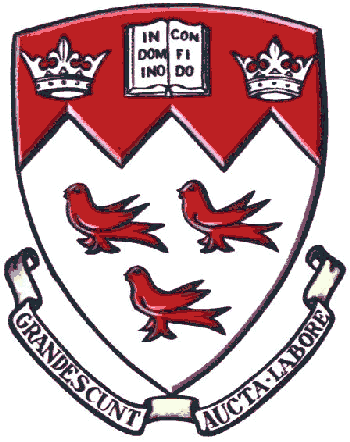
****

ECSE 543 – Numerical Methods

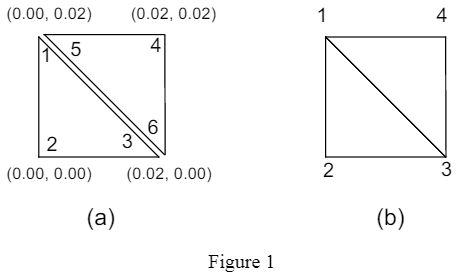
Assignment 2

Sharhad Bashar

260519664

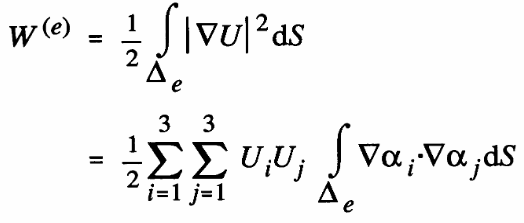
Nov 14th, 2016

**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.**

**Local S-Matrices:**

Energy is each triangle is given by the following equation (taken from lecture slides):



From the above equation, we can see that

Using the example provided in the lecture FE1stOrder, a general form for S is formed:

We also need the Area of each of the triangles:

1. Triangle 1,2,3:

2

1

3

Node 1:

Node 2:

Node 3:

Calculation of alphas (values in cm):

Using the above equation (which matches the general form above), the following local S – matrix was generated:

1. Triangle (4,5,6):

4

5

6

Node 4:

Node 5:

Node 6:

Calculation of alphas (values in cm):

Using the above equation (which matches the general form above), the following local S – matrix was generated:

**Global Matrix:**

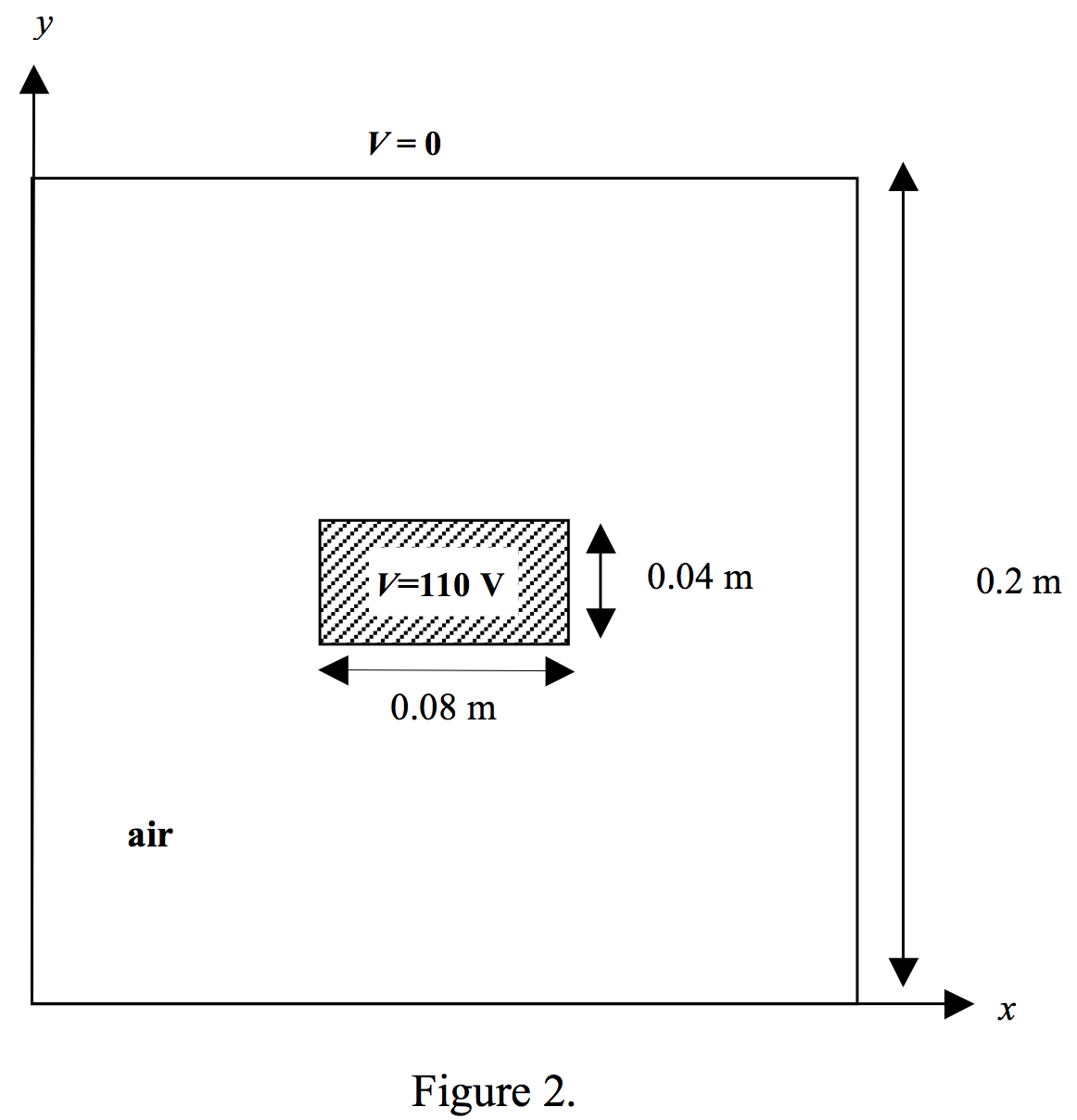
In order to find the Global Matrix, we use the following equation:

Where C is a conjoint numbering matrix and Sdis is a 6x6 matrix created by combining the two local S-matrices in the following manner:

Since each triangle has 3 nodes each separately, and when conjoint, there are 4 nodes (nodes 1 and 5 are common, and nodes 3 and 6 are common), C will be a 6x4 matrix:

And Sdis, a combination of the two local S-matrices, will be 6x6:

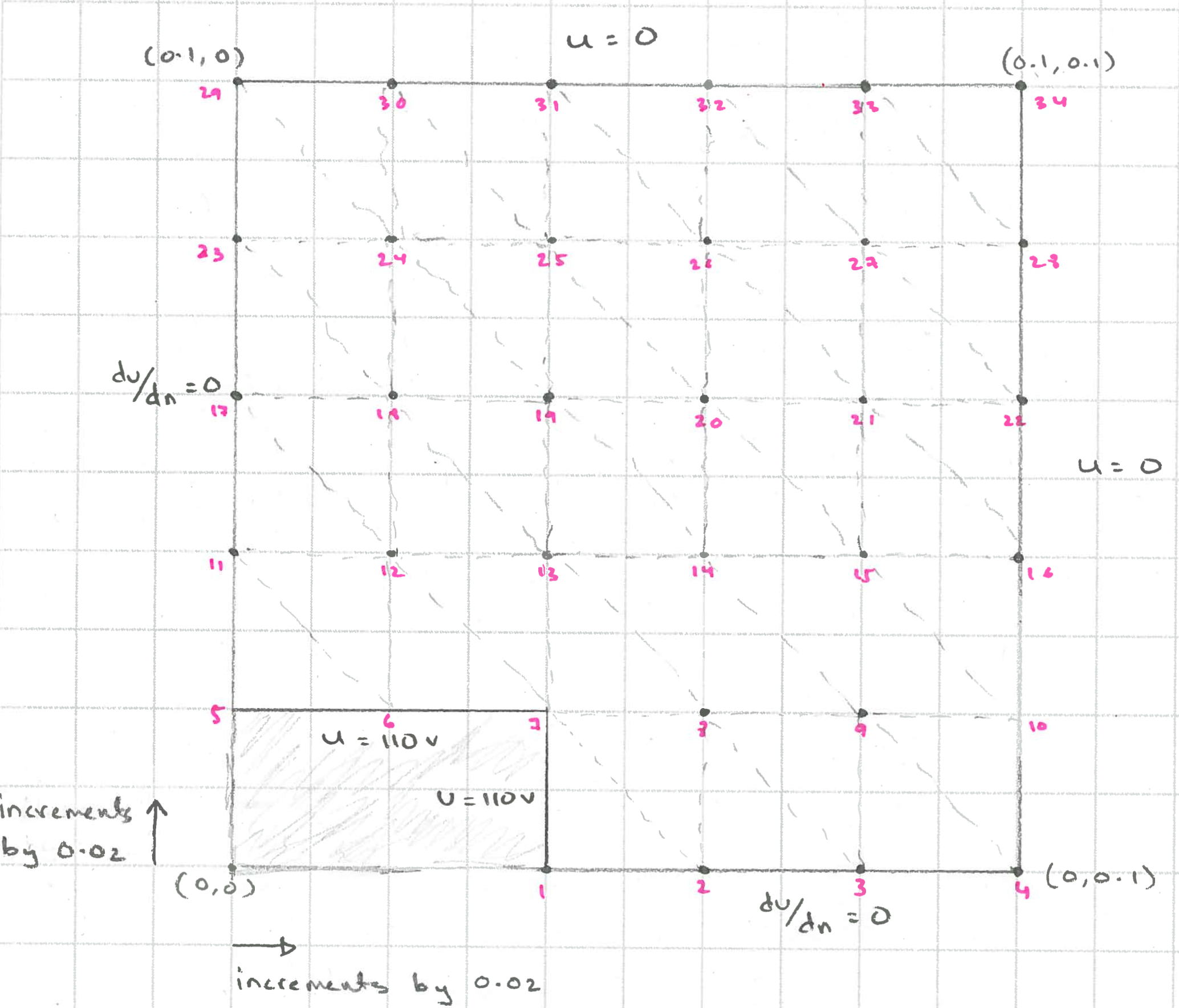
**Question 2**

**Figure 2 shows the cross-section of an electrostatic problem with translational symmetry: a rectangular coaxial cable. The inner conductor is held at 110 volts and the outer conductor is grounded.**

1. **Use the two-element mesh shown in Figure 1(b) as a “building block” to construct a finite element meshes 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.**

Using the example provided in page 20 of the FE1stOrder lecture, the python code genFile4Simple2D.py was created to produce the input file for the simple2D program. This code generated the file fileForSimple2D.txt, and it indeed had 46 elements. Both the code and the input file are included in the appendix.

