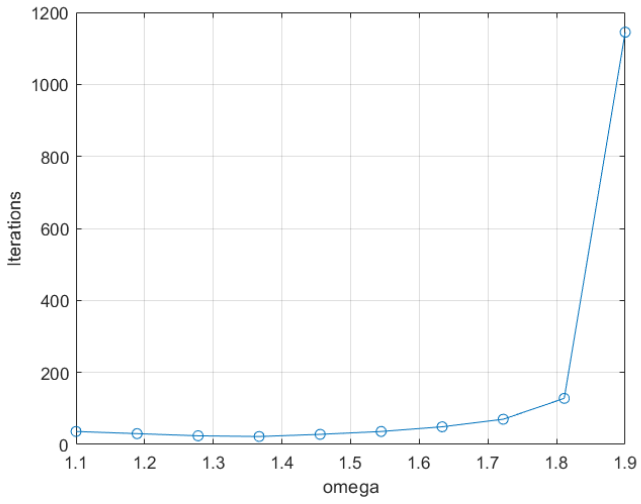


Figure 10:graph of iterations vs ω for SOR

(c) For ω=1.35, (chosen after reviewing data in Q3. (b)) the script **Q3_uniform_node.m** *(Appendix C.9)* calculates the potential at (x, y) = (0.06, 0.04) and iterations for decreasing values of h= [0.02,0.01,0.005,0.002,0.001,0.0005] using **SOR_final.m** *(Appendix C.3)*. The results are shown in figure 11 and tabulated in Table 4. Graph of 1/h vs V at (x, y) = (0.06, 0.04) and 1/h vs iterations are shown in figure 12 and figure 13 respectively. We observe that as 1/h increases, the potential at (0.06, 0.04) approaches approximately 5.23 V (up to three significant figures). From figure 13, we see that as 1/h increases, the number of iterations increases exponentially.

```
Command Window
>> Q3_uniform_node
>> V1

V1 =

  Columns 1 through 5

   5.526337806822303   5.350627378700452   5.289105828398012   5.260794186372436   5.249794140536849

  Column 6

   5.230126313158515

>> itr2

itr2 =

          21          74         258        1300        4322       13888
```

Figure 11: Output of SOR method showing potential at (0.06,0.04) and iterations for decreasing values of h at ω=1.35

1/h	Potential at (x, y) = (0.06, 0.04)	Iterations for SOR
50	5.526337806822303	21
100	5.350627378700452	74
200	5.289105828398012	258
500	5.260794186372436	1300
1000	5.249794140536849	4322
2000	5.230126313158515	13888

Table 4: Iterations and potential at (0.06, 0.04) for varying h (and 1/h) at ω=1.35

