# ECSE 543 Assignment 1

Garrett Kinman – 260763260

## Question 1

a) Write a program to solve the matrix equation Ax=b by Choleski decomposition. A is a real, symmetric, positive-definite matrix of order n.

One function was created to solve Ax=b with Choleski decomposition with function signature shown below. The rest of the function body can be found in the appendix.

```
"""
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})
```

b) Construct some small matrices (n = 2, 3, 4, or 5) to test the program. Remember that the matrices must be real, symmetric, and positive-definite. Explain how you chose the matrices.

Two example positive-definite matrices were chosen, one 2x2 and one 3x3. These were provided in the Wikipedia page on positive-definite matrices as examples. Because, as mentioned in lectures, successful Choleski decomposition of a matrix is the best proof of its positive-definiteness, only checks for squareness and real-ness were added in the function itself. The two matrices chosen are shown below. Of note is that the first one is simply a 2x2 identity matrix.

```
[1 0
 0 1]
```

```
[2 -1  0
-1  2 -1
 0 -1  2]
```

c) Test the program you wrote in (a) with each small matrix you built in (b) in the following way: invent an x, multiply it by A to get b, then give A and b to your program and check that it returns x correctly.

The Choleski decomposition function was tested with a set of unit tests. For each test A matrix, a simple test case was performed, then a series of 10 randomized x vectors were used. The results and the code are shown below. One thing to note is that using simple equality returned an error whenever the result was floating point, because of floating-point errors in the calculations. Because of this, a built-in Julia operator for approximate equals (an operator built with the express purpose of accounting for floating-point errors) was used.

```
Test Summary:                          | Pass  Total
real, symmetric, and positive-definite |   22     22
```

```
@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
```

d) Write a program that reads from a file a list of network branches (Jk, Rk, Ek) and a reduced incidence matrix, and finds the voltages at the nodes of the network. Use the code from part (a) to solve the matrix problem. Explain how the data is organized and read from the file. Test the program with a few small networks that you can check by hand. Compare the results for your test circuits with the analytical results you obtained by hand. Clearly specify each of the test circuits used with a labeled schematic diagram.

Circuit configurations were stored as standard *.toml* files, which indicates them as using *Tom's Obvious, Minimal Language*, a simple markup language. An example of the structure of the files is shown below.

```
num_nodes = 1
num_branches = 2
A = [ [ -1, 1 ] ]
J = [ 0, 0 ]
R = [ 10, 10 ]
E = [ 10, 0 ]
```

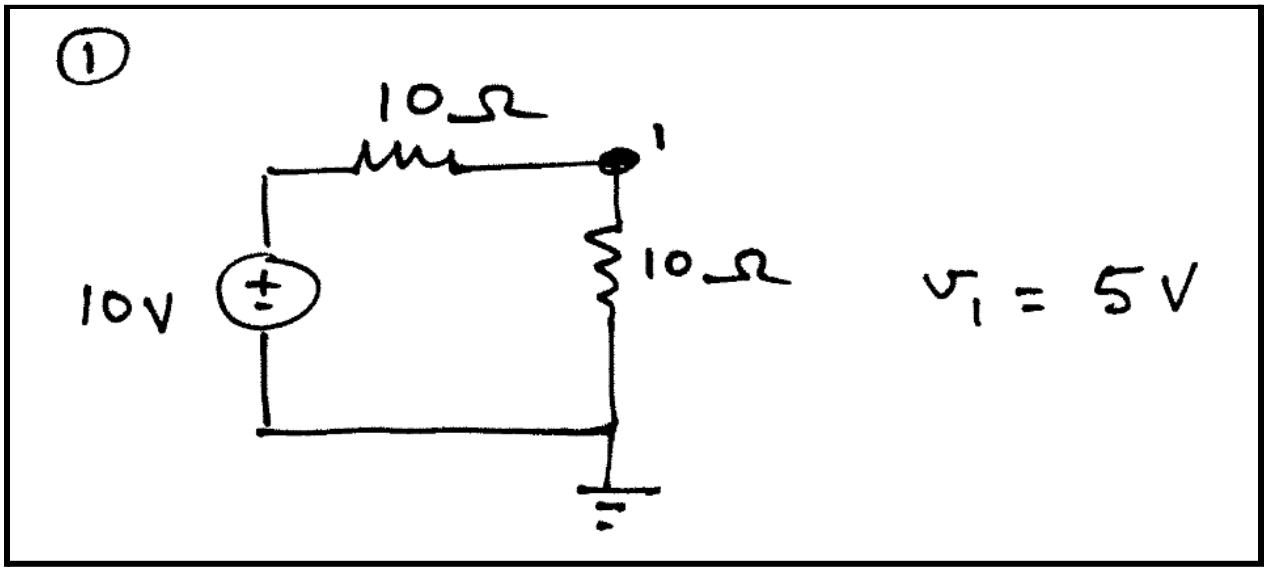
To parse the files, a Julia package for parsing TOMLs, *TOML.jl*, was used. The arrays as parsed by the TOML.jl packages were interpreted into vector or matrix datatypes as appropriate, in order to perform the necessary calculations. A function was then written for solving a given circuit, which calls the above function for Choleski decomposition. The function signature is given below, and the full function body is in the appendix.

```
"""
Solves a circuit given the reduced incidence matrix, A; current source
matrix, J;
resistance matrix, R; and voltage source matrix, E. Solves using Choleski
decomposition.

Tip: use the spread operator, `...`, to use the returned output of
`get_circuit(id)`
without having to access the members of the named tuple.
"""
function solve_circuit(A, J, R, E)
```

Following are the circuits it was tested on, as well as the results.

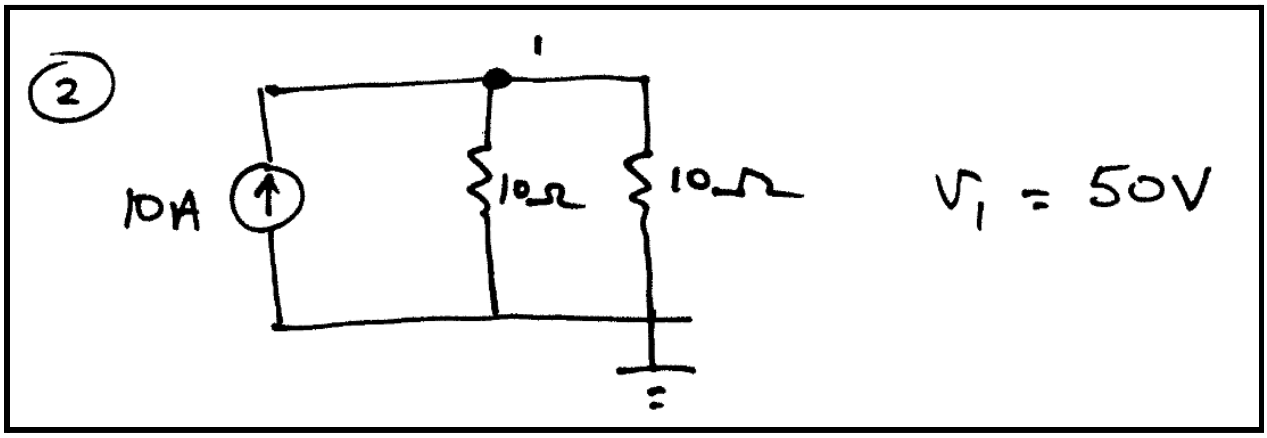
Circuit 1:




Result:

```
julia> solve_circuit(circuit1...)
1×1 Matrix{Float64}:
 5.0
```

Circuit 2:

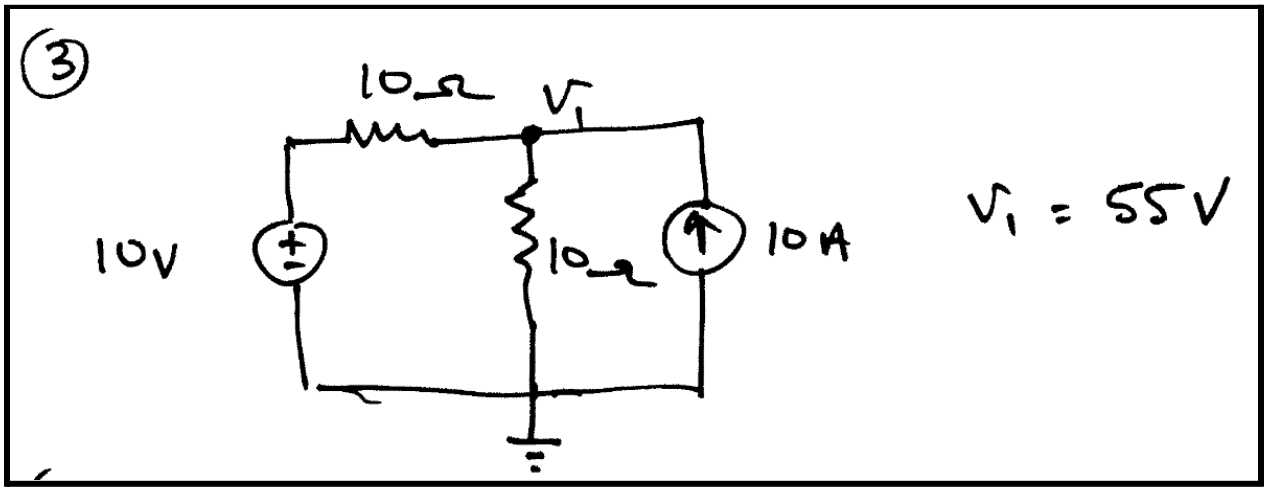



Result:

```
julia> solve_circuit(circuit2...)
1×1 Matrix{Float64}:
 50.00000000000001
```

Note the floating-point error here; this is what the approximate equals operator was used for in the unit tests.

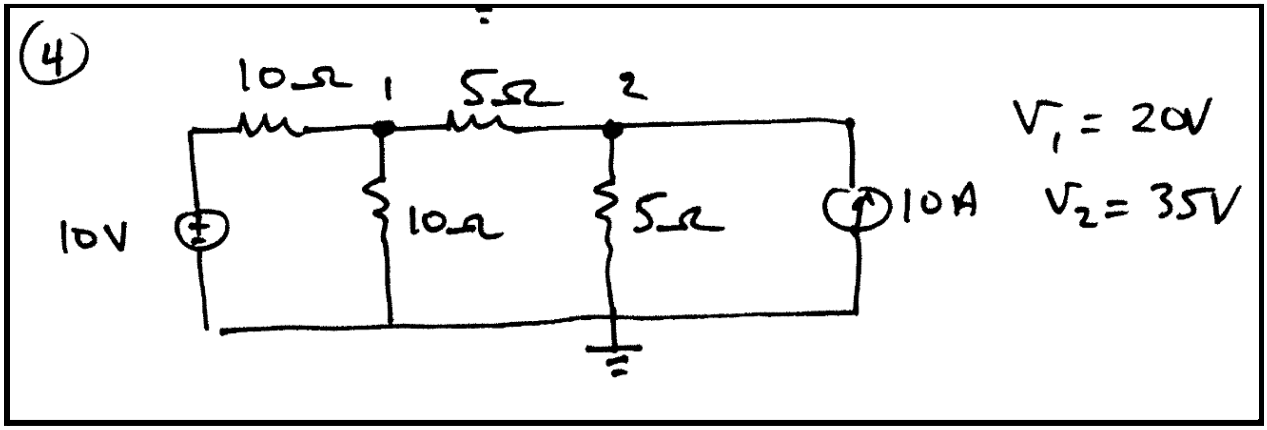
Circuit 3:




Result:

```
julia> solve_circuit(circuit3...)
1×1 Matrix{Float64}:
 55.00000000000001
```

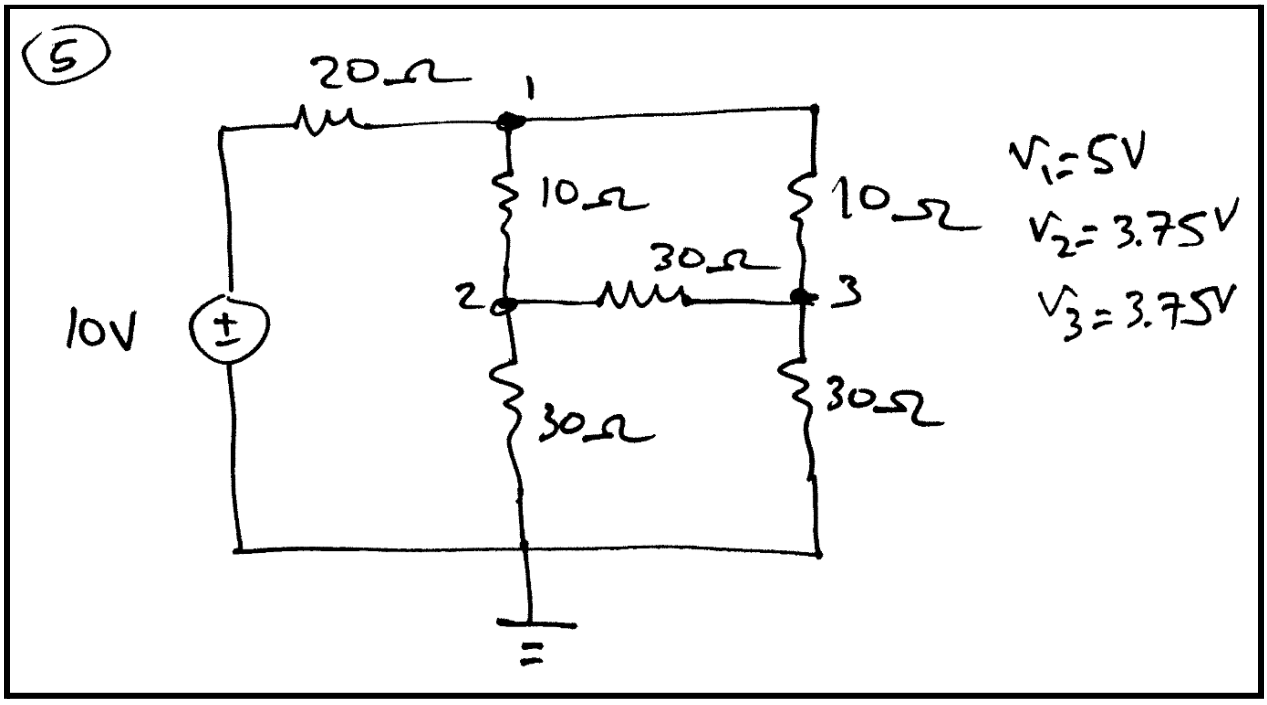
Circuit 4:




Result:

```
julia> solve_circuit(circuit4...)
2×1 Matrix{Float64}:
 19.999999999999993
 34.999999999999986
```

Circuit 5:



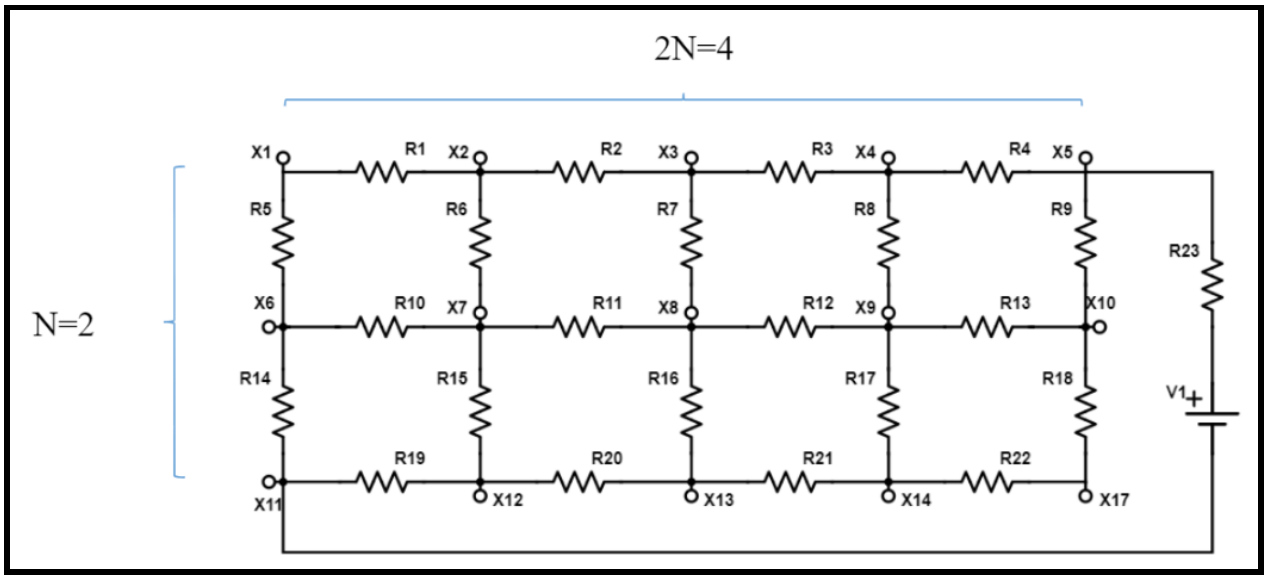

Result:

```
julia> solve_circuit(circuit5...)
3×1 Matrix{Float64}:
 5.000000000000001
 3.7500000000000004
 3.750000000000001
```

## Question 2: Take a regular N-by-2N finite-difference mesh and replace each horizontal and vertical line by a 1-kΩ resistor. This forms a linear, resistive network.

a) Using the program you developed in question 1, find the resistance, R, between the node at the bottom-left corner of the mesh and the node at the top right corner of the mesh, for N = 2, 3, ..., 10. (You will probably want to write a small program that generates the input file needed by the network analysis program. Constructing by hand the incidence matrix for a 200-node network is rather tedious).

A function was created to generate the key matrices and vectors for defining a circuit configuration for an N-by-2N finite-difference mesh, like shown in the diagram below.




To determine the equivalent resistance, a branch is added by the function, connecting the top-right corner to the bottom-left, with a voltage source and a resistor. By calculating the voltage across the network, the current through the driving branch is calculated. With the current and the voltage, the equivalent resistance was calculated using Ohm's Law. The results for N from 2 to 10 are shown in the table below.

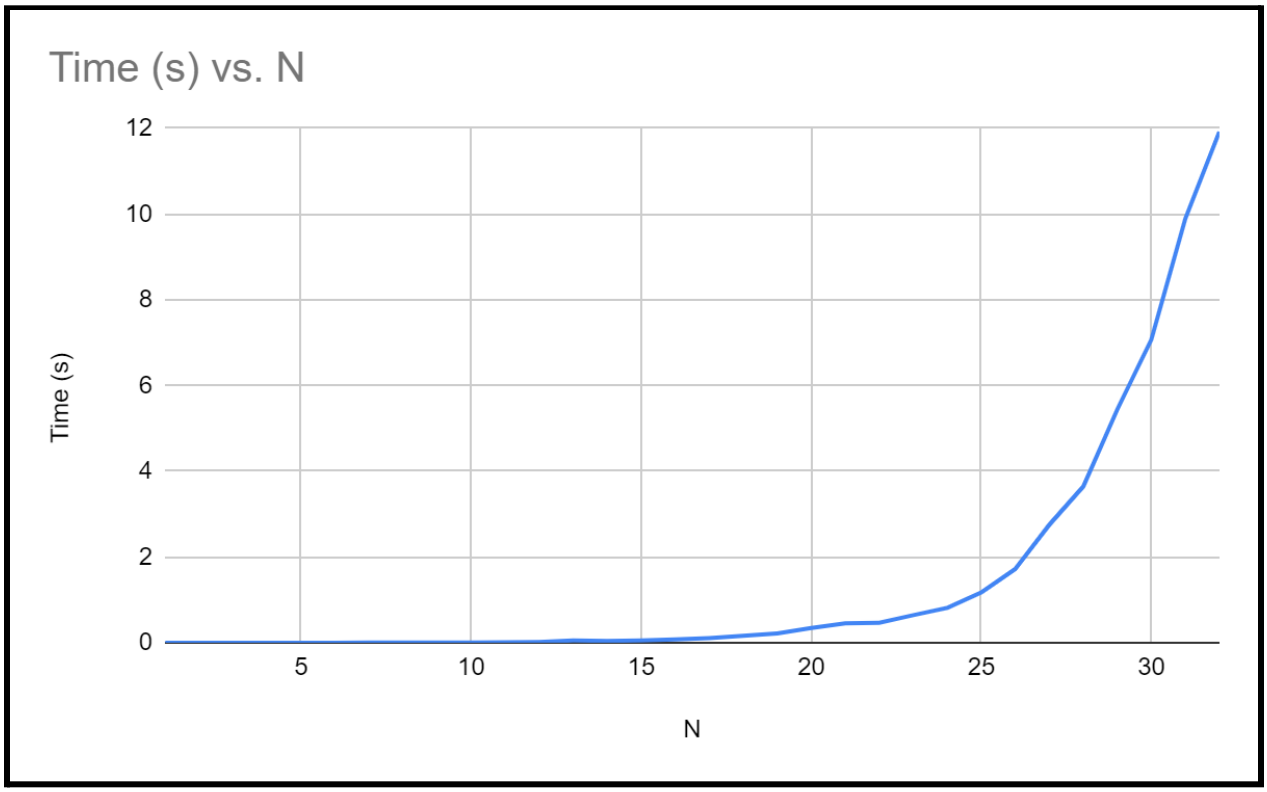
| N | Equivalent Resistance (Ω) |
|---|---|
| 2 | 2057.416268 |
| 3 | 2497.718031 |
| 4 | 2827.490806 |
| 5 | 3090.573861 |
| 6 | 3309.185198 |
| 7 | 3496.083544 |
| 8 | 3659.251365 |
| 9 | 3804.006422 |
| 10 | 3934.065475 |

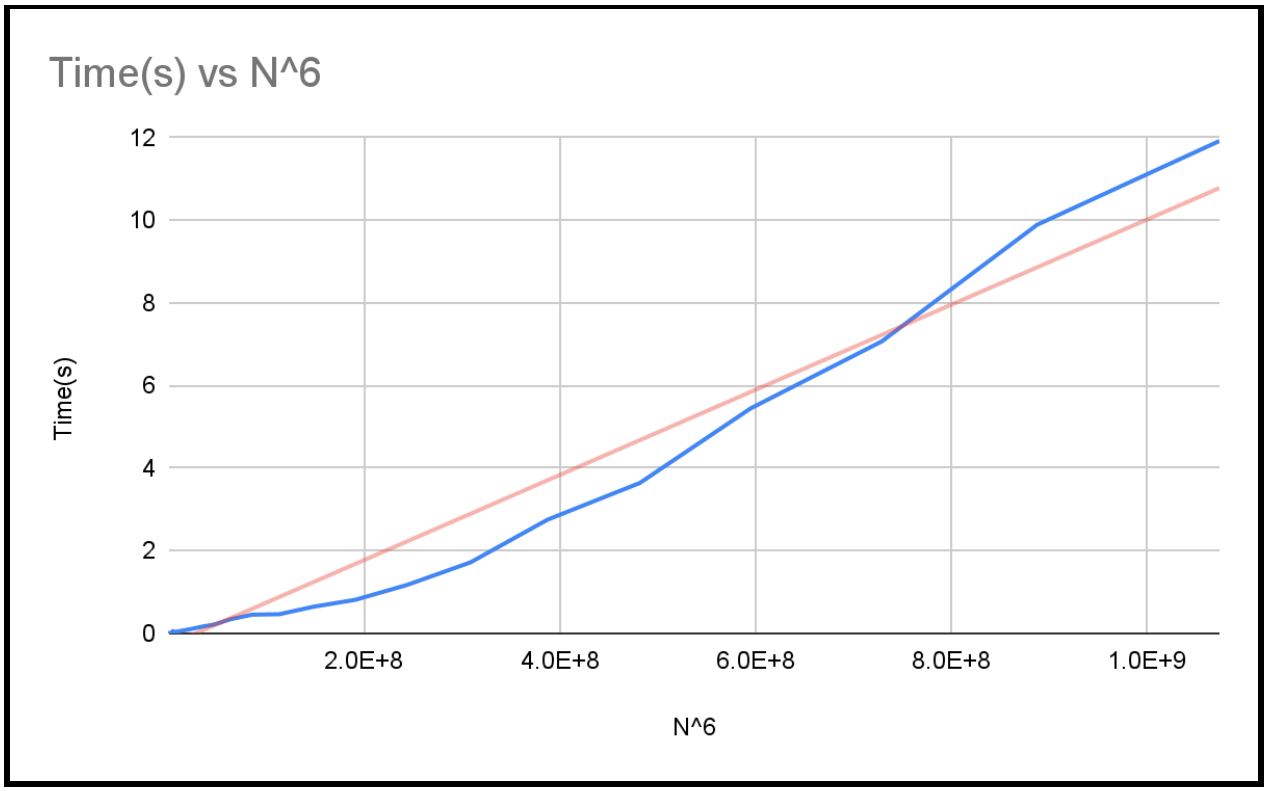
b) In theory, how does the computer time taken to solve this problem increase with N, for large N? Are the timings you observe for your practical implementation consistent with this? Explain your observations.

