ECSE 543: Numerical Methods

Assignment 1

Wenjie Wei
260685967

October 18, 2018

Contents

1 Introduction

All programs in this assignment are written and compiled with Python 3.6. This report is structured so that the individual problems are answered in respective sections. The python codes used to solve the assignment problems are attached in the appendices, with the file names labeled at the top of the code segments.

2 Choleski Decomposition

2.a Choleski Implementation

The implementation of Choleski decomposition is shown in Listing 2. There are two methods defined in `choleski.py`: `check_choleski(A, b, x)` and `choleski_decomposition(A, b)`. The latter method takes two matrices `A` and `b` as arguments, and returns `x` as the computational result of the decomposition. The first method takes these three matrices as arguments, and performs matrix production to check the result of

$$Ax = b$$

The precision of the equality is set to 0.001, as the program may end up with results with uncertainties with a quantity level of 10^{-8}.

2.b Simple Tester Matrices

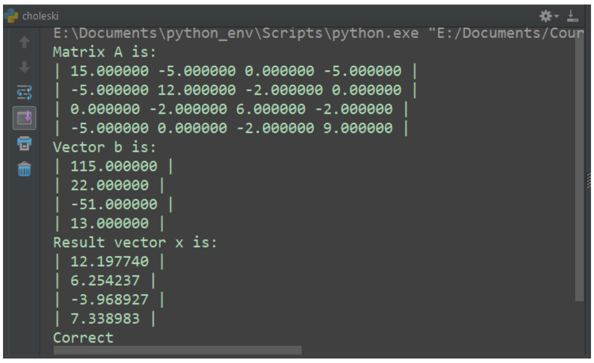
To test the functionality of the program, we construct tester matrices with size varying from 2 to 10. The matrices are constructed to be symmetric, positive definite.

To construct the testing matrices, start from the fact that A must be symmetric, positive definite if cholesky decomposition succeeds. Thus, we do the reverse process by constructing a lower-triangular matrix L, and thus obtain A by $A = LL^T$. Below are several test matrices, and the results of the running are shown in Figures 1 and 2.

Testing matrices 1:

$$\begin{bmatrix} 15 & -5 & 0 & -5 \\ -5 & 12 & -2 & 0 \\ 0 & -2 & 6 & -2 \\ -5 & 0 & -2 & 9 \end{bmatrix} x = \begin{bmatrix} 115 \\ 22 \\ -51 \\ 13 \end{bmatrix}$$

Output results:



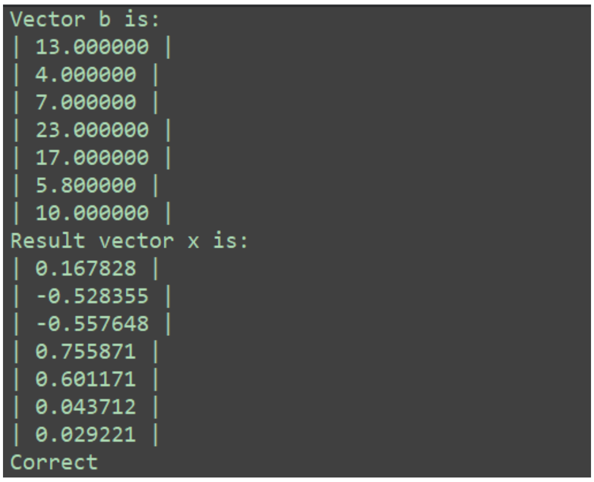
```
E:\Documents\python_env\Scripts\python.exe "E:/Documents/Cour
Matrix A is:
| 15.000000 -5.000000 0.000000 -5.000000 |
| -5.000000 12.000000 -2.000000 0.000000 |
| 0.000000 -2.000000 6.000000 -2.000000 |
| -5.000000 0.000000 -2.000000 9.000000 |
Vector b is:
| 115.000000 |
| 22.000000 |
| -51.000000 |
| 13.000000 |
Result vector x is:
| 12.197740 |
| 6.254237 |
| -3.968927 |
| 7.338983 |
Correct
```

Figure 1: Result of the First Choleski Decomposition Test

Testing matrix 2:

$$\begin{bmatrix} 38 & 23 & 31 & 22 & 29 & 25 & 31 \\ 23 & 44 & 36 & 27 & 35 & 24 & 33 \\ 31 & 36 & 65 & 36 & 45 & 34 & 45 \\ 22 & 27 & 36 & 46 & 29 & 15 & 27 \\ 29 & 35 & 45 & 29 & 52 & 32 & 39 \\ 25 & 24 & 34 & 15 & 32 & 37 & 36 \\ 31 & 33 & 45 & 27 & 39 & 36 & 65 \end{bmatrix} x = \begin{bmatrix} 13 \\ 4 \\ 7 \\ 23 \\ 17 \\ 5.8 \\ 10 \end{bmatrix}$$

Output results:



```
Vector b is:
| 13.000000 |
| 4.000000 |
| 7.000000 |
| 23.000000 |
| 17.000000 |
| 5.800000 |
| 10.000000 |
Result vector x is:
| 0.167828 |
| -0.528355 |
| -0.557648 |
| 0.755871 |
| 0.601171 |
| 0.043712 |
| 0.029221 |
Correct
```

Figure 2: Result of the Second Choleski Decompostion Test

2.c Testing of $Ax = b$

The outputs of two testing cases are shown in Figures 1 and 2 in the previous section. In the program implemented, there is a simple checking method which does a dot product of A and x and check if the results match the entries entered for b. Note that because of the reason of the Python interpreter, there is a tiny error with a quantity of

10^{-10}, therefore the precision is set to 10^{-5} to check the validity of the computation.

2.d Linear Resistive Networks

Linear resistive networks are now able to be solved by the Choleski decomposition implemented in the previous parts. Listing 3 shows the implementation of reading a circuit file with data organized in a .csv file.