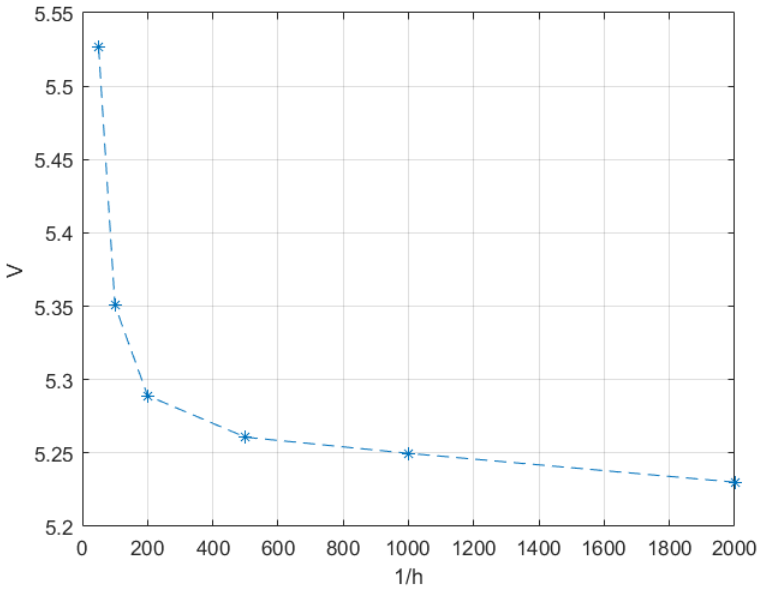


Figure 12: graph of V at (x, y) = (0.06, 0.04) vs 1/h for SOR method at ω=1.35

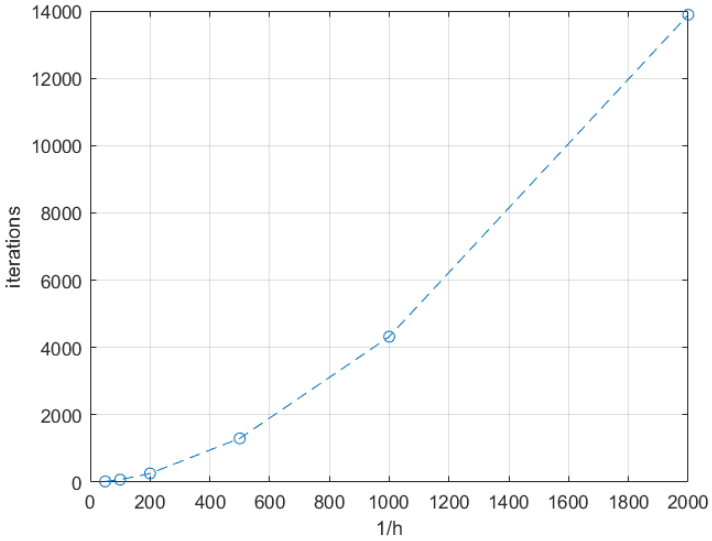


Figure 13: graph of iterations VS 1/h for SOR at ω=1.35

(d) The program **Jacobi_final.m** *(Appendix C.3)* applies the Jacobi formula to the matrix until the residual at all the nodes is < 10^{-5}. The script **Q3_uniform_node.m** *(Appendix C.9)* calculates the potential at (x, y) = (0.06, 0.04) for the same values of h as Q3. (c) using **Jacobi_final.m** *(Appendix C.3)*. The results are shown in figure 14 and tabulated in Table 5. Graph of 1/h vs V and 1/h vs iterations for Jacobi are shown in figure 15 and figure 16 respectively. We observe that as 1/h increases, the potential at (0.06, 0.04) approaches approximately 5.23 V (up to three significant figures). From Table 4, Table 5 and figure 16 we can see that the number of iterations increase exponentially for increase in 1/h but the number of iterations is significantly more for the Jacobi method as it is a less efficient method. The values for the potential at (x, y) = (0.06, 0.04) is the same for both methods.

```
Command Window
>> Q3_uniform_node
>> V2

V2 =

  Columns 1 through 5

   5.526328463163290   5.350625683628088   5.289096668354475   5.260793518964958   5.249803748093687

  Column 6

   5.230151577978697

>> itr3

itr3 =

          39         145         513        2647        8881       28678
```

Figure 14: Output of Jacobi method showing potential at (0.06,0.04) and iterations for decreasing h

1/h	Potential at (x, y) = (0.06, 0.04)	Iterations for Jacobi
50	5.526328463163290	39
100	5.350625683628088	145
200	5.289096668354475	513
500	5.260793518964958	2647
1000	5.249803748093687	8881
2000	5.230151577978697	28678

Table 5: Iterations and potential at (0.06, 0.04) for varying h (and 1/h) for Jacobi method