Choleski decomposition is $O(n^3)$, where n refers to the size of the matrix. However, because the time complexity we want for this is in terms of N, where N is the size of the finite-difference mesh, we need the complexity of n, the matrix size, with regards to N, the circuit size. Because the size of matrix A in Ax=b is nxn, where n is the number of nodes (each element of x

represents the voltage at one node), and there are $n = (N + 1)(2N + 1) = 2N^2 + 3N + 1$ nodes, the time complexity of solving the circuit is $O(n^3) = O((2N^2 + 3N + 1)^3) = O(N^6)$.

For the practical implementation, the results for N from 1 to 32 are plotted below, as well as plotted vs $N^6$, with the raw results shown further below. The observed timings appear to be generally consistent with the theoretical, with the only caveat that individual timings for samples vary, depending on the just-in-time compile time of the Julia programming language. This can be seen in the raw results, where it reports the percentage that is “gc time”.



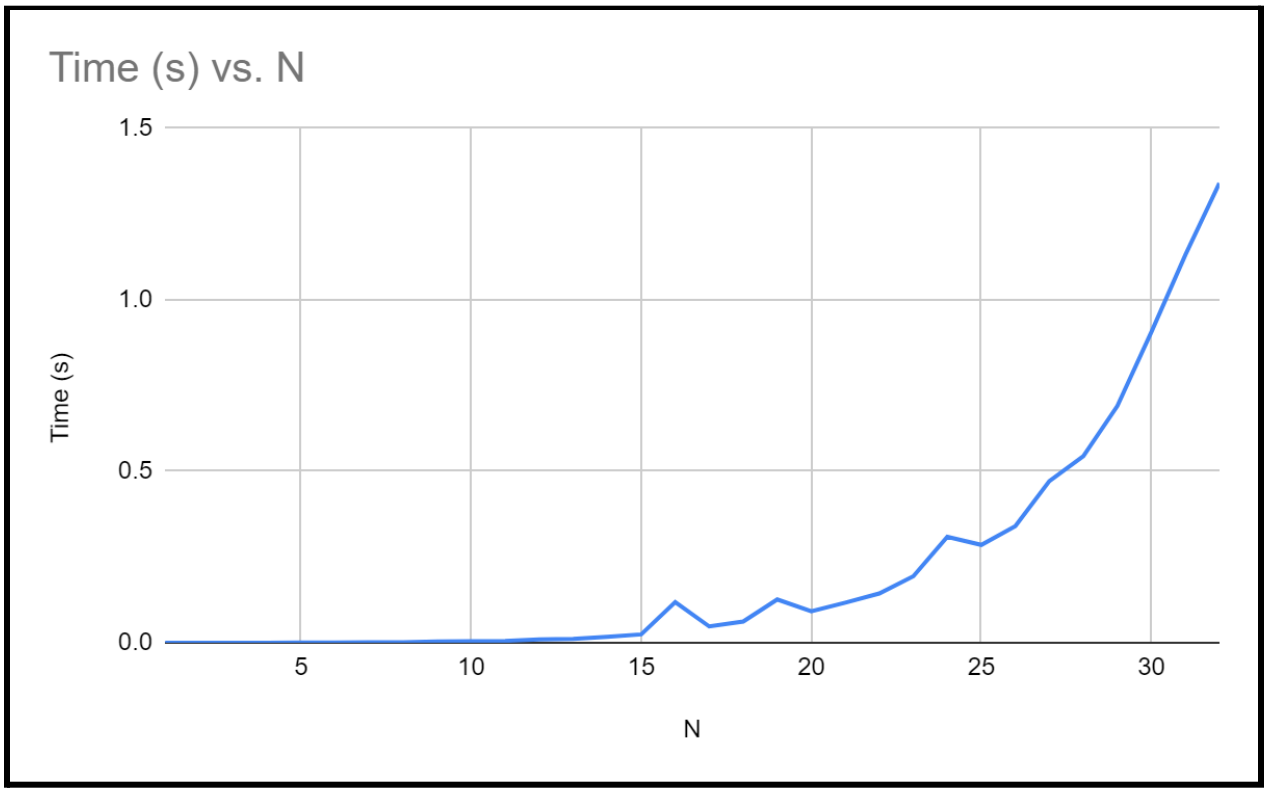


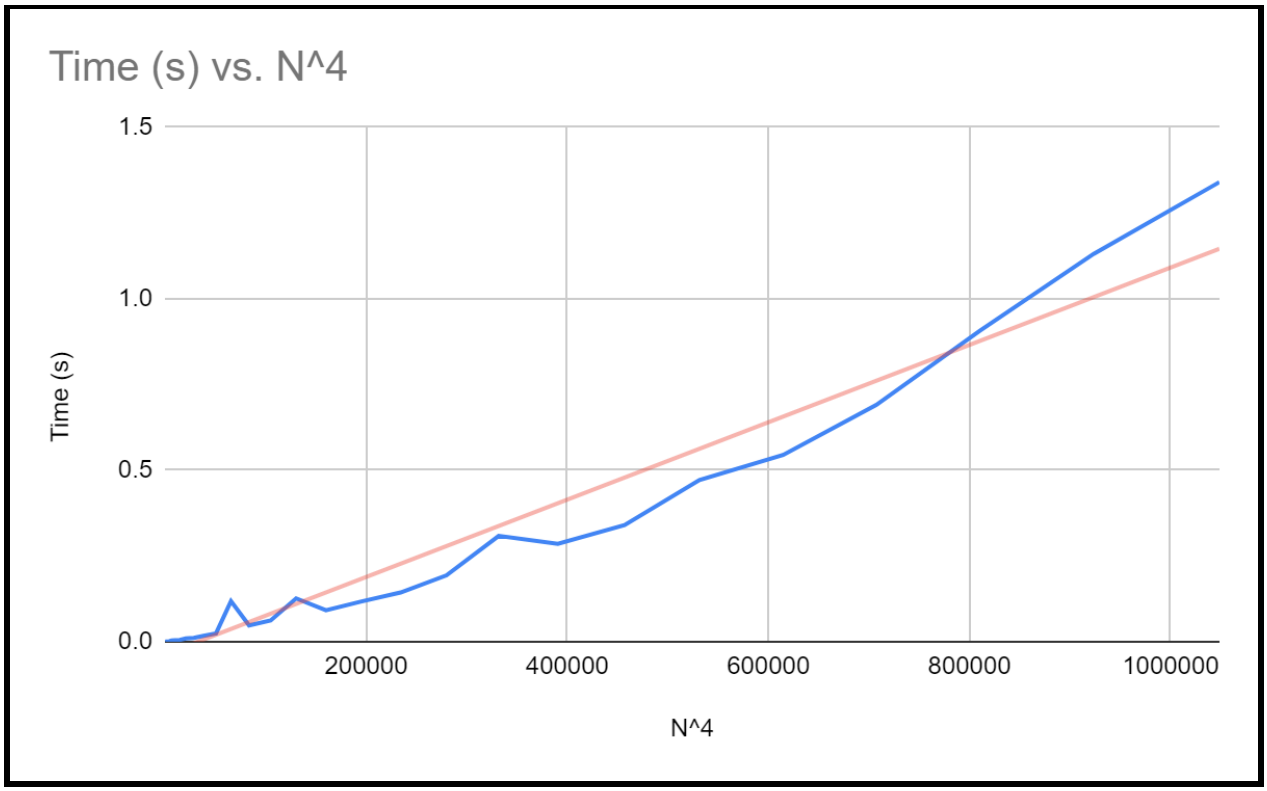

```
N = 1     0.000013 seconds (9 allocations: 2.219 KiB)
N = 2     0.000016 seconds (9 allocations: 11.625 KiB)
N = 3     0.000041 seconds (10 allocations: 40.031 KiB)
N = 4     0.000083 seconds (11 allocations: 106.281 KiB)
N = 5     0.000245 seconds (13 allocations: 234.297 KiB)
N = 6     0.000567 seconds (13 allocations: 454.484 KiB)
N = 7     0.001105 seconds (13 allocations: 802.844 KiB)
N = 8     0.002290 seconds (13 allocations: 1.290 MiB)
N = 9     0.006104 seconds (13 allocations: 2.009 MiB)
N = 10    0.005483 seconds (13 allocations: 2.994 MiB)
N = 11    0.010728 seconds (13 allocations: 4.303 MiB)
N = 12    0.015066 seconds (13 allocations: 6.002 MiB)
N = 13    0.052642 seconds (13 allocations: 8.160 MiB, 50.43% gc time)
N = 14    0.037217 seconds (13 allocations: 10.854 MiB)
N = 15    0.052096 seconds (13 allocations: 14.166 MiB)
N = 16    0.077089 seconds (13 allocations: 18.184 MiB)
N = 17    0.109164 seconds (13 allocations: 23.001 MiB)
N = 18    0.160886 seconds (13 allocations: 28.718 MiB, 6.47% gc time)
N = 19    0.215193 seconds (13 allocations: 35.440 MiB, 8.31% gc time)
N = 20    0.344996 seconds (13 allocations: 43.279 MiB, 22.39% gc time)
N = 21    0.452820 seconds (13 allocations: 52.351 MiB, 13.35% gc time)
N = 22    0.463546 seconds (13 allocations: 62.780 MiB, 0.60% gc time)
```

```
N = 23    0.643390 seconds (15 allocations: 74.695 MiB, 0.51% gc time)
N = 24    0.812997 seconds (15 allocations: 88.230 MiB, 0.40% gc time)
N = 25    1.173516 seconds (15 allocations: 103.527 MiB, 0.85% gc time)
N = 26    1.720299 seconds (15 allocations: 120.732 MiB, 0.56% gc time)
N = 27    2.751907 seconds (15 allocations: 139.997 MiB, 3.09% gc time)
N = 28    3.642942 seconds (15 allocations: 161.481 MiB, 0.31% gc time)
N = 29    5.444953 seconds (15 allocations: 185.348 MiB, 0.06% gc time)
N = 30    7.069585 seconds (15 allocations: 211.768 MiB, 0.28% gc time)
N = 31    9.890783 seconds (15 allocations: 240.917 MiB, 0.26% gc time)
