ECSE 543
Assignment 2

Sean Stappas
260639512

November 20$^{\text{th}}$, 2017

Contents

Introduction

The code for this assignment was created in Python 2.7 and can be seen in Appendix A. To perform the required tasks in this assignment, the `Matrix` class from Assignment 1 was used, with useful methods such as add, multiply, transpose, etc. This package can be seen in the `matrices.py` file shown in Listing 1. The only packages used that are not built-in are those for creating the plots for this report, i.e., `matplotlib` for plotting. The structure of the rest of the code will be discussed as appropriate for each question. Output logs of the program are provided in Appendix B. The `SIMPLE2D` input and output files can be seen in Appendix C.

1 Finite Element Triangles

The source code for the Question 1 program can be seen in the `q1.py` file shown in Listing 2.

1.a Local S-Matrix

The equation for the α parameter for a general vertex i of a finite element triangle can be seen in Equation (1), where $i+1$ and $i+2$ implicitly wraps around when exceeding 3.

$$\begin{aligned}\alpha_i(x,y) = \frac{1}{2A}\big[&(x_{i+1}y_{i+2} - x_{i+2}y_{i+1}) \\ &+ (y_{i+1} - y_{i+2})x \\ &+ (x_{i+2} - x_{i+1})y\big]\end{aligned} \tag{1}$$

Using Equation (1), we can solve for the entries of the local S matrix, as shown in Equation (2). This was used in the program to compute every entry for both example triangles.

$$\begin{aligned}S_{ij} &= \int_{\Delta_e} \nabla\alpha_i \cdot \nabla\alpha_j dS \\ &= \frac{1}{4A}\big[(y_{i+1} - y_{i+2})(y_{j+1} - y_{j+2}) \\ &\quad + (x_{i+2} - x_{i+1})(x_{j+2} - x_{j+1})\big]\end{aligned} \tag{2}$$

The local S-matrix for the first triangle can be seen in Equation (3).

$$S_1 = \begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \tag{3}$$

The local S-matrix for the second triangle can be seen in Equation (4).

$$S_2 = \begin{bmatrix} 1.0 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0.0 \\ -0.5 & 0.0 & 0.5 \end{bmatrix} \tag{4}$$

1.b Global S-Matrix

The disjoint S-matrix is then given by the following:

$$S_{dis} = \begin{bmatrix} 0.5 & -0.5 & 0.0 & 0 & 0 & 0 \\ -0.5 & 1.0 & -0.5 & 0 & 0 & 0 \\ 0.0 & -0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & -0.5 & -0.5 \\ 0 & 0 & 0 & -0.5 & 0.5 & 0.0 \\ 0 & 0 & 0 & -0.5 & 0.0 & 0.5 \end{bmatrix}$$

The connectivity matrix C is given by Equation (5).

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{5}$$

The global S-matrix is then given by Equation (6).

$$S = C^T S_{dis} C^T \tag{6}$$

Using Equations (5) and (6), we can solve for the global S-matrix, giving the value shown in Equation (7), which is computed by the `finite_element_triangles.py` script shown in Listing 3.

$$S = \begin{bmatrix} 1.0 & -0.5 & 0.0 & -0.5 \\ -0.5 & 1.0 & -0.5 & 0.0 \\ 0.0 & -0.5 & 1.0 & -0.5 \\ -0.5 & 0.0 & -0.5 & 1.0 \end{bmatrix} \tag{7}$$

2 Finite Element Coaxial Cable

The source code for the Question 2 program can be seen in the `q2.py` file shown in Listing 4.

2.a Mesh

The mesh to be used by the `SIMPLE2D` program is generated by the `finite_element_mesh_generator.py` script shown in Listing 5. This input and output files of the `SIMPLE2D` program are shown in Listings 14 and 15 of Appendix C.

2.b Electrostatic Potential

Based on the results from the `SIMPLE2D` program, the potential at (0.06, 0.04) is 5.5263 V. This corresponds to node 16 in the mesh arrangement we created.

2.c Capacitance

The finite element functional equation for two conjoint finite element triangles forming a square i can be seen in Equation (8).

$$W_i = \frac{1}{2} U_{con_i}^T S U_{con_i} \tag{8}$$

where S is given in Equation (7) and U_{con_i} is the conjoint potential vector for square i, giving the potential at the four corners of the square defining the combination of two finite element triangles. This can be seen in Equation (9).

$$U_{con} = \begin{bmatrix} U_{i_1} \\ U_{i_2} \\ U_{i_3} \\ U_{i_4} \end{bmatrix} \tag{9}$$

To find the total energy function W of the mesh, we must add the contributions from each square and multiply by 4, since our mesh is one quarter of the entire coaxial cable. This yields Equation (10).

$$W = 4 \sum_i^N W_i = 2 \sum_i^N U_{con_i}^T S U_{con_i} \tag{10}$$

where N is the number of finite difference squares in the mesh.

Note that W is not equal to the energy. The relation between the energy per unit length E and W is shown in Equation (11).

$$E = \epsilon_0 W \tag{11}$$

We then know that the energy per unit length E is related to the capacitance per unit length C as shown in Equation (12).

$$E = \frac{1}{2} C V^2 \tag{12}$$

where V is the voltage across the coaxial cable.