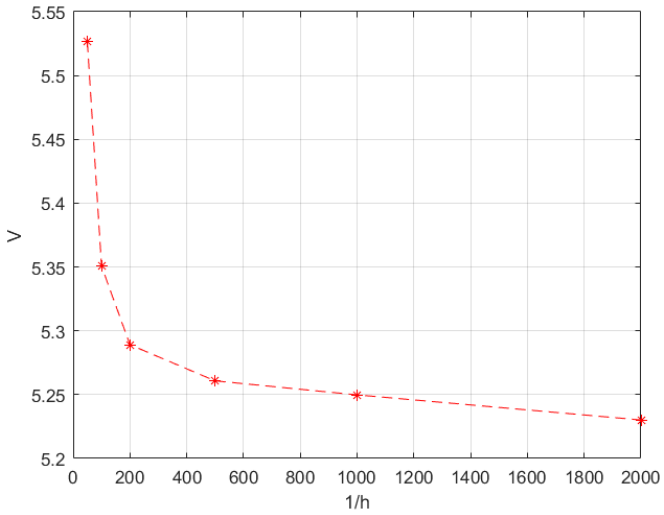


Figure 15: graph of V at (x, y) = (0.06, 0.04) vs 1/h for Jacobi method

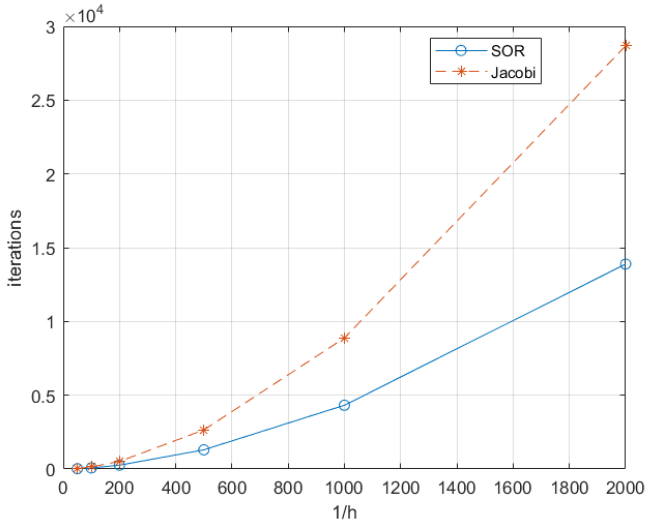


Figure 16: graph of iterations VS 1/h for Jacobi and SOR (at ω=1.35)

(e) Code from the previous questions were modified for non-uniform node spacing as follows:
mesh_nu.m *(Appendix C.6):* Creates the grid of nodes for non-uniform node spacing.
SOR_nu.m *(Appendix C.7):* performs SOR iterations for non-uniform node spacing.
max_residual_nu *(Appendix C.7):* Calculate maximum of all the residuals at the free nodes for the non-uniform node spacing.
The values of potential are calculated for following grid points and node spacing:
x= [0, 0.02, 0.03, 0.0525, 0.0575, 0.0595, 0.06, 0.0625, 0.08, 0.09, 0.1];
hx= [0.02, 0.01, 0.0225, 0.005, 0.002, 0.0005, 0.0025, 0.0175, 0.01, 0.01]
y= [0, 0.02, 0.04, 0.05, 0.07, 0.0725, 0.0775, 0.0795, 0.08, 0.0825, 0.1]
hy= [0.02, 0.02, 0.01, 0.02, 0.0025, 0.005, 0.002, 0.0005, 0.0025, 0.0175]

This creates the same number of nodes as the case for h=0.1 in uniform node spacing. The node spacing is chosen such that there is smaller node spacing close to the boundary of the core which is the more "difficult" part of the problem domain. Using the programs mentioned above we can calculate the potential at the grid points for ω=1.35 as shown in Figure 17. The value at **phi (3,7) = 5.3421 V** corresponds to the value at (x, y) = (0.06, 0.04) which is more accurate than the value obtained for uniform node spacing of 0.1(phi = 5.3506, see Table 4).

Command Window

```
>>
>> x=[0,0.02,0.03,0.0525,0.0575,0.0595,0.06,0.0625,0.08,0.09,0.1];
>> y=[0,0.02,0.04,0.05,0.07,0.075,0.0775,0.0795,0.08,0.0825,0.1];
>> A=mesh_nu(x,y);
>> [phi,itr]=SOR_nu(A,1.35,x,y);
>> itr

itr =

   552

fx >>
```

phi

11x11 double

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0.9299	1.3763	2.2948	2.4572	2.5174	2.5321	2.6026	2.9711	3.0797	3.1157	
3	0	1.9095	2.8465	4.8327	5.1821	5.3110	5.3421	5.4919	6.2385	6.4503	6.5191	
4	0	2.4169	3.6316	6.3023	6.7788	6.9530	6.9950	7.1950	8.1065	8.3502	8.4273	
5	0	3.4195	5.2531	9.7522	10.6737	11.0026	11.0802	11.4393	12.4473	12.6497	12.7062	
6	0	3.6184	5.5947	10.6857	12.0026	12.4765	12.5831	13.0381	13.7022	13.8158	13.8464	
7	0	3.7075	5.7490	11.1272	12.7378	13.3925	13.5288	13.9725	14.3479	14.4067	14.4223	
8	0	3.7731	5.8627	11.4532	13.3193	14.3228	14.5702	14.7903	14.8695	14.8813	14.8844	
9	0	3.7887	5.8896	11.5290	13.4417	14.5589	15	15	15	15	15	
10	0	3.8613	6.0149	11.8721	13.8651	14.7690	15	15	15	15	15	
11	0	4.1095	6.4073	12.5145	14.1601	14.8318	15	15	15	15	15	
12												

