ECSE 543
Assignment 1

Sean Stappas
260639512

October 17, 2017

1 Choleski Decomposition

1.a Choleski Program

1.b Constructing Test Matrices

1.c Test Runs

1.d Linear Networks

2 Finite Difference Mesh

2.a Equivalent Resistance

2.b Time Complexity

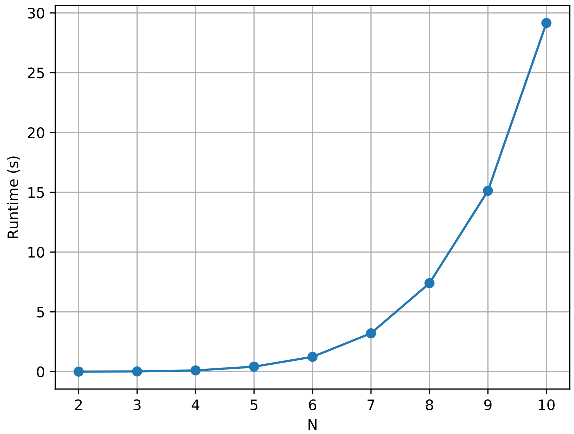

Figure 1: Runtime of program versus N.

2.c Sparsity Modification

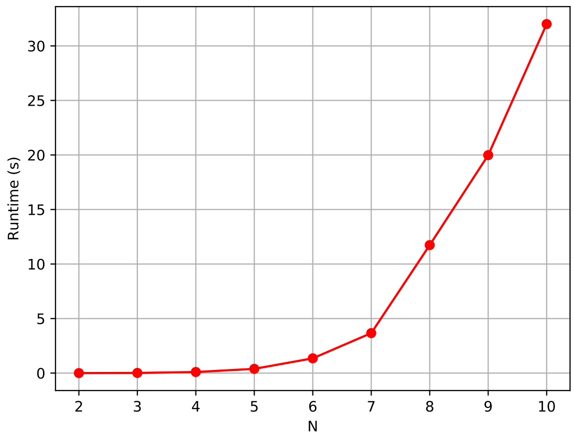

Figure 2: Runtime of banded program versus N.

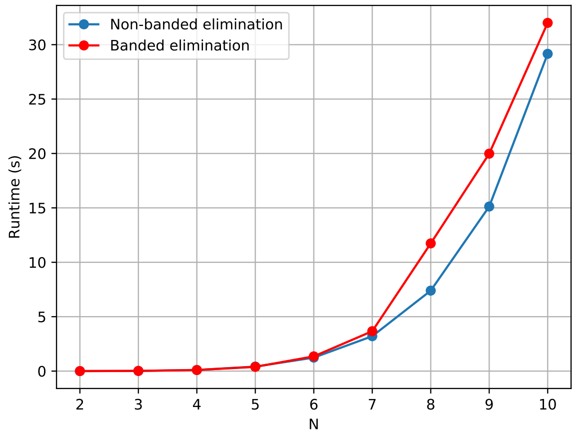

Figure 3: Comparison of runtime of banded and non-banded programs versus N.

2.d Resistance vs. Mesh Size

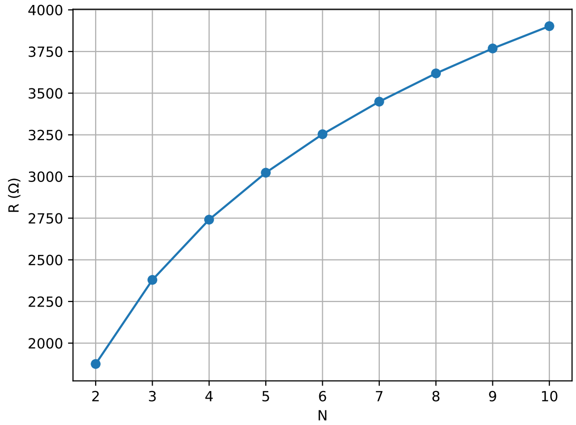

Figure 4: Resistance of mesh versus mesh size.