Combining Equations (8) and (10) to (12), we obtain an expression for the capacitance per unit length which can be easily calculated, as shown in Equation 13.

$$C = \frac{2E}{V^2} = \frac{4\epsilon_0}{V^2} \sum_i^N U_{con_i}^T S U_{con_i} \tag{13}$$

The capacitance per unit length is computed as 5.2137×10^{-11} F/m by the `finite_element_capacitance.py` script shown in Listing 6 with output shown in Listing 12.

3 Conjugate Gradient Coaxial Cable

The source code for the Question 3 program can be seen in the `q3.py` file shown in Listing 7.

3.a Positive Definite Test

To form the A matrix, we must consider all the free nodes in the mesh. The potential at the non-boundary free nodes is given by Equation (14).

$$-4\phi_{i,j} + \phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} = 0 \tag{14}$$

The free nodes along a boundary must satisfy the Neumann boundary condition for symmetry. Since our quarter-mesh is the bottom left corner of the overall mesh, these boundary nodes defining planes of symmetry are along the top and the right. The Neumann boundary condition for the top nodes is given by Equation (15) and that for the right nodes is given by Equation (16).

$$\phi_{i,j+1} - \phi_{i,j-1} = 0 \tag{15}$$

$$\phi_{i+1,j} - \phi_{i-1,j} = 0 \tag{16}$$

Now, the simplified potential for boundary free nodes can be calculated, as seen in Equations (17) and (18).

$$-4\phi_{i,j} + \phi_{i+1,j} + \phi_{i-1,j} + 2\phi_{i,j-1} = 0 \tag{17}$$

$$-4\phi_{i,j} + 2\phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} = 0 \tag{18}$$

The non-free nodes are fixed by the potentials of the conductors, i.e., 15 V and 0 V.

With Equations (14), (17) and (18), we can form the A matrix from every mesh node. This is done in `finite_difference_mesh_generator.py`, as shown in Listing 8. The output A matrix can be seen in Listing 13.

If the matrix A is not positive definite, one can simply multiply both sides of the $Ax = b$ equation by A^T, forming a new equation $A^T Ax = A^T b$. This is equivalent to $A'x = b'$, where $b' = A^T b$ and $A' = A^T A$. Here, A' is now positive definite.

In our case, the matrix A is indeed not positive definite, and multiplying by A^T made it positive definite. The before and after positive definite test can be seen in Listing 13.

3.b Matrix Solution

The matrix equation to be solved can be seen in Equation (19), where A is positive-definite matrix generated previously, ϕ_c is the unknown potential vector and b contains the initial potential values along the boundaries.

$$A\phi_c = b \tag{19}$$

The matrix equation is solved first by the Choleski method from Assignment 1 is applied, as found in the `choleski.py` shown in Listing 9. Then, the conjugate gradient method defined by `conjugate_gradient_solve` of the `conjugate_gradient.py` file shown in Listing 10 was applied. The solved x vector from both methods can be seen in Listing 13.

3.c Residual Norm

Consider a vector $\mathbf{v} = \{v_1, \ldots, v_n\}$. The infinity norm $\|\mathbf{v}\|_\infty$ of $\mathbf{v}$ is given by the maximum absolute element of $\mathbf{v}$, as shown in Equation (20).

$$\|\mathbf{v}\|_\infty = \max\{|v_1|, \ldots, |v_n|\} \tag{20}$$

Similarly, the 2-norm $\|\mathbf{v}\|_2$ of $\mathbf{v}$ is given by Equation (21).

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2} \tag{21}$$

The infinity norm of the residual vector versus conjugate gradient iterations can be seen in Figure 1. The 2-norm versus iterations can be seen in Figure 2. Both norms converge to 0 after $n = 19$ iterations, as expected.

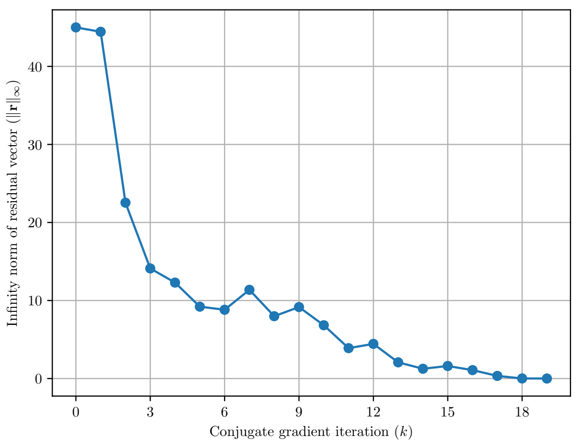

Figure 1: Value of the infinity norm $\|\mathbf{r}\|_\infty$ of the residual vector versus iterations of the conjugate gradient algorithm.

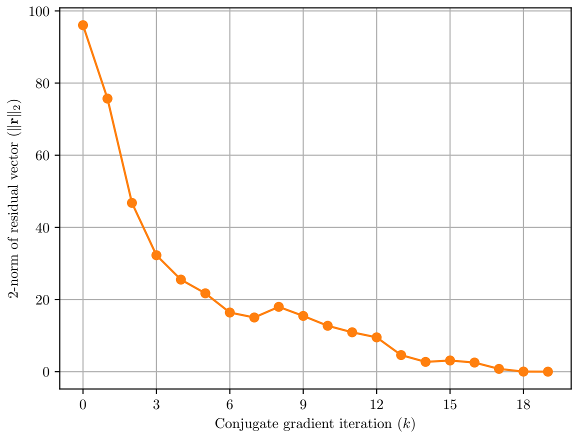

Figure 2: Value of the 2-norm $\|\mathbf{r}\|_2$ of the residual vector versus iterations of the conjugate gradient algorithm.