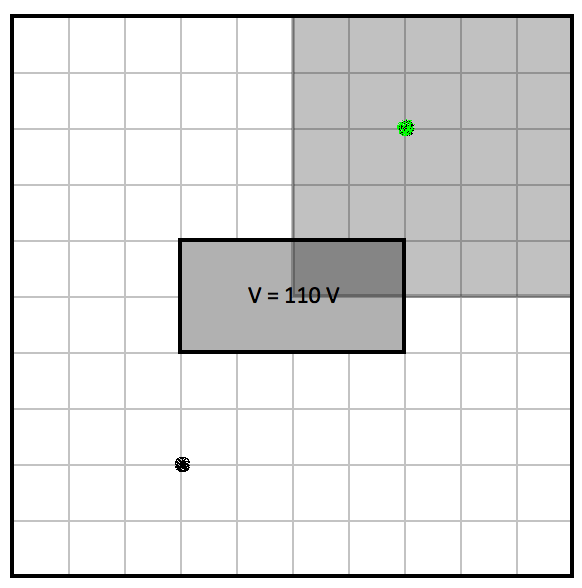
The top right quadrant was used cross-section of the coaxial cable was used. The following picture shows the node numbering of the triangles:

****

1. **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 input file was fed into the Matlab version of the Simple2D program, and it generated an output file, similar to the one described in page 22 of the FE1stOrder lecture. The output file is included in the Appendix. The diagram below shows the cable. We were asked to compute the potential at (x, y) = (0.06,0.04). This point is marked in the black dot. Since the top right quadrant (grey are in the diagram) was used, the equivalent potential is located at (x, y) = (1.04,1.06) (if we only consider the quadrant, (x, y) = (0.04,0.06)) or node 19, which is marked in the green dot:

(0.06,0.04)



(1.04,1.06)

The potential at (x,y) = (0.06,0.04) was found to be **40.5265V**

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

Energy is a capacitor is given by:

Which can also be written as follows:

Energy in a potential field is given by:

Where S is from Question 1, Ucon is a 1-D matrix with potential values from the four corners of the square from Figure 1(b), as shown below:

U1

U2

U3

U4

Ucon = [U1, U2, U3, U4]

Equating the two energy equations from above, gives us the capacitance per length, (confirmed by [1]):

The file findCapacitance.py, calculates the capacitance per length, using the equation given above. Matrix multiplication and transpose operations were done using functions that were created in the basicDefinitions.py from Assignment 1 (both included in Appendix).

The Capacitance per length of the cable is: **5.2137e-11 F/m**

**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 considered in Question 2 above. Use a starting solution of zero.**

The code for conjugate difference method is written in conjGradMethod.py. The Choleski decomposition is the same as the previous assignment, and can be found in basicDefinitions.py

The mesh is created similarly as the one from assignment 1, in q\_3Functions.py.

The A and b matrices were also generated in conjGradMethod.py. There were a total of 19 free nodes, so A is a 19 x 19 matrix, and b is a column vector with 19 elements.

A was computed as outlined in the lecture: FD\_LectureNotes.

Generating A:

4 nodes, as shown below, surround each free node:

-4

1

1

1

1

-4

1

2

1

2

Current node is the middle node (in yellow). A[currentNode][currentNode] = -4.0

Node Up, Node Down, Node Left, Node Right corresponds to the neighbouring nodes. The figure on the left shows free nodes are do not neighbour any bordering nodes. For these,

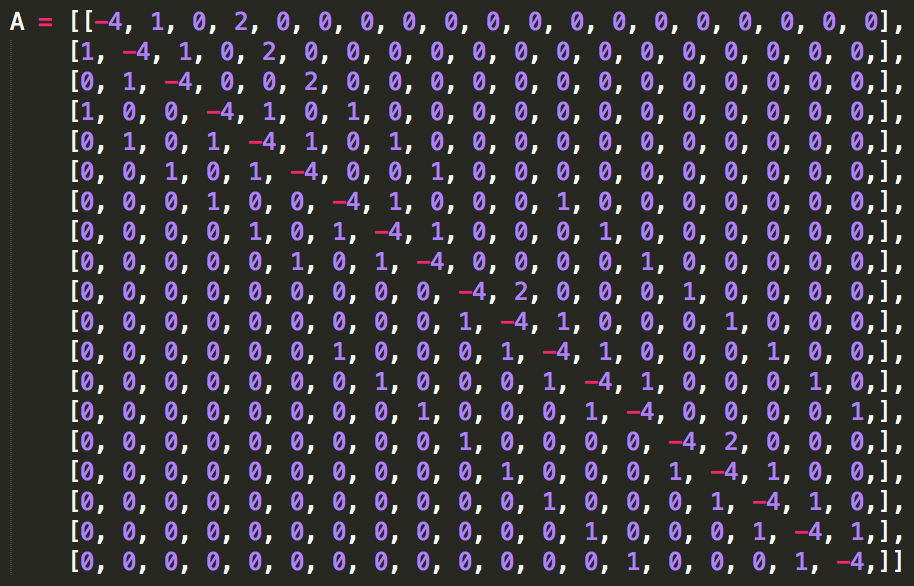
A[currentNode][nodeUp] = A[currentNode][nodeDown] = A[currentNode][nodeLeft] = A[currentNode][nodeRight] = 1.0

For the nodes that do neighbour bordering nodes (below and to the right only), refer to the diagram on the right. For these, A[currentNode][nodeUp] and A[currentNode][nodeRight] are 2.0, where applicable.

The b coloumn vector has a voltage value of -110.0 for the nodes that border the inner cable, and zero otherwise.