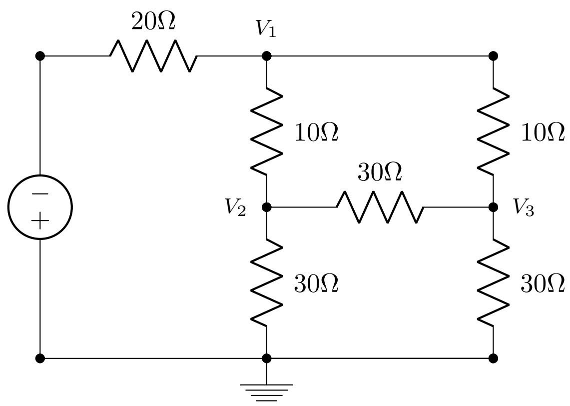


Figure 3: Test Circuit 5

```
wenjie@wenjie-XPS-13-9343:~/f18/numerical_method/a1$ python linearNetwork.py
| 5.000000 |
| 3.750000 |
| 3.750000 |
```

Figure 4: Result of the Testing Circuit 5

Take the 5th circuit provided by the TA for example, the circuit is shown in Figure 3, and the result of the running of the program on this circuit is shown in Figure 4.

The data of the circuit are organized in the way shown in Figure 5. The first line shows the

1	5	B	6	N	4
2	0	1	0	20	10
3	1	2	0	10	0
4	1	3	0	10	0
5	2	0	0	30	0
6	2	3	0	30	0
7	3	0	0	30	0

Figure 5: Circuit File Organizations

general information about the circuit, such as the circuit ID (the example shown in the figure is the 5th test circuit), number of branches, and number of nodes. The lines followed are the data of the branches, which contains the following data: the starting node, the end node, the current source J, the resistance R, and the voltage source E.

The convention of the input files should be well defined. In the program used for this test circuit, define the positive current direction is flowing from the start node to the end node. Current source must deliver positive current to the start node and the voltage source should deliver positive current to the end node. Following the conventions listed above, the program should be able to output desired node voltages in matrix form.

To verify the reliability of the program, four more simple test circuits are constructed. The input file as well as the result of the calculations are attached immediately after the circuit diagrams. The test runs below are proving that the program runs correctly as long as appropriate input files are passed into.

2.d.1 Testing Circuit 1

Figures 6 and 7 show the first test circuit. The desired output at node 1 can be calculated as $V_1 = 5V$, and the program is outputting the correct result.

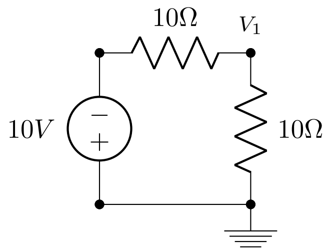


Figure 6: Test Circuit 1

```
wenjie@wenjie-XPS-13-9343:
| 5.000000 |
```

Figure 7: Output Result of the Testing Circuit 1

2.d.2 Testing Circuit 2

Figures 8 and 9 below show the testing circuit 2 and its result. The expected result is $V_1 = 50V$.

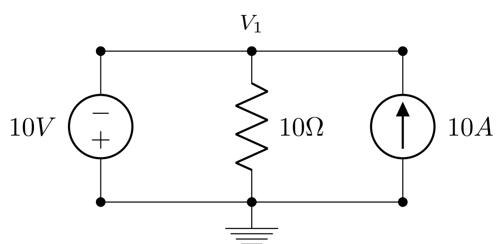


Figure 8: Test Circuit 2

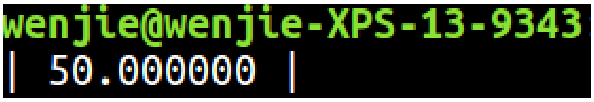


Figure 9: Output Result of the Testing Circuit 2

2.d.3 Testing Circuit 3

Figures 10 and 11 below show the results of the testing circuit 3. The expected result of the circuit is $V_1 = 55V$.

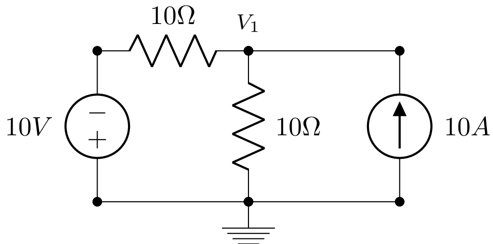


Figure 10: Test Circuit 3

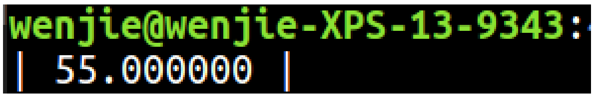


Figure 11: Output Result of the Testing Circuit 3

2.d.4 Testing Circuit 4

Figures 12 and 13 below show the results of the testing circuit 4. The expected results of the circuit is $V_1 = 20V$ and $V_2 = 35V$.

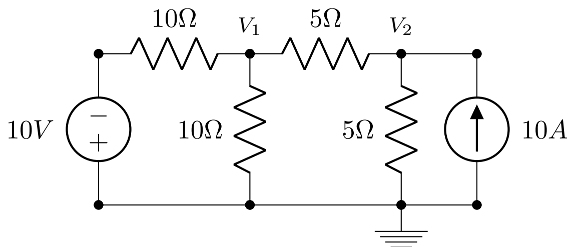


Figure 12: Test Circuit 4

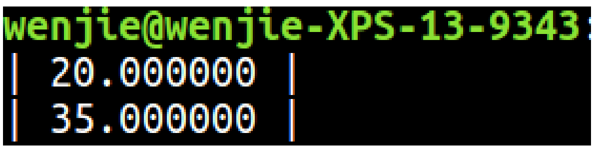


Figure 13: Output Result of the Testing Circuit 4

3 Resistor Mesh Network

3.a Implementation

The implementation of the program is shown in the appendix as Listing 3. The file firstly writes *.csv* files with N from 2 to 15, and then the program reads from the files constructed and compute the total resistance of the network.

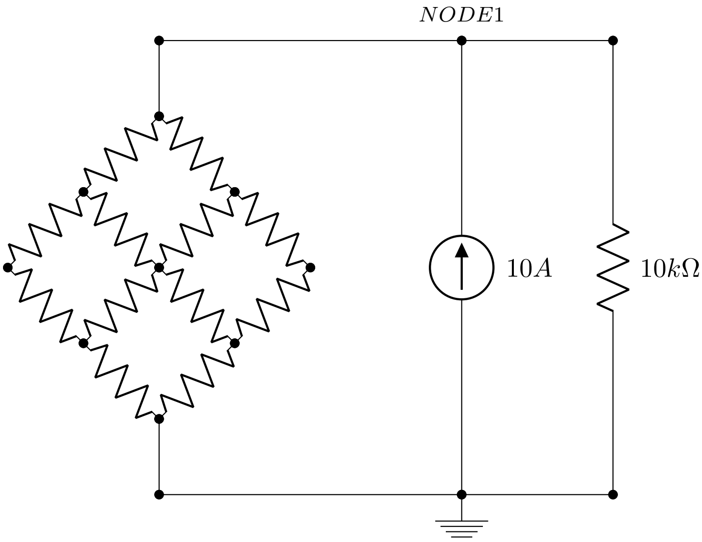


Figure 14: Resistor Mesh Example

Shown in Figure 14 is the circuit model that the program is using. The right most is a testing branch which provides both the source and resistance. The program calculates the node voltage at every node, the nodal voltage for $NODE$ 1.

