Thomas Christinck
260601793

ECSE 543 Assignment I

Note: All generic functions used in this assignment (matrix multiplication, transposition, etc.) can be found under Appendix 4 (pages 25 – 27). To test any code, see the code submission. Navigate to Assignment1/Q{'question number'}. Run test.m (For Q3, testnu.m runs the code for section 3(e)).

1. (a) A program to solve the matrix equation $Ax=b$ by Cholesky decomposition was written and can be found under Appendix 1. A function that implements Cholesky decomposition can be found as cholesky.m (Appendix 1.1., page 11). A function that takes the decomposition L and b and solves for x can be found as forwardelim_backsub.m (see Appendix 1.2, page 12).

1. (b) Symmetric, real, positive-definite matrices of size $n = 2, \ldots, 10$ were constructed to test the program with (Figure 1 illustrates program input/output results for n = 5). These matrices were created by generating a lower triangular matrix L with random entries $L_{i,j} \epsilon(0,10)$ and multiplying it by its transpose. We know that a matrix A is symmetric positive definite if it can be written as $A = LL^T$, where L is a lower triangular matrix. Note the diagonal entries of L must be non-zero (the columns of L must be independent). The program written to produce these matrices can be found under Appendix 1.3. as spd_generator.m (page 12).
   Note: To verify that a matrix A is positive-definite, we check that all the entries of its Cholesky decomposition matrix L are real (see Appendix 1.1, cholesky.m on page 11).

```
>> [A_5] = spd_generator(5);
A_5 = round(A_5, 2)

A_5 =

  Columns 1 through 4

   22.58   41.22   31.21   11.39
   41.22  157.37  136.89  109.4
   31.21  136.89  172.52  121.01
   11.39  109.4   121.01  108.83
   17.41   61.39   75.83   50.86

  Column 5

   17.41
   61.39
   75.83
   50.86
      41
```

Figure 1 - Example random real, positive definite, symmetric matrix for n = 5

1. (c) The program written in (a) was tested using the matrices in (b) by inventing an x, multiplying Ax = b, and then giving A and b to the program to verify that it returns x. Figure 2 illustrates the program input/output n=5. To test n = 2, …, 10, see test.m in the code submission (not included in the Appendix due to length).

```
>> tgt_5 = [1;2.789;-8.908;3.2098;-0.9087];
[b_5] = matrix_multiply(A_5, tgt_5);
[L_5] = cholesky(A_5);
[y_5, X_5] = forwardelim_backsub(L_5, b_5);
disp(round(X_5, 4))
                         1
                    2.789
                   -8.908
                    3.2098
                   -0.9087
```

Figure 2 - Example matrix solution given an invented x (tgt_5). The algorithm correctly solves the equation given matrices A and b.

1. (d) A program was written that reads from a .csv file a list of network branches (J_k, R_k, E_k) and a reduced incidence matric and can be found in Appendix 1.4 as csv_matrix.m (page 13). A program was written that finds the voltages at the nodes of the network can be found in Appendix 1.5 as voltage_solver.m (page 13). The data in the .csv file should be organized as: column for incidence matrix, empty column, J column, R column, and E column (an example is submitted as circuit_data.csv).
   The program was tested with the test circuits provided on myCourses (see Appendix 1.6 on page 14). The program inputs and outputs are shown in Figure 3. Note that the program outputs correspond to the correct expected values shown in Appendix 1.6.

```
>> [I_1, E_1, J_1, R_1] = csv_matrix(0,0,0,1);
[I_2, E_2, J_2, R_2] = csv_matrix(2,2,0,1);
[I_3, E_3, J_3, R_3] = csv_matrix(4,4,0,1);
[I_4, E_4, J_4, R_4] = csv_matrix(6,7,0,3);
[I_5, E_5, J_5, R_5] = csv_matrix(10,12,0,5);

[V_1] = voltage_solver(I_1, E_1, J_1, R_1)
[V_2] = voltage_solver(I_2, E_2, J_2, R_2)
[V_3] = voltage_solver(I_3, E_3, J_3, R_3)
[V_4] = voltage_solver(I_4, E_4, J_4, R_4)
[V_5] = voltage_solver(I_5, E_5, J_5, R_5)
```

```
V_1 =

     5

V_2 =

    50

V_3 =

    55

V_4 =

    20
    35

V_5 =

     5
  3.75
  3.75
```

Figure 3 - Node voltages for test circuits in Appendix 1.6

2. (a) Using the program written in 1(d) (voltage_solver.m, see Appendix 1.5 on page 13), we find the resistance R of the mesh for N = 2, …, 15. A program was written to generate incidence matrices *I*, current source vectors *J*, voltage source vectors *E*, and resistance vectors *R*. It can be found under Appendix 2.1. as generate_mesh.m (page 15). A sample input and output for *N = 4* is shown in Figure 4. The computed *R* values along with computation time are shown in Table 1.

```
>> [I, E, J, R] = generate_mesh(4);
        [V] = voltage_solver(I, E, J, R);
        R = V(1) / (1.00 - V(1))

R =

          18571.4285714958
```

Figure 4 – Example input/output for calculating mesh equivalent resistance for n = 4

N	R	Computation Time (s)
2	15000	0.26e-3
3	18571	0.24e-3
4	21364	0.28e-3
5	23657	0.60e-3
6	25601	1.90e-3
7	27290	2.65e-3
8	28781	6.57e-3
9	30117	1.28e-2
10	31326	2.51e-2
11	32430	4.64e-2
12	33447	6.97e-2
13	34388	0.108
14	35265	0.174
15	36090	0.276

Table 1 – R values and associated computation time for N = 2, …, 15

2. (b) The computation time for Cholesky decomposition should be $O(n^3)$ theoretically. Here, $n = N^2$ and so the computation time is $O(N^6)$. Figure 5 shows a scatter plot of computation time as N increases fit with a 6th degree polynomial, where time t is given by $t(N) = (5.21e\text{-}06)N^6 - 0.00052N^5 + 0.00148N^4 - 0.000649N^3 + 0.000109N^2 - (7.90e\text{-}06)N$. The equation fits well for large N (with some discrepancy for small N, where the Cholesky computation does not dominate).

