ECSE 543 Assignment 2

Garrett Kinman – 260763260

Question 1: Figure 1 shows two first-order triangular finite elements used to solve the Laplace equation for electrostatic potential. Find a local S-matrix for each triangle and a global S-matrix for the mesh, which consists of just these two triangles. The local (disjoint) and global (conjoint) node-numberings are shown in Figure 1(a) and (b), respectively. Also, Figure 1(a) shows the (x, y)-coordinates of the element vertices in meters.

Calculations are shown in the series of equations below.

$$A = \frac{1}{2} \times 0.02 \times 0.02 = 0.0002$$

$$\nabla\alpha_1 = \frac{1}{2A}\left((y_2 - y_3), (x_3 - x_2)\right) = (0, 50)$$

$$\nabla\alpha_2 = \frac{1}{2A}\left((y_3 - y_1), (x_1 - x_3)\right) = (-50, -50)$$

$$\nabla\alpha_3 = \frac{1}{2A}\left((y_1 - y_2), (x_2 - x_1)\right) = (50, 0)$$

$$\nabla\alpha_4 = \frac{1}{2A}\left((y_5 - y_6), (x_6 - x_5)\right) = (50, 50)$$

$$\nabla\alpha_5 = \frac{1}{2A}\left((y_6 - y_4), (x_4 - x_6)\right) = (-50, 0)$$

$$\nabla\alpha_6 = \frac{1}{2A}\left((y_4 - y_5), (x_5 - x_4)\right) = (0, -50)$$

$$S_{ij}^{(e)} = \nabla\alpha_i \times \nabla\alpha_j \times A$$

$$S^{(1)} = \begin{bmatrix} 0.5 & -0.5 & 0 \\ -0.5 & 1 & -0.5 \\ 0 & -0.5 & 0.5 \end{bmatrix} S^{(2)} = \begin{bmatrix} 1 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0 \\ -0.5 & 0 & 0.5 \end{bmatrix}$$

$$S_{dis} = \begin{bmatrix} S^{(1)} & 0 \\ 0 & S^{(2)} \end{bmatrix} = \begin{bmatrix} 0.5 & -0.5 & 0 & 0 & 0 & 0 \\ -0.5 & 1 & -0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -0.5 & -0.5 \\ 0 & 0 & 0 & -0.5 & 0.5 & 0 \\ 0 & 0 & 0 & -0.5 & 0 & 0.5 \end{bmatrix}$$

$$U_{dis} = CU_{dis} \Rightarrow \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

$$S_{con} = C^T S_{dis} C = \begin{bmatrix} 1 & -0.5 & 0 & -0.5 \\ -0.5 & 1 & -0.5 & 0 \\ 0 & -0.5 & 1 & -0.5 \\ -0.5 & 0 & -0.5 & 1 \end{bmatrix}$$

Question 2: Figure 2 shows the cross-section of an electrostatic problem with translational symmetry: a rectangular coaxial cable. The inner conductor is held at 15 volts and the outer conductor is grounded. (This is similar to the system considered in Question 3, Assignment 1.)

a) Use the two-element mesh shown in Figure 1(b) as a “building block” to construct a finite element mesh for one-quarter of the cross-section of the coaxial cable. Specify the mesh, including boundary conditions, in an input file following the format for the **SIMPLE2D** program as explained in the course notes. (Hint: Your mesh should consist of 46 elements.)

The upper-right quadrant of the coaxial cable cross-section was chosen and split up into two-element meshes, as shown in Figure 1 below. Because the potentials of the inner conductor are known (15 V), this region was ignored, providing a total of 46 elements and 34 nodes.

Based off of this, the input files for **SIMPLE2D** were manually created, and they are included as *file1.dat*, *file2.dat*, and *file3.dat* in the appendix.