3 Coaxial Cable

3.a SOR Program

3.b Varying ω

3.c Varying h

3.d Jacobi Method

3.e Non-uniform Node Spacing

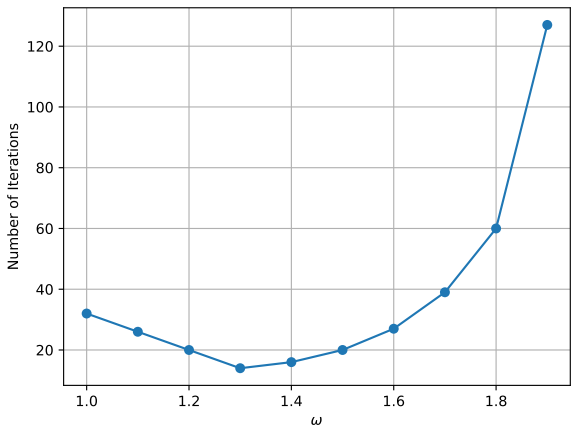

Figure 5: Number of iterations of SOR versus ω.

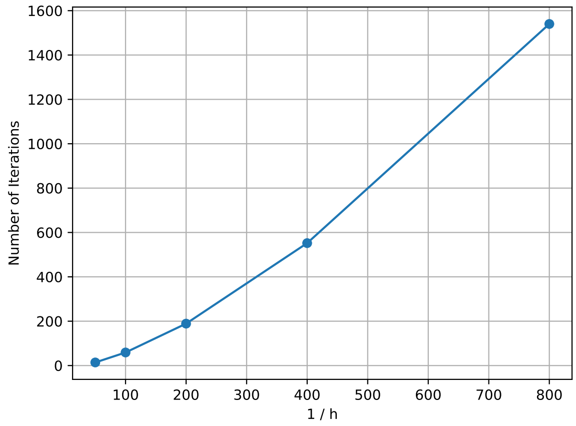

Figure 6: Number of iterations of SOR versus $1/h$.

Table 1: Number of iterations versus ω.

Omega	Iterations
1.0	32
1.1	26
1.2	20
1.3	14
1.4	16
1.5	20
1.6	27
1.7	39
1.8	60
1.9	127

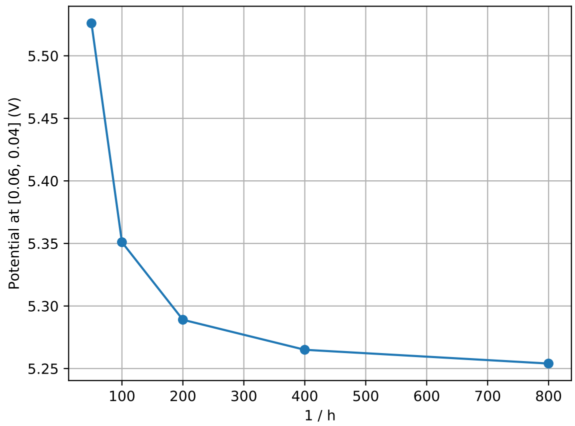

Figure 7: Potential at $(0.06, 0.04)$ found by SOR versus $1/h$.

Table 2: Potential versus ω.

Omega	Potential (V)
1.0	5.526
1.1	5.526
1.2	5.526
1.3	5.526
1.4	5.526
1.5	5.526
1.6	5.526
1.7	5.526
1.8	5.526
1.9	5.526

Table 4: Potential versus ω.

1/h	Potential (V)
50.0	5.526
100.0	5.351
200.0	5.289
400.0	5.265
800.0	5.254

Table 3: Number of iterations versus ω.

1/h	Iterations
50.0	14
100.0	59
200.0	189
400.0	552
800.0	1540

Table 5: Number of iterations versus ω.

1/h	Iterations
50.0	51
100.0	180
200.0	604
400.0	1935
800.0	5836

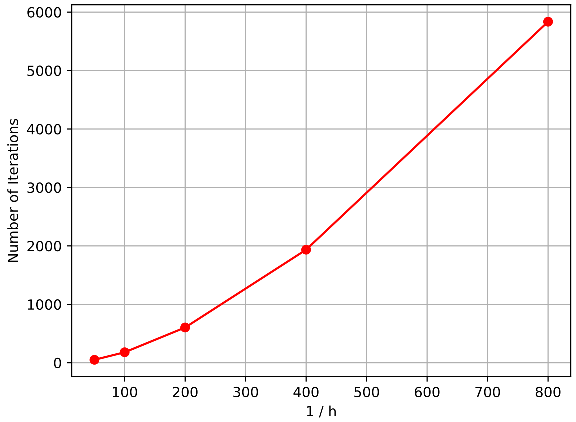

Figure 8: Number of iterations of the Jacobi method versus $1/h$.

