# ECSE 543 Assignemnt1

## **Question 1**

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

Cholesky decomposition has been implemented using the algorithm from lecture. An input file called matrix.txt has been used to pass matrix A and vector b in the format that each row is separated by a new line and each element in the row is separated by comma. The last line in the input file is taken as the b vector. Five methods have been developed to solve such a matrix problem: *read\_matrix(), check\_symmetric(A), check\_pd(A)*, and *chelosky(A, b).* These four methods are integrated together in *solve\_matrix().* Check Appendix for the implementation and sample input file.

### **(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.**

### **(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.**

Part (b) and (c) are explained together below.

Five tests were made. The first three tests were designed to validate the algorithm. The last two tests were used to check the error handling ability of the script.

1. Test 1:

A is SPD, can be used to test result correctness. (Determinant of A is positive)

A = 2, -1, -1, 1

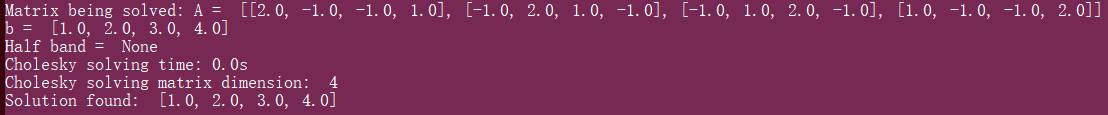
-1, 2, 1, -1  
 -1, 1, 2, -1  
 1, -1, -1, 2

Set x = [1, 2, 3, 4]

b = A\*x = [1, 2, 3, 4]

Set up matrix.txt file.

Script output: Solution found: [1.0, 2.0, 3.0, 4.0]



1. Test 2:

A is SPD. (Determinant of A is 29.0)

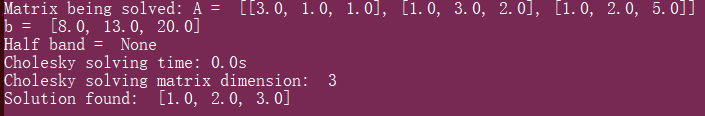
A = 3,1,1  
 1,3,2  
 1,2,5

Set x = [1,2,3]

b = A\*x = [8, 13, 20]

Set up matrix.txt file.

Script output: Solution found: [1.0, 2.0, 3.0]



1. Test 3:

A is PSD. (Determinant of A is 3.0)

A = 2,1

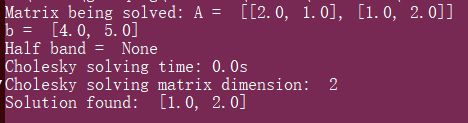
1,2

Set x = [1,2] (transpose)

b = A\*x = [4, 5]

Set up matrix.txt file.

Script output: Solution found: [1.0, 2.0]



1. Test 4:

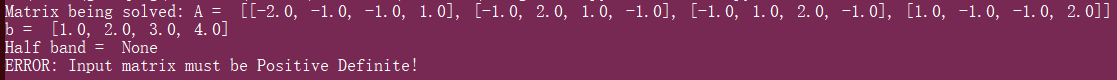
Error detection. Change the first element of A in Test 1 to -2, which makes A’s determinant negative (-11.0)

A = -2, -1, -1, 1

-1, 2, 1, -1  
 -1, 1, 2, -1  
 1, -1, -1, 2

b = [1, 2, 3, 4]

Script output: ERROR: Input matrix must be Positive Definite!



1. Test 5:

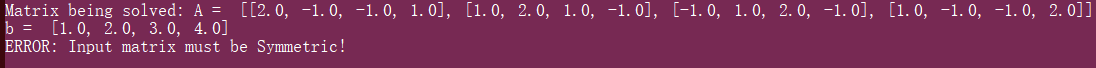
Error detection. Change the first element in row 2 of A in Test 1 to 1, which makes A not symmetric.

A = 2, -1, -1, 1

1, 2, 1, -1  
 -1, 1, 2, -1  
 1, -1, -1, 2

b = [1, 2, 3, 4]

Script output: ERROR: Input matrix must be Symmetric!



### **(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. Cleary specify each of the test circuits used with a labeled schematic diagram.**

Another input file called *networkBranch.txt* is used including four matrices/vectors: circuit relation matrix A, branch model vectors J, R, and E. The last three lines of the input correspond to J, R, E respectively.

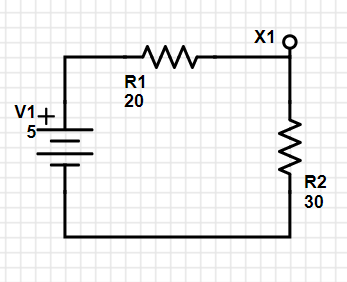
Several methods were integrated in method *solve\_network()*, including, first, *read\_network()* which returns A, J, R, and E matrices. Then based on the formula derived in class: *(A \* y \* AT) \* v = A \* (J - y \* E),* the two inputs required by *cholesky(A, b)* are calculated where *A = (A \* y \* AT), b = A \* (J - y \* E).* Before feeding A and b into cholesky(), the validation of matrix A has been checked by calling *check\_symmetric(A)* and *check\_pd(A).*

A few test cases can be seen below.

1. Test 1

This circuit has 2 branches and 1 node.

V1 = V / (R1+R2) \* R2 = 3V



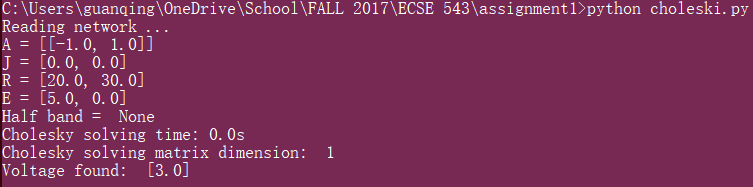
A = [-1, 1]

J = [0, 0]

R = [20, 30]

E = [5, 0]

Script output: *Voltage found: [3.0]*

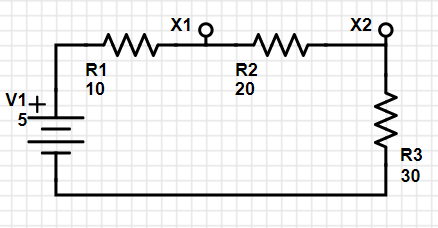


1. Test 2

This circuit has 3 branches and 2 nodes.

