a)

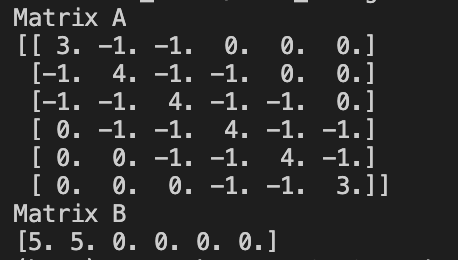
Using Ohm’s law and Kirchhoff’s law, for example, at the junction at V1:

Similarly, the same method can be used at other junctions to deduce the given equations.

The calculations for this part has been uploaded as Part\_a\_Calculations.pdf

b) The same method of using Ohm’s law and Kirchhoff’s law in part (a) can be used and generalized to achieve the equations given in the question. The equation can be express in

form following the code below. Where N is the desired size of the matrix. When N=6 had been used as an example, the following had been output, which seems correct.



import numpy as np

import math

# a = np.array([[3,-1,-1], [-1,4,-1], [-1,-1,3]])

# b = np.array([5,5,0])

# x = np.linalg.solve(a,b)

# print (x)

# create Av = w matrix

V\_in = 5

N = 6

A = np.zeros(shape=(N,N))

B = np.zeros(N)

B[0] = V\_in

B[1] = V\_in

for i in range(N):

if(i == 0):

A[i][0] = 3

A[i][1] = -1

A[i][2] = -1

if(i == 1):

A[i][0] = -1

A[i][1] = 4

A[i][2] = -1

if(N > 3):

A[i][3] = -1

if(i >= 2 and i<(N-2)):

A[i][i-2] = -1

A[i][i-1] = -1

A[i][i] = 4

A[i][i+1] = -1

A[i][i+2] = -1

if(i == N-2):

A[i][N-1-3] = -1

A[i][N-1-2] = -1

A[i][N-1-1] = 4

A[i][N-1] = -1

if(i == N-1):

A[i][N-1-2] = -1

A[i][N-1-1] = -1

A[i][N-1] = 3

print("Matrix A")

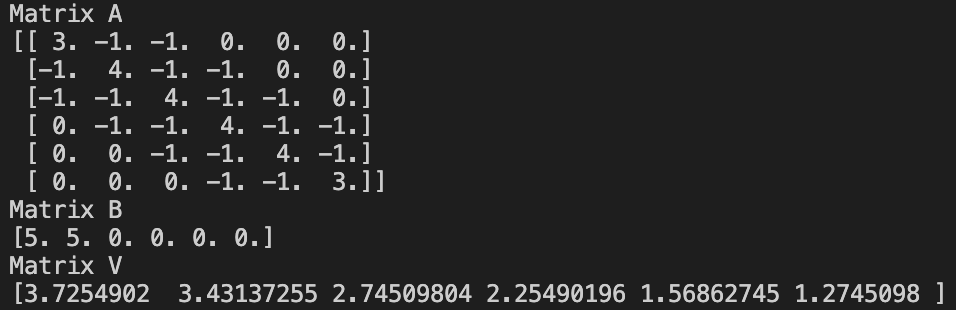
print(A)

print("Matrix B")

print(B)

c)

np.linalg.solve() had been used to solve for the unknown voltage values. The output was as follows for the equation form Av = B. Solving for v:



import numpy as np

import math

# a = np.array([[3,-1,-1], [-1,4,-1], [-1,-1,3]])

# b = np.array([5,5,0])

# x = np.linalg.solve(a,b)

# print (x)

# create Av = w matrix

V\_in = 5

N = 6

A = np.zeros(shape=(N,N))

B = np.zeros(N)

B[0] = V\_in

B[1] = V\_in

for i in range(N):

if(i == 0):

A[i][0] = 3

A[i][1] = -1

A[i][2] = -1

if(i == 1):

A[i][0] = -1

A[i][1] = 4

A[i][2] = -1

if(N > 3):

A[i][3] = -1

if(i >= 2 and i<(N-2)):

A[i][i-2] = -1

A[i][i-1] = -1

A[i][i] = 4

A[i][i+1] = -1

A[i][i+2] = -1

if(i == N-2):

A[i][N-1-3] = -1

A[i][N-1-2] = -1

A[i][N-1-1] = 4

A[i][N-1] = -1

if(i == N-1):

A[i][N-1-2] = -1

A[i][N-1-1] = -1

A[i][N-1] = 3

print("Matrix A")

print(A)

print("Matrix B")

print(B)

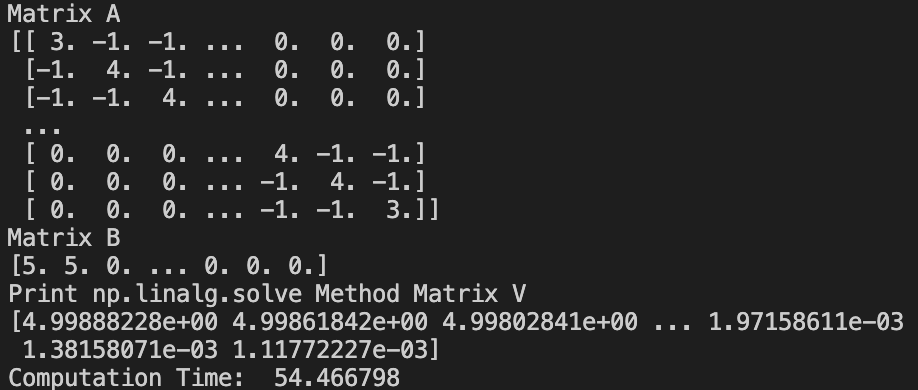
V = np.linalg.solve(A,B)

print("Matrix V")

print(V)

d)

In this question, my method and the banded method provided in the lecture notes had to be compared in computation time. As expected, my method took relatively long to perform.