3.d Potential Comparison

A comparison of the potential at (0.06, 0.04) to three decimal places for various numerical methods can be seen in Table 1. It can be seen that the potential found with all the methods is the same to three decimal places.

Table 1: Comparison of potential at (0.06, 0.04) to three decimal places for various numerical methods.

Method	Potential (V)
Choleski	5.526
Conjugate gradient	5.526
Finite element	5.526
Finite difference (SOR)	5.526

3.e Capacitance Computation

The capacitance can be calculated in the same way as in Question 2(c), i.e., with Equation (13). The node values must simply be mapped to same mesh used in the finite difference context.

A Code Listings

Listing 1: Custom matrix package (`matrices.py`).

```python
from __future__ import division

import copy
import csv
from ast import literal_eval

import math


class Matrix:
    def __init__(self, data):
        self.data = data
        self.num_rows = len(data)
        self.num_cols = len(data[0])

    def __str__(self):
        string = ''
        for row in self.data:
            string += '\n'
            for val in row:
                string += '{:6.2f} '.format(val)
        return string

    def integer_string(self):
        string = ''
        for row in self.data:
            string += '\n'
            for val in row:
                string += '{:3.0f} '.format(val)
        return string

    def __add__(self, other):
        if len(self) != len(other) or len(self[0]) != len(other[0]):
            raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is {}x{}.'
                             .format(len(self), len(self[0]), len(other), len(other[0])))

        return Matrix([[self[row][col] + other[row][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def __sub__(self, other):
        if len(self) != len(other) or len(self[0]) != len(other[0]):
            raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B is {}x{}.'
                             .format(len(self), len(self[0]), len(other), len(other[0])))

        return Matrix([[self[row][col] - other[row][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def __mul__(self, other):
        if type(other) == float or type(other) == int:
            return self.scalar_multiply(other)

        if self.num_cols != other.num_rows:
            raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix B is {}x{}.'
                             .format(self.num_rows, self.num_cols, other.num_rows, other.num_cols))

        # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
        product = Matrix.empty(self.num_rows, other.num_cols)
        for i in range(self.num_rows):
            for j in range(other.num_cols):
                row_sum = 0
                for k in range(self.num_cols):
                    row_sum += self[i][k] * other[k][j]
```

```python
                product[i][j] = row_sum
        return product

    def scalar_multiply(self, scalar):
        return Matrix([[self[row][col] * scalar for col in range(self.num_cols)] for row in
         ↪  range(self.num_rows)])

    def __div__(self, other):
        """
        Element-wise division.
        """
        if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
            raise ValueError('Incompatible matrix sizes.')
        return Matrix([[self[row][col] / other[row][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def __neg__(self):
        return Matrix([[-self[row][col] for col in range(self.num_cols)] for row in range(self.num_rows)])

    def __deepcopy__(self, memo):
        return Matrix(copy.deepcopy(self.data))

    def __getitem__(self, item):
        return self.data[item]

    def __len__(self):
        return len(self.data)

    def item(self):
        """
        :return: the single element contained by this matrix, if it is 1x1.
        """
        if not (self.num_rows == 1 and self.num_cols == 1):
            raise ValueError('Matrix is not 1x1')
        return self.data[0][0]

    def is_positive_definite(self):
        """
        :return: True if the matrix if positive-definite, False otherwise.
        """
        A = copy.deepcopy(self.data)
        for j in range(self.num_rows):
            if A[j][j] <= 0:
                return False
            A[j][j] = math.sqrt(A[j][j])
            for i in range(j + 1, self.num_rows):
                A[i][j] = A[i][j] / A[j][j]
                for k in range(j + 1, i + 1):
                    A[i][k] = A[i][k] - A[i][j] * A[k][j]
        return True

    def transpose(self):
        """
        :return: the transpose of the current matrix
        """
        return Matrix([[self.data[row][col] for row in range(self.num_rows)] for col in
         ↪  range(self.num_cols)])

    def mirror_horizontal(self):
        """
        :return: the horizontal mirror of the current matrix
        """
        return Matrix([[self.data[self.num_rows - row - 1][col] for col in range(self.num_cols)]
                       for row in range(self.num_rows)])

    def empty_copy(self):
        """
        :return: an empty matrix of the same size as the current matrix.
        """
        return Matrix.empty(self.num_rows, self.num_cols)
```

```python

    def infinity_norm(self):
        if self.num_cols > 1:
            raise ValueError('Not a column vector.')
        return max([abs(x) for x in self.transpose()[0]])

    def two_norm(self):
        if self.num_cols > 1:
            raise ValueError('Not a column vector.')
        return math.sqrt(sum([x ** 2 for x in self.transpose()[0]]))

    def save_to_csv(self, filename):
        """
        Saves the current matrix to a CSV file.

        :param filename: the name of the CSV file
        """
        with open(filename, "wb") as f:
            writer = csv.writer(f)
            for row in self.data:
                writer.writerow(row)

    def save_to_latex(self, filename):
        """
        Saves the current matrix to a latex-readable matrix.

        :param filename: the name of the CSV file
        """
        with open(filename, "wb") as f:
            for row in range(self.num_rows):
                for col in range(self.num_cols):
                    f.write('{}'.format(self.data[row][col]))
                    if col < self.num_cols - 1:
                        f.write('& ')
                if row < self.num_rows - 1:
                    f.write('\\\\\n')

    @staticmethod
    def multiply(*matrices):
        """
        Computes the product of the given matrices.

        :param matrices: the matrix objects
        :return: the product of the given matrices
        """
        n = matrices[0].rows
        product = Matrix.identity(n)
        for matrix in matrices:
            product = product * matrix
        return product

    @staticmethod
    def empty(num_rows, num_cols):
        """
        Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.

        :param num_rows: number of rows
        :param num_cols: number of columns
        :return: the empty matrix
        """
        return Matrix([[0 for _ in range(num_cols)] for _ in range(num_rows)])

    @staticmethod
    def identity(n):
        """
        Returns the identity matrix of the given size.

        :param n: the size of the identity matrix (number of rows or columns)
        :return: the identity matrix of size n
        """
```

```python
        return Matrix.diagonal_single_value(1, n)

    @staticmethod
    def diagonal(values):
        """
        Returns a diagonal matrix with the given values along the main diagonal.

        :param values: the values along the main diagonal
        :return: a diagonal matrix with the given values along the main diagonal
        """
        n = len(values)
        return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])

    @staticmethod
    def diagonal_single_value(value, n):
        """
        Returns a diagonal matrix of the given size with the given value along the diagonal.

        :param value: the value of each element on the main diagonal
        :param n: the size of the matrix
        :return: a diagonal matrix of the given size with the given value along the diagonal.
        """
        return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])

    @staticmethod
    def column_vector(values):
        """
        Transforms a row vector into a column vector.

        :param values: the values, one for each row of the column vector
        :return: the column vector
        """
        return Matrix([[value] for value in values])

    @staticmethod
    def csv_to_matrix(filename):
        """
        Reads a CSV file to a matrix.

        :param filename: the name of the CSV file
        :return: a matrix containing the values in the CSV file
        """
        with open(filename, 'r') as csv_file:
            reader = csv.reader(csv_file)
            data = []
            for row_number, row in enumerate(reader):
                data.append([literal_eval(val) for val in row])
            return Matrix(data)
```