V1 = V / (R1+R2+R3) \* (R2+R3) = 4.17V

V2 = V / (R1+R2+R3) \* R3 = 2.5V



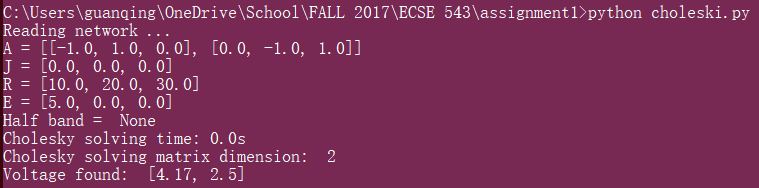
A = [[-1, 1, 0], [0, -1, 1]]

J = [0, 0, 0]

R = [10, 20, 30]

E = [5, 0, 0]

Script output: *Voltage found: [4.17, 2.5]*



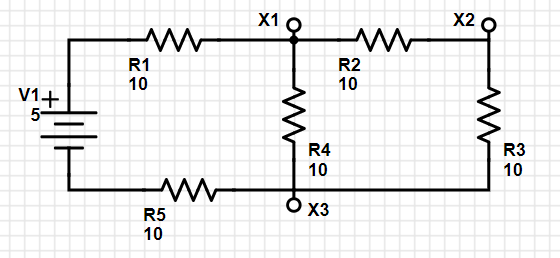
1. Test 3

This circuit has 5 branches and 3 nodes.

V1 = V / (R1+(R2+R3)//R4+R5) \* ((R2+R3)//R4+R5) = 3.12V

V2 = V3 + (V1-V3)/2 = 2.5V

V3 = V / (R1+(R2+R3)//R4+R5) \* R5 = 1.87V



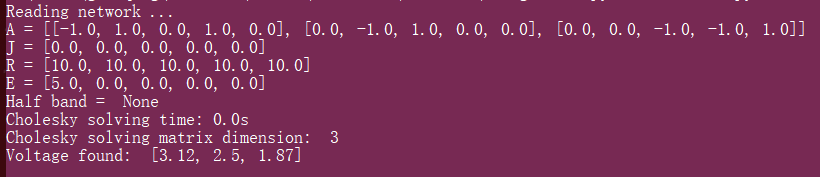
A = [[-1, 1, 0, 1, 0], [0, -1, 1, 0, 0], [0, 0, -1, -1, 1]]

J = [0, 0, 0, 0 ,0]

R = [10, 10, 10, 10, 10]

E = [5, 0, 0, 0, 0]

Script output: *Voltage found: [3.12, 2.5, 1.87]*

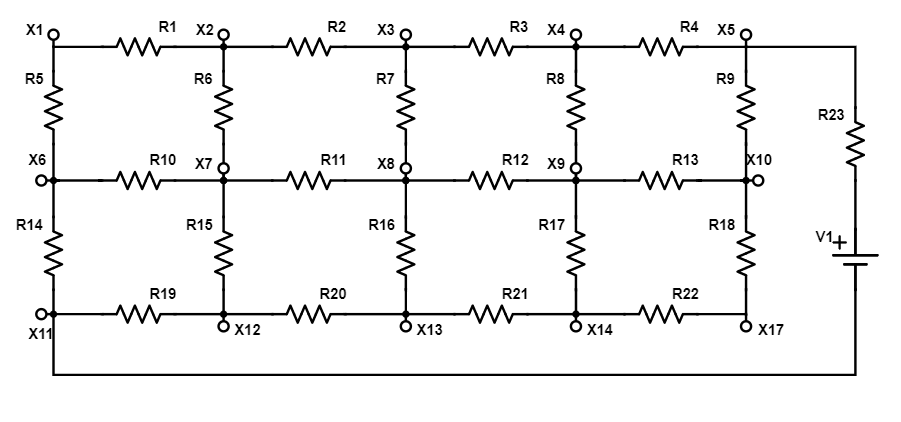


## **Question 2**

## **Take a regular N by 2N finite-difference mesh and replace each horizontal and vertical line by 1 k ohm 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).**

2N=4



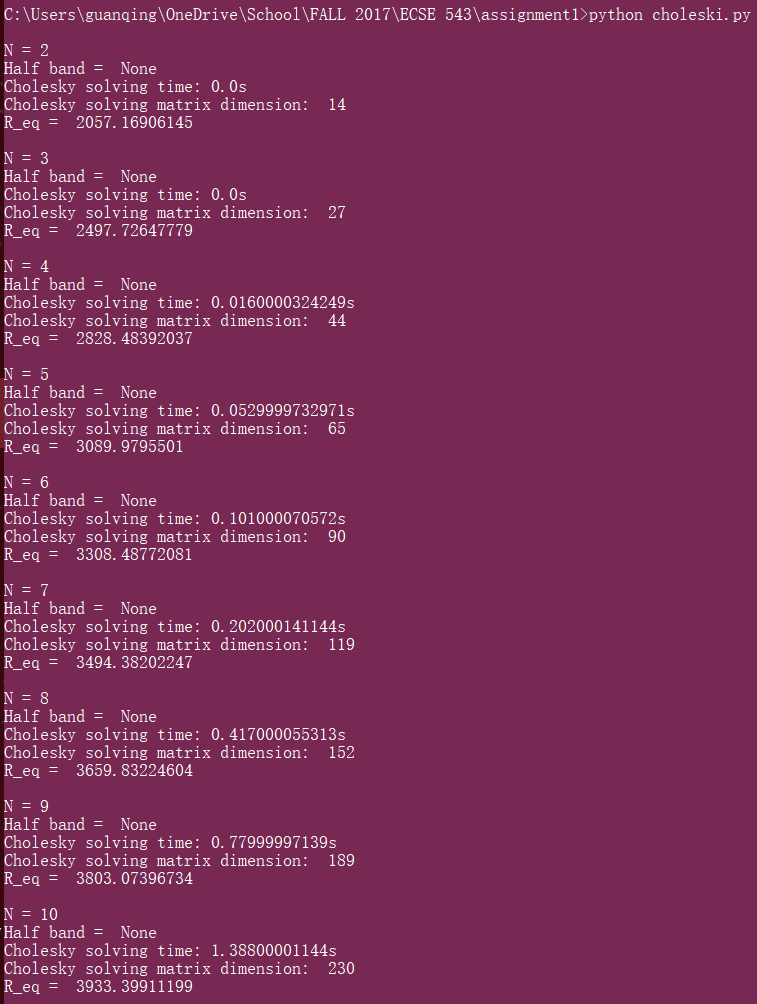
N=2

An example of N by 2N finite-difference mesh can be seen above where N=2. N is used as the number of branches, not number of nodes.