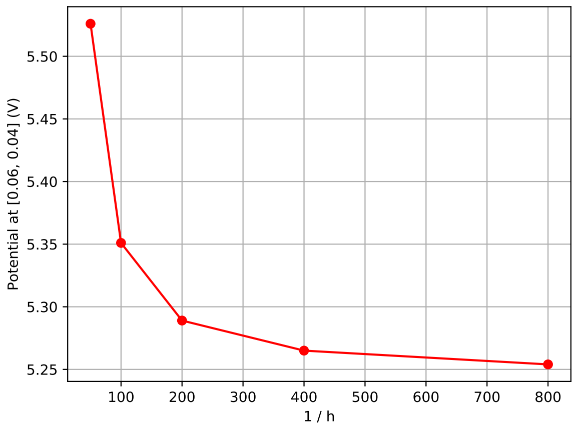

Figure 9: Potential at $(0.06, 0.04)$ found by the Jacobi method versus $1/h$.

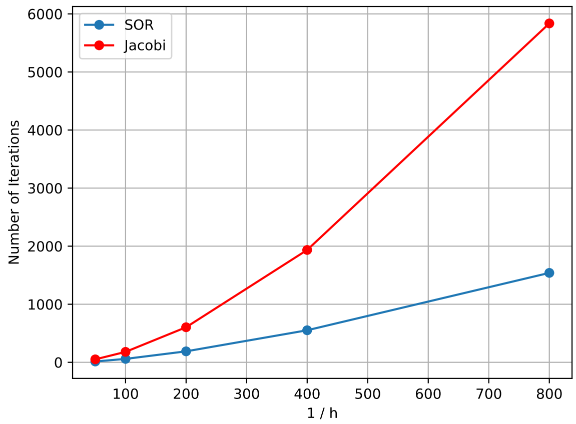

Figure 10: Comparison of number of iterations when using SOR and Jacobi methods versus $1/h$.

Table 6: Potential versus ω.

1/h	Potential (V)
50.0	5.526
100.0	5.351
200.0	5.289
400.0	5.265
800.0	5.254

Table 7: Potential versus ω.

1/h	Potential (V)
50.0	5.526
100.0	5.351
200.0	5.289
400.0	5.265
800.0	5.254

A Code Listings

Listing 1: Custom matrix package.

```
from __future__ import division

import copy
import csv
from ast import literal_eval

import math


class Matrix:

    def __init__(self, data):
        self.data = data

    def __str__(self):
        string = ''
        for row in self.data:
            string += '\n'
            for val in row:
                string += '{:6.2f} '.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])))
        rows = len(self)
        cols = len(self[0])

        return Matrix([[self[row][col] + other[row][col] for col in range(cols)] for row in range(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])))
        rows = len(self)
        cols = len(self[0])

        return Matrix([[self[row][col] - other[row][col] for col in range(cols)] for row in range(rows)])

    def __mul__(self, other):
        m = len(self[0])
        n = len(self)
        p = len(other[0])
        if m != len(other):
            raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix B is {}x{}.'
                             .format(n, m, len(other), p))

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

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

    def __getitem__(self, item):
```

```python
        return self.data[item]

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

    def is_positive_definite(self):
        A = copy.deepcopy(self.data)
        n = len(A)
        for j in range(n):
            if A[j][j] <= 0:
                return False
            A[j][j] = math.sqrt(A[j][j])
            for i in range(j + 1, n):
                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):
        rows = len(self)
        cols = len(self[0])
        return Matrix([[self.data[row][col] for row in range(rows)] for col in range(cols)])

    def mirror_horizontal(self):
        rows = len(self)
        cols = len(self[0])
        return Matrix([[self.data[rows - row - 1][col] for col in range(cols)] for row in range(rows)])

    def empty_copy(self):
        return Matrix.empty(len(self), len(self[0]))

    @staticmethod
    def multiply(*matrices):
        n = len(matrices[0])
        product = Matrix.identity(n)
        for matrix in matrices:
            product = product * matrix
        return product

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

        :param rows: number of rows
        :param cols: number of columns
        :return: the empty matrix
        """
        return Matrix([[0 for col in range(cols)] for row in range(rows)])

    @staticmethod
    def identity(n):
        return Matrix.diagonal_single_value(1, n)

    @staticmethod
    def diagonal(values):
        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):
        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):
        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: Choleski decomposition.