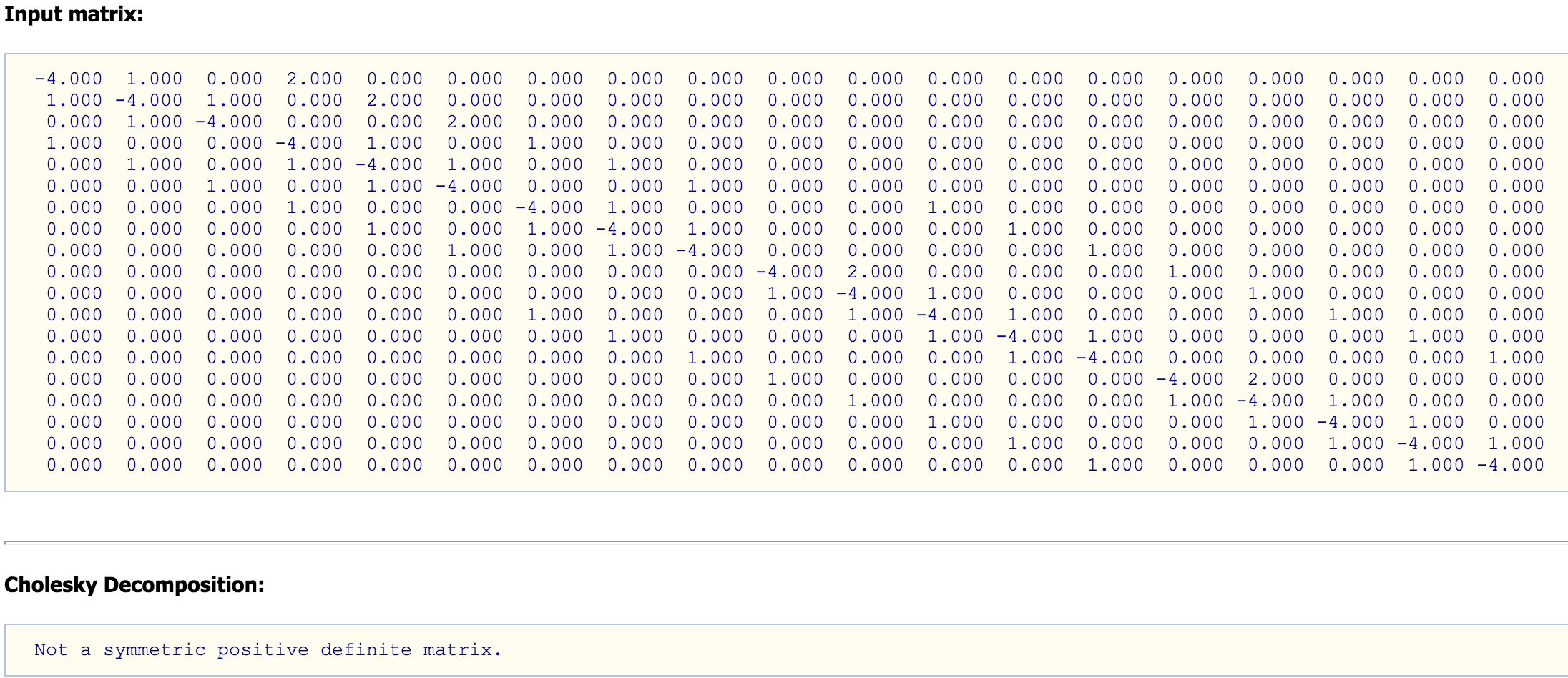
All in all, the A and b matricies are shown below:

Macintosh HD:Users:Sharhad:Desktop:Screen Shot 2016-11-14 at 10.04.00 PM.png



1. **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.**

The Matrix A is not Positive definite. This is because of the shared boundaries on both the x and y-axis. A visual check and an online decomposition was used to double check to make sure it indeed isn’t positive definite, as shown below:



One way to rectify this issue is to multiply both A and b by A^T, transforming the equation to:

This creates a positive definite matrix, which can be decomposed

1. **Once you have modified the problem 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.**

The code for both the decomposition and the Conjugate Gradient Method can be found in conjGradMethod.py. Multiplication and addition functions are in basicDefinitions.py.

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

The norm values are calculated in conjGradMethod.py:

|  |  |  |  |
| --- | --- | --- | --- |
| Iteration | 2-norm | Iteration | Infinity norm |
| 0 | 555.1319626 | 0 | 325.8732212 |
| 1 | 343.0812586 | 1 | 165.2642705 |
| 2 | 236.703743 | 2 | 103.465448 |
| 3 | 187.1606422 | 3 | 90.15753512 |
| 4 | 159.2824468 | 4 | 67.53607946 |
| 5 | 120.256896 | 5 | 64.62750605 |
| 6 | 110.1450255 | 6 | 83.36937733 |
| 7 | 131.7943728 | 7 | 58.57329838 |
| 8 | 113.3462218 | 8 | 67.17612169 |
| 9 | 93.30339828 | 9 | 50.07314807 |
| 10 | 80.07526259 | 10 | 28.55090197 |
| 11 | 69.76736927 | 11 | 32.57103069 |
| 12 | 33.7521252 | 12 | 15.21885024 |
| 13 | 19.89542491 | 13 | 9.18304894 |
| 14 | 22.75210701 | 14 | 11.78157676 |
| 15 | 18.51914185 | 15 | 7.903589879 |
| 16 | 5.653268685 | 16 | 2.47321541 |
| 17 | 0.153755093 | 17 | 0.065569253 |
| 18 | 1.24E-05 | 18 | 5.14E-06 |

The graphs are shown below:

1. **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.**

After using the decomposition to solve the equation, the voltage value we received is: **40.5265V.**

Running the conjugate gradient program, we get the same value: **40.5265V.**

This value matches the values calculated in question 2 of this assignment above, and that from Assignment 1.

All the potential values for the nodes for both Choleski and Conjugate Gradient method have been included in the Appendix.