Listing 2: Question 1 (q1.py).

```python
from finite_element_triangles import Triangle, find_local_s_matrix, find_global_s_matrix
from matrices import Matrix


def q1():
    print('\n=== Question 1 ===')
    S1 = build_triangle_and_find_local_S(
        [0, 0, 0.02],
        [0.02, 0, 0])
    S1.save_to_latex('report/matrices/S1.txt')
    print('S1: {}'.format(S1))

    S2 = build_triangle_and_find_local_S(
        [0.02, 0, 0.02],
        [0.02, 0.02, 0])
    S2.save_to_latex('report/matrices/S2.txt')
    print('S2: {}'.format(S2))

```

```python
    C = Matrix([
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1],
        [1, 0, 0, 0],
        [0, 0, 1, 0]])
    C.save_to_latex('report/matrices/C.txt')
    print('C: {}'.format(C))

    S = find_global_s_matrix(S1, S2, C)
    S.save_to_latex('report/matrices/S.txt')
    S.save_to_csv('report/csv/S.txt')
    print('S: {}'.format(S))


def build_triangle_and_find_local_S(x, y):
    triangle = Triangle(x, y)
    S = find_local_s_matrix(triangle)
    return S


if __name__ == '__main__':
    q1()
```

Listing 3: Finite element triangles (`finite_element_triangles.py`).

```python
from __future__ import division

from matrices import Matrix


class Triangle:
    """Represents a finite-difference triangle."""
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.area = (x[1] * y[2] - x[2] * y[1] - x[0] * y[2] + x[2] * y[0] + x[0] * y[1] - x[1] * y[0]) / 2


def find_local_s_matrix(triangle):
    """
    Finds the local S matrix for a finite-difference triangle.

    :param triangle: the finite-difference triangle
    :return: the local S matrix
    """
    x = triangle.x
    y = triangle.y
    S = Matrix.empty(3, 3)

    for i in range(3):
        for j in range(3):
            S[i][j] = ((y[(i + 1) % 3] - y[(i + 2) % 3]) * (y[(j + 1) % 3] - y[(j + 2) % 3])
                       + (x[(i + 1) % 3] - x[(i + 2) % 3]) * (x[(j + 1) % 3] - x[(j + 2) % 3])) / (4 * triangle.area)

    return S


def find_global_s_matrix(S1, S2, C):
    """
    Finds the global S matrix given by two local S matrices and the the connectivity matrix.

    :param S1: the first local S matrix
    :param S2: the second local S matrix
    :param C: the connectivity matrix
    :return: the global S matrix
```

```
    """
    S_dis = find_disjoint_s_matrix(S1, S2)
    S_dis.save_to_latex('report/matrices/S_dis.txt')
    print('S_dis: {}'.format(S_dis))
    return C.transpose() * S_dis * C


def find_disjoint_s_matrix(S1, S2):
    """
    Finds the disjoint S matrix given by the two provided local S matrices.

    :param S1: the first local S matrix
    :param S2: the second local S matrix
    :return: the disjoint S matrix
    """
    n = len(S1)
    S_dis = Matrix.empty(2 * n, 2 * n)
    for row in range(n):
        for col in range(n):
            S_dis[row][col] = S1[row][col]
            S_dis[row + n][col + n] = S2[row][col]
    return S_dis
```