```
from __future__ import division

import math

from matrices import Matrix


def choleski_solve(A, b, half_bandwidth=None):
    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):
    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):  # TODO: Keep limited band in memory
    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, min(j + half_bandwidth, 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 back_substitution(L, x, y):
    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 3: Linear resistive networks.

```python
from __future__ import division

import csv
from matrices import Matrix
from choleski import choleski_solve


def solve_linear_network(A, Y, J, E, half_bandwidth=None):
    A_new = A * Y * A.transpose()
    b = A * (J - Y * E)
    return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)


def csv_to_network_branch_matrices(filename):
    with open(filename, 'r') as csv_file:
        reader = csv.reader(csv_file)
        J = []
        R = []
        E = []
        for row in reader:
            J_k = float(row[0])
            R_k = float(row[1])
            E_k = float(row[2])
            J.append(J_k)
            R.append(1 / R_k)
            E.append(E_k)
        Y = Matrix.diagonal(R)
        J = Matrix.column_vector(J)
        E = Matrix.column_vector(E)
        return Y, J, E


def create_network_matrices_mesh(rows, cols, branch_resistance, test_current):
    num_horizontal_branches = (cols - 1) * rows
    num_vertical_branches = (rows - 1) * cols
    num_branches = num_horizontal_branches + num_vertical_branches + 1
    num_nodes = rows * cols - 1

    A = create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
    ↪  num_vertical_branches)
    Y, J, E = create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current)

    return A, Y, J, E


def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
↪  num_vertical_branches):
    A = Matrix.empty(num_nodes, num_branches)
    node_offset = -1
    for branch in range(num_horizontal_branches):
        if branch == num_horizontal_branches - cols + 1:
            A[branch + node_offset + 1][branch] = 1
        else:
            if branch % (cols - 1) == 0:
                node_offset += 1
            node_number = branch + node_offset
            A[node_number][branch] = -1
            A[node_number + 1][branch] = 1
    branch_offset = num_horizontal_branches
    node_offset = cols
    for branch in range(num_vertical_branches):
        if branch == num_vertical_branches - cols:
            node_offset -= 1
            A[branch][branch + branch_offset] = 1
        else:
            A[branch][branch + branch_offset] = 1
            A[branch + node_offset][branch + branch_offset] = -1
    if num_branches == 2:
```

```
        A[0][1] = -1
    else:
        A[cols - 1][num_branches - 1] = -1
    return A


def create_network_branch_matrices_mesh(num_branches, resistance, test_current):
    Y = Matrix.diagonal([1 / resistance if branch < num_branches - 1 else 0 for branch in range(num_branches)])
    # Negative test current here because we assume current is coming OUT of the test current node.
    J = Matrix.column_vector([0 if branch < num_branches - 1 else -test_current for branch in range(num_branches)])
    E = Matrix.column_vector([0 for branch in range(num_branches)])
    return Y, J, E


def find_mesh_resistance(n, branch_resistance, half_bandwidth=None):
    test_current = 0.01
    A, Y, J, E = create_network_matrices_mesh(n, 2 * n, branch_resistance, test_current)
    x = solve_linear_network(A, Y, J, E, half_bandwidth=half_bandwidth)
    test_voltage = x[2 * n - 1 if n > 1 else 0][0]
    equivalent_resistance = test_voltage / test_current
    return equivalent_resistance
```