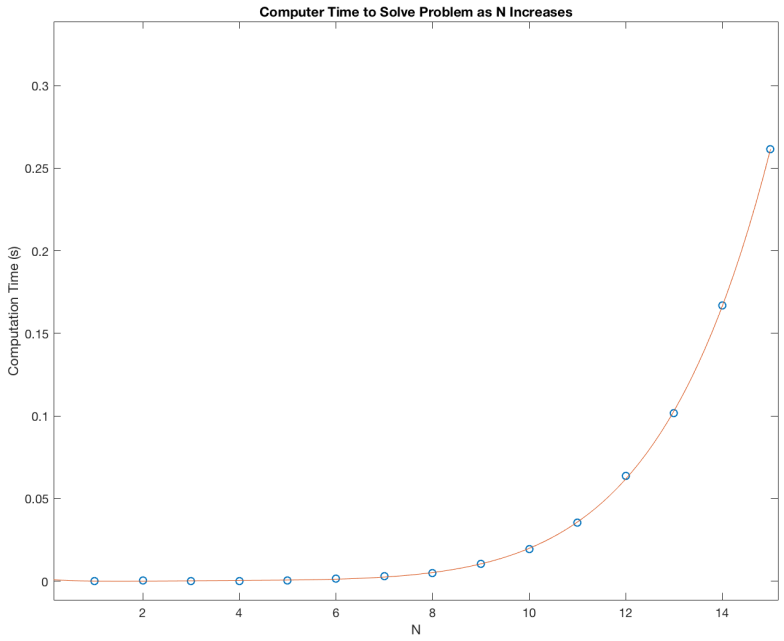


Figure 5 - Computation time to solve problem as N increases, with polynomial fit

2. (c) The program from 2(b) was modified to exploit the sparse nature of the matrices. The modified functions can be found in Appendix 2.2 and 2.3 as sparse_cholesky.m and sparse_fwdelim_bcksub.m, respectfully (pages 17 and 18). Theoretically, the problem now has computation time $O(b^2n) * O(N^2)$, where b is the mean half-bandwidth. Since b ~= N, this gives computation time $O(N^6)$. Note that the additional $O(N^2)$ term arises due to the pre-processing time introduced by the sparse implementations. Timings for the sparse implementation and the unoptimized implementation, along with the half-bandwidth, for N = 2, …, 15, are shown in Table 2. The difference in timing is more apparent for larger N, as shown in Figure 6.

N	Unoptimized Computation Time(s)	Sparse Computation Time (s)	Half Bandwidth
2	0.41e-3	0.31e-3	3
3	0.24e-3	0.30e-3	4
4	0.28e-3	0.28e-3	5
5	0.60e-3	0.64e-3	6
6	1.90e-3	2.44e-3	7
7	2.65e-3	2.78e-3	8
8	6.57e-3	5.90e-3	9
9	1.28e-2	1.28e-2	10
10	2.51e-2	2.50e-2	11
11	4.64e-2	4.10e-2	12
12	6.97e-2	6.92e-2	13
13	0.108	0.103	14
14	0.174	0.167	15
15	0.276	0.269	16

Table 2 - Computation time for unoptimized and sparse models. Half bandwidth is included. Note half bandwidth is equal to N + 1

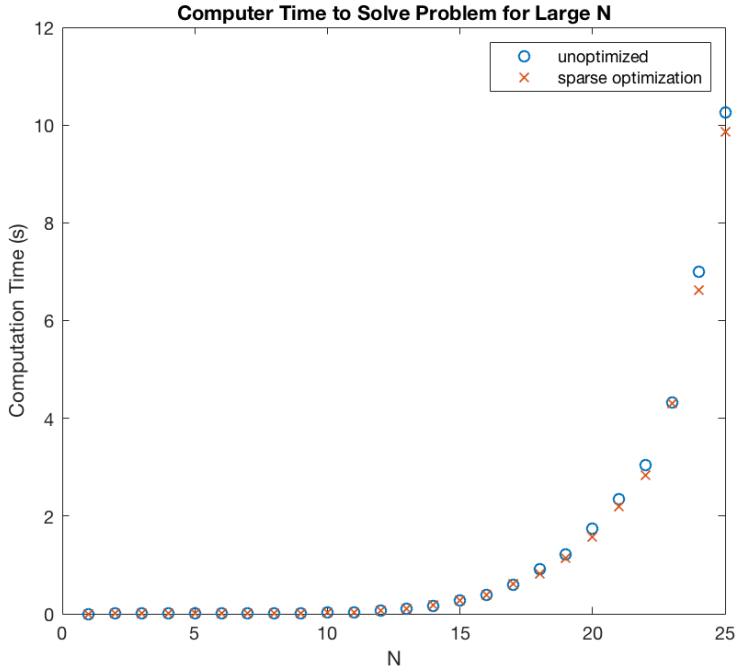


Figure 6 - Computer time to solve problem for N = 2, ..., 25. Note that the sparse optimization method performs better for larger N.

2. (d) A graph of R versus N was plotted in Figure 7. The equation *R(N) = 12564.07 ln(N) + 1193.4* fits the curve reasonably well. Since $\lim_{N\to\infty} \ln(N) = \infty$, the curve will approach infinity as N tends to infinity. This is correct, as the resistance of an infinite mesh will be infinite.

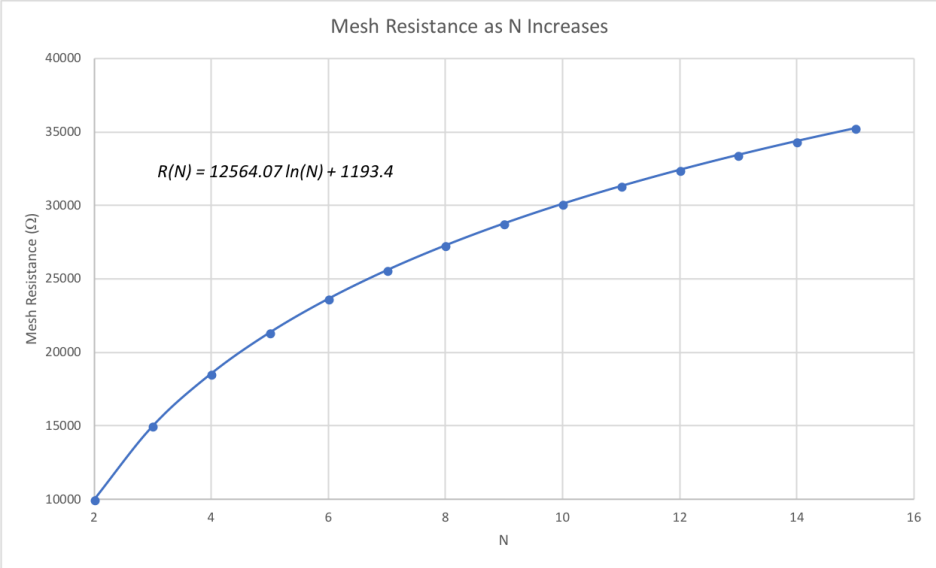


Figure 7 - Mesh resistance R (in ohms) as N increases with corresponding equation