The resistance of this mesh network can be found by plugging a voltage source across the two selected nodes. A new method *generate\_network(N, R, sJ, sR, sE)* has been developed to generate A, J ,R ,E matrices required by *solve\_network().* The parameter sJ, sR, and sE are the branch descriptions for the source branch. Both voltage and current source can be used to find the equivalent resistance, but using current source is harder due to parallel resistance. Therefore, a voltage source has been used as an example.

The schematic of N = 2 can be seen above. After solving all the node voltages, the current through R23 can be found with I = ((VX11+5)-VX5/R23). Then the mesh equivalent resistance can be found by Req = (VX1 - VX5)/I. The results for N = 2, 3 … 10 were calculated using the same idea and shown in the table below, followed by a screenshot of the outputs.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Req (Ω) | 2057.17 | 2497.73 | 2828.48 | 3089.98 | 3308.49 | 3494.38 | 3659.83 | 3803.07 | 3933.40 |



### **(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.**

The time complexity to solve Cholesky Decomposition is O(n3). The parameter n refers to the dimension of the mesh matrix A. In this question, n refers to the number of nodes in the circuit which can be found by n = (N+1)(2N+1) = 2N2+3N+1. Therefore, the time complexity regarding to N is O((N2)3) = O(N6).

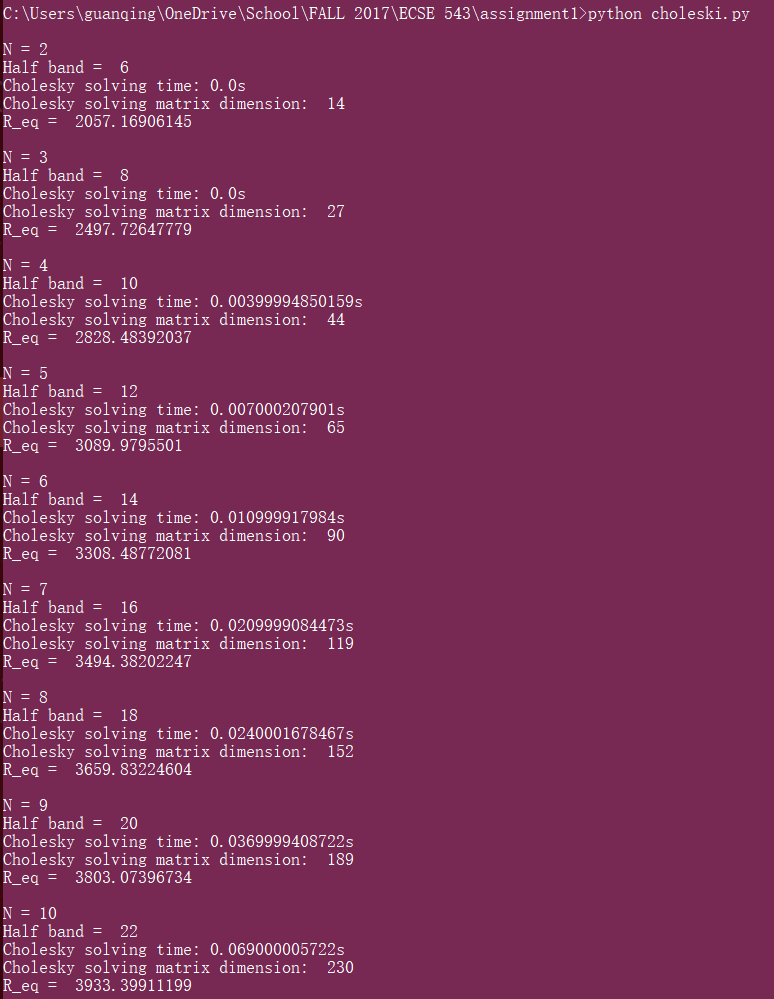
The time used to solve Cholesky Decomposition for each value of N is shown below, copied from the screenshot above.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Computing Time (ms) | 0 | 0 | 16 | 53 | 101 | 202 | 417 | 780 | 1388 |

Two line charts were plotted to present how N and N6 influence the computing time. The relation of computing time with N is exponential, while with N6 is almost linear. This observation is consistent with the time complexity O(N6) derived above.

### **(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 you for your practical sparse implementation consistent with this? Explain your observations.**

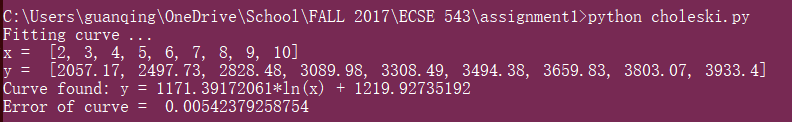
The half band has been found as 2N+2 by simply looking at the Cholesky matrices. The new computing time can be found by O(b2\*n) where both b2 and n are proportional to N2. So the computing time should have a linear relation with N4, instead of N6. A screenshot of outputs considering half band can be seen below, followed by a summary table, and two plots of computing time vs. N and N4.



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **9** | **10** |
| Computing Time (ms) | 0 | 0 | 4 | 7 | 11 | 21 | 24 | 37 | 69 |

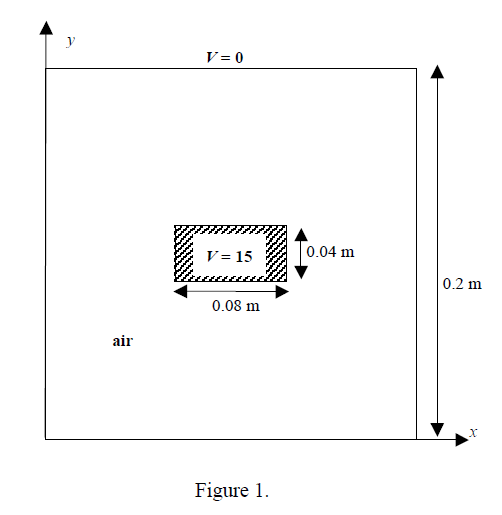
### **(d) Plot a graph of R versus N. Find a function R(N) that fits the curve reasonably well and is asymptotically correct as N tends to infinity, as far as you can tell.**

A method called *fit\_curve(x,y)* has been added for this part of question using Python Numpy. The fitting function found is *R = 1171.39\*ln(N) + 1219.93* with an error under 0.54% (check screenshot below for more details).This result is consistent with the trending line result found with Excel, which can be seen in the chart below.