Listing 4: Finite difference method.

```
from __future__ import division

import copy
import random
from abc import ABCMeta, abstractmethod

import time

import math

from matrices import Matrix


class Relaxer:
    __metaclass__ = ABCMeta

    @abstractmethod
    def relax(self, phi, i, j):
        raise NotImplementedError


class SimpleRelaxer(Relaxer):
    """Relaxer which can represent a Jacobi relaxer, if the 'old' phi is given, or a Gauss-Seidel relaxer, if phi is
    modified in place."""
    def relax(self, phi, i, j):
        return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4


class SuccessiveOverRelaxer(Relaxer):
    def __init__(self, omega):
        self.gauss_seidel = SimpleRelaxer()
        self.omega = omega

    def relax(self, phi, i, j):
        return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, i, j)


class Boundary:
    __metaclass__ = ABCMeta

    @abstractmethod
    def potential(self):
```

```python
        raise NotImplementedError

    @abstractmethod
    def contains_point(self, x, y):
        raise NotImplementedError


class OuterConductorBoundary(Boundary):
    def potential(self):
        return 0

    def contains_point(self, x, y):
        return x == 0 or y == 0 or x == 0.2 or y == 0.2


class QuarterInnerConductorBoundary(Boundary):
    def potential(self):
        return 15

    def contains_point(self, x, y):
        return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12


class Guesser:
    __metaclass__ = ABCMeta

    def __init__(self, minimum, maximum):
        self.minimum = minimum
        self.maximum = maximum

    @abstractmethod
    def guess(self, x, y):
        raise NotImplementedError


class RandomGuesser(Guesser):
    def guess(self, x, y):
        return random.randint(self.minimum, self.maximum)


class LinearGuesser(Guesser):
    def guess(self, x, y):
        return 150 * x if x < 0.06 else 150 * y


def radial(k, x, y, x_source, y_source):
    return k / (math.sqrt((x_source - x)**2 + (y_source - y)**2))


class RadialGuesser(Guesser):
    def guess(self, x, y):
        return 0.0225 * (radial(20, x, y, 0.1, 0.1) - radial(1, x, y, 0, y) - radial(1, x, y, x, 0))


class CoaxialCableMeshConstructor:
    def __init__(self):
        outer_boundary = OuterConductorBoundary()
        inner_boundary = QuarterInnerConductorBoundary()
        self.boundaries = (inner_boundary, outer_boundary)
        self.guesser = RadialGuesser(0, 15)
        self.boundary_size = 0.2

    def construct_simple_mesh(self, h):
        num_mesh_points_along_axis = int(self.boundary_size / h) + 1
        phi = Matrix.empty(num_mesh_points_along_axis, num_mesh_points_along_axis)
        for i in range(num_mesh_points_along_axis):
            y = i * h
            for j in range(num_mesh_points_along_axis):
                x = j * h
                boundary_pt = False
```

```
                for boundary in self.boundaries:
                    if boundary.contains_point(x, y):
                        boundary_pt = True
                        phi[i][j] = boundary.potential()
                if not boundary_pt:
                    phi[i][j] = self.guesser.guess(x, y)
        return phi

    def construct_symmetric_mesh(self, h):
        max_index = int(0.1 / h) + 2  # Only need to store up to middle
        phi = Matrix.empty(max_index, max_index)
        for i in range(max_index):
            y = i * h
            for j in range(max_index):
                x = j * h
                boundary_pt = False
                for boundary in self.boundaries:
                    if boundary.contains_point(x, y):
                        boundary_pt = True
                        phi[i][j] = boundary.potential()
                if not boundary_pt:
                    phi[i][j] = self.guesser.guess(x, y)
        return phi


def point_to_indices(x, y, h):
    i = int(y / h)
    j = int(x / h)
    return i, j


class IterativeRelaxer:
    def __init__(self, relaxer, epsilon, phi, h):
        self.relaxer = relaxer
        self.epsilon = epsilon
        self.phi = phi
        self.boundary = QuarterInnerConductorBoundary()
        self.h = h
        self.num_iterations = 0
        self.rows = len(phi)
        self.cols = len(phi[0])
        self.mid_index = int(0.1 / h)

    def relaxation_jacobi(self):
        # t = time.time()

        while not self.convergence():
            self.num_iterations += 1

            last_row = [0] * (self.cols - 1)
            for i in range(1, self.rows - 1):
                y = i * self.h
                for j in range(1, self.cols - 1):
                    x = j * self.h
                    if not self.boundary.contains_point(x, y):
                        last_val = last_row[j - 2] if j > 1 else 0
                        relaxed_value = (self.phi[i + 1][j] + last_row[j - 1] + self.phi[i][j + 1] +
                         ↪  last_val) / 4
                        last_row[j - 1] = self.phi[i][j]
                        self.phi[i][j] = relaxed_value
                        if i == self.mid_index - 1:
                            self.phi[i + 2][j] = relaxed_value
                        elif j == self.mid_index - 1:
                            self.phi[i][j + 2] = relaxed_value

        # print('Runtime: {} s'.format(time.time() - t))

    def relaxation_sor(self):
        while not self.convergence():
            self.num_iterations += 1
```

```
            for i in range(1, self.rows - 1):
                y = i * self.h
                for j in range(1, self.cols - 1):
                    x = j * self.h
                    if not self.boundary.contains_point(x, y):
                        relaxed_value = self.relaxer.relax(self.phi, i, j)
                        self.phi[i][j] = relaxed_value
                        if i == self.mid_index - 1:
                            self.phi[i + 2][j] = relaxed_value
                        elif j == self.mid_index - 1:
                            self.phi[i][j + 2] = relaxed_value

    def convergence(self):
        max_i, max_j = point_to_indices(0.1, 0.1, self.h)
        # Only need to compute for 1/4 of grid
        for i in range(1, max_i + 1):
            y = i * self.h
            for j in range(1, max_j + 1):
                x = j * self.h
                if not self.boundary.contains_point(x, y) and self.residual(i, j) >= self.epsilon:
                    return False
        return True

    def residual(self, i, j):
        return abs(self.phi[i+1][j] + self.phi[i-1][j] + self.phi[i][j+1] + self.phi[i][j-1] - 4 *
        ↪  self.phi[i][j])

    def get_potential(self, x, y):
        i, j = point_to_indices(x, y, self.h)
        return self.phi[i][j]

    def print_grid(self):
        header = ''
        for j in range(len(self.phi[0])):
            y = j * self.h
            header += '{:6.2f} '.format(y)
        print(header)
        print(self.phi)
        # for i in range(len(self.phi)):
        #     x = i * self.h
        #     print('{:6.2f} '.format(x))


def successive_over_relaxation(omega, epsilon, phi, h):
    relaxer = SuccessiveOverRelaxer(omega)
    iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
    iter_relaxer.relaxation_sor()
    return iter_relaxer


def jacobi_relaxation(epsilon, phi, h):
    relaxer = SimpleRelaxer()
    iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
    iter_relaxer.relaxation_jacobi()
    return iter_relaxer
```