With the nodal voltage, and the right most branch as a current divider, we perform the following calculation:

$$R_{mesh} = \frac{V_{node}}{10 - V_{node1}/10k\Omega}$$

3.b Theoretical Computing Time

Theoretically, the computing time of Choleski decomposition is $O(n^3)$, where n is the number of rows for matrix A in the equation

$$Ax = b$$

Since the number of branches B relates with the size N following: $B = N^2$, so theoretically, the computation time of this program is $O(N^6)$.

Figure 15 shows the computation time versus N. The formula found from the data is

$$t = 0.0001N^{5.58}$$

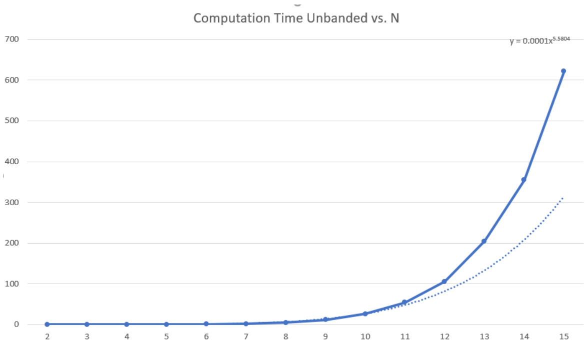


Figure 15: Mesh Resistance Calculation Time vs. N Unbanded

The curve generally agrees with the theoretical time $O(n^6)$.

3.c Sparse Matrix Computation Time

The theoretical computation time of a banded sparse matrix is $O(\bar{b}^2 n)$. By relating with the mesh size, the time complexity becomes $O(N^4)$.

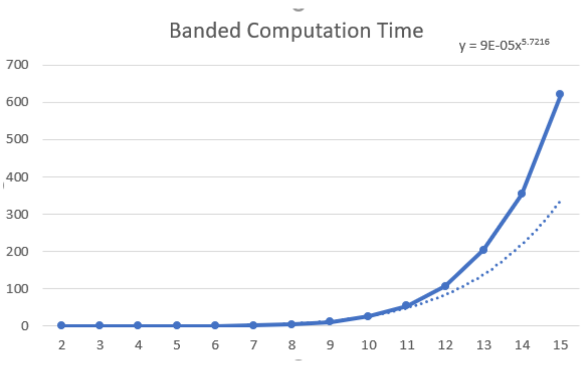


Figure 16: Mesh Resistance Calculation Time vs. N banded

Figure 16 shows the computation time for banded solution. My result does not agree with the theoretical computation time. I consider that there exists a lot of overheads during computations which slows down the overall time. For example, when I am using the banded algorithm, I keeps checking if my element of interest is about to exceed my half-bandwidth, which can cause a very big process overhead.

3.d Resistance vs. Mesh Size

Use the resistance computed by the program, plot the resistance R vs. N. Figure 17 shows the resistance changes versus the change of the mesh size.

The formula derived from this data set is found to be

$$R = 9880.3log(N) + 8452.9$$

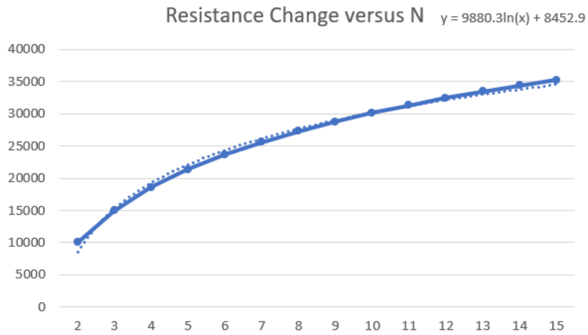


Figure 17: Mesh Resistance

4 Finite Difference for the Coaxial Cable

4.a Implementation of SOR

Method `successive_over_relaxation` in Listing 4 shows the implementation of successive over relaxation for the coaxial cable.

By using the symmetry of the problem. the implemented program constructs the quarter in the second quadrant of the cable. By doing this, the memory required to run the program is quartered.

4.b Effects of Varying ω

Fix the node distance $h = 0.02$, run the implemented program to get the voltage at $(0.06, 0.04)$. Because of symmetry, the program computes the value of the voltage at $(0.06, 0.16)$, which equals to the value at $(0.06, 0.04)$. Figure 18 shows how the number of iterations change with the change of ω. From the graph, we can tell that the optimal *omega* has a value of 1.3 since the number of iterations reaches the lowest point of 26 iterations.

4.c Effects of Varying h

With the optimal point $\omega = 1.3$, vary the distance between the nodes. From $h = 0.02$, decrease the distance by a factor of 2, until $h = 0.00125$. The data recorded are shown in Table 1, and the plots are shown in Figure 19 and 20.

From the two figures above, we can see that the precision of the calculation result is increasing,

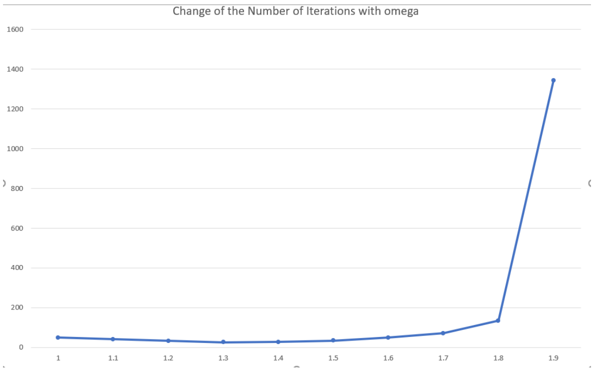


Figure 18: Change of Number of Iterations with ω

Table 1: Potential at (0.06, 0.04) versus h when using the SOR method.

h	1/h	value	iterations
0.02	50	40.52648569	26
0.01	100	39.23825657	96
0.005	200	38.78818592	341
0.0025	400	38.61727504	1195
0.00125	800	38.5479162	4141

