Assignment 1

ECSE 543

Professor: Prof. Dennis  Giannacopoulos

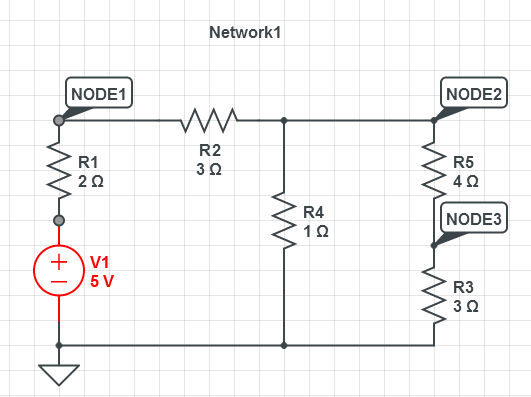
Student Name: Chris Morin

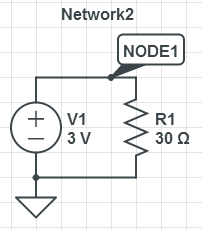
Student Number: 260344722

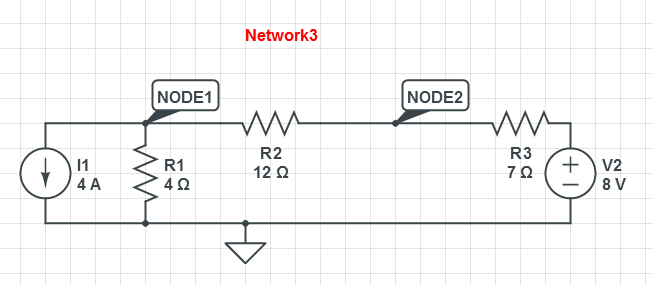
Question 1

There are 4 source files for this question. The first is called “Matrices.py” and it contains the Matrix classes. This is done to encapsulate the linear algebra from the problem specific code. The next file is called “NetworkSolver.py”. This contains a single class which parses the input file and transforms the physical problem to a mathematical one. It uses the Matrix classes to do the actual computation. The other two source files “TestCholesky.py” and “TestSolver.py” just test the aforementioned classes.

1. This is all contained in the “Matrices.py” file. A base class called “Matrix” contains all necessary matrix functions like multiplication or transposition along with doing a few sanity checks along the way. The class “BehavingMatrix” is a matrix that is real, symmetric and positive definite; it therefore has the Cholesky methods.
2. See “TestCholesky.py”.
3. “TestCholesky.py” contains a set of test cases used directly on the “Matrices.py” classes (i.e. not going through the NetworkSolver). All the test cases succeed.
4. “NetworkSolver.py” takes care of the translation from input file to matrix operations. The input file format is described in the comments of that file. The file “TestSovler.py” contains the test cases. There are three test circuits. Here are the schematics:







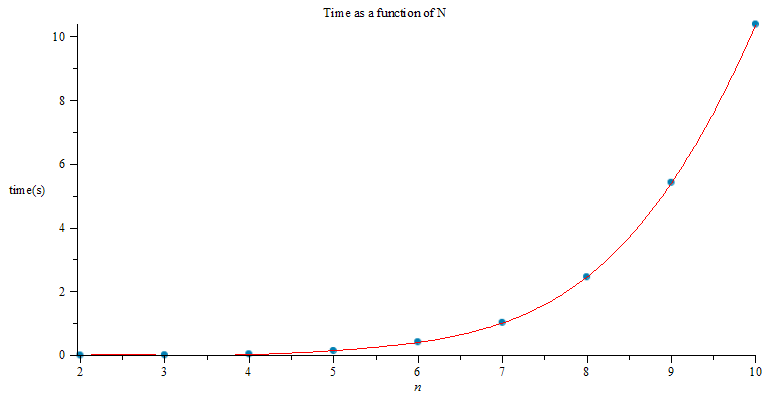
Question 2

Because this question uses modules from Q1, its source files are in the same directory. “resistorMeshGenerator.py” is the program that makes the network data representing the grids of varying sizes of N. “ResistorGridSolver.py” uses this data to calculate the resistances of the circuits; ResisterGridSolver doesn’t do any heavy lifting, this is done by by adding to the already existing source files. The BehavingSparseMatrix class was created; it inherits from BehavingMatrix adds the functionality of being able to take advantage of the matrix sparcity to speed up computation. SparseNetworkSolver was created as a child of NetworkSolver. The only difference between the two is that the sparse version uses the sparse matrix.

1. Well, the “small program” the generated the mesh was the hardest part of the assignment. The program works by creating a spatially accurate array of the nodes and branches (both nodes and branches occupy the same array). It then traverses the array, and when it hits a node, it looks around the node for the surrounding branches. It then converts it to the network data format and dumps is all to a file. Part a of ResisterGridSolver calls NetworkSolver with the appropriate arguments. Here are the resistance values along with computation time:

|  |  |  |
| --- | --- | --- |
| N | R (Ohms) | Time (s) |
| 2 | 1.875 | 0.001 |
| 3 | 2.379545 | 0.007 |
| 4 | 2.741025 | 0.037 |
| 5 | 3.022819 | 0.14 |
| 6 | 3.253676 | 0.415 |
| 7 | 3.449166 | 1.025 |
| 8 | 3.618675 | 2.455 |
| 9 | 3.768291 | 5.42 |
| 10 | 3.902189 | 10.387 |

1. In theory, the computational time should be O(n^3) = O(N^2^3) = O(N^6) as that is the computational complexity of the Cholesky algorithm. Here is a plot of the points with a third order curve fit:



The equation describing the curve fit is as follows:

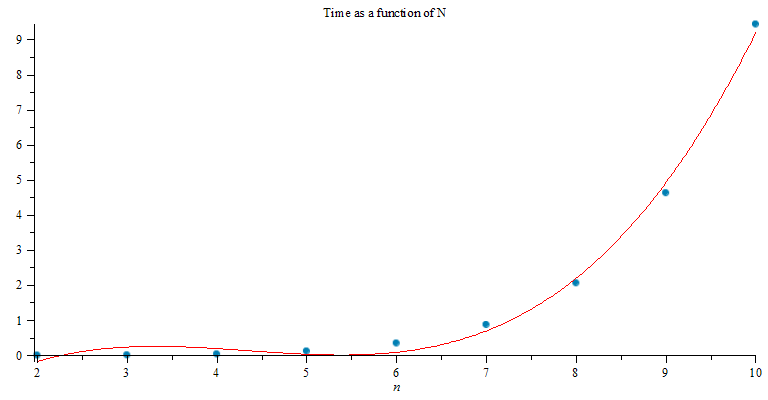


This equations fits very well, but there is a bit of error at the beginning. The discrepancy at the beginning might be attributable to the Cholesky Decomposition not dominating the computational time for small values of N.

1. As was explained above, BehavingMatrix was extended to make BehavingSparseMatrix. This add optimization by checking for the leftmost non-zero value in each row of A before computing L and only using values after those ones. It also computes the half-bandwidth of the matrix. Here are the results side by side with the un-optimized results.

|  |  |  |  |
| --- | --- | --- | --- |
| N | Un-optimized time(s) | Optimised time(s) | Half-bandwidth |
| 2 | 0.001 | 0.002 | 4 |
| 3 | 0.007 | 0.007 | 6 |
| 4 | 0.037 | 0.033 | 8 |
| 5 | 0.14 | 0.116 | 10 |
| 6 | 0.415 | 0.344 | 12 |
| 7 | 1.025 | 0.871 | 14 |
| 8 | 2.455 | 2.06 | 16 |
| 9 | 5.42 | 4.629 | 18 |
| 10 | 10.387 | 9.44 | 20 |