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

After we compute A and b using finite difference method, we can compute the potential at each node using either Choleski or Conjugate Gradient. And once we have the voltages at each of the node, we can simply use the same equation used in question 2 in the findCapacitance.py to calculate the capacitance per length.

# References

1. Whiteman, J. R. (1973). The mathematics of finite elements and applications: Proceedings of the Brunel University conference of the Institute of Mathematics and Its Applications held in April 1972. London: Academic Press.

# Appendix

#!/usr/bin/env python2

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

260519664

ECSE 543

Assignment 2

**genFile4Simple2D.py**

Creates the input file for the Simple2d\_M program

"""

output = open("fileForSimple2D.txt", 'w') #opens the file

#All code below will go in here

#Also has print statements, so we can see in the IDE if proper numbers were printed

horNodes = 6

verNodes = 5

pot = 110.0

botBound = [(x+1) for x in range (horNodes - 2)]

mesh = [[((x + 5) + y \* horNodes) for x in range(horNodes)] for y in range(verNodes)]

#########################################################################################

# First part of input

# Node -- xVal -- yVal

for i in range (horNodes - 2):

print("{:<3} {:>6} {:>6}\n".format(botBound[i], (i + 2) \* 0.02, i \* 0.00))

output.write("{:<3} {:>6} {:>6}\n".format(botBound[i], (i + 2) \* 0.02, i \* 0.00))

for y in range (verNodes):

for x in range (horNodes):

print("{:<3} {:>6} {:>6}\n".format(mesh[y][x], x \* 0.02, (y + 1) \* 0.02))

output.write("{:<3} {:>6} {:>6}\n".format(mesh[y][x], x \* 0.02, (y + 1) \* 0.02))

#########################################################################################

print("\n")

output.write("\n")

#########################################################################################

#Second part of the input

# Node1 -- Node2 -- Node3 -- 0.000

for i in range (horNodes - 3):

print("{:<3} {:<3} {:<3} {}\n".format(botBound[i],botBound[i + 1],mesh[0][i + 2],'0.000'))

print("{:<3} {:<3} {:<3} {}\n".format(botBound[i + 1],mesh[0][i + 3],mesh[0][i + 2],'0.000'))

output.write("{:<3} {:<3} {:<3} {}\n".format(botBound[i],botBound[i + 1],mesh[0][i + 2],'0.000'))

output.write("{:<3} {:<3} {:<3} {}\n".format(botBound[i + 1],mesh[0][i + 3],mesh[0][i + 2],'0.000'))

for y in range (verNodes - 1):

for x in range (horNodes - 1):

print("{:<3} {:<3} {:<3} {:>6}\n".format(mesh[y][x],mesh[y][x + 1],mesh[y + 1][x],'0.000'))

print("{:<3} {:<3} {:<3} {:>6}\n".format(mesh[y][x + 1],mesh[y + 1][x + 1],mesh[y + 1][x],'0.000'))

output.write("{:<3} {:<3} {:<3} {:>6}\n".format(mesh[y][x],mesh[y][x + 1],mesh[y + 1][x],'0.000'))

output.write("{:<3} {:<3} {:<3} {:>6}\n".format(mesh[y][x + 1],mesh[y + 1][x + 1],mesh[y + 1][x],'0.000'))

########################################################################################

print("\n")

output.write("\n")

#########################################################################################

#Third part of the input

#Boundary Conditions

#Boundary Node -- Boundary Conditions

#Dirchlet Condition:

print ("{:<3} {}\n".format(1,pot))

output.write ("{:<3} {}\n".format(1,pot))

for i in range (horNodes):

if (mesh[0][i] < 8):

print ("{:<3} {}\n".format(mesh[0][i],pot))

output.write ("{:<3} {}\n".format(mesh[0][i],pot))

for i in range (horNodes):

print ("{:<3} {}\n".format(mesh[verNodes-1][i],'0.000'))

output.write ("{:<3} {}\n".format(mesh[verNodes-1][i],'0.000'))

for i in range (verNodes - 1):

print ("{:<3} {}\n".format(mesh[i][horNodes - 1],'0.000'))

output.write ("{:<3} {}\n".format(mesh[i][horNodes - 1],'0.000'))

#####################################################################################

output.close() #closes the file

**fileForSimple2D.txt**

**Input 1**

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

**Input 2**

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

**Input 3**

1 110.0

5 110.0

6 110.0

7 110.0

29 0.000

30 0.000

31 0.000

32 0.000

33 0.000

34 0.000

10 0.000

16 0.000

22 0.000

28 0.000

4 0.000

**Outfile.txt**

Potential =

1.0000 0.0400 0 110.0000

2.0000 0.0600 0 66.6737

3.0000 0.0800 0 31.1849

4.0000 0.1000 0 0

5.0000 0 0.0200 110.0000

6.0000 0.0200 0.0200 110.0000

7.0000 0.0400 0.0200 110.0000

8.0000 0.0600 0.0200 62.7550

9.0000 0.0800 0.0200 29.0330

10.0000 0.1000 0.0200 0

11.0000 0 0.0400 77.3592

12.0000 0.0200 0.0400 75.4690

13.0000 0.0400 0.0400 67.8272

14.0000 0.0600 0.0400 45.3132

15.0000 0.0800 0.0400 22.1921

16.0000 0.1000 0.0400 0

17.0000 0 0.0600 48.4989

18.0000 0.0200 0.0600 46.6897

**19.0000 0.0400 0.0600 40.5265**

20.0000 0.0600 0.0600 28.4785

21.0000 0.0800 0.0600 14.4223

22.0000 0.1000 0.0600 0

23.0000 0 0.0800 23.2569

24.0000 0.0200 0.0800 22.2643

25.0000 0.0400 0.0800 19.1107

26.0000 0.0600 0.0800 13.6519

27.0000 0.0800 0.0800 7.0186

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

#!/usr/bin/env python2

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

260519664

ECSE 543

Assignment 2

**findCapacitance.py**

Computes the Capacitance per legnth of the cable

"""