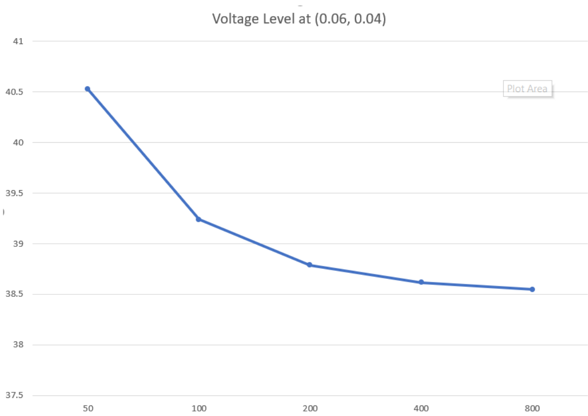


Figure 19: Change of Voltage Levels versus 1/h

with the price of a very big increase in the calculation time. Therefore, a trade-off decision should be made to decide if the increase of time is well-worthy for the precision. As far as I am concerned, the precision using $h = 0.005$ is enough for the purpose of getting a general idea of the voltage level. After $h = 0.005$, the increase in precision is limited, while the computation time sky-rockets.

4.d Varying h using the Jacobi Method

In Listing 4, the method `jacobi` implements the Jacobi method. Export the computing results to a

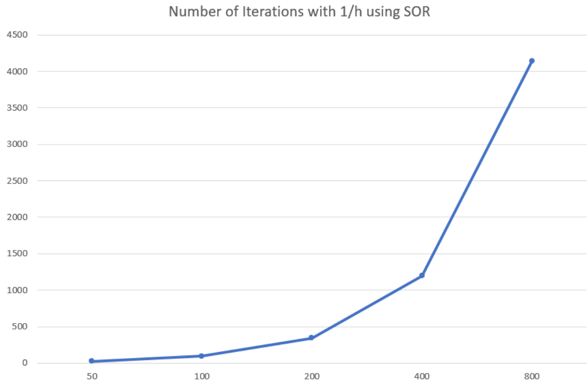


Figure 20: Number of Iterations versus 1/h

.csv file, and the results are shown in Table 2, and the plots of iterations and voltage levels are shown in Figures

Table 2: Potential at (0.06, 0.04) versus h when using Jacobi method.

h	1/h	value	iterations
0.02	50	40.52648913	88
0.01	100	39.23827224	337
0.005	200	38.78822284	1226
0.0025	400	38.61736454	4365
0.00125	800	38.54813767	15264

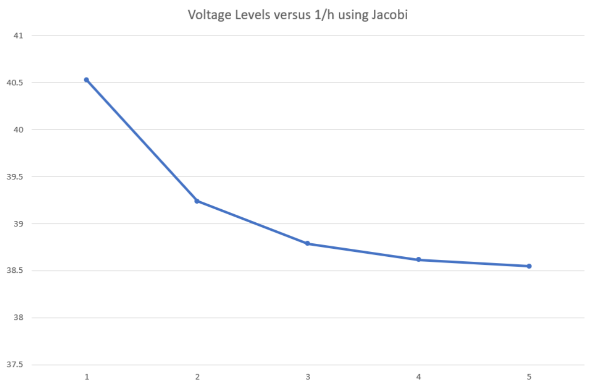


Figure 21: Change of Voltage Levels versus 1/h using Jacobi Method

From the two graphs, we can easily tell that the final calculated value is valid using Jacobi Method, but meanwhile it consumes much more computational time comparing with the successive over relaxation method. Similar to the SOR method, with the increase of precision (decrease of nodal distance), the computation time grows very rapidly.

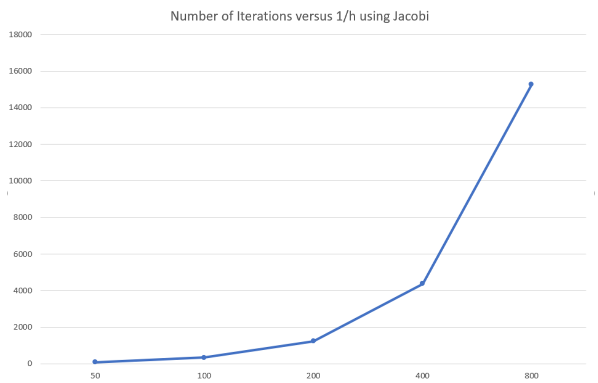


Figure 22: Number of Iterations versus 1/h using Jacobi Method

A Code Listings

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

```python
import math


class Matrix(object):
    def __init__(self, vec, rows, cols):
        self._vec = vec
        self._rows = rows
        self._cols = cols

    def set_row(self, n_rows):
        self._rows = n_rows

    def is_square(self):
        return self._rows == self._cols

    def is_symmetric(self):
        if not self.is_square():
            return False

        else:
            for i in range(self.rows):
                for j in range(self.cols):
                    if self[i][j] != self.T[i][j]:
                        return False

        return True

    def transpose(self):
        vec_trans = [[None for _ in range(self.rows)] for _ in range(self.cols)]
        for x in range(self.cols):
            for y in range(self.rows):
                vec_trans[x][y] = self.vec[y][x]

        transposed_matrix = Matrix(vec_trans, self.cols, self.rows)
        return transposed_matrix

    def minus(self, other):
        if self.cols != other.cols or self.rows != other.rows:
            raise ValueError("Incorrect dimension for matrix subtraction.")

        result_vec = [[None for _ in range(self.cols)] for _ in range(self.rows)]
        result = Matrix(result_vec, self.rows, self.cols)
        for i in range(self.rows):
            for j in range(self.cols):
                result[i][j] = self[i][j] - other[i][j]

        return result

    def dot_product(self, other):
        if self.cols != other.rows:
            raise ValueError("Incorrect dimension for matrix multiplication.")

        result_vec = [[None for _ in range(other.cols)] for _ in range(self.rows)]
        result = Matrix(result_vec, self.rows, other.cols)

        for i in range(self.rows):
            for j in range(other.cols):
                temp_sum = 0
                for k in range(other.rows):
                    temp_sum += self[i][k] * other[k][j]
                result[i][j] = temp_sum

        return result

    def __getitem__(self, item_number):
```

```
        if isinstance(item_number, int):
            return self._vec[item_number]

        if isinstance(item_number, tuple):
            x, y = item_number
            # use some "dummy entries" as a buffer to decrease the possibility of occurring out of
↪  boundary.
            if x < 0 or x >= self.rows or y < 0 or y >= self.cols:
                return 0
            else:
                return self._vec[x][y]

    def clone(self):
        cloned_matrix = Matrix(self.vec, self.rows, self.cols)
        return cloned_matrix

    def print_matrix(self):
        for i in range(self.rows):
            print("|", end=" ")
            for j in range(self.cols):
                print("%f" % self[i][j], end=" ")
            print("|")

    @property
    def vec(self):
        return self._vec

    @property
    def rows(self):
        return self._rows

    @property
    def cols(self):
        return self._cols

    @property
    def T(self):
        return self.transpose()
```