## **Question 3**

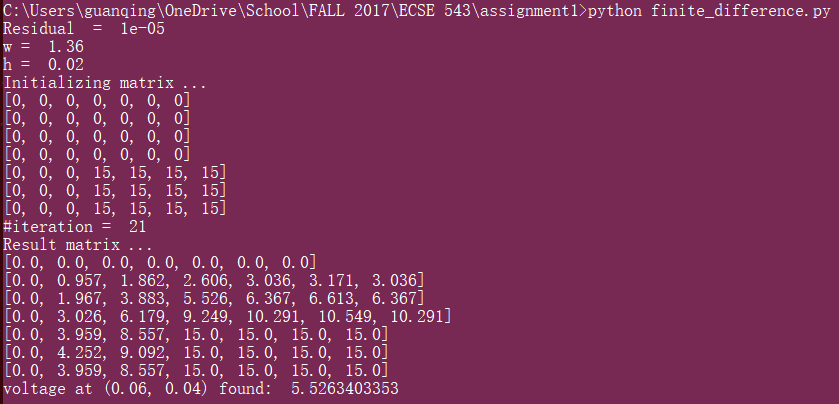
## **Figure 1 shows the cross-section of an electrostatic problem with translational symmetry: a coaxial cable with a square outer conductor and a rectangular inner conductor. The inner conductor is held at 15 volts and the outer conductor is grounded.**



### **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 w. Terminate the iteration when the magnitude of the residual at each free node is less than 10-5.**

Because of symmetry, only quarter of the cable needs to be modeled. An example is shown below with h = 0.02m, resulting in a regular 6-by-6 grid plus boundaries (7x7 matrix). There are totally 24 unknowns including 5 boundary nodes and 19 interior nodes. There are 25 fixed nodes with voltage potential equals to 0V or 15V.

There are several major methods in the program developed to solve this problem. First is *generate\_mesh()* method which analyzes width, height, fixed-node ranges, and fixed voltage values of the quarter cable and initializes a matrix expression of all the nodes in the mesh based on the value of node-spacing (h). The matrix below is an example of the output matrix with h = 0.02m. The nodes highlighted in blues are the unknowns and the nodes highlighted in yellow are the boundary nodes. Nodes circled in red are the ones locate on the symmetric plane.



[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, 15, 15, 15, 15],

[0, 0, 0, 15, 15, 15, 15],

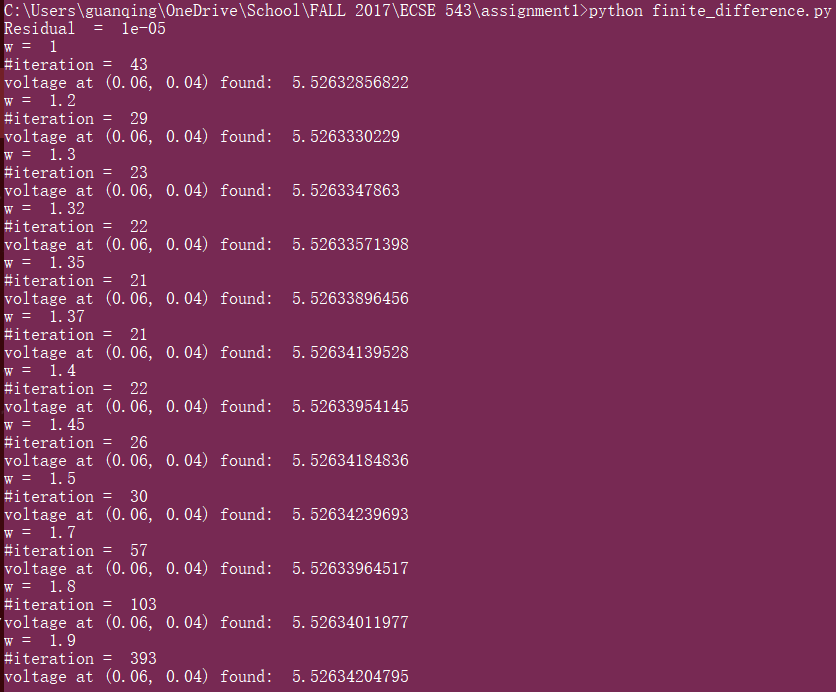
[0, 0, 0, 15, 15, 15, 15]

After the grid matrix has been initialized, a method called *update\_matrix(n, unknown\_range, boundary\_range)* is called where the matrix gets updated node by node using five-point difference formula and SOR-Gauss Method.

### **(b) With h = 0.02, explore the effect of varying w. For 10 values of w 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 number of iterations versus w.**

Some values of w between 1 and 2 have been tested. Points between 1.32 and 1.4 give the least number of iterations needed to converge. The average was taken as the estimated best choice for w, which is 1.36.

An interesting find is that the sequence of updating the unknowns influences number of iterations required to converge. The data above got from updating points from the top-left most unknown in the matric. Another way is starting the update from the bottom-right most unknown, which will take less iterations to converge because the algorithm is able to skip a lot of zero updates.