N = 32   11.919678 seconds (18 allocations: 272.975 MiB, 0.25% gc time)
```

c) Modify your program to exploit the sparse nature of the matrices to save computation time. What is the half-bandwidth b of your matrices? In theory, how does the computer time taken to solve this problem increase now with N, for large N? Are the timings for your practical sparse implementation consistent with this? Explain your observations.

By inspecting the matrices, the halfband b = 2N + 2. The complexity, then, of Cholesky decomposition is $O(b^2n) = O((4N^2 + 8N + 4)(2N^2 + 3N + 1)) = O(N^4)$. As can be seen in the plots below, the experimental results are generally generally consistent with the theory, with the same caveat as for without halfbanding.



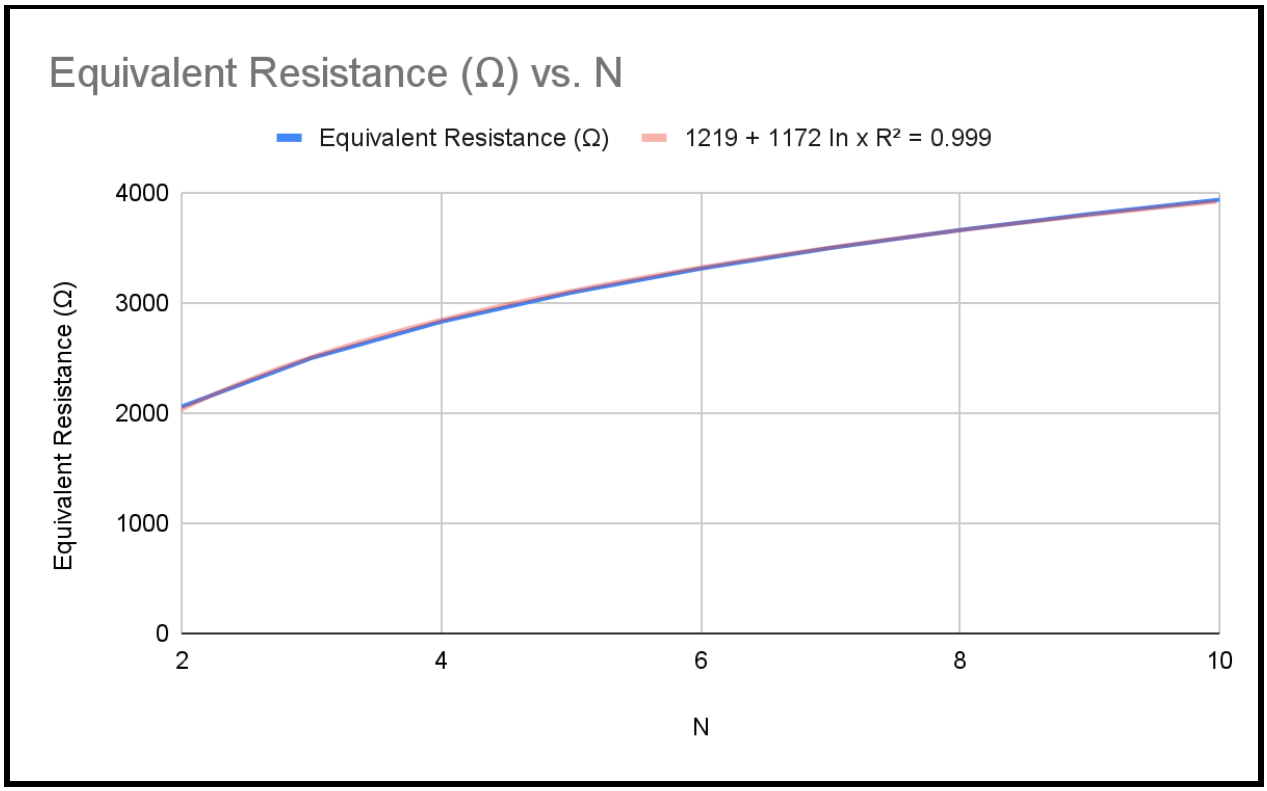



```
N = 1   b = 4     0.000012 seconds (10 allocations: 2.234 KiB)
N = 2   b = 6     0.000015 seconds (10 allocations: 11.641 KiB)
N = 3   b = 8     0.000032 seconds (11 allocations: 40.047 KiB)
N = 4   b = 10    0.000074 seconds (12 allocations: 106.297 KiB)
N = 5   b = 12    0.000260 seconds (14 allocations: 234.312 KiB)
N = 6   b = 14    0.000436 seconds (14 allocations: 454.500 KiB)
N = 7   b = 16    0.000852 seconds (14 allocations: 802.859 KiB)
N = 8   b = 18    0.001327 seconds (14 allocations: 1.290 MiB)
N = 9   b = 20    0.003338 seconds (14 allocations: 2.009 MiB)
N = 10  b = 22    0.004488 seconds (14 allocations: 2.994 MiB)
N = 11  b = 24    0.005189 seconds (14 allocations: 4.303 MiB)
N = 12  b = 26    0.009716 seconds (14 allocations: 6.002 MiB)
N = 13  b = 28    0.011336 seconds (14 allocations: 8.160 MiB)
N = 14  b = 30    0.017198 seconds (14 allocations: 10.854 MiB)
N = 15  b = 32    0.024166 seconds (14 allocations: 14.166 MiB)
N = 16  b = 34    0.118487 seconds (14 allocations: 18.184 MiB, 71.00% gc
time)
N = 17  b = 36    0.047366 seconds (14 allocations: 23.001 MiB)
N = 18  b = 38    0.061710 seconds (14 allocations: 28.718 MiB)
N = 19  b = 40    0.126183 seconds (14 allocations: 35.440 MiB, 45.87% gc
time)
N = 20  b = 42    0.091651 seconds (14 allocations: 43.279 MiB, 5.11% gc
```

```
time)
N = 21  b = 44    0.116694 seconds (14 allocations: 52.351 MiB, 4.56% gc
time)
N = 22  b = 46    0.143454 seconds (14 allocations: 62.780 MiB, 4.40% gc
time)
N = 23  b = 48    0.193593 seconds (16 allocations: 74.695 MiB, 3.39% gc
time)
N = 24  b = 50    0.308600 seconds (16 allocations: 88.230 MiB, 1.59% gc
time)
N = 25  b = 52    0.285202 seconds (16 allocations: 103.527 MiB, 1.28% gc
time)
N = 26  b = 54    0.339835 seconds (16 allocations: 120.732 MiB, 1.29% gc
time)
N = 27  b = 56    0.471387 seconds (16 allocations: 139.997 MiB, 2.85% gc
time)
N = 28  b = 58    0.544305 seconds (16 allocations: 161.481 MiB, 1.01% gc
time)
N = 29  b = 60    0.690168 seconds (16 allocations: 185.348 MiB, 0.91% gc
time)
N = 30  b = 62    0.905897 seconds (16 allocations: 211.768 MiB, 7.27% gc
time)
N = 31  b = 64    1.130770 seconds (16 allocations: 240.917 MiB, 1.93% gc
time)
N = 32  b = 66    1.340049 seconds (19 allocations: 272.975 MiB, 0.43% gc
time)
```