Listing 5: Question 1.

```
from __future__ import division

from linear_networks import solve_linear_network, csv_to_network_branch_matrices
from choleski import choleski_solve
from matrices import Matrix

NETWORK_DIRECTORY = 'network_data'

L_2 = Matrix([
    [5, 0],
    [1, 3]
```

```
])
L_3 = Matrix([
    [3, 0, 0],
    [1, 2, 0],
    [8, 5, 1]
])
L_4 = Matrix([
    [1, 0, 0, 0],
    [2, 8, 0, 0],
    [5, 5, 4, 0],
    [7, 2, 8, 7]
])
matrix_2 = L_2 * L_2.transpose()
matrix_3 = L_3 * L_3.transpose()
matrix_4 = L_4 * L_4.transpose()
positive_definite_matrices = [matrix_2, matrix_3, matrix_4]

x_2 = Matrix.column_vector([8, 3])
x_3 = Matrix.column_vector([9, 4, 3])
x_4 = Matrix.column_vector([5, 4, 1, 9])
xs = [x_2, x_3, x_4]


def q1b():
    print('=== Question 1(b) ===')
    for count, A in enumerate(positive_definite_matrices):
        n = count + 2
        print('n={} matrix is positive-definite: {}'.format(n, A.is_positive_definite()))


def q1c():
    print('=== Question 1(c) ===')
    for x, A in zip(xs, positive_definite_matrices):
        b = A * x
        # print('A: {}'.format(A))
        # print('b: {}'.format(b))

        x_choleski = choleski_solve(A, b)
        print('Expected x: {}'.format(x))
        print('Actual x: {}'.format(x_choleski))  # TODO: Assert equal here (to number of sig figs)


def q1d():
    print('=== Question 1(d) ===')
    for i in range(1, 6):
        A = Matrix.csv_to_matrix('{}/incidence_matrix_{}.csv'.format(NETWORK_DIRECTORY, i))
        Y, J, E = csv_to_network_branch_matrices('{}/network_branches_{}.csv'.format(NETWORK_DIRECTORY,
        ↪  i))
        # print('Y: {}'.format(Y))
        # print('J: {}'.format(J))
        # print('E: {}'.format(E))
        x = solve_linear_network(A, Y, J, E)
        print('Solved for x in network {}: {}'.format(i, x))  # TODO: Create my own test circuits here


def q1():
    q1b()
    q1c()
    q1d()


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

Listing 6: Question 2.