Figure 17: Output for non-uniform node spacing for SOR method at ω=1.35

APPENDIX

A. Code for Question 1

A.1 Cholesky_lookahead.m

```
function x = Cholesky_lookahead(A,b)
%Choleski decomposition with look ahead modification
[n,m]=size(A);

%Check whether input matrix is a square matrix
if n~=m
    x=nan;
    disp('not a square matrix');
    return
end

%to overwrite A by L and b by y
for j=1:n
    A(j,j)=sqrt(A(j,j));
    b(j,1)=b(j,1)/A(j,j);
    for i= j+1:n
        A(i,j)=A(i,j)/A(j,j);
        b(i)= b(i)-A(i,j)*b(j);
        for k=j+1:i
            A(i,k)=A(i,k)-A(i,j)*A(k,j);
        end
    end
end
U=mat_trans(A);%transpose the edited matrix A to get U

for i=n:-1:1%BACKWARD ELIMINATION(solve Ux=y for x)
        temp_sum=0;
        for j=n:-1:i+1
            temp_sum = temp_sum + U(i,j)*x(j,1);
        end
        x(i,1) = (b(i,1)-temp_sum)/U(i,i);

end
end
```

A.2 sym_posdef_gen.m

```
function A = sym_posdef_gen(n)
%function to create a symmetric positive definite matrix of size n
%If M is any n x n, real, non-singular matrix then,A=MM^t is symmetric positive definite.
    for i=1:n
        for j=1:n
            if i>=j
                P(i,j)=rand()+1;%a lower trianglular matrix is non singular if all diagonal elements are non
zero
                Pt(j,i)=P(i,j);
            else
                P(i,j)=0;
                Pt(j,i)=P(i,j);
            end
        end
    end
```

```
A=mat_mul(P,Pt);
end
```

A3. test_c.m

```
%testbench for Q1. (c)
clear()
A2=sym_posdef_gen(2);
A3=sym_posdef_gen(3);
A4=sym_posdef_gen(4);
A5=sym_posdef_gen(5);

x2=[0.23;-34.32];
x3=[1.23;0.23;1.435];
x4=[10.6;-12.5;1.9;0.234];
x5=[2.67;4.67;-5.78;0.56;-0.23];

b2=mat_mul(A2,x2);
b3=mat_mul(A3,x3);
b4=mat_mul(A4,x4);
b5=mat_mul(A5,x5);

X2=Cholesky_lookahead(A2,b2);
X3=Cholesky_lookahead(A3,b3);
X4=Cholesky_lookahead(A4,b4);
X5=Cholesky_lookahead(A5,b5);
```

A.4 read_data.m

```
function [A,Y,J,E] = read_data(filename,node,branch)
%reads circuit data from file
M=importdata(filename);
for i=1:node
    for j=1:branch
        A(i,j)=M(i,j);
    end
end
for i=1:branch
    J(i,1)=M(i,j+2);
    R(i,1)=M(i,j+3);
    E(i,1)=M(i,j+4);
end
for i=1:branch
    for j=1:branch
        if i==j
            Y(i,j)=1/R(i,1);
        else
            Y(i,j)=0;
        end
    end
end
end
```

A.5 circuit_solve.m

```
function V = circuit_solve(A,Y,J,E) %function to find voltage at the nodes of a given circuit

lhs1=mat_mul(A,Y);
lhs2=mat_trans(A);
lhs_final=mat_mul(lhs1,lhs2);%A*Y*(A^t)

rhs1=mat_mul(Y,E);
rhs2=mat_sub(J,rhs1);
rhs_final=mat_mul(A,rhs2);%A*(J-Y*E)

V=Cholesky_lookahead(lhs_final,rhs_final);%call cholesky function to solve AY(A^t)=A(J-YE)
end
```