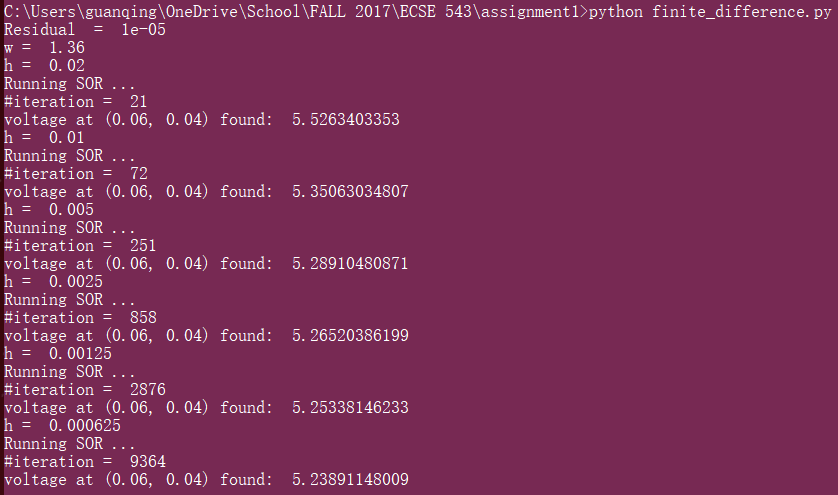


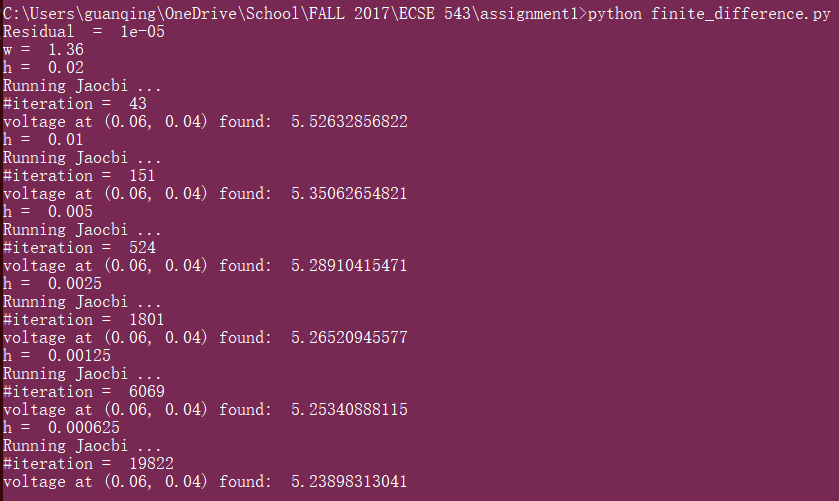
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| w | 1 | 1.2 | 1.3 | 1.32 | 1.35 | 1.37 | 1.4 | 1.45 | 1.5 | 1.7 | 1.8 | 1.9 |
| #iteration | 43 | 29 | 23 | 22 | 21 | 21 | 22 | 26 | 30 | 57 | 103 | 393 |
| V(0.06, 0.04) | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 | 5.526 |

### **(c) With an appropriate value of w, 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.**

### **(d) Use the Jacobi method to solve this problem for the same values of h used in part (c). Tabulate and plot the values of the potential at (x, y) = (0.06, 0.04) versus 1/h and the number of iterations versus 1/h. Comment on the properties of both plots and compare to those of SOR.**

Since (c) and (d) have very similar logic. The results of them are explained together. Screenshots of outputs using SOR and Jacobi can be seen first, followed by a summary table and plots.





|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| h | | 0.02 | 0.01 | 0.005 | 0.0025 | 0.00125 | 0.000625 |
| 1/h | | 50 | 100 | 200 | 400 | 800 | 1600 |
| SOR | #iteration | 21 | 72 | 251 | 858 | 2876 | 9364 |
| Potential (V) | 5.526 | 5.351 | 5.289 | 5.265 | 5.253 | 5.239 |
| Jacobi | #iteration | 43 | 151 | 524 | 1801 | 6069 | 19822 |
| Potential (V) | 5.526 | 5.351 | 5.289 | 5.265 | 5.253 | 5.239 |

From the graph of potential at point (0.06, 0.04) vs. 1/h, it can be seen that, for both SOR and Jacobi method, the value is approaching 5.24V dramatically at the beginning, but smoother after 1/h = 400. The precision using either method is the same. The potential values corresponding to 1/h = 1600 is the most accurate one, giving 5.239V.

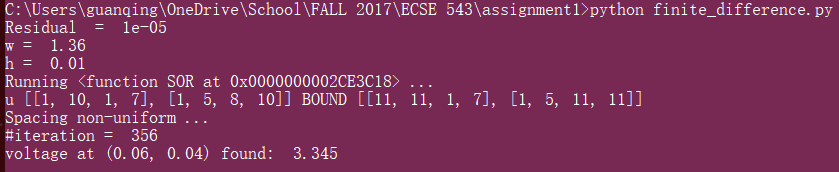
Number of iterations increases quadratic with 1/h in both cases. But obviously, SOR method needs much less number of iterations than Jacobi does to reach the same precision. In the data points that have been tested, the number of iterations required by SOR is about half of that required by Jacobi and the difference is going to be even more with increasing 1/h.

As a conclusion, SOR is a more efficient method than Jacobi and its advantage in computing efficiency becomes larger when the size of the problem increases.

### **(e) Modify the program you wrote in part (a) to use the five-point difference formula derived in class for non-uniform node spacing. An alternative to using equal node spacing, h, is to use smaller node spacing in more “difficult” parts of the problem domain. Experiment with a scheme of this kind and see how accurately you can compute the value of the potential at (x, y) = (0.06, 0.04) using only as many nodes as for the uniform case h = 0.01 in part (c).**

When non-uniformed spacing is used, the five-point difference formula needs to be generalized. A flag called *even\_spacing* was added to the SOR method. When flag values is False, SOR enters non-uniform spacing mode, where two extra arrays were used to specify horizontal and vertical node positions.

With non-uniform spacing and the same amount of nodes, slightly more accurate result can be found with more iterations as tradeoff.