3. (a) A program to find the potential at the nodes of a regular mesh in the air between conductors by the finite difference method was written and can be found in Appendix 3.1 as get_potential.m (page 20). The translational and rotational symmetry of the problem was exploited and ¼ of the cable is considered in calculations. A function that implements successive order relaxation can be found in Appendix 3.2 as sor.m (see page 20). A function that computes the max residual can be found in Appendix 3.3 as compute_residual.m (see page 21). A function that iterates over SOR until the maximum residual of the free nodes is less than 10^{-5} can be found in Appendix 3.4 as iterate.m (see page 21).

3. (b) With h = 0.02, we vary ω for 10 values from 1.0 to 2.0. Corresponding potentials at the point (*x, y*) = (0.06, 0.04), as well as the number of iterations taken to achieve convergence, are summarized in Table 3. A graph of the number of iterations versus ω can be found in Figure 8. The code for varying ω can be found in the code submission under Q3/test.m (not included in the Appendix due to length).

ω	Iterations to Convergence	Potential at (0.06, 0.04) (V)
1	31	42.265
1.11	23	42.265
1.22	14	42.265
1.33	16	42.265
1.44	21	42.265
1.56	29	42.265
1.67	41	42.265
1.78	68	42.265
1.89	145	42.265
2	>400 or DNC	-18.01

Table 3 - Iterations to convergence and computed potential at convergence at (0.06, 0.04) with varying omega

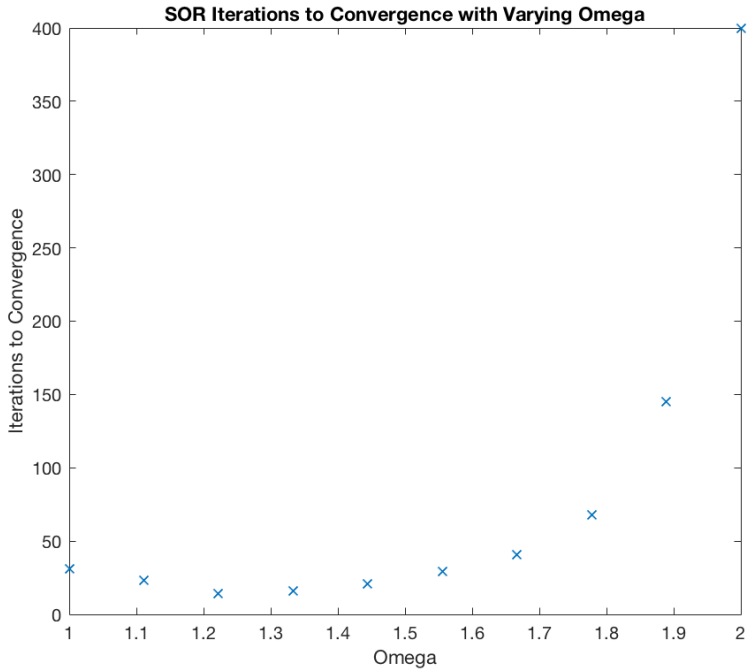


Figure 8 - SOR iterations to convergence as omega varies. Note the optimum omega occurs at approximately $\omega = 1.25$

3. (c) From the plot in Figure 8, an ω value of 1.25 was chosen. Using this ω, h is varied from 0.02 to 0.0005 and the corresponding value of the potential was calculated at the point (*x, y*) = (0.06, 0.04). Results are tabulated in Table 4. The potential at (0.06, 0.04) is plotted against 1/h in Figure 9. We can see from Figure 9 and Table 4 that as 1/h increases, the potential approaches approximately 40.4 V, which we estimate as the potential at (0.06, 0.04) to three significant figures. The number of iterations is plotted against 1/h in Figure 10. From the figure, we see that as 1/h increases, the number of iterations increases in an exponential fashion.

1/h	Potential at (0.06, 0.04) (V)	Iterations
50	42.265	13
100	41.082	68
200	40.690	261
250	40.629	397
500	40.525	1428
1000	40.483	5012
1250	40.474	7472
2000	40.456	17171

Table 4 - Number of iterations to convergence and potential at (0.06, 0.04) as 1/h increases, using the SOR method