B. Code for Question 2

B.1 mesh_final.m

```
function [A,Y,J,E] = mesh_final(n)
%function to create the input data for a N*2N finite difference mesh
node_total=(2*n)*(n);
branch_total=(4*n-3)*n;

%initialize incidence matrix
for i= 1:node_total
    for j=1:branch_total+1
        M(i,j)=0;
    end
end

%populate incidence matrix
for i=1:n
    for j=1:2*n
        node_pos= n*(j-1)+i;
        br_up= node_pos+(n-1)*(j-1);
        br_right=br_up+(n-1);
        br_down= br_up-1;
        br_left=br_up-n;
        if i==1
            if j==1
                M(node_pos,br_right)=1;
                M(node_pos,br_up)=1;
            elseif j==2*n
                M(node_pos,br_up)=1;
                M(node_pos,br_left)=-1;
            else
                M(node_pos,br_right)=1;
                M(node_pos,br_left)=-1;
                M(node_pos,br_up)=1;
            end

        elseif i==n
            if j==1
                M(node_pos,br_right)=1;
                M(node_pos,br_down)=-1;
            elseif j==2*n
                M(node_pos,br_down)=-1;
                M(node_pos,br_left)=-1;
            else
                M(node_pos,br_right)=1;
                M(node_pos,br_left)=-1;
                M(node_pos,br_down)=-1;
            end

        else
            if j==1
                M(node_pos,br_right)=1;
                M(node_pos,br_up)=1;
                M(node_pos,br_down)=-1;
            elseif j==2*n
                M(node_pos,br_up)=1;
                M(node_pos,br_left)=-1;
                M(node_pos,br_down)=-1;
            else
```

```
                M(node_pos,br_right)=1;
                M(node_pos,br_left)=-1;
                M(node_pos,br_up)=1;
                M(node_pos,br_down)=-1;
            end
        end
    end
end

%Add an extra branch between the the corner nodes with current source=1A
M(1,branch_total+1)=-1;
M(node_total,branch_total+1)=1;

%Create the reduced incidence matrix by grounding the last node
for i= 1:node_total-1
    for j=1:branch_total+1
        A(i,j)=M(i,j);
    end
end

%create the diagonal matrix Y with elements 1/R on the diagonal
for i=1:branch_total+1
    for j=1:branch_total+1
        if i==j
            if j~=branch_total+1
                Y(i,j)=1e-3;
            else
                Y(i,j)=0;
            end
        else
            Y(i,j)=0;
        end
    end
end

%create an empty E vector
for i=1:branch_total+1

        E(i,1)=0;

end

%create the J vector with one current source
for i=1:branch_total+1
    if i<branch_total+1
        J(i,1)=0;
    else
        J(i,1)=-1;
    end
end

end
```

B2. solve.m

```
function V = solve(A,Y,J,E,operation) %function to find voltage at the nodes of the mesh
lhs1=mat_mul(A,Y);
lhs2=mat_trans(A);
lhs_final=mat_mul(lhs1,lhs2);%A*Y*(A^t)
hbw=find_band(lhs_final);%find bandwidth of the final square matrix

rhs1=mat_mul(Y,E);
rhs2=mat_sub(J,rhs1);
rhs_final=mat_mul(A,rhs2);%A*(J-Y*E)
if operation =='n'%perform unoptimised cholesky decomposition
    V=Cholesky_lookahead(lhs_final,rhs_final);
elseif operation=='o'%perform optimised cholesky decomposition
    V=Cholesky_sparse(lhs_final,rhs_final,hbw);
end

end
```

B3. Cholesky_sparse.m

```
function x = Cholesky_sparse(A,b,hbw)%Function to solve Ax=b using Choleski algorithm using bandwidth
hbw=hbw-1;
[n,m]=size(A);
%Check whether input matrix is a square matrix
if n~=m
    x=nan;
    disp('not a square matrix');
    return
end

%to overwrite A by L and b by y
for j=1:n
    if A(j,j)<=0
        disp('not a positive definite matrix');
        x=nan;
        return
    end
    A(j,j)=sqrt(A(j,j));
    b(j,1)=b(j,1)/A(j,j);
    for i= j+1:min(n,(j+hbw))%minimising number of operations
        A(i,j)=A(i,j)/A(j,j);
        b(i)= b(i)-A(i,j)*b(j);
        for k=j+1:i
            A(i,k)=A(i,k)-A(i,j)*A(k,j);
        end
    end
end
U=mat_trans(A);

for i=n:-1:1%OPTIMISED BACKWARD ELIMINATION(solve Ux=y for x)
        temp_sum=0;
        for j=(min(n,i+hbw)):-1:i+1
            temp_sum = temp_sum + U(i,j)*x(j,1);
        end
        x(i,1) = (b(i,1)-temp_sum)/U(i,i);

end
end
```

B4. find_band.m

```
function hbw = find_band(A)%to find half bandwidth of a matrix
n=size(A,1);
hbw=0;
    for i=1:n
        for j=1:n
            if A(i,j)~=0
                if abs(i-j)+1>hbw
                    hbw =abs(i-j)+1;
                end
            end
        end
    end
end
```