Listing 2: Choleski decomposition (`choleski.py`).

```
import math
from matrix import Matrix


def check_choleski(A, b, x):
    """
    This method checks if the result of the choleski decomposition is correct.
    Precision is set to 0.001.

    :param A: n by n matrix A
    :param b: result vector, n by 1
    :param x: x vector, n by 1

    :return: True if the result is correct, other wise False
    """
    temp_result = A.dot_product(x)
    print("Matrix A is:")
    A.print_matrix()
    print("Vector b is:")
    b.print_matrix()
    print("Result vector x is:")
    x.print_matrix()

    for i in range(temp_result.rows):
        for j in range(temp_result.cols):
            if abs(temp_result[i][j] - b[i][j]) >= 0.001:
                return False
    return True
```

```python


def solve_chol(A, b, half_bandwidth=None):
    """
    This is the method implemented for solving the problem Ax = b,
    using Choleski Decomposition.

    Arguments:
        A: the matrix A, a real, S.P.D. (Symmetric positive definite) n * n matrix.
        b: Column vector with n rows.
        half_bandwidth: the half bandwidth of A.

    Returns:
        Column vector x with n rows.
    """
    if not A.is_symmetric():
        raise ValueError("Matrix must be symmetric to perform Choleski Decomposition.\n")

    if half_bandwidth is None:
        L = decomposition(A, half_bandwidth)

        # Now L and LT are all obtained, we can move to forward elimination
        y = forward_elimination(L, b, half_bandwidth)

        # Now perform back substitution to find x.
        v = backward_substitution(L, y, half_bandwidth)

    else:
        v = elimination(A, b, half_bandwidth)

    return v


def decomposition(A, half_bandwidth=None):
    n = A.rows
    empty_matrix = [[0 for _ in range(n)] for _ in range(n)]
    L = Matrix(empty_matrix, n, n)

    if half_bandwidth is None:
        for j in range(n):
            if A[j][j] <= 0:
                raise ValueError("Matrix is not positive definite.\n")

            temp_sum = 0
            for k in range(-1, j):
                temp_sum += math.pow(L[j][k], 2)
            if (A[j][j] - temp_sum) < 0:
                raise ValueError("Operand under square root is not positive. Matrix is not positive definite, exiting.")
            L[j][j] = math.sqrt(A[j][j] - temp_sum)

            for i in range(j + 1, n):
                temp_sum = 0
                for k in range(-1, j):
                    temp_sum += L[i][k] * L[j][k]
                L[i][j] = (A[i][j] - temp_sum) / L[j][j]
    else:
        for j in range(n):
            if A[j][j] <= 0:
                raise ValueError("Matrix is not positive definite.\n")

            temp_sum = 0
            k = j + 1 - half_bandwidth
            if k < 0:
                k = 0
            while k < j:
                temp_sum += math.pow(L[j][k], 2)
                k += 1

            if (A[j][j] - temp_sum) < 0:
```

```
                raise ValueError("Operand under the square root is not positive, matrix is not P.D. exiting")
            # Write the diagonal entry to matrix L
            L[j][j] = math.sqrt(A[j][j] - temp_sum)

            # Now we have found the diagonal entry
            # we move to calculate the entries below the diagonal entry, covered by HB.

            # Scenario 1: all entries below Ljj that are covered by HB are with the matrix bound.
            # However, some entries to the left covered by HB are out of bounds.
            # Scenario 2: all entries below and to the left of Ljj covered by HB are within the matrix bounds.
            # Scenario 3: some entries below Ljj are out of bounds,
            # but the entries to the left are within bounds.
            for i in range(j + 1, j + half_bandwidth):
                if i >= n:
                    break
                temp_sum = 0
                k = j + 1 - half_bandwidth
                if k < 0:
                    k = 0
                while k < j:
                    temp_sum += L[i][k] * L[j][k]
                    k += 1
                L[i][j] = (A[i][j] - temp_sum) / L[j][j]

    return L


def forward_elimination(L, b, half_bandwidth=None):
    n = L.rows
    y_vec = [[None for _ in range(1)] for _ in range(n)]
    y = Matrix(y_vec, n, 1)

    if half_bandwidth is None:
        for i in range(y.rows):
            temp_sum = 0
            if i > 0:
                for j in range(i):
                    temp_sum += L[i][j] * y[j][0]
                y[i][0] = (b[i][0] - temp_sum) / L[i][i]
            else:
                y[i][0] = b[i][0] / L[i][i]
    else:
        for i in range(y.rows):
            temp_sum = 0
            j = i + 1 - half_bandwidth
            if j < 0:
                j = 0
            while j < i:
                temp_sum += L[i][j] * y[j][0]
                j += 1

            y[j][0] = (b[j][0] - temp_sum) / L[i][i]

    return y


def elimination(A, b, half_bandwidth=None):
    n = A.rows
    for j in range(n):
        if A[j][j] <= 0:
            raise ValueError("Diagonal Entry is not positive, matrix is not P.D.")

        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0] / A[j][j]

        if half_bandwidth is None:
            finish_line = n
        else:
```

```python
            if j + half_bandwidth <= n:
                finish_line = j + half_bandwidth
            else:
                finish_line = n

        for i in range(j + 1, finish_line):
            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]

    x = backward_substitution(A, b, half_bandwidth)
    return x


def backward_substitution(L, y, half_bandwidth=None):
    n = L.rows
    x_vec = [[0 for _ in range(1)] for _ in range(n)]
    x = Matrix(x_vec, n, 1)

    for i in range(n - 1, -1, -1):
        temp_sum = 0
        for j in range(i + 1, n):
            temp_sum += L[j][i] * x[j][0]
        x[i][0] = (y[i][0] - temp_sum) / L[i][i]

    return x


if __name__ == "__main__":
    a_vec = [[38, 23, 31, 22, 29, 25, 31], [23, 44, 36, 27, 35, 24, 33]
        , [31, 36, 65, 36, 45, 34, 45], [22, 27, 36, 46, 29, 15, 27], [29, 35, 45, 29, 52, 32, 39]
             , [25, 24, 34, 15, 32, 37, 36], [31, 33, 45, 27, 39, 36, 65]]
    b_vec = [[13], [4], [7], [23], [17], [5.8], [10]]

    A = Matrix(a_vec, 7, 7)
    b = Matrix(b_vec, 7, 1)

    x = solve_chol(A, b)
    if check_choleski(A, b, x):
        print("Correct")
    else:
        print("Incorrect")
```