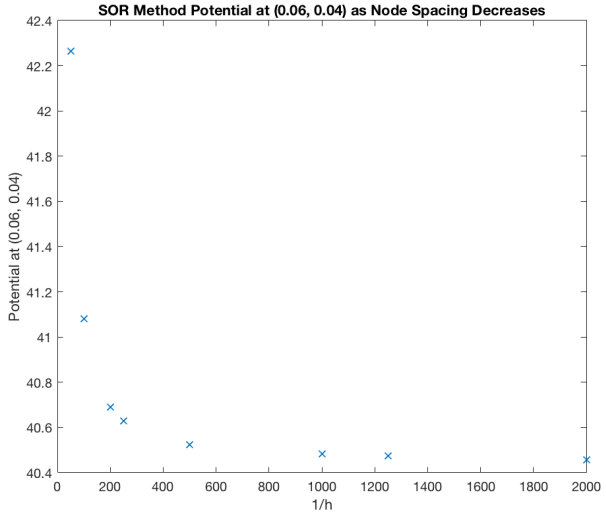


Figure 9 - Potential at (0.06, 0.04) as 1/h increases, using the SOR method

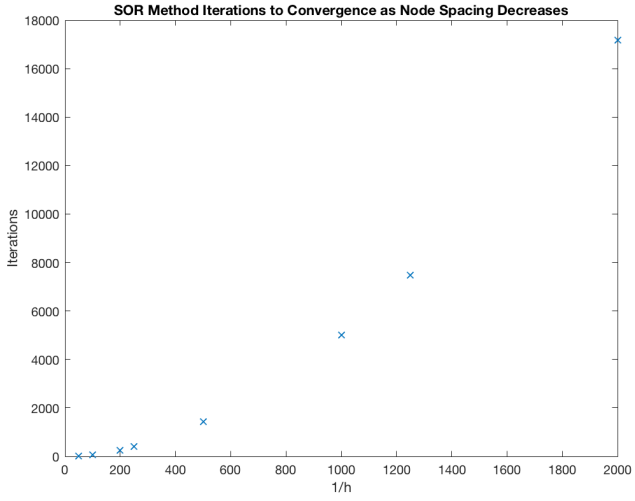


Figure 10 - Number of iterations to convergence as 1/h increases, using the SOR method

3. (d) The values of h from c) were used and the Jacobi method (see jacobian.m, Appendix 3.5, page 22) was implemented to solve for the potential at (0.06, 0.04). Results are tabulated in Table 5. The potential at (0.06, 0.04) is plotted against 1/h in Figure 11. We can see from Figure 11 and Table 5 that as 1/h increases, the potential approaches approximately 40.4 V, which we estimate as the potential at (0.06, 0.04) to three significant figures. The number of iterations is plotted against 1/h in Figure 12. From the figure, we see that as 1/h increases, the number of iterations increases in an exponential fashion. Note that the number of iterations to convergence is notably larger when using the Jacobi method as expected, as it is a less efficient algorithm. The results, however, are approximately identical.

1/h	Potential at (0.06, 0.04) (V)	Iterations
50	42.265	31
100	41.082	120
200	40.690	442
250	40.629	669
500	40.525	2391
1000	40.483	8374
1250	40.474	12478
2000	40.456	28657

Table 5 - Number of iterations to convergence and potential at (0.06, 0.04) as 1/h increases, using the Jacobi method

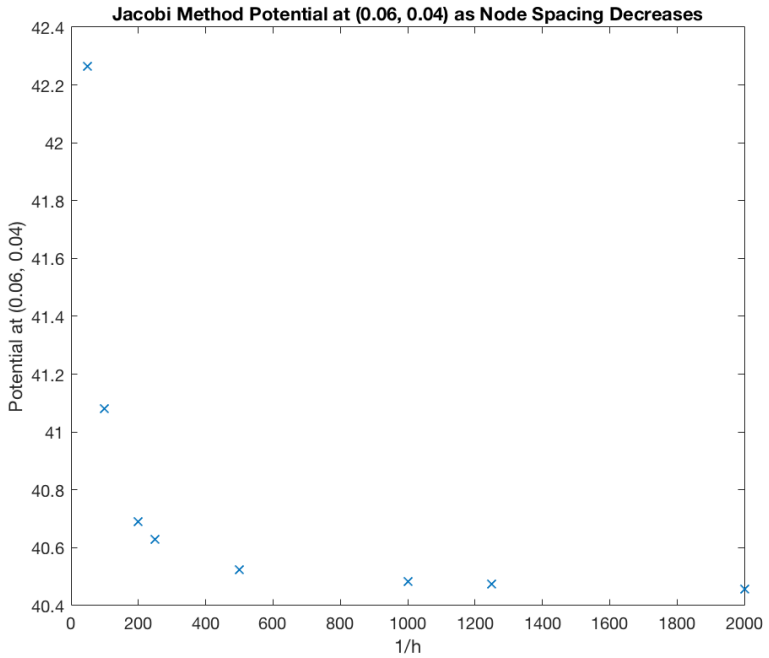


Figure 11 - Potential at (0.06, 0.04) as 1/h increases, using the Jacobi method

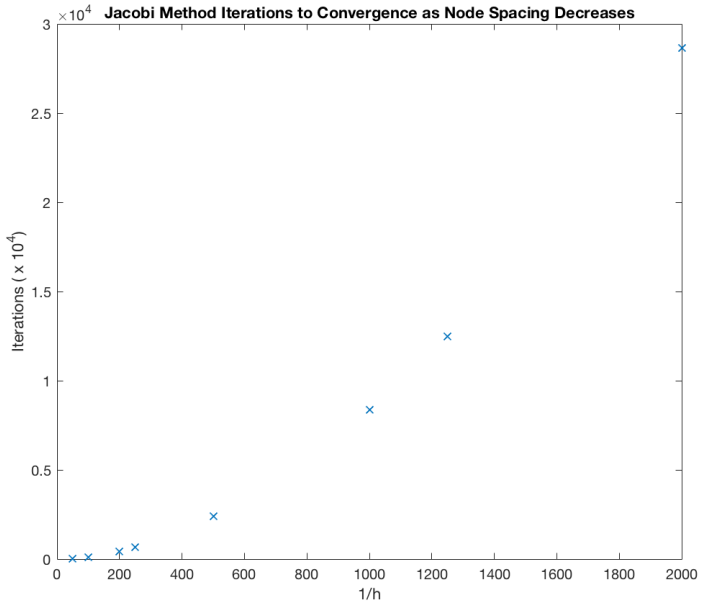


Figure 12 - Number of iterations to convergence as 1/h increases, using the Jacobi method

3. (e) The code from questions 3(a) – 3(d) was modified for non-uniform node spacing. See generate_mesh_nonuniform.m, sor_nonuniform.m, compute_res_nonuniform.m, and get_potential_nonuniform.m in Appendix 3.6, 3.7, 3.8, and 3.9, respectively (pages 22 to 24). First, when we test the code with a uniform node spacing of 0.01, we obtain the same result as in c) for h = 0.01. If we modify the grid so that the mesh resolution becomes slightly finer in the neighbourhood of the point of interest, we get a potential of 41.042 V, somewhat closer to the actual potential found in c) of 40.44 V found in c). See Figure 13 for printed results.

```
>> x_grid = [0 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1];
y_grid = [0 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1];
>> % A dummy h for testing
w = 1.25;
h = 0.01;
mesh = generate_mesh_nonuniform(x_grid, y_grid);
[new_mesh, iterations] = iterate(mesh, h, w, x_grid, y_grid, 'snu');
V = get_potential_nonuniform(new_mesh, x, y, x_grid, y_grid)

V =

          41.0817996169974
```

```
>> x_grid = [0 0.01 0.02 0.03 0.04 0.052 0.06 0.069 0.08 0.09 0.1];
y_grid = [0 0.01 0.02 0.031 0.04 0.05 0.06 0.07 0.08 0.09 0.1];
>> % A dummy h for testing
w = 1.25;
h = 0.01;
mesh = generate_mesh_nonuniform(x_grid, y_grid);
[new_mesh, iterations] = iterate(mesh, h, w, x_grid, y_grid, 'snu');
V = get_potential_nonuniform(new_mesh, x, y, x_grid, y_grid)

V =

          41.0419536112251
```