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
| 4/3 | 4/3 | 4/3 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | 1.5 |  |
|  |  |  |  |  |  |  |  |  | 4/3 |  |
|  |  |  |  |  |  |  |  |  | 4/3 |  |
|  |  |  |  |  |  |  |  |  | 4/3 |  |

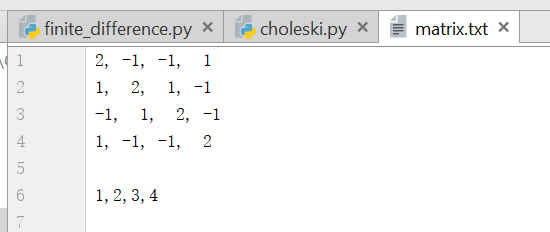
## Appendix

### Q1 & Q2

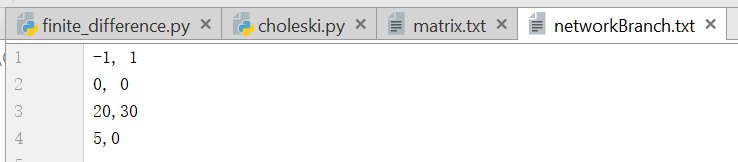
### Cholesky.py

*"""  
This script solves Ax=b matrix problem using Cholesky Decomposition, where A is represented by L\*transpose(L)  
Author: Guanqing Hu, Oct 3rd, 2017  
"""***import** math  
**import** numpy **as** np  
**import** time  
  
*# check S.P.D***def** check\_symmetric(A):  
 n = len(A)  
 **for** i **in** range(n):  
 **for** j **in** range(i + 1, n):  
 **if** A[i][j] != A[j][i]:  
 exit(**'ERROR: Input matrix must be Symmetric!'**)  
  
  
*# positive definite if determinant of A is positive  
# determinant equals to the product of all the eigenvalues of A***def** check\_pd(A):  
 det = np.linalg.det(A)  
 **print** det  
 **if** det <= 0:  
 exit(**'ERROR: Input matrix must be Positive Definite!'**)  
  
**def** read\_matrix():  
 *# read matrix, last line is b* **with** open(**'matrix.txt'**, **'r'**) **as** f:  
 A = []  
 **for** line **in** f:  
 **if** line.strip() == **''**:  
 **continue** A.append([float(x) **for** x **in** line.split(**','**)])  
 *# check if data complete  
 #* ***todo*** b = A[-1]  
 A = A[:-1]  
 **print 'Matrix being solved: A = '**, A, **'\nb = '**, b  
 **return** A,b  
  
  
**def** cholesky(A, b, hb=None):  
 *# Cholesky implementation* n = len(A)  
 **print 'Half band = '**, hb  
 start\_time = time.time()  
 **for** j **in** range(n - 1):  
 **if** A[j][j] <= 0:  
 exit(**'ERROR: Input matrix must be Positive Definite!'**)  
 A[j][j] = math.sqrt(A[j][j])  
 b[j] = b[j] / A[j][j]  
  
 i\_range = range(j + 1, n)  
 **if** (hb!=None):  
 **if** ((j+hb+1)<n):  
 i\_range = range(j+1, (j+hb+1))  
  
 **for** i **in** i\_range: *#range(j+1, n):  
 #print 'updating element A{}{}'.format(i,j)* A[i][j] = A[i][j] / A[j][j]  
 b[i] = b[i] - A[i][j] \* b[j]  
 **for** k **in** range(j + 1, i + 1):  
 A[i][k] = A[i][k] - A[i][j] \* A[k][j]  
 *# Back Substitution: T(L)x = y* x = [0.0 **for** i **in** range(n)]  
 **for** i **in** range(n - 1, -1, -1):  
 x[i] = round((b[i] - sum([A[j][i] \* x[j] **for** j **in** range(i + 1, n)])) / A[i][i], 2)  
 solving\_time = time.time()-start\_time  
 **print 'Cholesky solving time: {}s'**.format(solving\_time)  
 **print 'Cholesky solving matrix dimension: '**, len(A)  
  
 **return** x  
  
**def** solve\_matrix():  
 A, b = read\_matrix()  
 check\_symmetric(A)  
 *#check\_pd(A)* results = cholesky(A, b, None)  
 **print 'Solution found: '**, results  
  
**def** read\_network():  
 **with** open(**'networkBranch.txt'**, **'r'**) **as** f:  
 A = []  
 **for** line **in** f:  
 **if** line.strip() == **''**:  
 **continue** A.append([float(x) **for** x **in** line.split(**','**)])  
 J, R, E = A[-3], A[-2], A[-1]  
 A = A[:-3]  
 **return** A, J, R, E  
  
  
**def** m\_product(A, B):  
 *# return A\*B  
 # check dimension* **if** len(A[0]) != len(B):  
 exit(**'ERROR: Matrix production failed due to wrong dimensions!'**)  
 *# create correct dimension output matrix  
 #print 'product', A, B* C = [[0 **for** col **in** B[0]] **for** row **in** A]  
 **for** i **in** range(len(C)):  
 **for** j **in** range(len(C[0])):  
 C[i][j] = sum([A[i][k] \* B[k][j] **for** k **in** range(len(B))])  
 **return** C  
  
  
**def** transpose(A):  
 A2 = [[0 **for** i **in** A] **for** j **in** A[0]]  
 **for** i **in** range(len(A)):  
 **for** j **in** range(len(A[0])):  
 A2[j][i] = A[i][j]  
 **return** A2  
  
**def** m\_substract(A, B):  
 *# check dimension* **if** (len(A)!=len(B))|(len(A[0])!=len(B[0])):  
 exit(**'ERROR: Matrix dimension error in substraction!'**)  
 C = []  
 **for** i **in** range(len(A)):  
 C.append([A[i][k]-B[i][k] **for** k **in** range(len(A[0]))])  
 **return** C  
  