B5. Q2_final.m

```
%Test bench to solve the circuit mesh using both methods(sparse optimised and
%unoptimised) and compare computation time

clear()
% Solve for n=2,3,4....,10 with the normal unoptimized method
for n = 2:10
        [A, Y, J, E] = mesh_final(n);
        t_start=tic;%start timer
        V = solve(A, Y, J, E,'n');
        t_normal(n-1) = toc(t_start);%stop timer to measure computation time
    R=V(1);
        r(n-1) = R;
end
clearvars -except t_normal

% Solve for n=2,3,4....,10 with the optimized method
for n = 2:10
        [A, Y, J, E] = mesh_final(n);
        t_start=tic;%start timer
        V = solve(A, Y, J, E,'o');
        t_optimised(n-1) = toc(t_start);%stop timer to measure computation time
    R=V(1);
        r(n-1) = R;
end

%Plotting R versus N:
x = 2:10;
plot(x,r)
xlabel('N')
ylabel('Resistance(ohms)')
grid

% Plot unoptimized and sparse optimized computation time graphs
figure(2)
plot(x,t_normal,'o')
xlabel('N')
ylabel('Computation Time')
```

```
hold on
plot(x,t_optimised, '*')
grid
```

C. Code for Question 3

C.1 mesh_cable_final.m

```
function A = mesh_cable_final(h) %Create the grid of nodes
%Using symmetry along x and y axis , consider only one quarter of the area(bottom left)
pot=15;
s_cable=0.1;
core_l=0.04;
core_h=0.02;
xmax = round((s_cable/h)+1);
ymax = round((s_cable/h)+1);

for i=1:ymax %apply dirichlet boundary conditions and initialise every free node = 0
    for j=1:xmax
         if i>= (ymax-(core_h/h)) && j>= (xmax-(core_l/h))
             mesh(i,j) = pot;
         else
             mesh(i,j) = 0;
         end
    end
end
A = mesh;
end
```

C2. SOR_final.m

```
function [phi,itr] = SOR_final(mesh,w,h)
%function which performs successive over relaxation and gives final scaler
%potential and returns the number of iterations
s_cable=0.1;
core_l=0.04;
core_h=0.02;
xmax = round((s_cable/h)+1);
ymax = round((s_cable/h)+1);
r=max_residual_final(mesh,h);
itr=0;
A=mesh;
while r>1e-5
    for i=2:ymax
        for j=2:xmax
            if i< (ymax-(core_h/h)) || j< (xmax-(core_l/h))
                if  j==xmax      %Neuman boundary
                    mesh(i,j) = (1-w)*mesh(i,j)+(w/4)*(mesh(i,j-1)+A(i,j-1)+mesh(i-1,j)+mesh(i+1,j));
                elseif i==ymax   %Neuman boundary
                    mesh(i,j) = (1-w)*mesh(i,j)+(w/4)*(mesh(i,j-1)+mesh(i,j+1)+mesh(i-1,j)+A(i-1,j));
                else             %update mesh using SOR formula
                    mesh(i,j) = (1-w)*mesh(i,j)+(w/4)*(mesh(i,j-1)+mesh(i,j+1)+mesh(i-1,j)+mesh(i+1,j));
                end
            end
        end
    end
    A=mesh;
    itr=itr+1;
    r=max_residual_final(mesh,h);
end
    phi=mesh;
end
```

C3. Jacobi_final.m

```
function [phi,itr] = Jacobi_final(mesh,h)
%function which performs Jacobi method and gives final scaler
%potential and returns the number of iterations
s_cable=0.1;
core_l=0.04;
core_h=0.02;
xmax = round((s_cable/h)+1);
ymax = round((s_cable/h)+1);
r=max_residual_final(mesh,h);
itr=0;
while r>1e-5
    for i=2:ymax
        for j=2:xmax
            if i< (ymax-(core_h/h)) || j< (xmax-(core_l/h))
                if  j==xmax %Neuman boundary
                    mesh(i,j) = (1/4)*(2*mesh(i,j-1)+mesh(i-1,j)+mesh(i+1,j));
                elseif i==ymax %Neuman boundary
                    mesh(i,j) = (1/4)*(mesh(i,j-1)+mesh(i,j+1)+2*mesh(i-1,j));
                else %update mesh using Jacobi formula
                    mesh(i,j) = (1/4)*(mesh(i,j-1)+mesh(i,j+1)+mesh(i-1,j)+mesh(i+1,j));
                end
            end
        end
    end
    itr=itr+1;
    r=max_residual_final(mesh,h);
end
    phi=mesh;
end
```