```
import time

import matplotlib.pyplot as plt
```

```python
from matplotlib.ticker import MaxNLocator

from linear_networks import find_mesh_resistance


def find_mesh_resistances(banded=False):
    branch_resistance = 1000
    points = {}
    runtimes = {}
    for n in range(2, 11):
        start_time = time.time()
        half_bandwidth = 2 * n + 1 if banded else None
        equivalent_resistance = find_mesh_resistance(n, branch_resistance, half_bandwidth=half_bandwidth)
        print('Equivalent resistance for {}x{} mesh: {:.2f} Ohms.'.format(n, 2 * n,
        ↪   equivalent_resistance))
        points[n] = equivalent_resistance
        runtime = time.time() - start_time
        runtimes[n] = runtime
        print('Runtime: {} s.'.format(runtime))
    plot_runtime(runtimes, banded)
    return points, runtimes


def q2ab():
    print('=== Question 2(a)(b) ===')
    return find_mesh_resistances(banded=False)


def q2c():
    print('=== Question 2(c) ===')
    return find_mesh_resistances(banded=True)


def plot_runtime(points, banded):
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = points.keys()
    y_range = points.values()
    plt.plot(x_range, y_range, '{}o-'.format('r' if banded else ''))
    plt.xlabel('N')
    plt.ylabel('Runtime (s)')
    plt.grid(True)
    f.savefig('report/plots/q2{}.pdf'.format('c' if banded else 'b'), bbox_inches='tight')


def plot_runtimes(points1, points2):
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = points1.keys()
    y_range = points1.values()
    y_banded_range = points2.values()
    plt.plot(x_range, y_range, 'o-', label='Non-banded elimination')
    plt.plot(x_range, y_banded_range, 'ro-', label='Banded elimination')
    plt.xlabel('N')
    plt.ylabel('Runtime (s)')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q2bc.pdf', bbox_inches='tight')


def q2d(points):
    print('=== Question 2(d) ===')
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = points.keys()
    y_range = points.values()
    plt.plot(x_range, y_range, 'o-', label='Resistance')
```

```python
    plt.xlabel('N')
    plt.ylabel('R ($\Omega$)')
    plt.grid(True)
    # plt.legend()
    # plt.show()
    f.savefig('report/plots/q2d.pdf', bbox_inches='tight')


def q2():
    _, runtimes1 = q2ab()
    pts, runtimes2 = q2c()
    plot_runtimes(runtimes1, runtimes2)
    q2d(pts)