d) Plot a graph of R versus N. Find a function R(N) that fits the curve reasonably well and is asymptotically correct as N tends to infinity, as far as you can tell.

Below is a plot of equivalent resistance, R, vs N. Included is a logarithmic trendline, with function R(N) = 1219 + 1172ln(N). As shown below, the trendline has an $R^2$ value of 0.999, suggesting a very strong fit, as is also visible by how closely the trendline tracks the observed results.

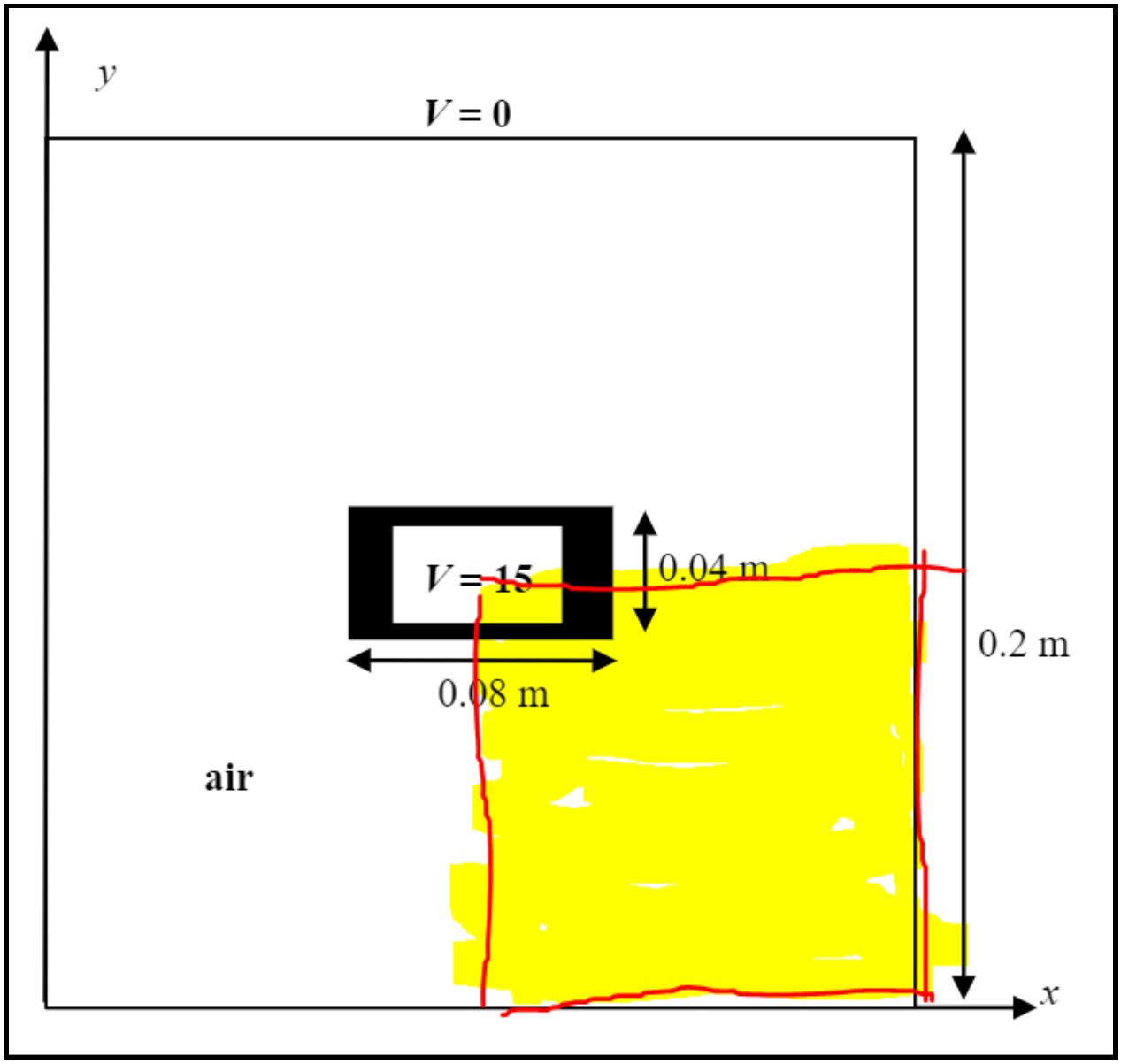



Question 3: Figure 1 shows the cross-section of an electrostatic problem with translational symmetry: a coaxial cable with a square outer conductor and a rectangular inner conductor. The inner conductor is held at 15 volts and the outer conductor is grounded.

a) Write a computer program to find the potential at the nodes of a regular mesh in the air between the conductors by the method of finite differences. Use a five-point difference formula. Exploit at least one of the planes of mirror symmetry that this problem has. Use an equal node-spacing, h, in the x and y directions. Solve the matrix equation by successive over-relaxation (SOR), with SOR parameter ω. Terminate the iteration when the magnitude of the residual at each free node is less than $10^{-5}$.

Because of symmetry, only one quadrant needs to be calculated and the results for any quadrant would be identical, only translated. The bottom-right quadrant was selected as shown below. Within the selected quadrant, (0,0) represents the top-left corner, or the center of the full

space. Increasing y-coordinates represent going down vertically, and increasing x-coordinates represent going to the right horizontally.