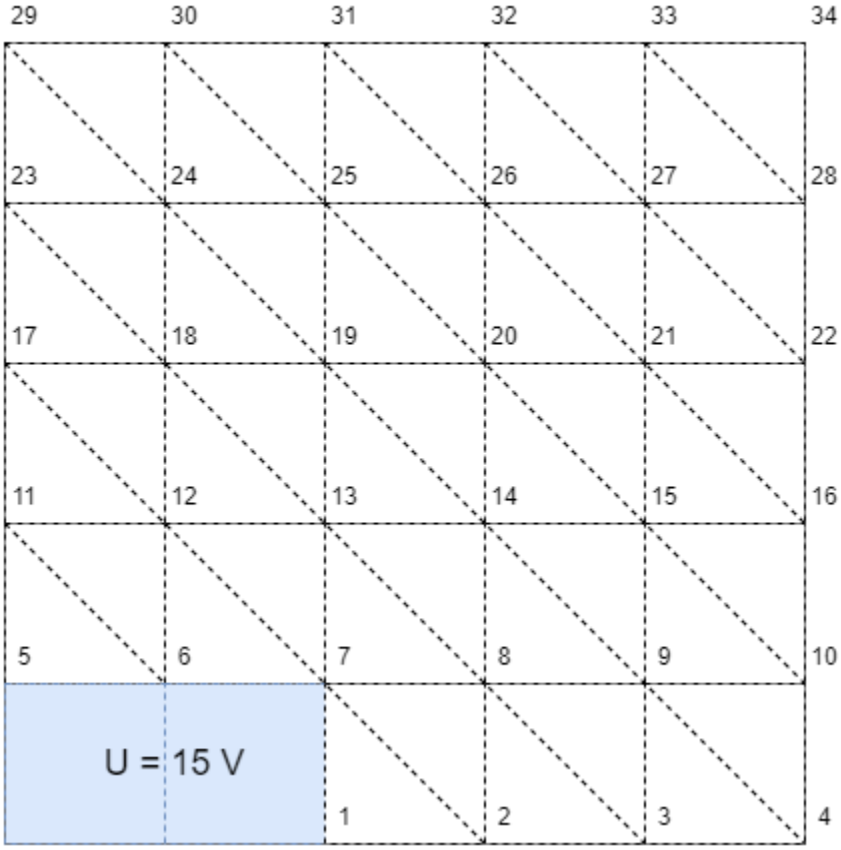

Figure 1: Upper-right quadrant of coaxial cable composed of two-element meshes

b) Use the **SIMPLE2D** program with the mesh from part (a) to compute the electrostatic potential solution. Determine the potential at (x,y) = (0.06, 0.04) from the data in the output file of the program.

The complete input files can be found in the appendix. Below are the results of the **SIMPLE2D** program. Note that, in declaring the inputs, the center of the coaxial cable was chosen as the zero-point for the quadrant. Within this coordinate system, node 14 corresponds to (0.06, 0.04), however, within the global coordinate system, it does not, as that point does not exist within the selected quadrant. By symmetry, however, node 19 in the results below corresponds with point (0.06, 0.04). Thus, we determine the potential at (0.06, 0.04) is 5.5263 V.

```
potential =

    1.0000    0.0400         0   15.0000
    2.0000    0.0600         0    9.0919
    3.0000    0.0800         0    4.2525
    4.0000    0.1000         0         0
```

```
 5.0000         0    0.0200   15.0000
 6.0000    0.0200    0.0200   15.0000
 7.0000    0.0400    0.0200   15.0000
 8.0000    0.0600    0.0200    8.5575
 9.0000    0.0800    0.0200    3.9590
10.0000    0.1000    0.0200         0
11.0000         0    0.0400   10.5490
12.0000    0.0200    0.0400   10.2912
13.0000    0.0400    0.0400    9.2492
14.0000    0.0600    0.0400    6.1791
15.0000    0.0800    0.0400    3.0262
16.0000    0.1000    0.0400         0
17.0000         0    0.0600    6.6135
18.0000    0.0200    0.0600    6.3668
19.0000    0.0400    0.0600    5.5263
20.0000    0.0600    0.0600    3.8834
21.0000    0.0800    0.0600    1.9667
22.0000    0.1000    0.0600         0
23.0000         0    0.0800    3.1714
24.0000    0.0200    0.0800    3.0360
25.0000    0.0400    0.0800    2.6060
26.0000    0.0600    0.0800    1.8616
27.0000    0.0800    0.0800    0.9571
28.0000    0.1000    0.0800         0
29.0000         0    0.1000         0
30.0000    0.0200    0.1000         0
31.0000    0.0400    0.1000         0
32.0000    0.0600    0.1000         0
33.0000    0.0800    0.1000         0
34.0000    0.1000    0.1000         0
```

c) Compute the capacitance per unit length of the system using the solution obtained from **SIMPLE2D**.

By using the equation for the energy contained in a capacitor, as shown in the equations below, an equation can be derived for capacitance in terms of total energy, E_{total}. The total energy can then be calculated as the sum of the energies of all the constituent meshes, whose energies are given by the equations below as well.

$$E_{total} = \frac{1}{2}CV^2 => C = 2\frac{E_{total}}{V^2}$$

$$E_{mesh} = \frac{1}{2}\epsilon_0 U_{con}^T S_{con} U_{con}$$

Using these equations, and where S_{con} was determined in Question 1, and U_{con} was obtained in part (b), a total capacitance can be tabulated. However, this is for just one quadrant, so multiplying by 4 is necessary. Thus, the capacitance per unit length was calculated to be 5.2136 pF, as shown below.

```
C =

   5.2136e-11
```

Question 3: Write a program implementing the conjugate gradient method (un-preconditioned). Solve the matrix equation corresponding to a finite difference node-spacing, h = 0.02m in x and y directions for the same one-quarter cross-section of the system shown in Figure 2 that you considered in Question 2 above. Use a starting solution of zero. (Hint: The program you wrote for Question 3 of Assignment 1 may be useful for generating the matrix equation.)