#Function Imports

import numpy as np #numpy, for reading the output file

from basicDefinitions import matTranspose, matrixMult #multiplication and transpose functions

#variable declaration

totalEnergy = 0.0

nodeHeight = 6

nodeWidth = 6

pot = 110.0

UCon = [0 for x in range (4)]

#S, which was calculated in Q1

S = [[1.0,-0.5,0.0,-0.5],

[-0.5,1.0,-0.5,0.0],

[0.0,-0.5,1.0,-0.5],

[-0.5,0.0,-0.5,1.0]]

###############################################################################

#Create the Mesh of voltages from the output file of Simple2d\_m

mesh = [[0 for x in range (nodeHeight)] for y in range(nodeWidth)]

values = np.loadtxt('outfile.txt', dtype = float, skiprows = 3, unpack= False)

for i in range (len(values)):

xNode = int(values[i][1]/0.02)

yNode = int(values[i][2]/0.02)

mesh[yNode][xNode] = values[i][3]

#Voltage of the inner conductor

mesh[0][0] = pot

mesh[0][1] = pot

###############################################################################

#Capacitance per lenght calculator

#c = eplison0\*|divU|^2/V^2

#Total energy: |divU|^2 = Ucon^T \* S \* Ucon:

for y in range (nodeHeight - 1):

for x in range(nodeWidth - 1):

UCon[0] = mesh[y][x]

UCon[1] = mesh[y][x + 1]

UCon[2] = mesh[y + 1][x + 1]

UCon[3] = mesh[y + 1][x]

UConT = matTranspose(UCon)

potential = matrixMult(matrixMult(UCon,S),UConT)

totalEnergy += potential[0][0]

#Calculates the Capacitance per length

#multiplies by 4, to include all 4 quadrants

epsilon = 8.854187817620e-12

Vsquared = pot\*pot

capPerLen = totalEnergy\* epsilon \* 4 / Vsquared

print (str(capPerLen) + ' F/m')

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

260519664

ECSE 543

Assignment 2

**basicDefinition.py**

A list of functions, used for this assignment

Nov 14th, 2016

"""

import math

import numpy as np

import csv

#####################################################################################

#Function that checks if a matrix is 1D or 2D

def is1Dor2D (A):

while True:

try:

length = len(A[0]) #If true, A is 2D

return A

break

except TypeError: #else A is 1D

return [A]

break

#####################################################################################

#function that creates floats from lists

def list2float(A):

length = len(A)

floatA = [0 for x in range(length)]

for i in range (length):

numVal = ''

stringVal = list(str(A[i]))

for j in range (1,len(stringVal)-1,1):

numVal = numVal + stringVal[j]

floatVal = float(numVal)

floatA [i] = floatVal

return floatA

#####################################################################################

#Function that transposes a Matrix

def matTranspose(A):

A = is1Dor2D(A)

rowsA = len(A)

colsA = len(A[0])

C = [[0 for rows in range (rowsA)] for cols in range(colsA)]

for i in range (colsA):

for j in range (rowsA):

C[i][j] = A[j][i]

return C

#####################################################################################

#Function that adds or subtracts two matricies

def matrixAddorSub(A,B,operation):

A = is1Dor2D(A)

B = is1Dor2D(B)

rowsA = len(A)

colsA = len(A[0])

rowsB = len(B)

colsB = len(B[0])

if (rowsA == rowsB and colsA == colsB):

C = [[0 for row in range(colsA)] for col in range(rowsA)]

if (operation == 'a'):

for i in range (rowsA):

for j in range (colsA):

C[i][j] = A[i][j]+B[i][j]

elif (operation == 's'):

for i in range (rowsA):

for j in range (colsA):

C[i][j] = A[i][j]-B[i][j]

return C

#####################################################################################

#function for scalar matrix multiplication

def scalarMult(scalar, A):

newMat = [0 for x in range (len(A))]

for i in range (len(A)):

newMat[i] = scalar\*A[i][0]

return newMat

#####################################################################################

#Function that multiplies two matricies

def matrixMult (A, B):

A = is1Dor2D(A)

B = is1Dor2D(B)

rowsA = len(A)

colsA = len(A[0])

rowsB = len(B)

colsB = len(B[0])

if (rowsA == colsB or colsA == rowsB):

C = [[0 for row in range(colsB)] for col in range(rowsA)]

for i in range(rowsA):

for j in range(colsB):

for k in range(colsA):

# Create the result matrix

# Dimensions would be rows\_A x cols\_B

C[i][j] += A[i][k] \* B[k][j]

else:

print ("Cannot multiply the two matrices. Incorrect dimensions.")

return

return C

#####################################################################################

#Function that creates a diagonal matrix

def diogMat (A):

length = len(A)

floatA = [0 for x in range(length)]

for i in range (length):

numVal = ''

stringVal = list(str(A[i]))

for j in range (1,len(stringVal)-1,1):

numVal = numVal + stringVal[j]

floatVal = float(numVal)

floatA [i] = floatVal

diognalMatrix = [[0 for x in range(length)] for y in range(length)]

for i in range (length):

diognalMatrix[i][i] = 1/floatA[i]

return diognalMatrix

#####################################################################################