Listing 4: Question 2 (`q2.py`).

```
from finite_element_capacitance import find_capacitance
from matrices import Matrix
from finite_element_mesh_generator import generate_simple_2d_mesh

INNER_CONDUCTOR_POINTS = [28, 29, 30, 34]
OUTER_CONDUCTOR_POINTS = [1, 2, 3, 4, 5, 6, 7, 13, 19, 25, 31]

MESH_SIZE = 6


def q2():
    print('\n=== Question 2 ===')
    q2a()
    q2c()


def q2a():
    generate_simple_2d_mesh(MESH_SIZE, INNER_CONDUCTOR_POINTS, OUTER_CONDUCTOR_POINTS)


def q2c():
    print('\n=== Question 2(c) ===')
    S = Matrix.csv_to_matrix('report/csv/S.txt')
    voltage = 15
    capacitance = find_capacitance(S, voltage, MESH_SIZE)
    print('Capacitance per unit length: {} F/m'.format(capacitance))


if __name__ == '__main__':
    q2()
```

Listing 5: Finite element mesh generator (`finite_element_mesh_generator.py`).

```
def generate_simple_2d_mesh(mesh_size, inner_conductor_points, outer_conductor_points):
    """
    Generates the input mesh needed for the SIMPLE2D program.

    :param mesh_size: the mesh size
    :param inner_conductor_points: the inner conductor points
    :param outer_conductor_points: the outer conductor points
    """
    with open('simple2d/mesh.dat', 'w') as f:
        generate_node_positions(f, mesh_size)
```

```python
        generate_triangle_coordinates(f, mesh_size)
        generate_initial_potentials(f, inner_conductor_points, outer_conductor_points)


def generate_node_positions(f, mesh_size):
    """
    Generates the node positions for the SIMPLE2D program.

    :param f: the mesh file
    :param mesh_size: the mesh size
    """
    for row in range(mesh_size):
        y = row * 0.02
        for col in range(mesh_size):
            x = col * 0.02
            node = row * mesh_size + (col + 1)
            if node <= 34:  # Inner conductor
                f.write('{} {} {}\n'.format(node, x, y))
    f.write('\n')


def generate_triangle_coordinates(f, mesh_size):
    # Left triangles (left halves of squares)
    """
    Generates the triangle coordinates for the SIMPLE2D program.

    :param f: the mesh file
    :param mesh_size: the mesh size
    """
    for row in range(mesh_size - 1):
        for col in range(mesh_size - 1):
            node = row * mesh_size + (col + 1)
            if node < 28:
                f.write('{} {} {} 0\n'.format(node, node + 1, node + mesh_size))

    # Right triangles (right halves of squares)
    for row in range(mesh_size - 1):
        for col in range(1, mesh_size):
            node = row * mesh_size + (col + 1)
            if node <= 28:
                f.write('{} {} {} 0\n'.format(node, node + mesh_size - 1, node + mesh_size))

    f.write('\n')


def generate_initial_potentials(f, inner_conductor_points, outer_conductor_points):
    """
    Generates the initial potentials for the SIMPLE2D program.

    :param f: the mesh file
    :param inner_conductor_points: the inner conductor points
    :param outer_conductor_points: the outer conductor points
    """
    for point in outer_conductor_points:
        f.write('{} {}\n'.format(point, 0))
    for point in inner_conductor_points:
        f.write('{} {}\n'.format(point, 15))
```

Listing 6: Finite element capacitance (`finite_element_capacitance.py`).

```python
from matrices import Matrix

E_0 = 8.854187817620E-12


def extract_mesh():
    """
    Extracts mesh information from the SIMPLE2D file.

```

```python
    :return: the extracted mesh
    """
    with open('simple2d/result.dat') as f:
        mesh = {}
        for line_number, line in enumerate(f):
            if line_number >= 2:
                vals = line.split()
                node = int(float(vals[0]))
                voltage = float(vals[3])
                mesh[node] = voltage
    return mesh


def compute_half_energy(S, mesh, mesh_size):
    """
    Computes the half-energy needed to compute the capacitance of the mesh.

    :param S: the S matrix
    :param mesh: the mesh
    :param mesh_size: the mesh size
    :return: the half-energy
    """
    U_con = Matrix.empty(4, 1)
    half_energy = 0
    for row in range(mesh_size - 1):
        for col in range(mesh_size - 1):
            node = row * mesh_size + (col + 1)  # 1-based
            if node < 28:
                U_con[0][0] = mesh[node + mesh_size]
                U_con[1][0] = mesh[node]
                U_con[2][0] = mesh[node + 1]
                U_con[3][0] = mesh[node + mesh_size + 1]
                half_energy_contribution = U_con.transpose() * S * U_con
                half_energy += half_energy_contribution[0][0]
    return half_energy


def find_capacitance(S, voltage, mesh_size):
    """
    Finds the capacitance per unit length of the mesh.

    :param S: the S matrix
    :param voltage: the voltage difference
    :param mesh_size: the mesh size
    :return: the capacitance per unit length
    """
    mesh = extract_mesh()
    half_energy = compute_half_energy(S, mesh, mesh_size)
    capacitance = (4 * E_0 * half_energy) / voltage ** 2
    return capacitance
```