In order to employ the conjugate gradient method, the matrix A and vector B are needed first. From Assignment 1, we have a method for constructing a mesh of 36 nodes to represent the upper-right quadrant. However, the nodes in or on the conductors are at fixed, known voltages. As seen in Figure 2 below, the free nodes—i.e., the nodes for which their voltages are yet to be determined—are 2, 3, 8, 9, 11–15, 17–21, and 23–27. Thus, a 19×19 A matrix and a 19×1 b vector are required to solve the system.

Using the five-point difference method, an equation can be expressed for each free node:

$$-4\varphi_{i,j} + \varphi_{i+1,j} + \varphi_{i-1,j} + \varphi_{i,j+1} + \varphi_{i,j-1} = 0$$

Additionally, because of symmetry, boundary nodes (2, 3, 11, 17, 23) have slightly modified equations. For bottom boundary (nodes 2 and 3), the equation becomes:

$$-4\varphi_{i,j} + 2\varphi_{i+1,j} + \varphi_{i,j+1} + \varphi_{i,j-1} = 0$$

And for left-boundary (nodes 11, 17, and 23), the equation becomes:

$$-4\varphi_{i,j} + \varphi_{i+1,j} + \varphi_{i-1,j} + 2\varphi_{i,j+1} = 0$$

There is one more complication, however. In the five-point difference equation for nodes neighboring the non-free nodes invokes the non-free nodes. This is a problem, as the non-free nodes cannot be represented within our A matrix. If the equation is rearranged for these values, however, the constant, known value for these (i.e., the known voltage) can be subtracted from both sides. This corresponds to setting the value of the appropriate row in the vector b to -15, for instance.