#Function that performs the Cholesky decomposition and returns L

def cholesky(A,length):

global sum

L = [[0 for x in range(length)] for y in range(length)]

for i in range(length):

for k in range(i + 1):

sum = 0

for j in range(k):

sum += L[i][j] \* L[k][j]

if (i == k):

L[i][k] = math.sqrt(abs(A[i][i] - sum))

else:

L[i][k] = (A[i][k] - sum) / L[k][k]

return L

#####################################################################################

#Function that solves y in Ly = b

def forwElim (L,b,length):

b = list2float(b)

global sum

y = [0 for x in range (length)]

for i in range (length):

sum = 0.00

for j in range (i):

sum += L[i][j] \* y[j]

y[i] = (b[i]-sum)/L[i][i]

return y

#####################################################################################

#Function that solves for x in L^Tx = y

def backSub (L,y,length):

global sum

X = [0 for x in range(length)]

for i in range (length - 1, -1, -1):

sum = 0

for j in range (i + 1, length, 1):

sum += L[j][i] \* X[j]

X[i] = (y[i] - sum) / L[i][i]

return X

#####################################################################################

#!/usr/bin/env python2

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

260519664

ECSE 543

Assignment 2

**conjGradMethod.py**

Calculates the conjugate gradient for the cable

Nov 14th, 2016

"""

import numpy as np

from basicDefinitions import matrixMult, matTranspose, cholesky, forwElim, backSub, matrixAddorSub, scalarMult

from q\_3Functions import genMesh

import math

def genAb(h,freeNode):

cableHeight = 0.1

cableWidth = 0.1

coreHeight = 0.02

coreWidth = 0.04

corePot = 110.0

mesh = genMesh(0.02)

nodeHeight = (int)(cableHeight/h + 1)

nodeWidth = (int)(cableWidth/h + 1)

coreHeightNode = (int)(coreHeight/h + 1)

codeWidthNode = (int)(coreWidth/h + 1)

freeNode = (nodeHeight \* nodeWidth) - (coreHeightNode \* codeWidthNode) - (nodeHeight + nodeWidth) + 1

A = [[0 for x in range (freeNode)] for y in range (freeNode)]

b = [0 for x in range(freeNode)]

for y in range (nodeHeight):

for x in range (nodeWidth):

if (mesh[y][x][0] != None):

cord = mesh[y][x][0]

cordRight = mesh[y][x + 1][0]

cordLeft = mesh[y][x - 1][0]

cordUp = mesh[y + 1][x][0]

cordDown = mesh[y - 1][x][0]

A[cord][cord] = -4.0

#print (cord, cordLeft, cordRight, cordUp, cordDown)

#right

if (cordRight != None):

A[cord][cordRight] = 1

if (cord == 0):

A[cord][cordRight] = 2

b[cord] = -corePot

#left

if (cordLeft != None):

A[cord][cordLeft] = 1

if (cord == 1):

A[cord][cordLeft] = 2

b[cord + 1] = -corePot

if (cordLeft == 4 or cordLeft == 9 or cordLeft == 14):

A[cord][cordLeft] = 2

#up

if (cordUp != None):

A[cord][cordUp] = 1

if (cord == 4 or cord == 9):

A[cord][cordUp] = 2

if (cord == 4):

b[cord] = -corePot

b[cord + 1] = -corePot

b[cord + 2] = -corePot

#down

if (cordDown != None):

A[cord][cordDown] = 1

if (cord == 14 or cord == 9):

A[cord][cordDown] = 2

if (cordDown == 0 or cordDown == 1):

A[cord][cordDown] = 2

return (A,b)

##################################################

#Choleski

def Cholesky (A, b):

AT = matTranspose(A)

b = matTranspose(b)

newA = matrixMult(AT,A)

newb = matrixMult(AT,b)

length = 19#len(newA)

L = cholesky(newA,length)

y = forwElim (L, newb, length)

X = backSub (L, y, length)

print ('Choleski: ')

print X

##################################################

#Conjugate Gradient

def conjGrad(A,b,freeNode):

AT = matTranspose(A)

b = matTranspose(b)

newA = matrixMult(AT,A)

newb = matrixMult(AT,b)

X = [0 for x in range (freeNode)]

X = matTranspose(X)

r = matrixAddorSub(newb, matrixMult(newA,X),'s')

p = r

for i in range(freeNode):

alpha = float(matrixMult(matTranspose(p),r)[0][0])/float((matrixMult(matrixMult(matTranspose(p),newA),p))[0][0])

X = matrixAddorSub(X, matTranspose(scalarMult(alpha,p)),'a')

r = matrixAddorSub(newb, matrixMult(newA,X),'s')

beta = (-1)\*float(matrixMult((matrixMult(matTranspose(p),newA)),r)[0][0])/float((matrixMult(matrixMult(matTranspose(p),newA),p))[0][0])

p = matrixAddorSub(r, matTranspose(scalarMult(beta,p)),'a')

#finding the norms

infNorm = 0

twoNorm = 0

for j in range (freeNode):

value = abs(r[j][0])

if (value > infNorm):

infNorm = value

twoNorm += r[j][0]\*\*2

twoNorm = math.sqrt(twoNorm)

print (i,twoNorm,infNorm)

print X

##################################################

cableHeight = 0.1

cableWidth = 0.1

coreHeight = 0.02

coreWidth = 0.04

corePot = 110.0

h = 0.02

nodeHeight = (int)(cableHeight/h + 1)

nodeWidth = (int)(cableWidth/h + 1)

coreHeightNode = (int)(coreHeight/h + 1)

codeWidthNode = (int)(coreWidth/h + 1)

freeNode = (nodeHeight \* nodeWidth) - (coreHeightNode \* codeWidthNode) - (nodeHeight + nodeWidth) + 1