C4. max_residual_final.m

```
function max_res = max_residual_final(mesh,h)%calculate maximum of all the residuals at the free nodes

s_cable=0.1;
core_l=0.04;
core_h=0.02;
xmax = round((s_cable/h)+1);
ymax = round((s_cable/h)+1);
max_res=0;
for i=2:ymax
    for j=2:xmax
        if i< (ymax-(core_h/h)) || j< (xmax-(core_l/h))
            if  j==xmax %Neuman boundary
                res = 2*mesh(i,j-1)+mesh(i-1,j)+mesh(i+1,j)-4*mesh(i,j);
                res = abs(res);
            elseif  i==ymax %Neuman boundary
                res = mesh(i,j-1)+mesh(i,j+1)+2*mesh(i-1,j)-4*mesh(i,j);
                res = abs(res);
            else
                res = mesh(i,j-1)+mesh(i,j+1)+mesh(i-1,j)+mesh(i+1,j)-4*mesh(i,j);
                res = abs(res);
            end

            if res > max_res
                max_res = res;
            end
```

```
        end
    end
end

end
```

C5. get_potential.m

```
function V = get_potential(x,y,phi,h)%to find potential at any point (x,y) accross entire cable
    if x<=0.1
        if y<=0.1
            V=phi(round(y/h+1),round(x/h+1));
        elseif y>0.1 && y<=0.2
            V=phi(round((0.2-y)/h+1),round(x/h+1));
        else
            disp('given coordinate doesnt exist on the cable');
            V=nan;
        end

    elseif x>0.1 && x<=0.2
        if y<=0.1
            V=phi(round(y/h+1),round((0.2-x)/h+1));
        elseif y>0.1 && y<=0.2
            V=phi(round((0.2-y)/h+1),round((0.2-x)/h+1));
        else
            disp('given coordinate doesnt exist on the cable');
            V=nan;
        end

    else
        disp('given coordinate doesnt exist on the cable');
            V=nan;
    end

end
```

C6. mesh_nu.m

```
function A = mesh_nu(x,y)
%Create the grid of nodes for non uniform node spacing
%Using symmetry along x and y axis , consider only one quarter of the
%area(bottom left)
pot=15;
s_cable=0.1;
for i=1:length(y)
    for j=1:length(x)
        if x(j)>s_cable || y(i)>s_cable
            disp('out of bounds');
            A=nan;
        else
            if x(j)>=0.06 && y(i)>=0.08%apply dirichlet boundary conditions
                mesh(i,j)=pot;
            else
                mesh(i,j)=0;
            end
        end
    end
end
```

```
A=mesh;
end
```

C7. SOR_nu.m

```
function [phi,itr] = SOR_nu(mesh,w,x,y)
%function which performs successive over relaxation and gives final scaler
%potential and returns the number of iterations

for i=2:length(x)
    hx(i-1)=x(i)-x(i-1);
end
for i=2:length(y)
    hy(i-1)=y(i)-y(i-1);
end
r=max_residual_nu(mesh,x,y);
itr=0;
A=mesh;
while r>1e-5
    for i=2:length(y)
        for j=2:length(x)
            if x(j)< 0.06 || y(i)< 0.08
                if j==length(x) %Neuman boundary
                    alpha= 1/(hx(j-1)^2) + 1/(hy(i-1)*(hy(i)+hy(i-1))) + 1/(hy(i)*(hy(i)+hy(i-1)));
                    mesh(i,j) = (1-w)*mesh(i,j)+(w/alpha)*(mesh(i,j-1)/(2*(hx(j-1)^2)) + A(i,j-1)/(2*(hx(j-
1)^2)) + mesh(i-1,j)/(hy(i-1)*(hy(i)+hy(i-1))) + mesh(i+1,j)/(hy(i)*(hy(i)+hy(i-1))));
                elseif i==length(y) %Neuman boundary
                    alpha= 1/(hx(j-1)*(hx(j)+hx(j-1))) + 1/(hx(j)*(hx(j)+hx(j-1))) + 1/(hy(i-1)^2);
                    mesh(i,j) = (1-w)*mesh(i,j)+(w/alpha)*(mesh(i,j-1)/(hx(j-1)*(hx(j)+hx(j-1))) +
mesh(i,j+1)/(hx(j)*(hx(j)+hx(j-1))) + mesh(i-1,j)/(2*(hy(i-1)^2)) + A(i-1,j)/(2*(hy(i-1)^2)));
                else %update mesh using SOR formula
                    alpha= 1/(hx(j-1)*(hx(j)+hx(j-1))) + 1/(hx(j)*(hx(j)+hx(j-1))) + 1/(hy(i-1)*(hy(i)+hy(i-
1))) + 1/(hy(i)*(hy(i)+hy(i-1)));
                    mesh(i,j) = (1-w)*mesh(i,j)+(w/alpha)*(mesh(i,j-1)/(hx(j-1)*(hx(j)+hx(j-1))) +
mesh(i,j+1)/(hx(j)*(hx(j)+hx(j-1))) + mesh(i-1,j)/(hy(i-1)*(hy(i)+hy(i-1))) +
mesh(i+1,j)/(hy(i)*(hy(i)+hy(i-1))));
                end
            end
        end
    end
    A=mesh;
    itr=itr+1;
    r=max_residual_nu(mesh,x,y);
end
    phi=mesh;
end
```