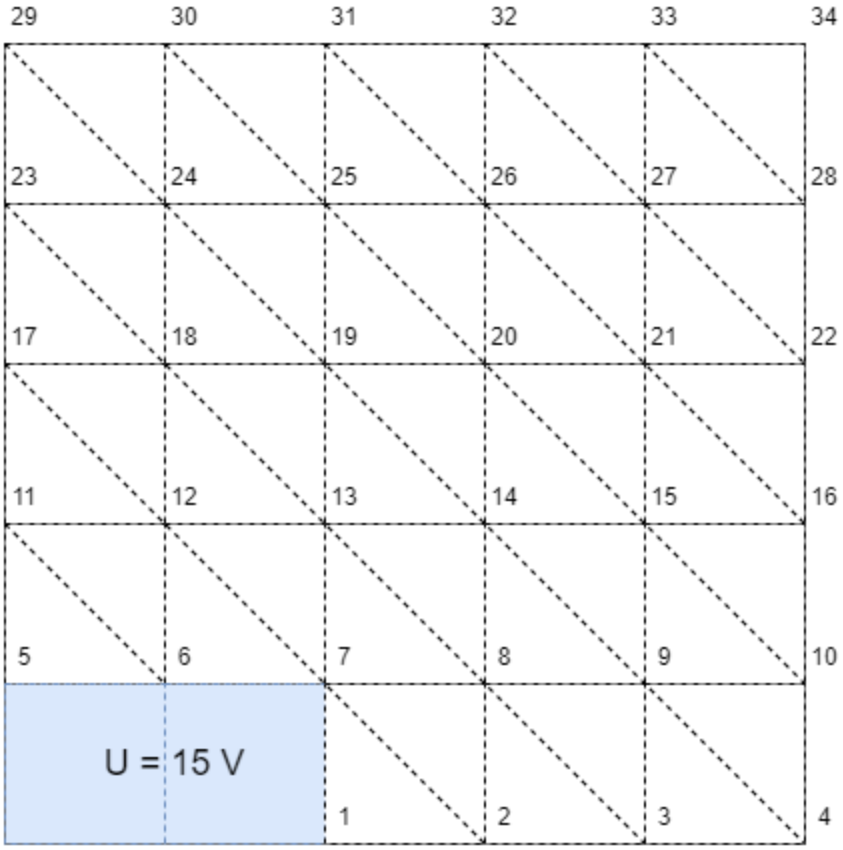

Figure 2: Node numbering of the mesh

In *conjugate-gradient.jl* in the appendix is the function for generating the A matrix and b vector from the mesh. The results look as below in Figure 3. Additionally in *conjugate-gradient.jl* in the appendix is the function for applying the conjugate gradient method to a given A matrix and b vector. In performing the conjugate gradient method to the generated A and b, it returns reasonable-looking results, shown below in Figure 4.

```
julia> mesh_A
19×19 Matrix{Float64}:
 -4.0   1.0   2.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  …  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
  1.0  -4.0  -0.0   2.0  -0.0  -0.0  -0.0  -0.0  -0.0     -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
  1.0  -0.0  -4.0   1.0  -0.0  -0.0  -0.0   1.0  -0.0     -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
 -0.0   1.0   1.0  -4.0  -0.0  -0.0  -0.0  -0.0   1.0     -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
 -0.0  -0.0  -0.0  -0.0  -4.0   2.0  -0.0  -0.0  -0.0     -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
 -0.0  -0.0  -0.0  -0.0   1.0  -4.0   1.0  -0.0  -0.0  …  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
 -0.0  -0.0  -0.0  -0.0  -0.0   1.0  -4.0   1.0  -0.0      1.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
 -0.0  -0.0   1.0  -0.0  -0.0  -0.0   1.0  -4.0   1.0     -0.0   1.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0
 -0.0  -0.0  -0.0   1.0  -0.0  -0.0  -0.0   1.0  -4.0     -0.0  -0.0   1.0  -0.0  -0.0  -0.0  -0.0  -0.0
  ⋮                        ⋮                       ⋱                       ⋮
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0   1.0  -0.0  -0.0     -4.0   1.0  -0.0  -0.0  -0.0   1.0  -0.0  -0.0
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0   1.0  -0.0      1.0  -4.0   1.0  -0.0  -0.0  -0.0   1.0  -0.0
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0   1.0     -0.0   1.0  -4.0  -0.0  -0.0  -0.0  -0.0   1.0
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0     -0.0  -0.0  -0.0  -4.0   2.0  -0.0  -0.0  -0.0
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  …  -0.0  -0.0  -0.0   1.0  -4.0   1.0  -0.0  -0.0
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0      1.0  -0.0  -0.0  -0.0   1.0  -4.0   1.0  -0.0
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0     -0.0   1.0  -0.0  -0.0  -0.0   1.0  -4.0   1.0
 -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0  -0.0     -0.0  -0.0   1.0  -0.0  -0.0  -0.0   1.0  -4.0
```

```
julia> mesh_b
19-element Vector{Float64}:
 -15.0
  -0.0
 -15.0
  -0.0
 -15.0
 -15.0
 -15.0
   0.0
  -0.0
   ⋮
   0.0
   0.0
  -0.0
  -0.0
  -0.0
  -0.0
  -0.0
  -0.0
```

Figure 3: Generated A matrix and b vector

```
19×1 Matrix{Float64}:
  9.091885102934521
  4.25253287558052
  8.557508161060534
  3.9590714862479492
 10.548963141642117
 10.291196848473763
  9.249144548281341
  6.1789885810838845
  3.026172046136854
  ⋮
  5.526274961557273
  3.8833778702531827
```

```
1.9666081295798907
3.1713245121813225
3.0360001463297577
2.605966333074788
1.8615846544143282
0.9570846222194286
```

Figure 4: Output of conjugate gradient method on the generated A and b

a) Test your matrix using your Choleski decomposition program that you wrote for Question 1 of Assignment 1 to ensure that it is positive definite. If it is not, suggest how you could modify the matrix equation in order to use the conjugate gradient method for this problem.

In performing Choleski decomposition on the same A matrix and b vector as before, the Choleski decomposition threw an error. This suggests A was not positive-definite. One way to modify the equation is to multiply both sides by the transpose of A as such:

$$Ax = b$$
$$\left(A^T A\right) x = \left(A^T b\right)$$

As can be seen in the equation above, the solution to the modified equation will still be x, unchanged. All that has to be done differently is to multiply the inputs to the Choleski decomposition by the transpose of A first.

b) Once you have modified the problem, if necessary, so that the matrix is positive definite, solve the matrix equation first using the Choleski decomposition program from Assignment 1, and then the conjugate gradient program written for this assignment.

Below in Figure 5 are the side-by-side results of the conjugate gradient method and Choleski decomposition.