Figure 13 - Potential at (0.06, 0.04) for a non-uniform mesh. Using uniform spacing we get a potential of 41.08 V. Using a finr resolution in the neighbourhood of the point we get a potential of 41.04 V.

APPENDIX 1

1. cholesky.m

```
function [L] = cholesky(A)
%  Cholesky decomposition algorithm.
%  [L] = cholesky(A,b) solves the equation A = LL^T. A must be a
%  symmetric, positive-definite, real matrix.
%  Author: Thomas Christinck, 2018.

  size_A = size(A);
  length = size_A(1);
  L(1:length,1:length)=0;

  if length ==1
    L = A;
  else
    for i = 1:length
      for k = first_nonzero(i):i
        sum = 0;
        if A(i,k) ~= A(k,i)
          error("The matrix needs to be symmetric")
        end;
        for j = 1:k
          sum = sum + (L(i,j) * L(k,j));
        end;
        if (i == k)
          if ((A(i,i) - sum)) < 0
            error("The matrix needs to be positive definite")
          end;
          L(i,k) = sqrt(abs(A(i,i) - sum));
        else
          L(i,k) = (A(i,k) - sum) / L(k,k);
        end;
      end;
    end;
  end;
```

2. forwardelim_backsub.m

```
function[y, x] = forwardelim_backsub(L, b)
%  Function that performs both forward elimination and back substitution
%  i.e. in forward elimination it solves for y in Ly = b and in back
%  substitution it solves for x in L^Tx = y
%  Author: Thomas Christinck, 2018.

  size_L = size(L);
  length = size_L(1);

  y(1:length)=0;
  for i = 1:length
     sum = 0;
     for j = 1:i
        sum = sum + (L(i, j) * y(j));
     end;
     y(i) = (b(i) - sum) / L(i, i);
  end;

  x(1:length, 1)=0;
  for i = length:-1:1
     sum = 0;
     for j = i:length
        sum = sum + (L(j, i) * x(j));
     end;
     x(i) = (y(i) - sum) / L(i,i);
  end;
```

3. spd_generator.m

```
function[A] = spd_generator(length)
%  Function that creates a random symmetric positive definite
%  matrix A, given a length input.

    for i = 1:length
       for j = 1:i
           L(i,j) = 10 * rand();
        end;
    end;
    A(1:length,1:length)=0;
    A = matrix_multiply(L, mat_transpose(L));
    A = matrix_multiply(L, mat_transpose(L));
```

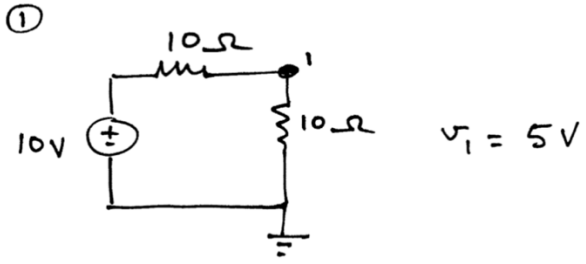
4. csv_matrix.m

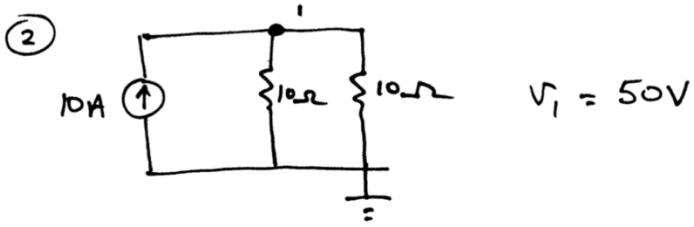
```
function[I, E, J, R] = csv_matrix(row_start, row_end, col_start, col_end)
% Reads a csv file into incidence matrix I, voltage source matrix E, current
% source matrix J, and resistance matrix R. Must specify where in the csv the
% incidence matrix is specified.
   file_name = 'circuit_data.csv';
   EJR_column = col_end + 2;
   I = csvread(file_name, row_start, col_start, [row_start, col_start, row_end, col_end]);
   E= csvread(file_name, row_start, EJR_column, …
                  [row_start,EJR_column,row_start+col_end,EJR_column]);
   J = csvread(file_name, row_start, EJR_column + 1, [row_start,EJR_column …
                  + 1,row_start+col_end,EJR_column + 1]);
   R = csvread(file_name, row_start, EJR_column + 2, [row_start,EJR_column …
                  + 2,row_start+col_end,EJR_column + 2]);
```