C8. max_residual_nu.m

```
function max_res = max_residual_nu(mesh,x,y)
%calculate maximum of all the residuals at the free nodes for the non
%uniform node spacing

for i=2:length(x)
    hx(i-1)=x(i)-x(i-1);
end
for i=2:length(y)
    hy(i-1)=y(i)-y(i-1);
end
```

```
max_res=0;
for i=2:length(y)
    for j=2:length(x)
        if x(j)< 0.06 || y(i)< 0.08
            if j==length(x) %Neuman boundary
                alpha= 1/(hx(j-1)^2) + 1/(hy(i-1)*(hy(i)+hy(i-1))) + 1/(hy(i)*(hy(i)+hy(i-1)));
                res=(mesh(i,j-1)/(hx(j-1)^2) + mesh(i-1,j)/(hy(i-1)*(hy(i)+hy(i-1))) +
mesh(i+1,j)/(hy(i)*(hy(i)+hy(i-1))))-(alpha*mesh(i,j));
                res = abs(res);
            elseif  i==length(y) %Neuman boundary
                alpha= 1/(hx(j-1)*(hx(j)+hx(j-1))) + 1/(hx(j)*(hx(j)+hx(j-1))) + 1/(hy(i-1)^2);
                res = (mesh(i,j-1)/(hx(j-1)*(hx(j)+hx(j-1))) + mesh(i,j+1)/(hx(j)*(hx(j)+hx(j-1))) + mesh(i-
1,j)/(hy(i-1)^2))-(alpha*mesh(i,j));
                res = abs(res);
            else
                alpha= 1/(hx(j-1)*(hx(j)+hx(j-1))) + 1/(hx(j)*(hx(j)+hx(j-1))) + 1/(hy(i-1)*(hy(i)+hy(i-1)))
+ 1/(hy(i)*(hy(i)+hy(i-1)));
                res = (mesh(i,j-1)/(hx(j-1)*(hx(j)+hx(j-1))) + mesh(i,j+1)/(hx(j)*(hx(j)+hx(j-1))) + mesh(i-
1,j)/(hy(i-1)*(hy(i)+hy(i-1))) + mesh(i+1,j)/(hy(i)*(hy(i)+hy(i-1))))-(alpha*mesh(i,j));
                res = abs(res);
            end

            if res > max_res
                max_res = res;
            end
        end
    end
end

end
```

C9. Q3_uniform_node.m

```
%test Bench for Q3 to apply SOR and Jacobi methods for uniform node spacing
clear()
h=[0.02,0.01,0.005,0.002,0.001,0.0005];
w=linspace(1.1,1.9,10);
A=mesh_cable_final(h(1));
%SOR method for different values of w at h=0.02
for i=1:10
    [phi,itr1(i)]=SOR_final(A,w(i),h(1));
    V1=get_potential(0.06,0.04,phi,h(1));
end
% Plot iterations vs w for SOR method
figure(1)
plot(w,itr1,'-o')
xlabel('omega')
ylabel('Iterations')
grid

for i=1:6 %SOR method for different values of h at optimal w=1.35
    A=mesh_cable_final(h(i));
    [phi,itr2(i)]=SOR_final(A,1.35,h(i));
    V(i)=get_potential(0.06,0.04,phi,h(i));
end

for i=1:6%Jacobi method for different values of h at optimal w=1.35
    A=mesh_cable_final(h(i));
    [phi,itr3(i)]=Jacobi_final(A,h(i));
    V2(i)=get_potential(0.06,0.04,phi,h(i));
```

```
end
% Plot potential at (0.06,0.04) vs 1/h for SOR and Jacobi method
figure(2)
plot(1./h,V1,1./h,V2,'--')
xlabel('1/h')
ylabel('V')
grid
% Plot iterations vs 1/h for SOR and Jacobi method
figure(3)
plot(1./h,itr2,1./h,itr3,'--')
xlabel('1/h')
ylabel('iterations')
grid
```

D. Code for General Matrix Operations

D1. mat_trans.m

```
function At = mat_trans(A)
%function to transpose a matrix

[n,m]=size(A);
for i=1:n
    for j=1:m
        At(j,i)=A(i,j);
    end
end
end
```

D2. 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
```

D3. 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
```

D4. 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
```