Listing 3: Linear resistive networks (`linear_networks.py`).

```python
from matrix import Matrix
from choleski import solve_chol
import csv, math, time, os

resistance = 10000
TEST_CURRENT = 10
TEST_BRANCH_RESISTANCE = 10000

class LinearResistiveNetwork(object):
    def __init__(self, num, branch, node, a, y, j, e, size):
        self._num = num
        self._branch_number = branch
        self._node_number = node
        self._curr_vec = j
        self._volt_vec = e
        self._red_ind_mat = a
        self._rev_res_mat = y
        self._size = size

    def solve_circuit_banded(self):
        return solve_chol(self.A, self.b, self.size + 1)

```

```python
    def solve_circuit(self):
        return solve_chol(self.A, self.b)

    @property
    def size(self):
        return self._size

    @property
    def J(self):
        return self._curr_vec

    @property
    def E(self):
        return self._volt_vec

    @property
    def Y(self):
        return self._rev_res_mat

    @property
    def re_A(self):
        return self._red_ind_mat

    @property
    def A(self):
        return self.re_A.dot_product(self.Y.dot_product(self.re_A.T))

    @property
    def b(self):
        YE = self.Y.dot_product(self.E)
        J_YE = self.J.minus(YE)
        result = self.re_A.dot_product(J_YE)
        return result


def read_circuits(filename):
    """
    This is the method to read the circuit information that is contained in csv files in a directory.
    Upon success, the method will create the required calculation information such as J, E, vectors
    and reduced indices matrices.

    :return: a LinearResistiveNetwork object containing the key matrices for calculations.
    """
    with open(filename) as csv_file:
        # Use CSV reader to read from circuit files
        # row[0] = start node ID
        # row[1] = end node ID
        # row[2] = J value of a branch
        # row[3] = R value of a branch
        # row[4] = E value of a branch
        csv_reader = csv.reader(csv_file, delimiter=',')
        row = next(csv_reader)
        circuit_id = int(row[0])
        n_branch = int(row[2])
        n_node = int(row[4])
        size = int(math.sqrt(n_node))

        branch_id = 0
        current_vec = [[0] for _ in range(n_branch)]
        volt_vec = [[0] for _ in range(n_branch)]
        rev_res_mat = [[0 for _ in range(n_branch)] for _ in range(n_branch)]
        incident_mat = [[0 for _ in range(n_branch)] for _ in range(n_node)]

        j_vec = Matrix(current_vec, n_branch, 1)
        e_vec = Matrix(volt_vec, n_branch, 1)
        y_mat = Matrix(rev_res_mat, n_branch, n_branch)
        a_mat = Matrix(incident_mat, n_node, n_branch)

        for row in csv_reader:
            j_vec[branch_id][0] = float(row[2])
```

```python
            e_vec[branch_id][0] = float(row[4])
            if int(row[3]) != 0:
                y_mat[branch_id][branch_id] = 1 / float(row[3])
            else:
                print("The input resistance is 0.")

            # create un-reduced A matrix
            a_mat[int(row[0])][branch_id] = 1
            a_mat[int(row[1])][branch_id] = -1

            branch_id += 1

        # By default, Node 0 is grounded, remove node 0
        # and create new reduced incidence matrix
        a_mat = Matrix(a_mat.vec[1:], n_node - 1, n_branch)

        linear_network = LinearResistiveNetwork(circuit_id, n_branch, n_node, a_mat, y_mat, j_vec, e_vec, size)
        return linear_network


def network_constructor(size):
    """
    This method generates a linear resistive network.
    The size of the network is defined by the argument size, and it's an N*N square network.

    This method generates a new input .csv file, for future uses.

    :param size: a.k.a, N, the number of nodes in a row or in a column.
    :return: No return value.
    """
    n_node = int(math.pow(size, 2))
    n_branch = 2 * size * (size - 1) + 1

    row_count = 0
    node_id = 0

    first_row = [str(size), 'B', str(n_branch), 'N', str(n_node)]
    first_branch = [str(n_node - 1), '0', str(TEST_CURRENT), str(TEST_BRANCH_RESISTANCE), '0']
    general_branch = [None for _ in range(5)]

    with open('res_mesh' + str(size) + '.csv', 'w', newline='') as csv_file:
        row_writer = csv.writer(csv_file, delimiter=',', quoting=csv.QUOTE_NONE, escapechar=' ')

        if row_count == 0:
            row_writer.writerow(r for r in first_row)
            row_writer.writerow(r for r in first_branch)
            row_count += 2

        for row_count in range(row_count, n_branch):
            if node_id == n_node - 1:
                break

            elif (node_id + 1) % size == 0:
                general_branch[0] = str(node_id)
                general_branch[1] = str(node_id + size)
                general_branch[2] = '0'
                general_branch[3] = str(resistance)
                general_branch[4] = '0'
                row_writer.writerow(r for r in general_branch)

            elif (node_id + size) >= n_node:
                general_branch[0] = str(node_id)
                general_branch[1] = str(node_id + 1)
                general_branch[2] = '0'
                general_branch[3] = str(resistance)
                general_branch[4] = '0'
                row_writer.writerow(r for r in general_branch)

            else:
```

```
                general_branch[0] = str(node_id)
                general_branch[1] = str(node_id + 1)
                general_branch[2] = '0'
                general_branch[3] = str(resistance)
                general_branch[4] = '0'
                row_writer.writerow(r for r in general_branch)

                general_branch[0] = str(node_id)
                general_branch[1] = str(node_id + size)
                general_branch[2] = '0'
                general_branch[3] = str(resistance)
                general_branch[4] = '0'
                row_writer.writerow(r for r in general_branch)
            node_id += 1


if __name__ == "__main__":
    os.chdir('circuits')
    """
    for size in range(2, 16):
        print("Constructing resistor mesh, n = " + str(size))
        network_constructor(size)
        print("Done.")
    """

    with open('result.csv', 'w', newline='') as csv_file:
        row_writer = csv.writer(csv_file, delimiter=',', quoting=csv.QUOTE_NONE, escapechar=' ')
        first_row = ['size', '', 'Resistance', 'Time of Calculation']
        row_writer.writerow(r for r in first_row)
        for size in range(2, 16):
            print("Writing result of N = " + str(size) + ", banded = False")
            start_time_unbanded = time.time()

            network = read_circuits('res_mesh' + str(size) + '.csv')
            x_unbanded = network.solve_circuit()

            v = x_unbanded[x_unbanded.rows - 1][0]
            i1 = v / TEST_BRANCH_RESISTANCE
            i2 = TEST_CURRENT - i1
            resistance = v / i2
            finish_time_unbanded = time.time()
            print("Resistance = " + str(resistance) + ", calculation time = "
                  + str(finish_time_unbanded - start_time_unbanded))
            result_arr = [str(size), 'unbanded', str(resistance), str(finish_time_unbanded - start_time_unbanded)]
            row_writer.writerow(r for r in result_arr)

            print("Writing result of N = " + str(size) + ", banded = True")
            start_time_banded = time.time()
            x_banded = network.solve_circuit_banded()

            v = x_banded[x_banded.rows - 1][0]
            i1 = v / 1000
            i2 = 10 - i1
            banded_resistance = v / i2
            finish_time_banded = time.time()
            print("Resistance = " + str(resistance) + ", calculation time = "
                  + str(finish_time_banded - start_time_banded))
            result_arr = [str(size), 'banded', str(resistance), str(finish_time_banded - start_time_banded)]
            row_writer.writerow(r for r in result_arr)
```