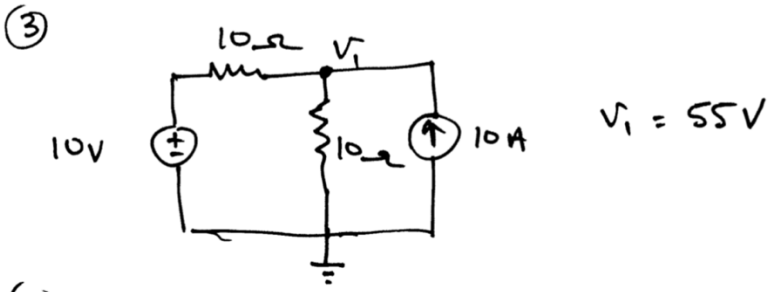
5. voltage_solver.m

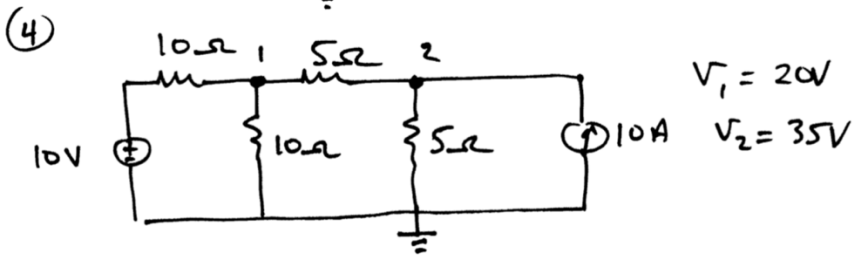
```
function[V] = voltage_solver(incident_matrix, E, J, R)
%Equation to solve: (A . Y . A^T)Vn = A(J-Y . E)
%step_1 = Y . E
%Step_2 = J-Step_1
%Step_3 = A . Step_2
%Step_4 = A^T
%Step_5 = Y . Step_4
%Step_6 = A . Step_5

   %Gives B
   Y = diag_mat(R);
   step_1 = matrix_multiply(Y, E);
   step_2 = matrix_add_subtract(J,step_1,'s');
   step_3 = matrix_multiply(incident_matrix,step_2);

   % Gives A
   step_4 = mat_transpose(incident_matrix);
   step_5 = matrix_multiply(Y, step_4);
   step_6 = matrix_multiply(incident_matrix,step_5);

   % Now we want to compute the voltage
   step_7a = cholesky(step_6);
   % Forward elim solves for y in Ly =b  Ly = 100, Lx = y
   % Back sub for x in L^T X = y
   [step_7b, step_7c]  = forwardelim_backsub(step_7a, step_3);
   V = step_7c;
```

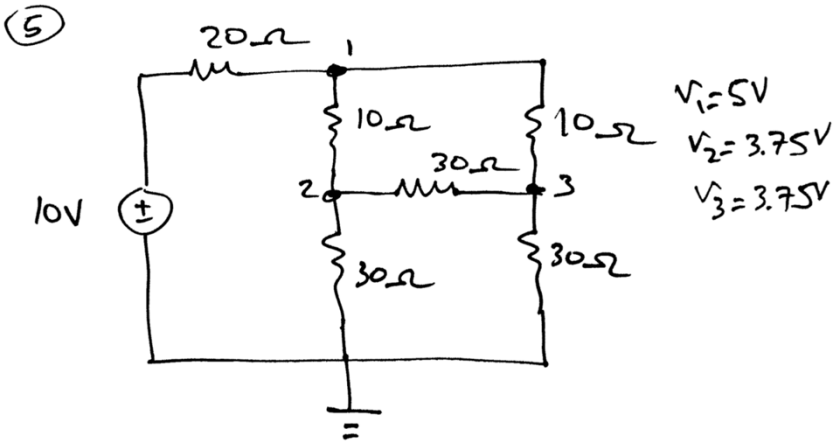
6. **Test Circuits**











APPENDIX 2

1. generate_mesh.m

```
function [I, E, J, R] = generate_mesh(N)

%  Mesh generation algorithm.
%  Given an N, creates an N x N mesh
%  Author: Thomas Christinck, 2018.

  I = generate_incidence_mat(N);
  E = generate_E_vector(N);
  J = generate_J_vector(N);
  R = generate_R_vector(N);

  function [I] = generate_incidence_mat(N)
    % Set up node count (total nodes in mesh), branch count (total branches), and
    % incidence matrix
    node_count = N^2;
    branch_count = 2 * N * (N-1);
    incidence_matrix(1:node_count, 1:(branch_count + 1)) = 0;

    for i = 1:N
      for j = 1:N
        node_idx = N * (j -1) + i;
        branch_above = node_idx + (N - 1) * (j - 1);
        branch_below = branch_above - 1;
        branch_left = branch_above - N;
        branch_right = branch_above + N - 1;

        % Now we populate the incidence matrix to describe our mesh. First, check
        % if we're in the far right column. We use the convention +1 for current leaving
        % a node, -1 for entering a node

        if j == N
          incidence_matrix(node_idx, branch_left) = -1;
          if i == 1
            incidence_matrix(node_idx, branch_above) = 1;
          elseif i == N
            incidence_matrix(node_idx, branch_below) = -1;
          else
            incidence_matrix(node_idx, branch_above) = 1;
            incidence_matrix(node_idx, branch_below) = -1;
          end;
```

```
        elseif j == 1
          incidence_matrix(node_idx, branch_right) = 1;
          if i == 1
            incidence_matrix(node_idx, branch_above) = 1;
          elseif i == N
            incidence_matrix(node_idx, branch_below) = -1;
          else
            incidence_matrix(node_idx, branch_above) = 1;
            incidence_matrix(node_idx, branch_below) = -1;
          end;
        else
          incidence_matrix(node_idx, branch_left) = -1;
          incidence_matrix(node_idx, branch_right) = 1;
          if i == 1
            incidence_matrix(node_idx, branch_above) = 1;
          elseif i == N
            incidence_matrix(node_idx, branch_below) = -1;
          else
            incidence_matrix(node_idx, branch_above) = 1;
            incidence_matrix(node_idx, branch_below) = -1;
          end;
        end;
        % Now we connect a voltage source between the bottom left and top right of
        % the mesh.
        incidence_matrix(1,branch_count + 1) = -1;
        incidence_matrix(node_count, branch_count + 1) = 1;
      end;
    end;

    incidence_matrix_reduced(1:node_count-1, 1:branch_count + 1) = 0;
    for i = 1:(branch_count + 1)
      for j = 1:(node_count - 1)
        incidence_matrix_reduced(j, i) = incidence_matrix(j, i);
      end;
    end;
    I = incidence_matrix_reduced;

function [E] = generate_E_vector(N)
    % The E vector will consist of 0 V everywhere except for the last branch, where we
    % will place a 1 V test source
    branch_count = 2 * N * (N-1);
    E(1:branch_count + 1, 1) = 0;
    E(branch_count + 1, 1) = 1.00 ;
```

```
function [J] = generate_J_vector(N)
    % The J vector will consist of 0 A everywhere (there are no current sources)
    branch_count = 2 * N * (N-1);
    J(1:branch_count + 1, 1) = 0;

function [R] = generate_R_vector(N)
    % There is a 10 kohm resistor connecting all the nodes, except for the last branch
    % where we will place a 1 ohm resistor in series with the test source
    branch_count = 2 * N * (N-1);
    R(1:branch_count + 1, 1) = 10000;
    R(branch_count + 1, 1) = 1.00;
```