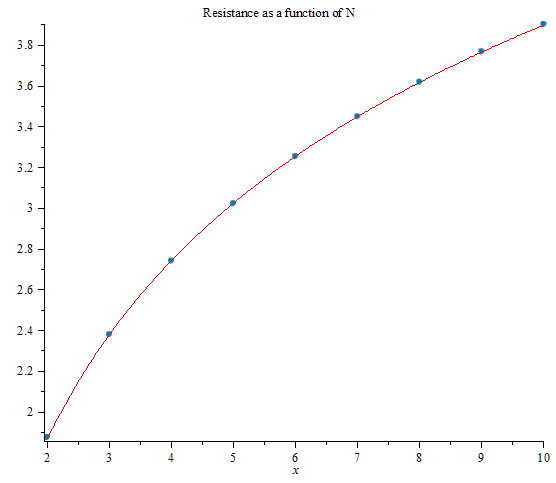
For low N=2, the “optimized” time is actually worse than the un-optimized. This is because of the additional time required in finding the left most non-zeros combined with the fact that that computing them gives us very little benefit as the matrix isn’t very sparse (the first non-zeros will be close to if not at the left edge). As N increases, the times diverge and it can be seen that the optimized time is increasing at a slower rate. But is it a lower order? Well, this is where things get a bit tricky. If we had a half-bandwidth that wasn’t varying with N (or was increasing much more slowly), the order would be about N^2, but the scheme we used to number the branches and nodes makes it so that far branch numbers become physically closer on the mesh with increasing N. This can be seen as the half-bandwidth is equal to 2\*N, so b is O(n) (I didn’t expect it to be that exact but I’m guessing there are some topological rules coming into play that I don’t know). Had we picked a better numbering scheme, we **may** have been able to decrease the order, or at least decrease the “weight” of the third order term. Based on my intuition and some reading I’ve done, I hypothesize that picking an ideal numbering scheme is NP-Hard so there wasn’t much that could be done besides some heuristic methods. In its current state, the complexity is O(b^2\*n) = O(n^3).



Fitted equation: 

It’s not a perfect fit, but I believe this is once again because of the counter productiveness of the optimization at low values of n. The curve fit is better when we remove the first 2 data points

1. Below is the plot of R vs N. It looks logarithmic, and a logarithmic curve fit works very well. I should note however that a sum or roots also worked well even though these two functions have very different asymptotic behavior. I suspect the log function is the correct one as there seems to be some kind of exponential thing going on with the amount of resistors in the mesh. Logarithmic equation: 



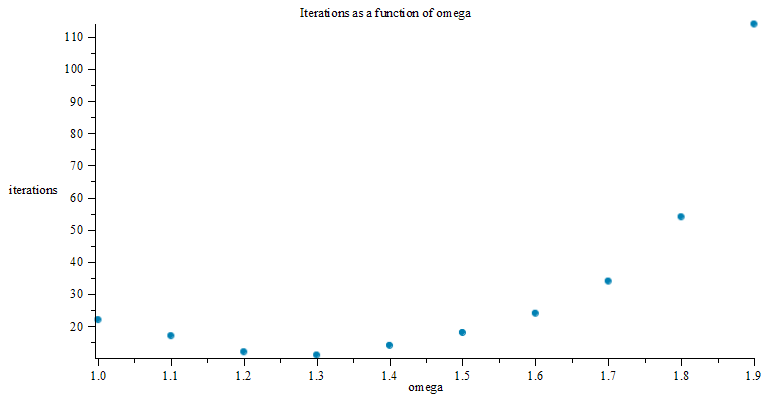
Question 3

The code for this is in folder “Q3” since none of it is dependent on the previous code. “FiniteDiffSovler.py” is the source file that solves the uniformly distributed mesh. I was thinking of sub-classing it to make the non-uniformly distributed mesh, but the relation isn’t parent-child; the uniformly distributed mesh is just a special case of the non-uniform one so I decided to just make a second source file called “StrongFiniteDiffSolver.py”. They then each have their appropriate testers.