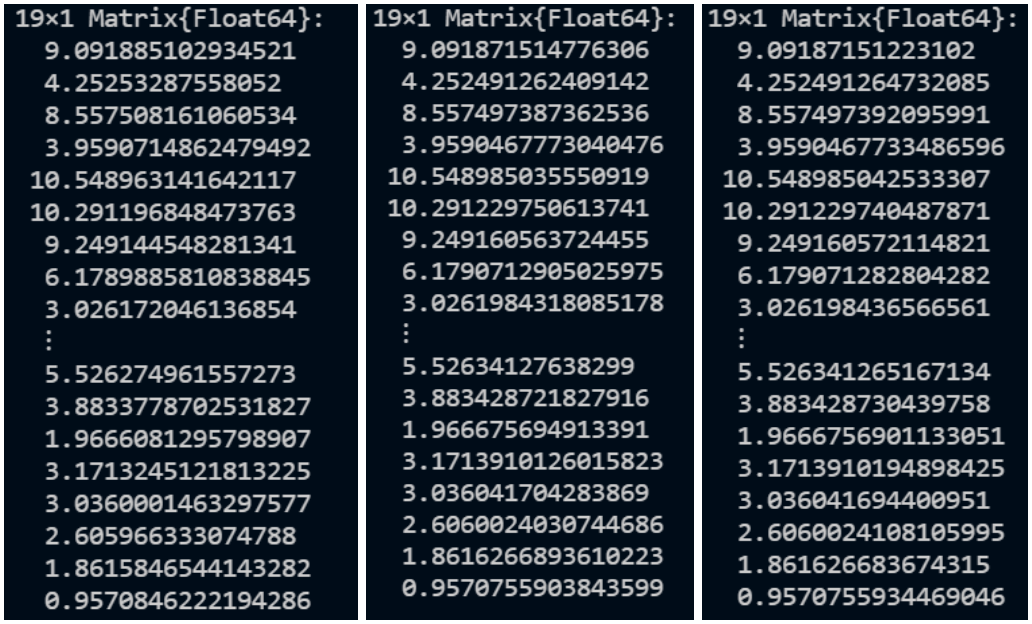

Figure 5: Results of conjugate gradient (left), conjugate gradient on modified inputs (center), and Choleski decomposition on modified input (right)

Note that, when using modified inputs (as described above), the conjugate gradient method provides answers the same to more decimal points as Choleski decomposition, as compared to conjugate gradient descent without modified inputs. This suggests at least some of the difference between the results is because of the slight additional floating-point error introduced by modifying the inputs. Nonetheless, they are still the same to about three decimal places.

c) Plot a graph of the infinity norm and the 2-norm of the residual vector versus the number of iterations for the conjugate program.

Below are the raw results as well as a plot of the infinity norm and 2-norm of the residual vector.

```
iteration,infinity_norm,two_norm
0,45.0,96.04686356149273
1,44.4372574385511,75.69981308722257
2,22.536036887462586,46.7838079972879
3,14.108924721980028,32.277783133313584
4,12.294209334571644,25.521905754958844
5,9.209465380683746,21.720333656180525
6,8.812841734612107,16.398667638416732
7,11.368551453748506,15.01977620495332
8,7.98726796035853,17.971959931926563
9,9.160380229877454,15.456302968522579
10,6.828156554955598,12.723190674362003
11,3.8933048145954814,10.919353989669144
12,4.441504185387444,9.513732172518129
```

```
13,2.0752977605230853,4.602562527817184
14,1.2522339460334138,2.7130124882485935
15,1.6065786480168356,3.102560046961435
16,1.0777622612499647,2.5253375244892275
17,0.337256599593843,0.7709002752078636
18,0.008941369378241859,0.020966603625365992
19,7.742933831877963e-7,1.8378850170682827e-6
```

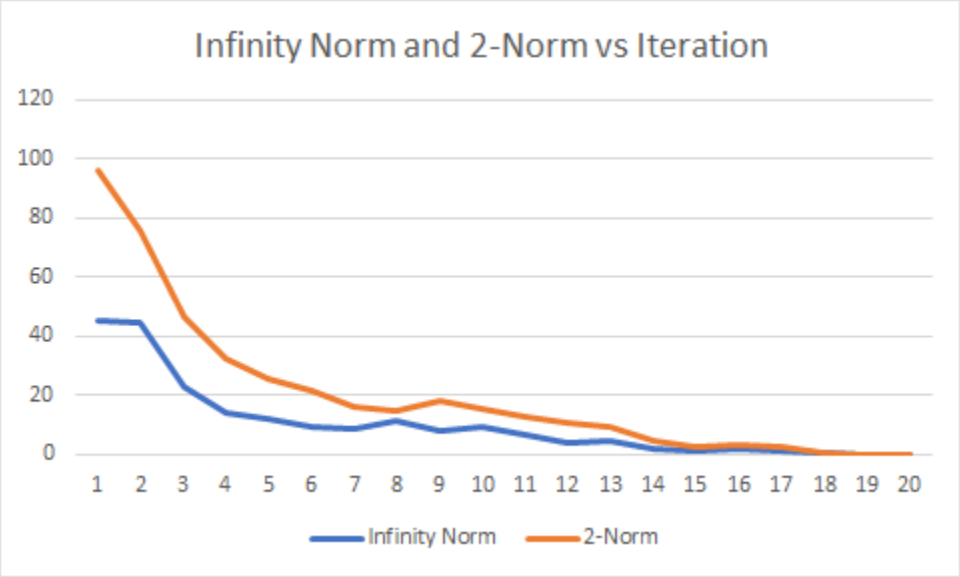

Figure 6: Plot of the infinity norm and 2-norm vs iteration count

d) What is the potential at (x,y) = (0.06, 0.04), using the Choleski decomposition and the conjugate gradient programs, and how do they compare with the value you computed in Question 2(b) above. How do they compare with the value at the same (x,y) location and for the same node spacing that you computed in Assignment 1 using SOR?