2. <u>sparse_cholesky.m</u>

```
function [L, first_nonzero, half_bw] = sparse_cholesky(A)
%   Cholesky decomposition algorithm with sparse optimization.
%   [L] = cholesky(A,b) solves the equation A = LL^T. A must be a
%   symmetric, positive-definite, real matrix.


%   Author: Thomas Christinck, 2018.
  size_A = size(A);
  length = size_A(1);
  L(1:length,1:length)=0;

  if length ==1
    L = A;
  else
    % Find first non-zero indices for half band width
    first_nonzero(1:length) = length;
    half_bw = 0;
    for i = 1:length
      for j = 1:length
        if A(i,j) ~= 0
          first_nonzero(i) = j;
          if (i - j + 1) > half_bw
            half_bw = i - j + 1;
          end;
        break
        end;
      end;
    end;
```

```
    for i = 1:length
        for k = first_nonzero(i):min(i, half_bw)
            sum = 0;
            if A(i,k) ~= A(k,i)
                error("The matrix needs to be symmetric")
            end;
            for j = 1:k
                sum = sum + (L(i,j) * L(k,j));
            end;
            if (i == k)
                if ((A(i,i) - sum)) < 0
                    error("The matrix needs to be positive definite")
                end;
                L(i,k) = sqrt(abs(A(i,i) - sum));
            else
                L(i,k) = (A(i,k) - sum) / L(k,k);
            end;
        end;
    end;
end;
```

3. sparse_fwdelim_bcksub.m

```
function[y, x] = sparse_fwdelim_bcksub(L, b, first_nonzero)
%   Function that performs bth forward elimination and back substitution
%   i.e. in forward elimination it solves for y in Ly = b and in back
%   substitution it solves for x in L^Tx = y

%   Author: Thomas Christinck, 2018.
    size_L = size(L);
    length = size_L(1);

    y(1:length)=0;
    for i = 1:length
        sum = 0;
        for j = first_nonzero(i):i
            sum = sum + (L(i, j) * y(j));
        end;
        y(i) = (b(i) - sum) / L(i, i);
    end;

    x(1:length, 1)=0;
    for i = length:-1:1
        sum = 0;
        for j = i:length
```

```
        sum = sum + (L(j, i) * x(j));
    end;
    x(i) = (y(i) - sum) / L(i,i);
end;
```

APPENDIX 3

1. get_potential.m

```
function [potential] = get_potential(mesh, x, y, h)
%  Gets the potential in mesh with node spacing h at (x,y) cable_height = 0.1;

   cable_width = 0.1;
   node_height = round(cable_height / h + 1);
   node_width = round(cable_width / h + 1);
   x_node = round(node_width - x / h);
   y_node = round(node_height - y / h);
   potential = mesh(y_node, x_node);
```

2. sor.m

```
function [M] = SOR(mesh, h, w)
%  Performs successive order relaxation on mesh with parameter w, and
%  node spacing h.
%  Author: Thomas Christinck, 2018.

   core_height = 0.02;
   core_width = 0.04;
   core_potential = 110.0;
   cable_height = 0.1;
   cable_width = 0.1;

   node_height = round((cable_height / h) + 1);
   node_width = round((cable_width / h) + 1);

   for i = 2:node_height - 1
         for j = 2:node_width - 1
            if (i-1) > round(core_height / h) || (j-1) > round(core_width / h)
               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;
   M = mesh;
```

3. compute_residual.m

```
function [max_res] = compute_residual(mesh, h)
%   Computes the maximum residual for mesh with node spacing h
%   Author: Thomas Christinck, 2018.

   core_height = 0.02;
   core_width = 0.04;
   cable_height = 0.1;
   cable_width = 0.1;
   node_height = round((cable_height / h) + 1);
   node_width = round((cable_width / h) + 1);
   max_res = 0;

   for i = 2:node_height - 1
          for j = 2:node_width - 1
             if (i-1) > (core_height / h) || (j-1) > (core_width / h)
                % Calculate the residual of the free points
                 res = mesh(i, j-1) + mesh(i, j+1) + mesh(i-1, j) + mesh(i+1, j) - 4 * mesh(i, j);
                 res = abs(res);
                if res > max_res
                    max_res = res;
                end;
            end;
          end;
    end;
```

4. iterate.m

```
function [mesh, iterations] = iterate(mesh, h, w, x_grid, y_grid, mode)
%   Performs residual minimization for mesh with node spacing h using sor
%   or jacobi method specified as 's' or 'j' in the mode parameter. If sor,
%         omega is specified by w. For a non-uniform grid using the SOR method, specify
%         mode as 'snu' and provide x_grid, y_grid
%   Author: Thomas Christinck, 2018.

   threshold = 0.00001;
   iterations = 1;
   % Check if we are using SOR or Jacobi method
   if mode == 's'
          mesh = sor(mesh, h, w);
          while compute_residual(mesh, h) >= threshold
                    mesh = sor(mesh, h, w);
                    iterations = iterations + 1;
          end;
   elseif mode == 'j'
```

```
        mesh = jacobian(mesh,h) ;
        while compute_residual(mesh,h) >= threshold
        mesh = jacobian(mesh,h);
        iterations = iterations + 1;
         end;
   elseif mode == 'snu'
        mesh = sor_nonuniform(mesh, x_grid, y_grid, w);
        while compute_res_nonuniform(mesh, x_grid, y_grid) >= threshold  …
                && iterations < 20000
                mesh = sor_nonuniform(mesh, x_grid, y_grid, w);
                iterations = iterations + 1;
        end;
    else
        error("Specify mode as 's' or 'j'!")
    end;
```