Listing 7: Question 3 (`q3.py`).

```python
from copy import deepcopy

import matplotlib.pyplot as plt

from matplotlib import rc
from matplotlib.ticker import MaxNLocator

from choleski import choleski_solve
from conjugate_gradient import conjugate_gradient_solve
from finite_difference_mesh_generator import generate_finite_diff_mesh

MESH_SIZE = 6
NUM_FREE_NODES = 19
rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)

```

```python

def q3():
    print('\n=== Question 3 ===')
    A, b = q3a()
    choleski_potential, cg_potential, residual_vectors = q3b(A, b)
    q3c(residual_vectors)
    q3d(choleski_potential, cg_potential)


def q3a():
    print('\n=== Question 3(a) ===')
    A, b = generate_finite_diff_mesh(MESH_SIZE, NUM_FREE_NODES)
    print('A: {}'.format(A.integer_string()))
    print('b: {}'.format(b.integer_string()))
    print('A is positive definite: {}'.format(A.is_positive_definite()))
    A_prime = A.transpose() * A
    b_prime = A.transpose() * b
    print("A' is positive definite: {}".format(A_prime.is_positive_definite()))
    return A_prime, b_prime


def q3b(A, b):
    print('\n=== Question 3(b) ===')
    A_copy = deepcopy(A)
    b_copy = deepcopy(b)
    x_choleski = choleski_solve(A_copy, b_copy)
    print('Choleski x: {}'.format(x_choleski))
    residual_vectors = []
    x_cg = conjugate_gradient_solve(A, b, residual_vectors)
    print('Conjugate gradient x: {}'.format(x_cg))
    node_6_4 = 7
    return x_choleski[node_6_4][0], x_cg[node_6_4][0], residual_vectors


def q3c(residual_vectors):
    print('\n=== Question 3(c) ===')
    plot_residual_norms(residual_vectors, infinity_norm=False)
    plot_residual_norms(residual_vectors, infinity_norm=True)


def q3d(choleski_potential, cg_potential):
    print('\n=== Question 3(d) ===')
    print('Choleski potential at (0.06, 0.04): {} V'.format(choleski_potential))
    print('Conjugate gradient potential at (0.06, 0.04): {} V'.format(cg_potential))


def plot_residual_norms(residual_vectors, infinity_norm=False):
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = [i for i in range(len(residual_vectors))]
    y_range = [v.infinity_norm() if infinity_norm else v.two_norm() for v in residual_vectors]
    plt.plot(x_range, y_range, 'o-{}'.format('C0' if infinity_norm else 'C1'),
             label=''.format('Infinity norm' if infinity_norm else '2-norm'))
    plt.xlabel('Conjugate gradient iteration ($k$)')
    plt.ylabel('Infinity norm of residual vector $(\\|\\textbf{r}\\|_\\infty)$' if infinity_norm
               else '2-norm of residual vector $(\\|\\textbf{r}\\|_2)$')
    plt.grid(True)
    f.savefig('report/plots/q3c_{}.pdf'.format('infinity' if infinity_norm else '2'), bbox_inches='tight')


if __name__ == '__main__':
    q3()
```

Listing 8: Finite difference mesh generator (`finite_difference_mesh_generator.py`).

```python
from matrices import Matrix


```

```python
def generate_finite_diff_mesh(mesh_size, num_free_nodes):
    """
    Generates a finite-difference mesh with the given size and number of free nodes.

    :param mesh_size: the mesh size
    :param num_free_nodes: the number of free nodes
    :return: the A and b matrices defining the mesh equation (Ax = b)
    """
    A = Matrix.empty(num_free_nodes, num_free_nodes)
    b = Matrix.empty(num_free_nodes, 1)
    for row in range(mesh_size - 3):
        for col in range(mesh_size - 1):
            node = row * (mesh_size - 1) + col
            A[node][node] = -4

            if row != 0:
                A[node][node - mesh_size + 1] = 1
            if 12 <= node <= 14:
                b[node][0] = -15
            else:
                A[node][node + mesh_size - 1] = 1

            # Right Neumann boundary
            if col == mesh_size - 2:
                A[node][node - 1] = 2
            else:
                if col != 0:
                    A[node][node - 1] = 1
                A[node][node + 1] = 1

    # Special nodes
    A[15][10] = 1
    A[15][15] = -4
    A[15][16] = 1
    A[15][17] = 1

    A[16][11] = 1
    A[16][15] = 1
    A[16][16] = -4
    A[16][18] = 1
    b[16][0] = -15

    A[17][15] = 2
    A[17][17] = -4
    A[17][18] = 1

    A[18][16] = 2
    A[18][17] = 1
    A[18][18] = -4
    b[18][0] = -15

    return A, b
```