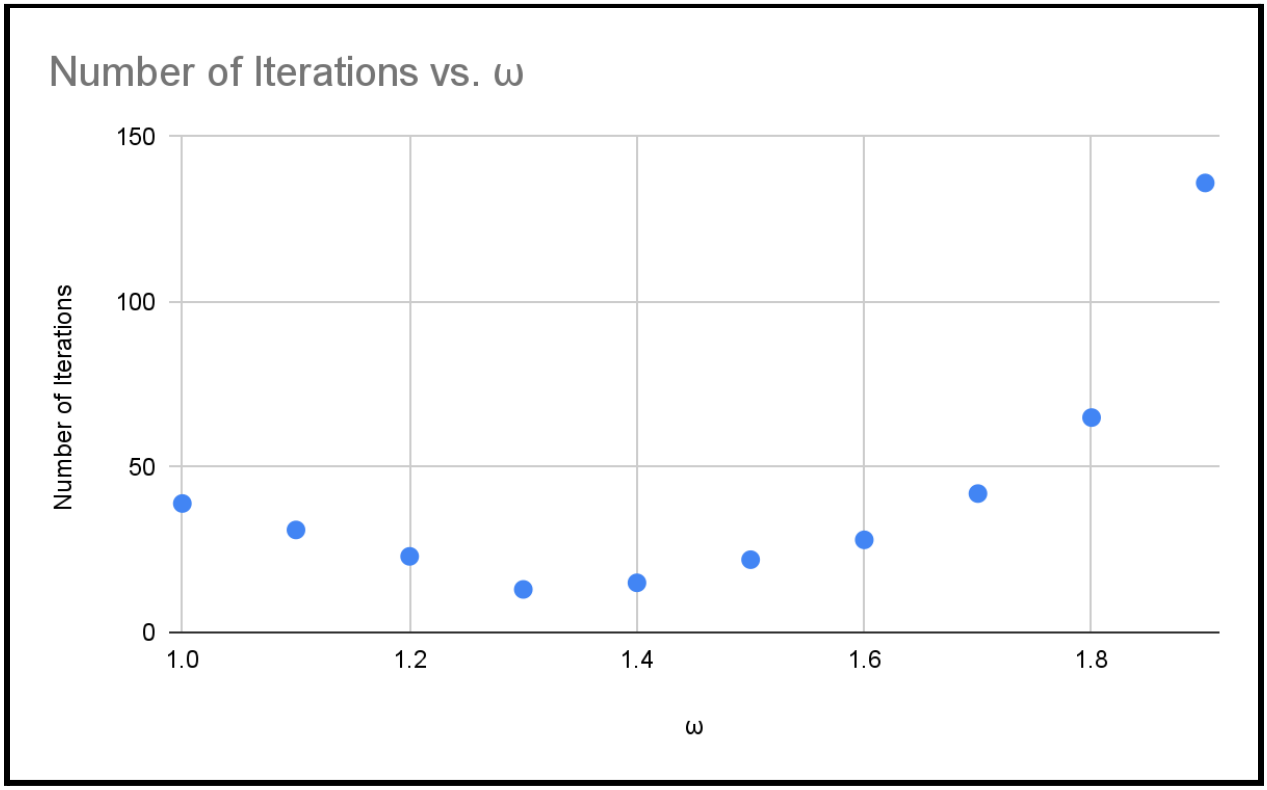

To solve this problem, a struct was made to hold the matrix representation of the nodes (generated by the h parameter), various height and width attributes, and so forth (full struct is available in the source code in the appendix). Additionally, two helper functions, *SOR_iterate!* and *get_residual*, were created to, respectively, perform one iteration of SOR and calculate the residual at the current step. These were called in the main *SOR!* function that directed the conditional iterations. As a simple test, the output for h=0.02 and ω=1.5 is shown below. Note that the top-left corner corresponds with the inner conductor, and the right and bottom sides correspond with the outer conductor.

```
15.0      15.0      15.0      9.09187   4.25249   0.0
15.0      15.0      15.0      8.5575    3.95905   0.0
```

```
10.549    10.2912    9.24916  6.17907  3.0262    0.0
 6.61348   6.36677   5.52634  3.88343  1.96668   0.0
 3.17139   3.03604   2.606    1.86163  0.957076  0.0
 0.0       0.0       0.0      0.0      0.0       0.0
```

b) With h = 0.02, explore the effect of varying ω. For 10 values of ω between 1.0 and 2.0, tabulate the number of iterations taken to achieve convergence, and the corresponding value of potential at the point (x ,y) = (0.06, 0.04). Plot a graph of the number of iterations versus ω.

Due to the symmetry, the point (0.06, 0.04) is mirrored at (0.14, 0.04) in the simulated quadrant. Hence, values are taken from this point. The results of number of iterations vs ω is shown below. Of note is the global minimum around ω=1.3. Due to the small sample size, however, the shape of the curve cannot be accurately extrapolated. It is apparent, however, that, as ω approaches 2, the number of iterations approaches infinity. This is in line with how it diverges at ω=2.




In the table below are the calculated potentials for (0.06, 0.04). Note that the values are all the same to 4 decimal points.

| ω | Potential at (0. 6, 0. 4) |
|---|---|
| 1 | 5.526336278 |

| | |
|---|---|
| 1.1 | 5.526338094 |
| 1.2 | 5.526338924 |
| 1.3 | 5.526340786 |
| 1.4 | 5.526338744 |
| 1.5 | 5.526341886 |
| 1.6 | 5.526339448 |
| 1.7 | 5.52634287 |
| 1.8 | 5.526339761 |
| 1.9 | 5.526339093 |

c) With an appropriate value of ω, chosen from the above experiment, explore the effect of decreasing h on the potential. Use values of h = 0.02, 0.01, 0.005, etc., and both tabulate and plot the corresponding values of potential at (x, y) = (0.06, 0.04) versus 1/h. What do you think is the potential at (0.06, 0.04), to three significant figures? Also, tabulate and plot the number of iterations versus 1/h. Comment on the properties of both plots.

## Appendix

*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
```

*circuit.jl*

```
using TOML
```

```
"""
Retrieves a circuit of a given ID and returns its reduced incidence matrix, A; current source matrix, J;
resistance matrix, R; and voltage source matrix, E. These are returned as a named tuple.
"""
function get_circuit(id::Int)
    circuit = TOML.tryparsefile("Assignment 1/circuits/circuit$(id).toml")

    # return as named tuple
    # need to flatten A into a matrix instead of nested vectors
    return (A = transpose(hcat(circuit["A"]...)), J = circuit["J"], R = circuit["R"], E = circuit["E"])
end

"""
Solves a circuit given the reduced incidence matrix, A; current source matrix, J;
resistance matrix, R; and voltage source matrix, E. Solves using Choleski decomposition.

Tip: use the spread operator, `...`, to use the returned output of `get_circuit(id)`
without having to access the members of the named tuple.
"""
function solve_circuit(A, J, R, E, halfband=nothing)
    # (AYA^T)v = A(J - YE)

    # fill in Y with zeros everywhere but the main diagonal
    Y = zeros(length(R), length(R))
    for i ∈ 1:length(R)
        Y[i,i] = 1.0 / R[i]
    end

    # perform choleski decomposition with constructed A and b
    v = choleski(A * Y * transpose(A), A * (J - (Y * E)), halfband)
    return v
end

"""
Generates an N×2N finite-difference mesh where each branch is a resistor of given resistance.
It also has attached a branch connecting the top-right node to the
```

```
bottom-left node
with a voltage source and resistor. This is to allow measurement of
equivalent resistance.