Method	**Potential at (0.06, 0.04) (V)**
Choleski Decomposition	5.526341265167134
Conjugate Gradient	5.526274961557273
SIMPLE2D_M	5.5263
SOR (ω = 1.3)	5.526340786

Note that SOR was chosen with ω = 1.3, as it had required the fewest iterations to converge at 13, compared to 19 iterations for the conjugate gradient method. Also note that all four methods provide the same result when rounded to four decimal places.

e) Suggest how you could compute the capacitance per unit length of the system from the finite difference solution.

The capacitance can be found using the same method as in Question 2, part (c). Because this yields the potentials at each node, this can be used to calculate the energy of each finite-element mesh, which can be used to find the total energy of the cross-section, which can be used to find the capacitance.

Appendix

file1.dat

```
1 0.04 0.0
2 0.06 0.0
3 0.08 0.0
4 0.1 0.0
5 0.0 0.02
6 0.02 0.02
7 0.04 0.02
8 0.06 0.02
9 0.08 0.02
10 0.1 0.02
11 0.0 0.04
12 0.02 0.04
13 0.04 0.04
14 0.06 0.04
15 0.08 0.04
16 0.1 0.04
17 0.0 0.06
18 0.02 0.06
19 0.04 0.06
20 0.06 0.06
21 0.08 0.06
22 0.1 0.06
23 0.0 0.08
24 0.02 0.08
25 0.04 0.08
26 0.06 0.08
27 0.08 0.08
28 0.1 0.08
```

```
29 0.0 0.1
30 0.02 0.1
31 0.04 0.1
32 0.06 0.1
33 0.08 0.1
34 0.1 0.1
```

file2.dat

```
1 2 7 0.000
2 8 7 0.000
2 3 8 0.000
3 9 8 0.000
3 4 9 0.000
4 10 9 0.000
5 6 11 0.000
6 12 11 0.000
6 7 12 0.000
7 13 12 0.000
7 8 13 0.000
8 14 13 0.000
8 9 14 0.000
9 15 14 0.000
9 10 15 0.000
10 16 15 0.000
11 12 17 0.000
12 18 17 0.000
12 13 18 0.000
13 19 18 0.000
13 14 19 0.000
14 20 19 0.000
14 15 20 0.000
15 21 20 0.000
15 16 21 0.000
16 22 21 0.000
17 18 23 0.000
18 24 23 0.000
18 19 24 0.000
19 25 24 0.000
19 20 25 0.000
20 26 25 0.000
20 21 26 0.000
21 27 26 0.000
```

```
21 22 27 0.000
22 28 27 0.000
23 24 29 0.000
24 30 29 0.000
24 25 30 0.000
25 31 30 0.000
25 26 31 0.000
26 32 31 0.000
26 27 32 0.000
27 33 32 0.000
27 28 33 0.000
28 34 33 0.000
```

file3.dat

```
1 15.0
5 15.0
6 15.0
7 15.0
29 0.000
30 0.000
31 0.000
32 0.000
33 0.000
34 0.000
28 0.000
22 0.000
16 0.000
10 0.000
4 0.000
```

capacitance.m

```
function Capacitance = capacitance(filename1, filename2, filename3)
    clc;

    % zero initialize
    Etot = 0;
    U = zeros(1, 4);

    % potential between inner and outer
    V = 15 - 0;
```

```
    % use potentials from SIMPLE2D_M and S from Question 1
    potentials = SIMPLE2D_M(filename1, filename2, filename3);
    S = [1, -0.5, 0, -.5; -0.5, 1, -0.5, 0; 0, -0.5, 1, -0.5; -0.5, 0,
-0.5, 1];

    % e0 = epsilon naught
    e0 = 8.854e-12;

    for i = 1:length(potentials)
        % use all nodes not on the outer edge
        if potentials(i, 2) < 0.1 && potentials(i, 3) < 0.1
            U(1) = potentials(i, 4);
            U(2) = potentials(i + 1, 4);
            U(3) = potentials(i + 7, 4);
            U(4) = potentials(i + 6, 4);

            % add the current Emesh to Etot
            Etot = Etot + (0.5 * e0 * U * S * transpose(U));
        end
    end
    Capacitance = 4 * 2 * Etot / V^2;
    return
end
```

