ECSE 543: Numerical Methods

Assignment 1

Wenjie Wei
260685967

October 6, 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}.

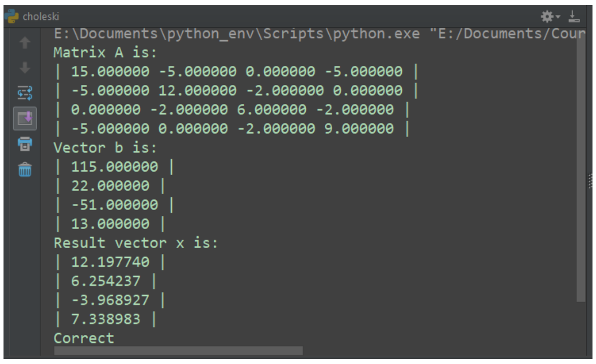
2.b Simple Tester Matrices

To examine the functionality of the implementation, some tester matrices are constructed. The first tester matrix has randomly chosen entries, under the condition that the matrix is a non-singular, symmetric, positive definite matrix:

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

To ensure non-singularity and positiveness, the entries on the primary diagonal must be chosen to be positive, otherwise the program with raise errors, meaning that the matrix does not meet the requirement. If the Choleski Decomposition succeeds, the matrix is proven to be positive definite.

Figure 1 shows the result of the test of this certain tester matrix. This result is found to be correct by checking the dot product (which is implemented in file `matrix.py`) of matrix A and vector x. This result is also verified by MATLAB using the back slash operator.



```
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

2.c 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.

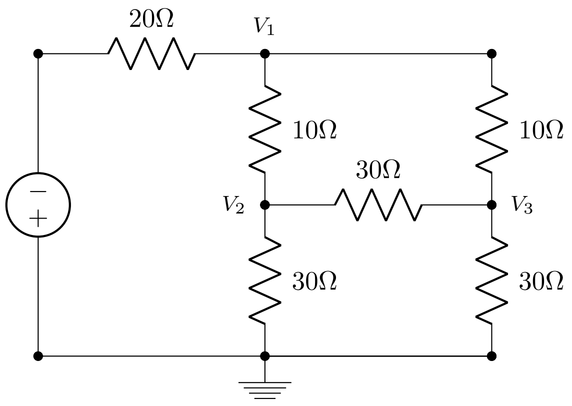
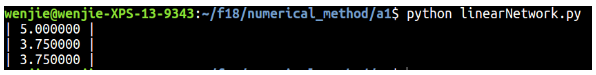


Figure 2: Test Circuit 5

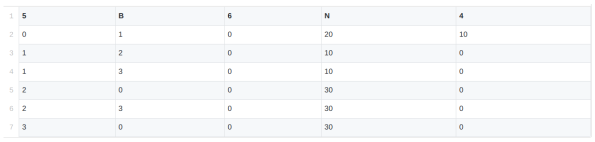


```
wenjie@wenjie-XPS-13-9343:~/f18/numerical_method/a1$ python linearNetwork.py
| 5.000000 |
| 3.750000 |
| 3.750000 |
```

Figure 3: Result of the Testing Circuit 5

Take the 5th circuit provided by the TA for example, the circuit is shown in Figure 2, and the result of the running of the program on this circuit is shown in Figure 3.

The data of the circuit are organized in the way shown in Figure 4. The first line shows the 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



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

Figure 4: Circuit File Organizations

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.c.1 Testing Circuit 1

Figures 5 and 6 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.

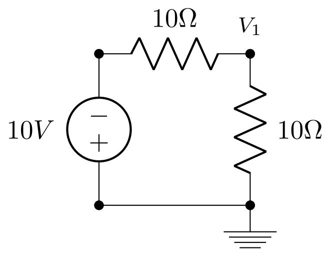
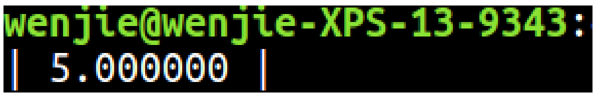


Figure 5: Test Circuit 1



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

Figure 6: Output Result of the Testing Circuit 1

2.c.2 Testing Circuit 2

Figures 7 and 8 below show the testing circuit 2 and its result. The expected result is $V_1 = 50V$.

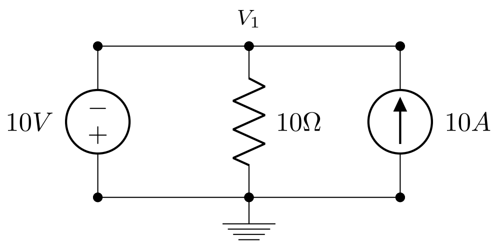
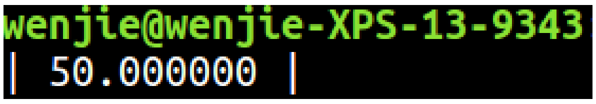


Figure 7: Test Circuit 2