  
*# this method generates network for a regular N by 2N finite-difference mesh and replace each horizontal and vertical  
# line by a 1 kW resistor***def** generate\_network(N, R, sJ, sR, sE):  
 *# calculate number of branches* nbranch = (N+1)\*2\*N+N\*(2\*N+1)  
 J = [0]\*nbranch  
 R = [R]\*nbranch  
 E = [0]\*nbranch  
 *# add teasting branch: voltage source branch* J.append(sJ)  
 R.append(sR)  
 E.append(sE)  
 *# calculate nodes* nnode = (N+1)\*(2\*N+1)  
 A = [[0]\*(nbranch+1) **for** i **in** range(nnode)]  
 *#print len(A), len(A[0])* **for** i **in** range(1, nnode+1):  
 level = i/(2\*N+1)  
 offset = i%(2\*N+1)  
 **if** offset==0:  
 level -= 1  
 offset = 2\*N+1  
 branch\_per\_level = (2\*N+(2\*N+1))  
 *# calculate surrounding branch indices* right = branch\_per\_level \* level + offset  
 left = right - 1  
 top = 2\*N\*level + (2\*N+1)\*(level-1)+ offset  
 bottom = top + branch\_per\_level  
 *#print i, 'r',right, 'l', left, 't',top, 'b', bottom* i -= 1  
 *# node on top border* **if** (level==0):  
 *# top left* **if** (offset==1):  
 A[i][right - 1], A[i][bottom - 1] = -1, 1  
 *# top right* **elif** (offset==2\*N+1):  
 A[i][left - 1], A[i][bottom - 1] = 1, 1  
 **else**:  
 A[i][right - 1], A[i][left - 1], A[i][bottom - 1] = -1, 1, 1  
 *# node on bottom border* **elif** (level == N):  
 *# bottom left* **if** (offset==1):  
 A[i][right - 1], A[i][top - 1] = -1, -1  
 *# bottom right* **elif** (offset==2\*N+1):  
 A[i][top - 1], A[i][left - 1] = -1, 1  
 **else**:  
 A[i][right - 1], A[i][top - 1], A[i][left - 1] = -1, -1, 1  
 *# node on left border* **elif** (offset == 1):  
 A[i][right - 1], A[i][top - 1], A[i][bottom - 1] = -1, -1, 1  
 *# node on right border* **elif** (offset == 2\*N+1):  
 A[i][top - 1], A[i][left - 1], A[i][bottom - 1] = -1, 1, 1  
 *# nodes in middle* **else**:  
 A[i][right - 1], A[i][top - 1], A[i][left - 1], A[i][bottom - 1] = -1, -1, 1, 1  
  
 *# setup the voltage source  
 # right top corner* **if** (level==0)&(offset==2\*N+1):  
 A[i][-1] = -1  
 **elif** (level==N)&(offset==1):  
 A[i][-1] = 1  
 *# set up ground node by removing that node from A  
 # ground left-bottom corner node* **del** A[-(2\*N+1)]  
  
 **return** A, J, R, E  
  
  
**def** solve\_network(A, J, R, E, hb=None):  
 *# (A \* y \* T(A)) \* v = A \* (J - y \* E)* y = [[0 **for** r **in** R] **for** r **in** R]  
 **for** i **in** range(len(y)):  
 y[i][i] = 1.0/R[i]  
  
 A2 = m\_product(m\_product(A, y), transpose(A))  
 check\_symmetric(A2)  
 *#check\_pd(A2)* **'''print '### start of matrix'  
 for i in A2:  
 print i  
 '''** b = m\_product(A, m\_substract([[j] **for** j **in** J], m\_product(y, [[e] **for** e **in** E])))  
 *# example of b: [[0.5], [0.0]], flat it* b = [i[0] **for** i **in** b]  
 *#print 'b', b, 'A2', A2* v = cholesky(A2, b, hb)  
 **return** v  
  
**def** fit\_curve(x,y):  
 **print 'Fitting curve ... '  
 print 'x = '**,x  
 **print 'y = '**,y  
 x = np.array(x)  
 y = np.array(y)  
 k, b = np.polyfit(np.log(x), y, 1)  
 **print 'Curve found: y = {}\*ln(x) + {}'**.format(k,b)  
 r = []  
 **for** n **in** x:  
 r.append(k \* math.log(n, math.e) + b)  
 error = max([(a0-a)/a **for** (a0,a) **in** zip(r, y)])  
 **print 'Error of curve = '**, error  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 *### q1* solve\_matrix()  
  
 *### q1* A, J, R, E = read\_network()  
 **print 'Reading network ...\nA = {}\nJ = {}\nR = {}\nE = {}'**.format(A,J,R,E)  
 v = solve\_network(A, J, R, E)  
 **print 'Voltage found: '**,v  
  
 *### q2  
 # J, R ,E of the source branch* sJ = 0  
 sR = 1000.0  
 sE = 100.0  
 **for** N **in** range(2,2):  
 **print '\nN = {}'**.format(N)  
 A, J, R, E = generate\_network(N, 1000, sJ, sR, sE) *#N, R, sJ, sR, sE  
 # half band, not sure ...  
 #hb = 2\*N+2* v = solve\_network(A, J, R, E)  
 *# get the voltage at top right and bottom left* v1 = v[2\*N]  
 v2 = 0*#v[-1-2\*N]  
 # R\_total = sR+R\_mesh* I = ((v2+sE)-v1)/sR  
 R\_mesh = (v1-v2)/I  
 *#print 'Voltage of {} nodes found. '.format(len(v))#, v* **print 'R\_eq = '**, R\_mesh  
  
 *# fit curve* N = [2, 3, 4, 5, 6, 7, 8, 9, 10]  
 R = [2057.17, 2497.73, 2828.48, 3089.98, 3308.49, 3494.38, 3659.83, 3803.07, 3933.4]  
 fit\_curve(N, R)

### Matrix.txt



### networkBranch.txt



## Q3

**import** math  
  
**def** initial\_matrix(r, n, value):  
 i\_min, i\_max, j\_min, j\_max = r  
 **for** j **in** range(j\_min, j\_max + 1):  
 **for** i **in** range(i\_min, i\_max + 1):  
 n[j][i] = value  
 **return** n  
  
**def** generate\_meshnodes():  
 *# find matrix dimension, +h : add boundary* j = height/h  
 i = width/h  
 *#check if h qualifies* **if** (j!=int(j))|(i!=int(i)):  
 **print 'ERROR: h should be selected so that width/d and hight/h are both integers.'** exit()  
 *# build matrix:* n = [[None **for** x **in** range(int(i+1))] **for** y **in** range(int(j+1))]  
 *# decide boundary, unknown, and fixed nodes* ground\_range = []  
 **for** area **in** ground:  
 *# [i\_min i\_max, j\_min, j\_max]* ground\_range.append([int(area[0][0]/h), int(area[1][0]/h), int(area[0][1]/h), int(area[1][1]/h)])  
 **for** r **in** ground\_range:  
 n = initial\_matrix(r, n, 0)  
 v1\_range = []  
 **for** area **in** v1:  
 v1\_range.append([int(area[0][0]/h), int(area[1][0]/h), int(area[0][1]/h), int(area[1][1]/h)])  
 **for** r **in** v1\_range:  
 n = initial\_matrix(r, n, v1\_value)  
  