5. jacobian.m

```
function [M] = jacobian(mesh, h)
%   Performs Jacobian method on mesh with node spacing h.
%   Author: Thomas Christinck, 2018.

   core_height = 0.02;
   core_width = 0.04;
   cable_height = 0.1;
   cable_width = 0.1;

   node_height = round((cable_height / h) + 1);
   node_width = round((cable_width / h) + 1);

   for i = 2:node_height - 1
        for j = 2:node_width - 1
          if (i-1) > round(core_height / h) || (j-1) > round(core_width / h)
             mesh(i, j) = (1 / 4) * (mesh(i, j-1) + mesh(i, j+1) + mesh(i-1,j) + mesh(i+1,j));
          end;
        end;
   end;
   M = mesh;
```

6. generate_mesh_nonuniform.m

```
function [M] = generate_mesh_nonuniform(x_grid, y_grid)
%   Generates mesh with initial boundary conditions (considering 1/4 of the problem,
%   due to symmetry)
%   Author: Thomas Christinck, 2018.
```

```
core_height = 0.02;
core_width = 0.04;
core_potential = 110.0;
cable_height = 0.1;
cable_width = 0.1;

% First, we'll set up the mesh
for i = 1:length(y_grid)
   for j = 1:length(x_grid)
      if x_grid(j) <= (core_width) && y_grid(i) <= (core_height)
         mesh(i, j) = core_potential;
      else
         mesh(i, j) = 0;
      end;
   end;
end;
% Consider the Neumann boundary conditions
delta_x = core_potential / (cable_width - core_width);
delta_y = core_potential / (cable_height - core_height);
for x = 1:length(x_grid)
   if x_grid(x) > core_width
      mesh(1, x) = core_potential - delta_x * (x_grid(x) - core_width);
   end;
end;
for y = 1:length(y_grid)
   if y_grid(y) > core_height
      mesh(y, 1) = core_potential - delta_y * (y_grid(y) - core_height);
   end;
end;

M = mesh;
```

7. <u>sor_nonuniform.m</u>

```
function [M] = sor_nonuniform(mesh, x_grid, y_grid, w)
%   Performs successive order relaxation on mesh with parameter w, and
%   node spacing specified by y_grid and x_grid.
%   Author: Thomas Christinck, 2018.

   core_height = 0.02;
   core_width = 0.04;
   M = mesh;
   for i = 2:(length(y_grid) - 1)
      for j = 2:(length(x_grid) - 1)
         if y_grid(i) > core_height || x_grid(j) > core_width
```

```
        surround = (M(i - 1, j) + M(i, j - 1) ...
                    + mesh(i + 1, j) + mesh(i, j + 1));
        old_value = mesh(i, j);
        new_value = (1 - w) * old_value + w / 4 * surround;
        M(i,j) = new_value;
       end;
     end;
   end;
```

8. compute_res_nonuniform.m

```
function [max_res] = compute_res_nonuniform(mesh, x_grid, y_grid)
%   Computes the maximum residual for mesh with node spacing h
%   Author: Thomas Christinck, 2018.

   core_height = 0.02;
   core_width = 0.04;
   max_res = 0;

   for i = 2:(length(y_grid)- 1)
      for j = 2:(length(x_grid) - 1)
         if x_grid(j) > core_width || y_grid(i) > core_height
            % Calculate the residual of the free points

            res = abs(mesh(i - 1, j) + mesh(i, j - 1) + mesh(i + 1, j) ...
                  + mesh(i, j + 1) - (4 * mesh(i, j)));
            res = abs(res);


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

9. get_potential_nonuniform.m

```
function [potential] = get_potential_nonuniform(mesh, x, y, x_grid, y_grid)
%   Gets the potential in mesh described by x_grid, y_grid at (x,y)
   x_node = find(x_grid == x);
   y_node = find(y_grid == y);
   potential = mesh(x_node, y_node);
```

APPENDIX 4

1. diag_matrix.m (diagonal matrix with entries of vector A)

```
function[D] = diag_mat(A):
   size_A = size(A);
   row_len = size_A(1);
   col_len = size_A(2);

   D(1:row_len,1:col_len)=0;

   for i = 1:row_len
      for j = 1:col_len
         if i == j
            D(i,j) = 1/A(i,j);
         else
            D(i,j) = 0;
         end;
      end;
   end;
```

2. mat_transpose.m

```
function[A_T] = mat_transpose(A)
  % Function that transposes a matrix A and returns the matrix's transpose
  % A_T
  %   Author: Thomas Christinck, 2018.

  size_A = size(A);
  rows_A = size_A(1);
  cols_A = size_A(2);

   A_T(1:cols_A,1:rows_A)=0;
  for i = 1:cols_A
    for j = 1:rows_A
      A_T(i,j) = A(j,i);
    end;
  end;
```

3. mat_add_or_subtract.m

```
function[C] = matrix_add_or_subtract(A,B,operation)
  %  Function that performs vector/matrix addition/subtraction
  %  Matrices A and B must be the same size. Operation should be specified
  %  as 's' for subtract and 'a' for add. Returns the result C
  %  Author: Thomas Christinck, 2018.

  size_A = size(A);
  size_B = size(B);
  rows_A = size_A(1);
  cols_A = size_A(2);
  rows_B = size_B(1);
  cols_B = size_B(2);
  if rows_A == rows_B & cols_A == cols_B
  C(1:rows_A,1:cols_A)=0;
    if operation == 'a'
      for i = 1:rows_A
        for j = 1:cols_A
          C(i,j) = A(i,j)+B(i,j);
        end;
      end;
    elseif operation == 's'
      for i = 1:rows_A
        for j = 1:cols_A
          C(i,j) = A(i,j)-B(i,j);
        end;
      end;
   end;
  else
    error("The two matrices to be added have to be equal in size")
  end;
```

4. matrix_multiply.m

```
function[C] = matrix_multiply(A, B)
  %   Function that multiplies two matricies A and B
  %   The number of columns in matrix A must equal the number of rows in matrix B. The
  %   product matrix, C, is returned.

  size_A = size(A);
  size_B = size(B);
 rows_A = size_A(1);
 cols_A = size_A(2);
 rows_B = size_B(1);
 cols_B = size_B(2);

 if cols_A == rows_B
    C(1:rows_A,1:cols_B)=0;
    for i = 1:rows_A
      for j = 1:cols_B
        for k = 1:cols_A
           C(i,j) = C(i,j) + (A(i, k) * B(k,j));
        end;
     end;
  end;
 else
    error("Cannot multiply the two matrices. Cols_A must equal Rows_B")
 end;
```