Listing 9: Choleski decomposition (`choleski.py`).

```python
from __future__ import division

import math

from matrices import Matrix


def choleski_solve(A, b, half_bandwidth=None):
    """
    Solves an Ax = b matrix equation by Choleski decomposition.
    :param A: the A matrix
    :param b: the b matrix
    :param half_bandwidth: the half-bandwidth of the A matrix
    :return: the solved x vector
```

```
    """
    n = len(A[0])
    if half_bandwidth is None:
        elimination(A, b)
    else:
        elimination_banded(A, b, half_bandwidth)
    x = Matrix.empty(n, 1)
    back_substitution(A, x, b)
    return x


def elimination(A, b):
    """
    Performs the elimination step of Choleski decomposition.
    :param A: the A matrix
    :param b: the b matrix
    """
    n = len(A)
    for j in range(n):
        if A[j][j] <= 0:
            raise ValueError('Matrix A is not positive definite.')
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0] / A[j][j]
        for i in range(j + 1, n):
            A[i][j] = A[i][j] / A[j][j]
            b[i][0] = b[i][0] - A[i][j] * b[j][0]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]


def elimination_banded(A, b, half_bandwidth):
    """
    Performs the banded elimination step of Choleski decomposition.
    :param A: the A matrix
    :param b: the b matrix
    :param half_bandwidth: the half_bandwidth to be used for the banded elimination
    """
    n = len(A)
    for j in range(n):
        if A[j][j] <= 0:
            raise ValueError('Matrix A is not positive definite.')
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0] / A[j][j]
        max_row = min(j + half_bandwidth, n)
        for i in range(j + 1, max_row):
            A[i][j] = A[i][j] / A[j][j]
            b[i][0] = b[i][0] - A[i][j] * b[j][0]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]


def back_substitution(L, x, y):
    """
    Performs the back-substitution step of Choleski decomposition.
    :param L: the L matrix
    :param x: the x matrix
    :param y: the y matrix
    """
    n = len(L)
    for i in range(n - 1, -1, -1):
        prev_sum = 0
        for j in range(i + 1, n):
            prev_sum += L[j][i] * x[j][0]
        x[i][0] = (y[i][0] - prev_sum) / L[i][i]
```

Listing 10: Conjugate gradient (`conjugate_gradient.py`).

```
from copy import deepcopy

```

```python
from matrices import Matrix


def conjugate_gradient_solve(A, b, residual_vectors=None):
    """
    Solves the Ax = b matrix equation given by the given A and b matrices

    :param A: the A matrix
    :param b: the b matrix
    :param residual_vectors: the list to store the residual vectors in
    :return: the solved x vector
    """
    n = len(A)
    x = Matrix.empty(n, 1)
    r = b - A * x
    p = deepcopy(r)
    if residual_vectors is not None:
        residual_vectors.append(r)
    for _ in range(n):
        denom = p.transpose() * A * p
        alpha = (p.transpose() * r) / denom
        x = x + p * alpha.item()
        r = b - A * x
        beta = - (p.transpose() * A * r) / denom
        p = r + p * beta.item()
        if residual_vectors is not None:
            residual_vectors.append(r)
    return x
```

B Output Logs

Listing 11: Output of Question 1 program (q1.txt).

```
=== Question 1 ===
S1:
  0.50  -0.50   0.00
 -0.50   1.00  -0.50
  0.00  -0.50   0.50
S2:
  1.00  -0.50  -0.50
 -0.50   0.50   0.00
 -0.50   0.00   0.50
C:
  1.00   0.00   0.00   0.00
  0.00   1.00   0.00   0.00
  0.00   0.00   1.00   0.00
  0.00   0.00   0.00   1.00
  1.00   0.00   0.00   0.00
  0.00   0.00   1.00   0.00
S_dis:
  0.50  -0.50   0.00   0.00   0.00   0.00
 -0.50   1.00  -0.50   0.00   0.00   0.00
  0.00  -0.50   0.50   0.00   0.00   0.00
  0.00   0.00   0.00   1.00  -0.50  -0.50
  0.00   0.00   0.00  -0.50   0.50   0.00
  0.00   0.00   0.00  -0.50   0.00   0.50
S:
  1.00  -0.50   0.00  -0.50
 -0.50   1.00  -0.50   0.00
  0.00  -0.50   1.00  -0.50
 -0.50   0.00  -0.50   1.00
```

Listing 12: Output of Question 2 program (q2.txt).

```
=== Question 2 ===

```

```
=== Question 2(c) ===
Capacitance per unit length: 5.21374340427e-11 F/m
```