 *#****todo make it more dynamic*** *# [i\_min i\_max, j\_min, j\_max]* unknowns = [[1, len(n)-2, 1, v1\_range[0][2]-1], [1,v1\_range[0][0]-1,v1\_range[0][2],v1\_range[0][3]-1]]  
 boundary = [[len(n[0])-1, len(n[0])-1, 1, v1\_range[0][2]-1],[1,v1\_range[0][0]-1, len(n)-1, len(n)-1]]  
 **for** r **in** unknowns+boundary:  
 n = initial\_matrix(r, n, 0)  
 *#print ground\_range, v1\_range, unknowns, boundary* **return** n, unknowns, boundary  
  
**def** error\_small\_enough(n,unknowns, thr):  
 error = []  
 **for** r **in** unknowns:  
 **for** j **in** range(r[2],r[3]+1):  
 **for** i **in** range(r[0],r[1]+1):  
 error.append(n[j][i-1]+n[j][i+1]+n[j-1][i]+n[j+1][i]-4\*n[j][i])  
 **if** max(error)>thr:  
 **print** max(error)  
 **return** False  
 **return** True  
  
**def** SOR(unknown, boundary, n, even\_spacing=True, horizontal\_split=[], vertical\_split=[]):  
 **if** even\_spacing:  
 **for** j **in** range(unknown[2],unknown[3]+1):  
 **for** i **in** range(unknown[0],unknown[1]+1):  
 n[j][i] = (1-w)\*n[j][i] + w\*0.25\*(n[j][i-1]+n[j][i+1]+n[j-1][i]+n[j+1][i])  
 *# check orientation of boundary* **if** boundary[2]==boundary[3]:  
 n[boundary[2]][i] = n[boundary[2]-2][i]  
 *# check orientation of boundary* **if** boundary[0]==boundary[1]:  
 n[j][boundary[0]] = n[j][boundary[0]-2]  
 **else**:  
 **for** j **in** range(unknown[2],unknown[3]+1):  
 **for** i **in** range(unknown[0],unknown[1]+1):  
 a1 = horizontal\_split[i] - horizontal\_split[i-1]  
 a2 = horizontal\_split[i+1] - horizontal\_split[i]  
 b1 = vertical\_split[j] - vertical\_split[j-1]  
 b2 = vertical\_split[j+1] - vertical\_split[j]  
 e = (math.pow(a1,2)+math.pow(a2,2))  
 f = (math.pow(b1,2)+math.pow(b2,2))  
 c1 = f/2/(e+f)  
 c2 = e/2/(e+f)  
 *#print 'c1,c2 = ',c1,c2* n[j][i] = (1-w)\*n[j][i] + w\*(c1\*(n[j][i-1]+n[j][i+1]) + c2\*(n[j-1][i]+n[j+1][i]))  
 *# check orientation of boundary* **if** boundary[2]==boundary[3]:  
 n[boundary[2]][i] = n[boundary[2]-2][i]  
 *# check orientation of boundary* **if** boundary[0]==boundary[1]:  
 n[j][boundary[0]] = n[j][boundary[0]-2]  
 *#print '###',i,j,a,b,c,d,coeff1,coeff2* **return** n  
  
**def** Jacobi(r,b,n\_old):  
 n\_new = [i **for** i **in** n\_old]  
 **for** j **in** range(r[2],r[3]+1):  
 **for** i **in** range(r[0],r[1]+1):  
 n\_new[j][i] = 0.25\*(n\_old[j][i-1]+n\_old[j][i+1]+n\_old[j-1][i]+n\_old[j+1][i])  
 *# check orientation of boundary: if horizontal* **if** b[2]==b[3]:  
 n\_new[b[2]][i] = n\_old[b[2]-2][i]  
 *# check orientation of boundary: if vertical* **if** b[0]==b[1]:  
 n\_new[j][b[0]] = n\_old[j][b[0]-2]  
 **return** n\_new  
 **if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 thr = math.pow(10, -5)  
 **print 'Residual = '**,thr  
  
 *# width or hight / h should both be integer* **for** w **in** [1]:*#[1, 1.2, 1.3, 1.32, 1.35, 1.37, 1.4, 1.45, 1.5, 1.7, 1.8, 1.9]:* **print 'w = '**, w  
 **for** h **in** [0.01]:*#[0.02, 0.01, 0.005, 0.0025, 0.00125, 0.000625]:* **print 'h = '**, h  
 *# specify cable dimension in meter* width = 0.1 + h  
 height = 0.1 + h  
 *# specify fixed voltage area* ground = [[(0, 0), (0, 0.1 + h)], [(0, 0), (0.1 + h, 0)]]  
 v1 = [[(0.06, 0.08), (0.1 + h, 0.1 + h)]]  
 v1\_value = 15  
  
 target = [0.06, 0.04]  
 target\_pos = [int(x / h) **for** x **in** target]  
  
 methodToRun = SOR  
 **print 'Running {} ... '**.format(methodToRun)  
 *#run(method, h, even\_spacing)* n, unknowns, boundarys = generate\_meshnodes()  
 **print 'u'**, unknowns, **'BOUND'**, boundarys  
  
 even\_spacing = True  
 horizontal\_split = [0,0.001,0.002,0.003,0.004,0.005,0.006,0.007,0.008,0.009,0.01,0.011]  
 vertical\_split = [0,0.001,0.002,0.003,0.004,0.005,0.006,0.007,0.008,0.009,0.01,0.011]  
 i = 0  
 **while not** error\_small\_enough(n, unknowns, thr):  
 i += 1  
 *#n = update\_matrix(n, unknowns, boundary, methodToRun)* **for** unknown, boundary **in** zip(unknowns, boundarys):  
 n = methodToRun(unknown, boundary, n, even\_spacing=even\_spacing, horizontal\_split=horizontal\_split, vertical\_split=vertical\_split)  
 *#print n[target\_pos[1]][target\_pos[0]]  
 #for row in n:  
 # print row* **print '#iteration = '**, i  
 *# print 'Result matrix ... '  
 # for i in n:  
 # print [round(x,3) for x in i]* **print 'voltage at (0.06, 0.04) found: '**, n[target\_pos[1]][target\_pos[0]]