Listing 4: Finite Difference (`finite_difference.py`).

```
import math, csv
from matrix import Matrix


SHIELD_SIZE = 0.2
```

```python
CORE_WIDTH = 0.08
CORE_HEIGHT = 0.04
CORE_VOLTAGE = 110
TOLERANCE = 0.00001

class Node(object):
    def __init__(self, value):
        self._value = value
        self._is_free = True

    def set_value(self, value):
        self._value = value

    def set_free(self):
        self._is_free = True

    def set_fixed(self):
        self._is_free = False

    @property
    def value(self):
        return self._value

    @property
    def is_free(self):
        return self._is_free

class UniformMesh(object):
    """
    This class generates a uniform mesh between the cable core and the outer shield.
    One of the corners of the mesh lies at the center of the core and the diagonal connects with a corner of the shield.
    This way we create a mesh with uniform spaced nodes with clear boundary conditions.
    """
    def __init__(self, width, height, x, y, h):
        self._width = width
        self._height = height
        self._node_distance = h

        # Coord of the bottom left corner
        self._bottom_left_x = x
        self._bottom_left_y = y

        # Coord of the bottom right
        self._bottom_right_x = x + width
        self._bottom_right_y = y

        # Coord of top left
        self._top_left_x = x
        self._top_left_y = y + height

        # Coord of top right
        self._top_right_x = x + width
        self._top_right_y = y + height

        # Calculate how many nodes are there in a row and a column.
        # Assume there is no remainder after the division.
        # Then construct a matrix for this mesh
        # The matrix has one extra row and one extra column
        # This addition can act as a buffer to the matrix, prevent out of bounds exceptions
        # Also makes use of the symmetry.
        self._row_nodes = int(width / h) + 1
        self._col_nodes = int(height / h) + 1

        self._mesh_vec = [[Node(0) for _ in range(self._row_nodes + 1)] for _ in range(self._col_nodes + 1)]
        self._mesh_matrix = Matrix(self._mesh_vec, self._row_nodes + 1, self._col_nodes + 1)

    def initialize_values_second_quadrant(self):
        """
```

```python
        This method initializes a mesh in the second quadrant w.r.t. the core.
        The left most and top most boundaries are initialized and fixed to 0.
        The nodes lying on the edge of the cores are initialized to 110V.

        * Assume that width and height are completely divisible without remainder by h.
        :return: void
        """
        h = self._node_distance
        # Initialize the shield boundary conditions
        # Start from the top boundary
        i = 0
        for j in range(self.matrix.cols):
            node = self.matrix[i][j]
            node.set_value(0)
            node.set_fixed()

        # Now do the left side boundary
        j = 0
        for i in range(self.matrix.rows):
            node = self.matrix[i][j]
            node.set_value(0)
            node.set_fixed()

        # Now do the boundary of the core
        core_center_i = self.matrix.rows - 1
        core_center_j = self.matrix.cols - 1

        shift_j = int((CORE_WIDTH / 2) / h) + 1
        shift_i = int((CORE_HEIGHT / 2) / h) + 1

        core_boundary_i = core_center_i - shift_i
        core_boundary_j = core_center_j - shift_j

        for i in range(core_boundary_i, self.matrix.rows):
            for j in range(core_boundary_j, self.matrix.cols):
                node = self.matrix[i][j]
                node.set_value(CORE_VOLTAGE)
                node.set_fixed()


    def print_mesh(self):
        for i in range(self.matrix.rows):
            print("|", end=" ")
            for j in range(self.matrix.cols):
                node = self._mesh_matrix[i][j]
                print("%f" % node.value, end=" ")
            print("|")

    def copy_mesh(self):
        new_mesh = UniformMesh(self._width, self._height,
                               self._bottom_left_x, self._bottom_left_y, self._node_distance)
        for i in range(self.matrix.rows):
            for j in range(self.matrix.cols):
                new_node = new_mesh.matrix[i][j]
                old_node = self.matrix[i][j]
                new_node.set_value(old_node.value)
                if old_node.is_free:
                    new_node.set_free()
                else:
                    new_node.set_fixed()
        return new_mesh

    @property
    def width(self):
        return self._width

    @property
    def height(self):
        return self._height