choleski.jl

```
using Test

"""
Uses Choleski decomposition to solve `Ax = b`, where A is real, symmetric,
and positive-definite. Returns the vector x.
"""
function choleski(A::AbstractMatrix{<:Real}, b::AbstractVector{<:Real},
halfband=nothing)
    n, m = size(A)

    # simple error checking
    if n != m
        throw(DimensionMismatch("Matrix A must be square"))
    elseif length(b) != n
        throw(DimensionMismatch("Matrix A must be n×n, and vector b must be
n×1"))
    end
```

```
# initialize
L = zeros(n,n)
y = zeros(n,1)
x = zeros(n,1)

#decompose L
for j ∈ 1:n
    sum  = 0
    for q = 1:j-1
        @inbounds sum  = sum  + L[j,q]^2
    end
    @inbounds L[j,j] = sqrt(A[j,j] - sum )

    sumᵢ = 0

    i_range = (j+1):n
    if !isnothing(halfband)
        if (j + halfband + 1) < n
            i_range = (j+1):(j+halfband)
        end
    end
    for i ∈ i_range

        for k ∈ 1:j-1
            if k == 1
                @inbounds sumᵢ = L[i,k]*L[j,k]
            else
                @inbounds sumᵢ = sumᵢ + L[i,k]*L[j,k]
            end
        end
        @inbounds L[i,j] = (A[i,j] - sumᵢ) / L[j,j]
    end
end

#get y
for i ∈ 1:n
    sumᵧ = 0
    for j ∈ 1:i-1
        @inbounds sumᵧ = sumᵧ + L[i,j]*y[j]
    end
    @inbounds y[i] = (b[i] - sumᵧ) / L[i,i]
end
```

```
    #calculate x
    for i ∈ n:-1:1
        sumₓ = 0
        for j ∈ (i+1):n
            @inbounds sumₓ = sumₓ + L[j,i]*x[j]
        end
        @inbounds x[i] = (y[i] - sumₓ) / L[i,i]
    end

    return x
end

@testset "real, symmetric, and positive-definite" begin
    # test a simple 2×2 case
    test_A = [1 0; 0 1]
    test_x = [1; 1]
    test_b = test_A * test_x
    @test choleski(test_A, test_b) ≈ test_x # approx is to account for
floating point errors

    # test 10 random 2×2 cases
    for i ∈ 1:10
        test_x = rand(2)
        test_b = test_A * test_x
        @test choleski(test_A, test_b) ≈ test_x
    end

    # test a simple 3×3 case
    test_A = [2 -1 0; -1 2 -1; 0 -1 2]
    test_x = [1; 1; 1]
    test_b = [1; 0; 1]
    @test choleski(test_A, test_b) ≈ test_x

    # test 10 random 3×3 cases
    for i ∈ 1:10
        test_x = rand(3)
        test_b = test_A * test_x
        @test choleski(test_A, test_b) ≈ test_x
    end
end
```

mesh.jl

```
mutable struct PotentialMesh
    mesh::AbstractMatrix{AbstractFloat}
    h::AbstractFloat
    inner_size::Tuple{Integer, Integer}
    outer_height::AbstractFloat
    outer_width::AbstractFloat
    outer_potential::AbstractFloat
    inner_height::AbstractFloat
    inner_width::AbstractFloat
    inner_potential::AbstractFloat

    """
    Constructor for PotentialMesh with optional kwargs for `h`, the mesh
spacing, and `residual_limit`,
    the cutoff point for SOR.
    """
    function PotentialMesh(;h::AbstractFloat=0.02)
        # constant parameters for defining the operating geometry and
potentials
        params = (outer_height=0.1, outer_width=0.1, outer_potential=0.0,
inner_height=0.02, inner_width=0.04, inner_potential=15.0)

        # construct a matrix where each element is a node at the
intersection of neighboring meshes
        # initialize values to outer_potential
        num_rows, num_cols = Int.(((params.outer_height / h) + 1,
(params.outer_width / h) + 1))
        mesh = fill(params.outer_potential, (num_rows, num_cols))

        # fill in the nodes corresponding to the inner conductor with
inner_potential
        inner_rows, inner_cols = Int.(((params.inner_height / h) + 1,
(params.inner_width / h) + 1))
        mesh[1:inner_rows, 1:inner_cols] .= params.inner_potential

        # construct full struct with the calculated and constant parameters
        return new(mesh, h, (inner_rows, inner_cols), params...)
    end
end
```