```
wenjie@wenjie-XPS-13-9343
| 50.000000 |
```

Figure 8: Output Result of the Testing Circuit 2

2.c.3 Testing Circuit 3

Figures 9 and 10 below show the results of the testing circuit 3. The expected result of the circuit is $V_1 = 55V$.

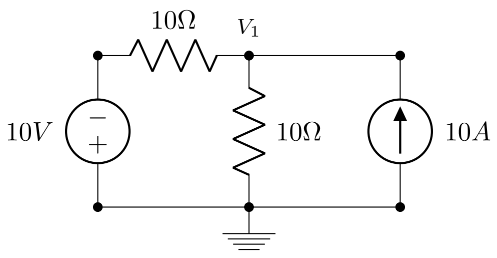
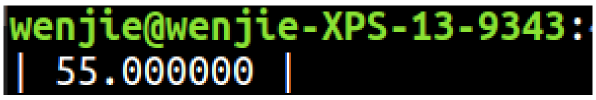


Figure 9: Test Circuit 3



```
wenjie@wenjie-XPS-13-9343:
| 55.000000 |
```

Figure 10: Output Result of the Testing Circuit 3

2.c.4 Testing Circuit 4

Figures 11 and 12 below show the results of the testing circuit 4. The expected results of the circuit is $V_1 = 20V$ and $V_2 = 35V$.

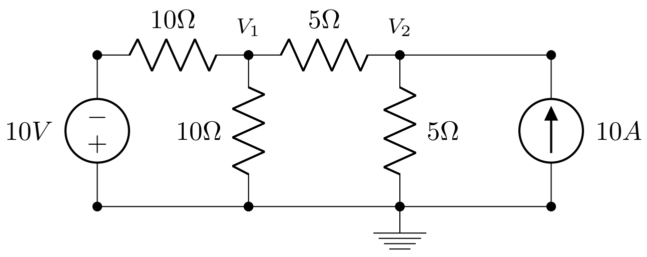
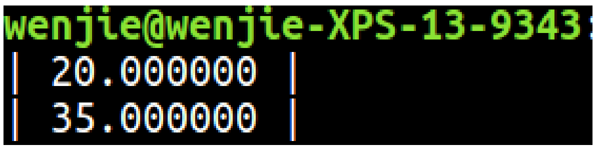


Figure 11: Test Circuit 4



```
wenjie@wenjie-XPS-13-9343
| 20.000000 |
| 35.000000 |
```

Figure 12: Output Result of the Testing Circuit 4

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 choleski_decomposition(A, b):
    """
    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.

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

    n = A.rows
    sparse_matrix = [[0 for _ in range(n)] for _ in range(n)]
    L = Matrix(sparse_matrix, n, n)

    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]
    # Now L and LT are all obtained, we can move to forward elimination

    y_vec = [[None for _ in range(1)] for _ in range(n)]
    y = Matrix(y_vec, n, 1)
    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]

    # Now perform back substitution to find x.
    x_vec = [[None 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 = [[0.1167, -0.0667, 0, -0.05, 0], [-0.0667, 0.2333, -0.1, 0, 0], [0, -0.1, 0.2667, -0.1, 0],
↪  [-0.05, 0, -0.1, 0.2, -0.05], [0, 0, 0, -0.05, 0.1167]]
    b_vec = [[0], [0.6667], [0], [0], [0]]

    A = Matrix(a_vec, 5, 5)
    b = Matrix(b_vec, 5, 1)

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

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

```
from matrix import Matrix
from choleski import choleski_decomposition
import csv, math, time


class LinearResistiveNetwork(object):
    def __init__(self, num, branch, node, a, y, j, e):
        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

    def solve_circuit(self):
        return choleski_decomposition(self.A, self.b)

    @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])

        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])
            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)
        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
    resistance = 1000
    test_current = 10
    res_branch = 1000

    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(res_branch), '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)
```

```python
            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__":
    size = 15
    #network_constructor(size)
    start_time = time.time()

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

    v = x[x.rows - 1][0]
    i1 = v / 1000
    i2 = 10 - i1
    resistance = v / i2
    print("Computation time = " + str(time.time() - start_time))
    print("Resistance of Mesh = " + str(resistance) + " Omega")
```