1. All the code is in “FiniteDiffSolver.py”. I used BOTH planes of symmetry because I’m hardcore like that. The code is pretty straightforward. First, the mesh is generated (which includes setting the boundary values), then we iterate over it using the chosen technique (only SOR in part a), the residual is computed, rinse, wash, repeat until the residual is smaller than the max threshold.
2. Here is a table of iterations vs. omega:

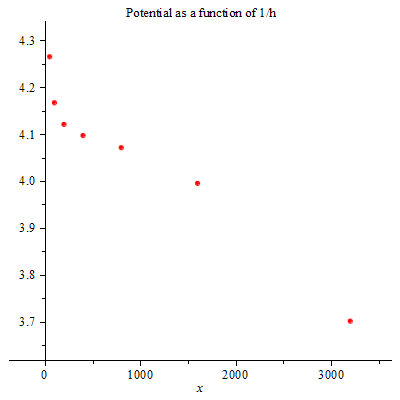
|  |  |  |
| --- | --- | --- |
| Omega | Iterations | Potential at (0.06, 0.04) |
| 1 | 22 | 4.264763 |
| 1.1 | 17 | 4.264777 |
| 1.2 | 12 | 4.264797 |
| 1.3 | 11 | 4.264819 |
| 1.4 | 14 | 4.264819 |
| 1.5 | 18 | 4.264829 |
| 1.6 | 24 | 4.264807 |
| 1.7 | 34 | 4.264813 |
| 1.8 | 54 | 4.264813 |
| 1.9 | 114 | 4.264794 |

We can see that the sweet spot seems to be around 1.3. I redid the simulation with a higher omega resolution around that point and found that a good omega was 1.24

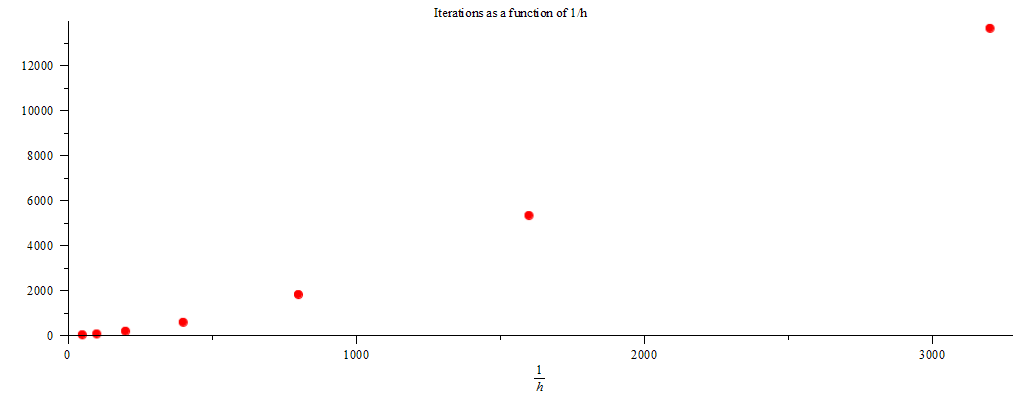


1. Following are the table and plot:

|  |  |  |
| --- | --- | --- |
| 1/h | potential | iterations |
| 50 | 4.264825 | 9 |
| 100 | 4.166708 | 47 |
| 200 | 4.120471 | 167 |
| 400 | 4.096795 | 564 |
| 800 | 4.07089 | 1797 |
| 1600 | 3.994811 | 5313 |
| 3200 | 3.700902 | 13641 |