conjugate-gradient.jl

```
using Test
using LinearAlgebra
include("Assignment 2/mesh.jl")
include("Assignment 2/choleski.jl")

## matrix generation

"""
Generates the A matrix and b vector for the free nodes of a given mesh.
"""
function generate_matrix(potentials::PotentialMesh, num_nodes::Integer)
    n, m = size(potentials.mesh)

    # initialize A as diagonal matrix with the -4φᵢ  set as all the
diagonals
    # need to fill in all the other 1s and 2s below
    A = one(zeros(num_nodes, num_nodes)) * -4

    # zero-initialize b; need to fill in some of the values below
    b = zeros(num_nodes)

    # k is used for indexing into our constructed A and b
    k = 1

    # manually fill in rest of A and b
    for i ∈ 1:(n-1) # right boundary are all non-free nodes
        for j ∈ 1:(m-1) # top boundary are all non-free nodes
            if j > 2 && potentials.mesh[i, j] == 0 && potentials.mesh[i, j
- 1] == potentials.inner_potential # nodes 2, 8
                if i == 1 # node 2
                    A[k, k + 1] = 1
                    A[k, k + 2] = 2
                    b[k] = -potentials.inner_potential
                elseif i == 2 # node 8
                    A[k, k + 1] = 1
                    A[k, k - 2] = 1
                    A[k, k + 5] = 1
                    b[k] = -potentials.inner_potential
                end
                k += 1
            elseif j == m - 1 # nodes 3, 9, 15, 21, 27
                if i == 1 # node 3
```

```
                    A[k, k - 1] = 1
                    A[k, k + 2] = 2
                    b[k] = -potentials.outer_potential
                elseif i == 2 # node 9
                    A[k, k - 1] = 1
                    A[k, k + 5] = 1
                    A[k, k - 2] = 1
                    b[k] = -potentials.outer_potential
                elseif i == n - 1 # node 27
                    A[k, k - 1] = 1
                    A[k, k - 5] = 1
                    b[k] = -potentials.outer_potential * 2
                else # nodes 15, 21
                    A[k, k - 1] = 1
                    A[k, k + 5] = 1
                    A[k, k - 5] = 1
                    b[k] = -potentials.outer_potential
                end
                k += 1
            elseif j == 1 && i > 2 # nodes 11, 17, 23
                if potentials.mesh[i - 1, j] == potentials.inner_potential
# node 11
                    A[k, k + 1] = 2
                    A[k, k + 5] = 1
                    b[k] = -potentials.inner_potential
                elseif i == n - 1 # node 23
                    A[k, k + 1] = 2
                    A[k, k - 5] = 1
                    b[k] = -potentials.outer_potential
                else # node 17
                    A[k, k + 1] = 2
                    A[k, k + 5] = 1
                    A[k, k - 5] = 1
                    b[k] = 0
                end
                k += 1
            elseif i == 3 && potentials.mesh[i - 1, j] ==
potentials.inner_potential # nodes 12, 13
                A[k, k - 1] = 1
                A[k, k + 1] = 1
                A[k, k + 5] = 1
                b[k] = -potentials.inner_potential
                k += 1
```

```
            elseif i == n - 1 # nodes 24, 25, 26
                A[k, k - 1] = 1
                A[k, k + 1] = 1
                A[k, k - 5] = 1
                b[k] = -potentials.outer_potential
                k += 1
            elseif i > 2 && j > 1 # nodes 14, 18, 19, 20
                A[k, k - 1] = 1
                A[k, k + 1] = 1
                A[k, k - 5] = 1
                A[k, k + 5] = 1
                b[k] = 0
                k += 1
            end
        end
    end
    return A, b
end

## conjugate gradient

"""
Uses Conjugate Gradient Method to solve `Ax = b`, where A is real and
symmetric, but not necessarily positive-definite. Returns the vector x.
"""
function conjugate_gradient(A::AbstractMatrix{<:Real},
b::AbstractVector{<:Real})
    n, m = size(A)

    # simple error checking
    if n != m
        throw(DimensionMismatch("Matrix A must be square"))
    elseif length(b) != n
        throw(DimensionMismatch("Matrix A must be n×n, and vector b must be
n×1"))
    end

    # initialize
    x = zeros(n,1)
    r = b - (A * x)
    p = copy(r)

    inf_norm_ini = 0
```

```
    two_norm_ini = 0

    for i ∈ 1:n
        if abs(r[i, 1]) > inf_norm_ini
            inf_norm_ini = abs(r[i, 1])
        end
        two_norm_ini += r[i, 1]^2
    end
    two_norm_ini = √(two_norm_ini)

    println("iteration,infinity_norm,two_norm")
    println("0,$inf_norm_ini,$two_norm_ini")

    # perform conjugate gradient method
    for i ∈ 1:n
        α = (transpose(p) * r)[1, 1] / (transpose(p) * A * p)[1, 1]
        x = x + (α * p)
        r = b - (A * x)
        β = -(transpose(p) * A * r)[1, 1] / (transpose(p) * A * p)[1, 1]
        p = r + (β * p)

        # finding the norms
        inf_norm = 0
        two_norm = 0

        for j ∈ 1:n
            if abs(r[j, 1]) > inf_norm
                inf_norm = abs(r[j, 1])
            end
            two_norm += r[j, 1]^2
        end

        two_norm = √(two_norm)
        println("$i,$inf_norm,$two_norm")
    end

    return x
end

## test

potentials = PotentialMesh(h=0.02)
mesh_A, mesh_b = generate_matrix(potentials, 19)
```

```
conjugate_gradient(mesh_A, mesh_b)
conjugate_gradient(transpose(mesh_A) * mesh_A, transpose(mesh_A) * mesh_b)
choleski(transpose(mesh_A) * mesh_A, transpose(mesh_A) * mesh_b)

## test

@testset "generated A and b from mesh" begin
    potentials = PotentialMesh(h=0.02)
    test_A, test_b = generate_matrix(potentials, 19)

    @test conjugate_gradient(transpose(test_A) * test_A, transpose(test_A)
* test_b) ≈ choleski(transpose(test_A) * test_A, transpose(test_A) *
test_b)
end
```

References