Listing 13: Output of Question 3 program (`q3.txt`).

```
=== Question 3 ===

=== Question 3(a) ===
A:
 -4   1   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0
  1  -4   1   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
  0   1  -4   1   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
  0   0   1  -4   1   0   0   0   1   0   0   0   0   0   0   0   0   0   0
  0   0   0   2  -4   0   0   0   0   1   0   0   0   0   0   0   0   0   0
  1   0   0   0   0  -4   1   0   0   0   1   0   0   0   0   0   0   0   0
  0   1   0   0   0   1  -4   1   0   0   0   1   0   0   0   0   0   0   0
  0   0   1   0   0   0   1  -4   1   0   0   0   1   0   0   0   0   0   0
  0   0   0   1   0   0   0   1  -4   1   0   0   0   1   0   0   0   0   0
  0   0   0   0   1   0   0   0   2  -4   0   0   0   0   1   0   0   0   0
  0   0   0   0   0   1   0   0   0   0  -4   1   0   0   0   1   0   0   0
  0   0   0   0   0   0   1   0   0   0   1  -4   1   0   0   0   1   0   0
  0   0   0   0   0   0   0   1   0   0   0   1  -4   1   0   0   0   0   0
  0   0   0   0   0   0   0   0   1   0   0   0   1  -4   1   0   0   0   0
  0   0   0   0   0   0   0   0   0   1   0   0   0   2  -4   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0  -4   1   1   0
  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   1  -4   0   1
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   2   0  -4   1
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   2   1  -4
b:
  0
  0
  0
  0
  0
  0
  0
  0
  0
  0
  0
  0
-15
-15
-15
  0
-15
  0
-15
A is positive definite: False
A' is positive definite: True

=== Question 3(b) ===
Choleski x:
  0.96
  1.86
  2.61
  3.04
  3.17
  1.97
  3.88
  5.53
  6.37
  6.61
  3.03
  6.18
  9.25
 10.29
 10.55
  3.96
```

```
  8.56
  4.25
  9.09
Conjugate gradient x:
  0.96
  1.86
  2.61
  3.04
  3.17
  1.97
  3.88
  5.53
  6.37
  6.61
  3.03
  6.18
  9.25
 10.29
 10.55
  3.96
  8.56
  4.25
  9.09

=== Question 3(c) ===

=== Question 3(d) ===
Choleski potential at (0.06, 0.04): 5.52634126517 V
Conjugate gradient potential at (0.06, 0.04): 5.52634127414 V
```

C Simple2D Data Files

Listing 14: Input mesh for the SIMPLE2D program.

```
1 0.0 0.0
2 0.02 0.0
3 0.04 0.0
4 0.06 0.0
5 0.08 0.0
6 0.1 0.0
7 0.0 0.02
8 0.02 0.02
9 0.04 0.02
10 0.06 0.02
11 0.08 0.02
12 0.1 0.02
13 0.0 0.04
14 0.02 0.04
15 0.04 0.04
16 0.06 0.04
17 0.08 0.04
18 0.1 0.04
19 0.0 0.06
20 0.02 0.06
21 0.04 0.06
22 0.06 0.06
23 0.08 0.06
24 0.1 0.06
25 0.0 0.08
26 0.02 0.08
27 0.04 0.08
28 0.06 0.08
29 0.08 0.08
30 0.1 0.08
31 0.0 0.1
32 0.02 0.1
33 0.04 0.1
```

```
34 0.06 0.1

1 2 7 0
2 3 8 0
3 4 9 0
4 5 10 0
5 6 11 0
7 8 13 0
8 9 14 0
9 10 15 0
10 11 16 0
11 12 17 0
13 14 19 0
14 15 20 0
15 16 21 0
16 17 22 0
17 18 23 0
19 20 25 0
20 21 26 0
21 22 27 0
22 23 28 0
23 24 29 0
25 26 31 0
26 27 32 0
27 28 33 0
2 7 8 0
3 8 9 0
4 9 10 0
5 10 11 0
6 11 12 0
8 13 14 0
9 14 15 0
10 15 16 0
11 16 17 0
12 17 18 0
14 19 20 0
15 20 21 0
16 21 22 0
17 22 23 0
18 23 24 0
20 25 26 0
21 26 27 0
22 27 28 0
23 28 29 0
24 29 30 0
26 31 32 0
27 32 33 0
28 33 34 0

1 0
2 0
3 0
4 0
5 0
6 0
7 0
13 0
19 0
25 0
31 0
28 15
29 15
30 15
34 15
```

Listing 15: Resulting potentials generated by the `SIMPLE2D` program.

```
ans =

```

```
    1.0000         0         0         0
    2.0000    0.0200         0         0
    3.0000    0.0400         0         0
    4.0000    0.0600         0         0
    5.0000    0.0800         0         0
    6.0000    0.1000         0         0
    7.0000         0    0.0200         0
    8.0000    0.0200    0.0200    0.9571
    9.0000    0.0400    0.0200    1.8616
   10.0000    0.0600    0.0200    2.6060
   11.0000    0.0800    0.0200    3.0360
   12.0000    0.1000    0.0200    3.1714
   13.0000         0    0.0400         0
   14.0000    0.0200    0.0400    1.9667
   15.0000    0.0400    0.0400    3.8834
   16.0000    0.0600    0.0400    5.5263
   17.0000    0.0800    0.0400    6.3668
   18.0000    0.1000    0.0400    6.6135
   19.0000         0    0.0600         0
   20.0000    0.0200    0.0600    3.0262
   21.0000    0.0400    0.0600    6.1791
   22.0000    0.0600    0.0600    9.2492
   23.0000    0.0800    0.0600   10.2912
   24.0000    0.1000    0.0600   10.5490
   25.0000         0    0.0800         0
   26.0000    0.0200    0.0800    3.9590
   27.0000    0.0400    0.0800    8.5575
   28.0000    0.0600    0.0800   15.0000
   29.0000    0.0800    0.0800   15.0000
   30.0000    0.1000    0.0800   15.0000
   31.0000         0    0.1000         0
   32.0000    0.0200    0.1000    4.2525
   33.0000    0.0400    0.1000    9.0919
   34.0000    0.0600    0.1000   15.0000
```