#Creates A and b

(A,b) = genAb(0.02,freeNode)

#Cholesky(A,b)

conjGrad (A,b,freeNode)

**RESULT FROM CHOLESKI**

[77.35922364524424,

48.498858387154726,

23.256867476258822,

75.46901809691109,

46.68967121355792,

22.264305758940274,

67.82717752884203,

**40.52650261122562,**

19.110684345944378,

66.67372442302747,

62.75498087537059,

45.31318940723139,

28.47847735655821,

13.651929013611632,

31.184935941368614,

29.033009671223503,

22.192121868154786,

14.422288394164239,

7.018554351943966]

**RESULTS FROM CONJUGATE GRADIENT**

[77.3592236913273],

[48.49885832259556],

[23.256867521606033],

[75.46901802963987],

[46.68967130700212],

[22.264305693582298],

[67.82717758581617],

**[40.526502535612636],**

[19.110684397876476],

[66.67372440401857],

[62.75498090994626],

[45.31318935318992],

[28.478477416035485],

[13.651928974644253],

[31.18493595888895],

[29.033009641909214],

[22.192121902250346],

[14.422288360411832],

[7.01855437327436]]

# -\*- coding: utf-8 -\*-

"""

Sharhad Bashar

ECSE 543

Assignment 2

**q\_3Functions.py**

Functons for generating the matrix from Assign 1

Nov 14th, 2016

"""

#using the top right quater of the cable, due to symmetry

import math

#####################################################################################

#Generates the initial mesh, taking into considering the boundary conditions

def genMesh (h):

cableHeight = 0.1

cableWidth = 0.1

coreHeight = 0.02

coreWidth = 0.04

corePot = 110.0

nodeHeight = (int)(cableHeight/h + 1)

nodeWidth = (int)(cableWidth/h + 1)

#Create the mesh, with Dirchlet conditions

mesh = [[(None,corePot) if x <= coreWidth/h and y <= coreHeight/h \

else ((None,0.0) if x == nodeHeight - 1 or y == nodeWidth - 1 \

else (0,0.0)) for x in range(nodeWidth)] for y in range(nodeHeight)]

#update the mesh to take into account the Neuman conditions

nodeForA = 0

for y in range (nodeHeight):

for x in range(nodeWidth):

if (mesh[y][x][0] == 0):

mesh[y][x] = (nodeForA,0.0)

nodeForA += 1

return mesh

#####################################################################################

#The Equation that calculates SOR

def SOR(mesh,h,w):

cableHeight = 0.1

cableWidth = 0.1

coreHeight = 0.02

coreWidth = 0.04

nodeHeight = (int)(cableHeight/h + 1)

nodeWidth = (int)(cableWidth/h + 1)

for y in range (1,nodeHeight - 1):

for x in range (1,nodeWidth - 1):

if (x > (int)(coreWidth/h) or y > (int)(coreHeight/h)):

mesh[y][x] = (1 - w) \* mesh[y][x] + (w/4) \* (mesh[y][x-1] + mesh[y][x+1] + mesh[y-1][x] + mesh[y+1][x])

return mesh

#####################################################################################

#The Equation that calculates Jacobian

def jacobian(mesh,h):

cableHeight = 0.1

cableWidth = 0.1

coreHeight = 0.02

coreWidth = 0.04

nodeHeight = (int)(cableHeight/h + 1)

nodeWidth = (int)(cableWidth/h + 1)

for y in range (1,nodeHeight - 1):

for x in range (1,nodeWidth - 1):

if (x > (int)(coreWidth/h) or y > (int)(coreHeight/h)):

mesh[y][x] = (1/4) \* (mesh[y][x-1] + mesh[y][x+1] + mesh[y-1][x] + mesh[y+1][x])

return mesh

#####################################################################################

#Equation that computes the residue

def computeMaxRes(mesh,h):

cableHeight = 0.1

cableWidth = 0.1

coreHeight = 0.02

coreWidth = 0.04

nodeHeight = (int)(cableHeight/h + 1)

nodeWidth = (int)(cableWidth/h + 1)

maxRes = 0

for y in range(1, nodeHeight - 1):

for x in range(1, nodeWidth - 1):

if (x > coreWidth/h or y > coreHeight/h):

#calculate the residue of each free point

res = mesh[y][x-1] + mesh[y][x+1] + mesh[y-1][x] + mesh[y+1][x] - 4 \* mesh[y][x]

res = math.fabs(res)

if (res > maxRes):

#Updates variable with the biggest residue amongst the free point

maxRes = res

return maxRes

#####################################################################################

#Function that computes the number of iterations

def numIteration (initialMesh,h,w,method):

minRes = 0.0001

iteration = 1

if (method == 's'):

mesh = SOR(initialMesh,h,w)

while (computeMaxRes(mesh,h) >= minRes):

mesh = SOR(mesh,h,w)

iteration += 1

elif (method == 'j'):

mesh = jacobian(initialMesh,h)

while (computeMaxRes(mesh,h) >= minRes):

mesh = jacobian(mesh,h)

iteration += 1

print ('Number of iterations: '+ str(iteration))

return(mesh)

#####################################################################################

#Function that returns the potential at a free node

def getPot(mesh, x, y, h):

cableHeight = 0.1

cableWidth = 0.1

nodeHeight = (int)(cableHeight/h + 1)

nodeWidth = (int)(cableWidth/h + 1)

xNode = int(nodeWidth - x/h - 1)

yNode = int(nodeHeight - y/h - 1)

return mesh[yNode][xNode]

#####################################################################################