```

```
    @property
    def bottom_left(self):
        return self._bottom_left_x, self._bottom_left_y

    @property
    def bottom_right(self):
        return self._bottom_right_x, self._bottom_right_y

    @property
    def top_left(self):
        return self._top_left_x, self._top_left_y

    @property
    def top_right(self):
        return self._top_right_x, self._top_right_y

    @property
    def matrix(self):
        return self._mesh_matrix

    @property
    def width_nodes(self):
        return self._row_nodes

    @property
    def height_nodes(self):
        return self._col_nodes

def successive_over_relaxation(mesh, omega, uni_spacing=True):
    iteration = 0
    new_mesh = mesh.copy_mesh()

    if uni_spacing:
        while not relaxation_succeeded(mesh, new_mesh):
            mesh = new_mesh.copy_mesh()
            new_mesh = mesh.copy_mesh()
            iteration += 1
            for i in range(mesh.height_nodes):
                for j in range(mesh.width_nodes):
                    if mesh.matrix[i][j].is_free:
                        sum = (new_mesh.matrix[i - 1][j].value +
                               new_mesh.matrix[i][j - 1].value +
                               mesh.matrix[i + 1][j].value +
                               mesh.matrix[i][j + 1].value)

                        temp_val = mesh.matrix[i][j].value
                        overwrite = (1 - omega) * temp_val + omega * sum * 0.25
                        new_mesh.matrix[i][j].set_value(overwrite)

                        if i == mesh.matrix.rows - 2:
                            new_mesh.matrix[i + 1][j].set_value(new_mesh.matrix[i - 1][j].value)

                        elif j == mesh.matrix.cols - 2:
                            new_mesh.matrix[i][j + 1].set_value(new_mesh.matrix[i][j - 1].value)
    else:
        pass

    return iteration, mesh

def jacobi(mesh, omega):
    iteration = 0
    new_mesh = mesh.copy_mesh()

    while not relaxation_succeeded(mesh, new_mesh):
        mesh = new_mesh.copy_mesh()
        new_mesh = mesh.copy_mesh()
        iteration += 1

        for i in range(mesh.height_nodes):
            for j in range(mesh.width_nodes):
```

```
                if mesh.matrix[i][j].is_free:
                    sum = (mesh.matrix[i - 1][j].value +
                           mesh.matrix[i][j - 1].value +
                           mesh.matrix[i + 1][j].value +
                           mesh.matrix[i][j + 1].value)
                    overwrite = sum / 4
                    new_mesh.matrix[i][j].set_value(overwrite)

                    # deal with symmetry
                    if i == mesh.matrix.rows - 2:
                        new_mesh.matrix[i + 1][j].set_value(new_mesh.matrix[i - 1][j].value)

                    elif j == mesh.matrix.cols - 2:
                        new_mesh.matrix[i][j + 1].set_value(new_mesh.matrix[i][j - 1].value)

    return iteration, mesh

def relaxation_succeeded(mesh, new_mesh):
    for i in range(mesh.width_nodes):
        for j in range(mesh.height_nodes):
            if mesh.matrix[i][j].is_free:
                residual = abs(new_mesh.matrix[i - 1][j].value
                               + new_mesh.matrix[i][j - 1].value
                               + new_mesh.matrix[i + 1][j].value
                               + new_mesh.matrix[i][j + 1].value
                               - 4 * new_mesh.matrix[i][j].value)
                if residual > TOLERANCE:
                    return False
    return True

if __name__ == "__main__":
    omega_list = [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
    h_list = [0.02, 0.01, 0.005, 0.0025, 0.00125]
    list_iteration = [0 for _ in range(10)]

    row = [0 for _ in range(3)]

    question = 4

    if question == 2:
        file_name = 'w '
    elif question == 3:
        file_name = 'h'
    else:
        file_name = 'h_jacobi'
    #iter, result = successive_over_relaxation(rect, 1.3)
    #result.print_mesh()

    with open(file_name + '_result.csv', 'w', newline='') as csv_file:
        first_row = ['omega', 'value', 'iterations']
        row_writer = csv.writer(csv_file, delimiter=',', quoting=csv.QUOTE_NONE, escapechar=' ')
        row_writer.writerow(r for r in first_row)

        if question == 2:
            for i in range(10):
                rect = UniformMesh(0.1, 0.1, 0, 0.1, 0.02)
                omega = omega_list[i]
                iteration, result = successive_over_relaxation(rect, omega)
                list_iteration[i] = iteration
                print("Number of iterations with omega = " + str(omega) + " is " + str(iteration))

                #0.06, 0.04 with h = 0.02
                target_node = result.matrix[2][3]

                row[0] = omega
                row[1] = target_node.value
                row[2] = iteration

                row_writer.writerow(r for r in row)
        elif question == 3:
```

```
            first_row = ['h', 'value', 'iterations']
            for i in range(5):
                omega = 1.3
                h = h_list[i]
                print("Now performing FD on a uniform spacing mesh using SOR with h = " + str(h))
                rect = UniformMesh(0.1, 0.1, 0, 0.1, h)
                rect.initialize_values_second_quadrant()

                iteration, result = successive_over_relaxation(rect, omega)
                list_iteration[i] = iteration
                print("Number of iterations with h = " + str(h) + " is " + str(iteration))
                if h == 0.02:
                    target_node = result.matrix[2][3]
                elif h == 0.01:
                    target_node = result.matrix[4][6]
                elif h == 0.005:
                    target_node = result.matrix[8][12]
                elif h == 0.0025:
                    target_node = result.matrix[16][24]
                else:
                    target_node = result.matrix[32][48]

                row[0] = h
                row[1] = target_node.value
                row[2] = iteration

                row_writer.writerow(r for r in row)
        elif question == 4:
            first_row = ['h', 'value', 'iterations']
            for i in range(5):
                omega = 1.3
                h = h_list[i]
                print("Now performing FD on a uniform spacing mesh using Jacobi with h = " + str(h))
                rect = UniformMesh(0.1, 0.1, 0, 0.1, h)
                rect.initialize_values_second_quadrant()

                iteration, result = jacobi(rect, omega)
                list_iteration[i] = iteration
                print("Number of iterations with h = " + str(h) + " is " + str(iteration))
                if h == 0.02:
                    target_node = result.matrix[2][3]
                elif h == 0.01:
                    target_node = result.matrix[4][6]
                elif h == 0.005:
                    target_node = result.matrix[8][12]
                elif h == 0.0025:
                    target_node = result.matrix[16][24]
                else:
                    target_node = result.matrix[32][48]

                row[0] = h
                row[1] = target_node.value
                row[2] = iteration

                row_writer.writerow(r for r in row)
```