Return value is in the same form as for `get_circuit`, allowing easy piping
into `solve_circuit`
with the spread operator, `...`.
"""
function generate_circuit(N, R, sJ, sR, sE)
    # calculate number of branches
    n_branch = (N + 1)*2*N + N*(2*N + 1)

    # construct 𝐉, 𝐑, 𝐄 for the main N×2N mesh
    𝐉 = zeros(n_branch)
    𝐑 = R * ones(n_branch)
    𝐄 = zeros(n_branch)

    # add voltage-source branch for testing equivalent resistance across
the mesh
    push!(𝐉, sJ)
    push!(𝐑, sR)
    push!(𝐄, sE)

    # calculate nodes
    n_node = (N + 1) * (2*N + 1)

    # zero-initialize 𝐀
    𝐀 = zeros(n_node, n_branch + 1)
    #print len(𝐀), len(𝐀[0])

    # set all the correct values in 𝐀
    for i_node ∈ 1:n_node

        # level = which row within the mesh, 1-indexed
        level = ((i_node - 1) ÷ (2*N + 1)) + 1

        # offset = which column within the mesh, 1-indexed
        offset = ((i_node - 1) % (2*N + 1)) + 1

        branch_per_level = 2*N + (2*N + 1)

        # calculate surrounding branch indices
        right = branch_per_level * (level - 1) + offset
```

```
        left = right - 1
        top = right - (2*N + 1)
        bottom = top + branch_per_level
        #print i, 'r',right, 'l', left, 't',top, 'b', bottom

        # node on top border
        if level == 1
            # top left
            if offset == 1
                A[i_node, right], A[i_node, bottom] = -1, 1
            # top right
            elseif offset == (2*N + 1)
                A[i_node, left], A[i_node, bottom] = 1, 1
            else
                A[i_node, right], A[i_node, left], A[i_node, bottom] = -1,
1, 1
            end
        # node on bottom border
        elseif level == (N + 1)
            # bottom left
            if offset == 1
                A[i_node, right], A[i_node, top] = -1, -1
            # bottom right
            elseif offset == (2*N + 1)
                A[i_node, top], A[i_node, left] = -1, 1
            else
                A[i_node, right], A[i_node, top], A[i_node, left] = -1,
-1, 1
            end
        # node on left border
        elseif offset == 1
            A[i_node, right], A[i_node, top], A[i_node, bottom] = -1, -1,
1
        # node on right border
        elseif offset == (2*N + 1)
            A[i_node, top], A[i_node, left], A[i_node, bottom] = -1, 1, 1
        # nodes in middle
        else
            A[i_node, right], A[i_node, top], A[i_node, left], A[i_node,
bottom] = -1, -1, 1, 1
        end

        # setup the voltage source
```

```
        # right top corner
        if level == 1 && offset == (2*N + 1)
            A[i_node, n_branch + 1] = -1
        elseif level == (N + 1) && offset == 1
            A[i_node, n_branch + 1] = 1
        end
    end

    # set up ground node by removing that node from A
    # i.e., remove the row from A corresponding to the bottom-left corner
node
    A = A[setdiff(1:end, N*(2*N + 1) + 1), :]

    return (A = A, J = J, R = R, E = E)
end
```

*main.jl*

```
include("Assignment 1/choleski.jl")
include("Assignment 1/circuit.jl")

# solve circuit 1 from /circuits/circuit1.toml
circuit1 = get_circuit(1)
solve_circuit(circuit1...)

# solve circuit 2 from /circuits/circuit2.toml
circuit2 = get_circuit(2)
solve_circuit(circuit2...)

# solve circuit 3 from /circuits/circuit3.toml
circuit3 = get_circuit(3)
solve_circuit(circuit3...)

# solve circuit 4 from /circuits/circuit4.toml
circuit4 = get_circuit(4)
solve_circuit(circuit4...)

# solve circuit 5 from /circuits/circuit5.toml
circuit5 = get_circuit(5)
solve_circuit(circuit5...)

# solve N×2N finite-difference mesh for N = 2
R, sJ, sR, sE = 1000, 0, 1000, 100
```

```
R_equivalent = []
for N ∈ 1:32
    circuit_mesh = generate_circuit(N, R, sJ, sR, sE)
    halfband = 2*N + 2
    print("N = $N\tb = $halfband\t")
    @time v_mesh =  solve_circuit(circuit_mesh..., halfband)
    v₁ = v_mesh[2*N + 1]
    v₂ = 0
    I = ((v₂ + sE) - v₁) / sR
    R_mesh = (v₁ - v₂) / I

    push!(R_equivalent, R_mesh)
end
```

```
## question 3

include("Assignment 1/finite-difference.jl")

# use SOR on a mesh with h=0.02 and residual limit of 1e-5
result_potentials = []
for ω ∈ 1.0:0.1:1.9
    mesh = PotentialMesh()
    SOR!(mesh, ω)
    push!(result_potentials, mesh.mesh[4,3])
end
result_potentials
```

*finite-difference.jl*

```
mutable struct PotentialMesh
    mesh::AbstractMatrix{AbstractFloat}
    h::AbstractFloat
    residual_limit::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,
residual_limit::AbstractFloat=1e-5)
        # 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, residual_limit, (inner_rows, inner_cols),
params...)
    end
end

"""
Performs one iteration of SOR on a potential mesh.
"""
function SOR_iterate!(potentials::PotentialMesh, ω::AbstractFloat)
    num_rows, num_cols = size(potentials.mesh)
    inner_rows, inner_cols = potentials.inner_size

    for j ∈ 1:num_rows-1
        for i ∈ 1:num_cols-1
            if i == 1 && j > inner_rows # if along y-axis but not in inner
conductor
                # because of symmetry, we can assume the potential at
```

```
[j,i-1] = [j,i+1], hence the 2*[j,i+1]
                potentials.mesh[j, i] = (1 - ω) * potentials.mesh[j, i] +
(ω / 4) * (2*potentials.mesh[j, i+1] + potentials.mesh[j-1, i] +
potentials.mesh[j+1, i])
            elseif j == 1 && i > inner_cols # if along x-axis but not in
inner conductor
                # same argument as above, just changed for x-axis
                potentials.mesh[j, i] = (1 - ω) * potentials.mesh[j, i] +
(ω / 4) * (potentials.mesh[j, i-1] + potentials.mesh[j, i+1] +
2*potentials.mesh[j+1, i])
            elseif i > inner_cols || j > inner_rows # if somewhere else
outside inner conductor
                potentials.mesh[j, i] = (1 - ω) * potentials.mesh[j, i] +
(ω / 4) * (potentials.mesh[j, i-1] + potentials.mesh[j, i+1] +
potentials.mesh[j-1, i] + potentials.mesh[j+1, i])
            end # don't do anything for anything in the inner conductor
        end
    end

    return potentials
end

"""
Calculates the residual of a given potential mesh state.
"""
function get_residual(potentials::PotentialMesh)
    residual_cur = 0
    residual_fin = 0

    num_rows, num_cols = size(potentials.mesh)
    inner_rows, inner_cols = potentials.inner_size

    for j ∈ 1:num_rows-1
        for i ∈ 1:num_cols-1
            if i == 1 && j > inner_rows # if along y-axis but not in inner
conductor
                # because of symmetry, we can assume the potential at
[j,i-1] = [j,i+1], hence the 2*[j,i+1]
                residual_cur = 2*potentials.mesh[j, i+1] +
potentials.mesh[j-1, i] + potentials.mesh[j+1, i] - 4*potentials.mesh[j, i]
            elseif j == 1 && i > inner_cols # if along x-axis but not in
inner conductor
                # same argument as above, just changed for x-axis
```

```
                residual_cur = potentials.mesh[j, i-1] + potentials.mesh[j,
i+1] + 2*potentials.mesh[j+1, i] - 4*potentials.mesh[j, i]
            elseif i > inner_cols || j > inner_rows # if somewhere else
outside inner conductor
                residual_cur = potentials.mesh[j, i-1] + potentials.mesh[j,
i+1] + potentials.mesh[j-1, i] + potentials.mesh[j+1, i] -
4*potentials.mesh[j, i]
            end # don't do anything for anything in the inner conductor

            residual_cur = abs(residual_cur)
            if residual_cur > residual_fin
                # update variable with the biggest residue amongst the free
point
                residual_fin = residual_cur
            end
        end
    end
    return residual_fin
end

"""
Performs the whole SOR algorithm.
"""
function SOR!(potentials::PotentialMesh, ω::AbstractFloat)
    iteration = 0
    SOR_iterate!(potentials, ω)
    while get_residual(potentials) >= potentials.residual_limit
        SOR_iterate!(potentials, ω)
        iteration += 1
    end
    println("total iteration is: $iteration")
    return potentials
end
```

## References


https://en.wikipedia.org/wiki/Definite_matrix#Examples
https://toml.io/en/
https://github.com/JuliaLang/TOML.jl