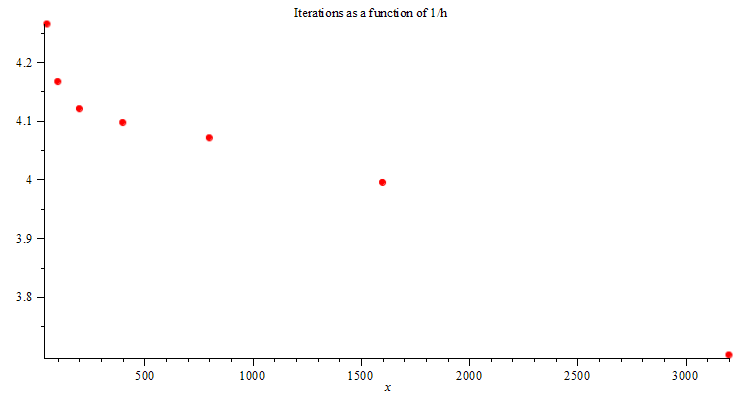
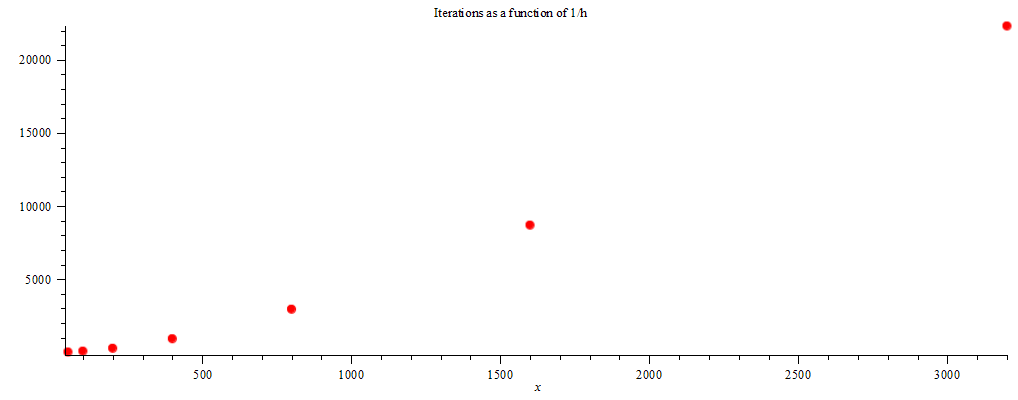
It doesn’t really look like is asymptotically converges which is strange. It’s harder to get more data points because of the order of complexity of the problem (takes too long to compute). I hypothesis that is has something to do with the residual not being small enough for smaller values of h. I shrank h, but I still ran into the issue of not having enough data points. My guess is like 3.56V but there really shouldn’t be that many significant figures. I can’t really do better than that with this data though.

This graph is a little clearer. This equation fits the graph very well: 

We can see that the third order term is very small, but it starts to take effect near the end since 1/h becomes large. For the range we worked in however, the equation is very linear.

1. The code for this one was actually a little more complex than the SOR code even though it is less efficient. Since the algorithm isn’t done “in place” meaning you write to the same grid that you read from, I had to make a copy of the grid at each iteration to “buffer” it. Here are the table and plots.

|  |  |  |
| --- | --- | --- |
| 1/h | iterations | potential |
| 50 | 22 | 4.264763 |
| 100 | 80 | 4.166646 |
| 200 | 277 | 4.120426 |
| 400 | 927 | 4.096895 |
| 800 | 2947 | 4.070893 |
| 1600 | 8696 | 3.994927 |
| 3200 | 22310 | 3.701151 |

 Both plots look similar to their SOR counterparts. The main difference is that the number of iterations for Jacobi was higher. The potential was more or less the same.

1. As explained above, this is in file “StrongDiffSolver.py”. By playing with the node spacing, I was able to compute a potential 3.11 V using the same amount of nodes. This is closer to the value than the uniform node spacing using the same number of nodes: 4.1667 V. The way this program work is by taking as a parameter a set of horizontal and vertical lines which then intersect to make the mesh nodes.