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

Listing 7: Question 3.

```python
from __future__ import division

import csv

import matplotlib.pyplot as plt
import time

from finite_diff import CoaxialCableMeshConstructor, successive_over_relaxation, jacobi_relaxation

epsilon = 0.00001
x = 0.06
y = 0.04

NUM_H_ITERATIONS = 5


def q3b():
    print('=== Question 3(b) ===')
    h = 0.02
    min_num_iterations = float('inf')
    best_omega = float('inf')

    omegas = []
    num_iterations = []
    potentials = []

    for omega_diff in range(10):
        omega = 1 + omega_diff / 10
        print('Omega: {}'.format(omega))
        phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
        print('Initial guess:')
        print(phi.mirror_horizontal())
        iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
        # print(iter_relaxer.phi)
        print('Num iterations: {}'.format(iter_relaxer.num_iterations))
        potential = iter_relaxer.get_potential(x, y)
        print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
        if iter_relaxer.num_iterations < min_num_iterations:
            best_omega = omega
        min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)

        omegas.append(omega)
        num_iterations.append(iter_relaxer.num_iterations)
        potentials.append('{:.3f}'.format(potential))
        print('Relaxed:')
        print(phi.mirror_horizontal())

    print('Best number of iterations: {}'.format(min_num_iterations))
    print('Best omega: {}'.format(best_omega))
```

```python

    f = plt.figure()
    x_range = omegas
    y_range = num_iterations
    plt.plot(x_range, y_range, 'o-', label='Number of iterations')
    plt.xlabel('$\omega$')
    plt.ylabel('Number of Iterations')
    plt.grid(True)
    f.savefig('report/plots/q3b.pdf', bbox_inches='tight')

    save_rows_to_csv('report/csv/q3b_potential.csv', zip(omegas, potentials), header=('Omega', 'Potential (V)'))
    save_rows_to_csv('report/csv/q3b_iterations.csv', zip(omegas, num_iterations), header=('Omega', 'Iterations'))

    return best_omega


def q3c(omega):
    print('=== Question 3(c): SOR ===')
    h = 0.04
    h_values = []
    potential_values = []
    iterations_values = []
    for i in range(NUM_H_ITERATIONS):
        h = h / 2
        print('h: {}'.format(h))
        print('1/h: {}'.format(1 / h))
        phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
        iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
        # print(phi.mirror_horizontal())
        potential = iter_relaxer.get_potential(x, y)
        num_iterations = iter_relaxer.num_iterations

        print('Num iterations: {}'.format(num_iterations))
        print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))

        h_values.append(1 / h)
        potential_values.append('{:.3f}'.format(potential))
        iterations_values.append(num_iterations)

    f = plt.figure()
    x_range = h_values
    y_range = potential_values
    plt.plot(x_range, y_range, 'o-', label='Potential at (0.06, 0.04)')
    plt.xlabel('1 / h')
    plt.ylabel('Potential at [0.06, 0.04] (V)')
    plt.grid(True)
    f.savefig('report/plots/q3c_potential.pdf', bbox_inches='tight')

    f = plt.figure()
    x_range = h_values
    y_range = iterations_values
    plt.plot(x_range, y_range, 'o-', label='Number of Iterations')
    plt.xlabel('1 / h')
    plt.ylabel('Number of Iterations')
    plt.grid(True)
    f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')

    save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h', 'Potential (V)'))
    save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h', 'Iterations'))

    return h_values, potential_values, iterations_values


def q3d():
    print('=== Question 3(d): Jacobi ===')
    h = 0.04
```

```
    h_values = []
    potential_values = []
    iterations_values = []
    for i in range(NUM_H_ITERATIONS):
        h = h / 2
        print('h: {}'.format(h))
        phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
        iter_relaxer = jacobi_relaxation(epsilon, phi, h)
        potential = iter_relaxer.get_potential(x, y)
        num_iterations = iter_relaxer.num_iterations

        print('Num iterations: {}'.format(num_iterations))
        print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))

        h_values.append(1 / h)
        potential_values.append('{:.3f}'.format(potential))
        iterations_values.append(num_iterations)

    f = plt.figure()
    x_range = h_values
    y_range = potential_values
    plt.plot(x_range, y_range, 'ro-', label='Potential at (0.06, 0.04)')
    plt.xlabel('1 / h')
    plt.ylabel('Potential at [0.06, 0.04] (V)')
    plt.grid(True)
    f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')

    f = plt.figure()
    x_range = h_values
    y_range = iterations_values
    plt.plot(x_range, y_range, 'ro-', label='Number of Iterations')
    plt.xlabel('1 / h')
    plt.ylabel('Number of Iterations')
    plt.grid(True)
    f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')

    save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h', 'Potential (V)'))
    save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h', 'Iterations'))

    return h_values, potential_values, iterations_values


def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values, iterations_values_jacobi):
    f = plt.figure()
    plt.plot(h_values, potential_values, 'o-', label='SOR')
    plt.plot(h_values, potential_values_jacobi, 'ro-', label='Jacobi')
    plt.xlabel('1 / h')
    plt.ylabel('Potential at [0.06, 0.04] (V)')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')

    f = plt.figure()
    plt.plot(h_values, iterations_values, 'o-', label='SOR')
    plt.plot(h_values, iterations_values_jacobi, 'ro-', label='Jacobi')
    plt.xlabel('1 / h')
    plt.ylabel('Number of Iterations')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')


def save_rows_to_csv(filename, rows, header=None):
    with open(filename, "wb") as f:
        writer = csv.writer(f)
        if header is not None:
            writer.writerow(header)
```

```
        for row in rows:
            writer.writerow(row)


def q3():
    o = q3b()
    h_values, potential_values, iterations_values = q3c(o)
    _, potential_values_jacobi, iterations_values_jacobi = q3d()
    plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
     ↪  iterations_values_jacobi)


if __name__ == '__main__':
    t = time.time()
    q3()
    print('Total runtime: {}'.format(time.time() - t))
```