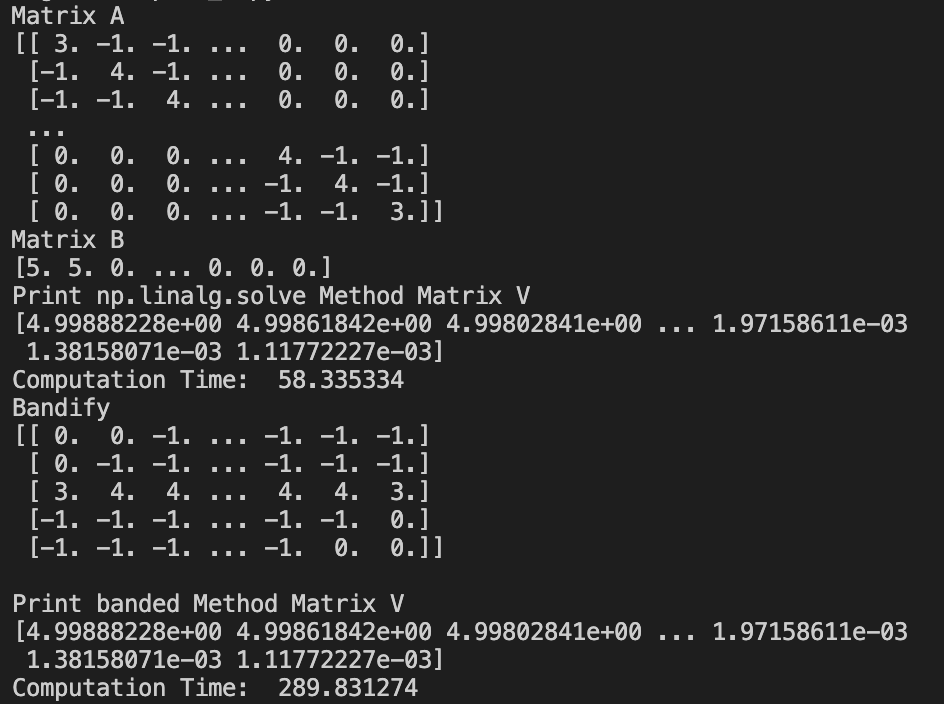


**Hypothesis**  
However, the banded function provided in the lecture notes did not work as intended, and even after tweaking and changing the code to remove errors, the function either continued to give errors or did not output the correct values. However, I can hypothesize even without running the code that the banded function should give a better computation time (faster computation time), because it is much more efficient in memory saving. This will not be obvious for small values of N, but for larger values of N such as 10000, the amount of memory saved should be significant enough to affect the computation time. It should be analogous to solving a size 10000 matrix to something much smaller.

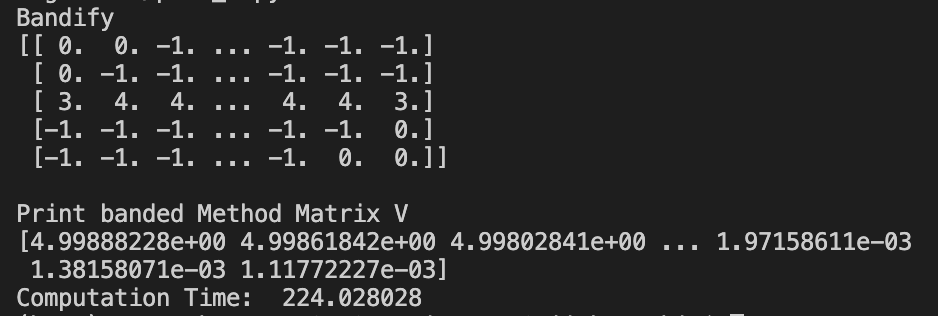
(My code has been debugged and the results are shown in the following page)

**After debugging and running the code**

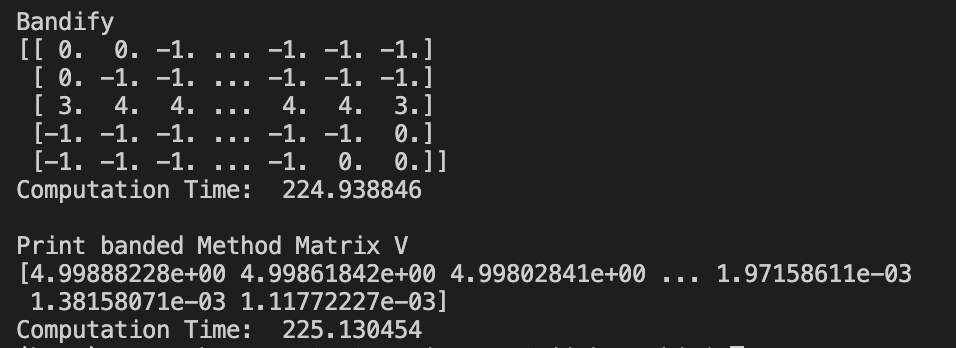
After long hours of efforts to debug the code, the banded function finally worked as intended, but the result was very shocking.



The computation time was cumulative, so the actual processing time for the banded method was 289.831274-58.335334=231.49594. The actual processing time of 231.49594 is still nearly 4 times longer than my method. As opposed to my prediction, the banded function, which was supposed to be much more memory efficient, did not perform the computation better than my method. This was very strange. Therefore, I tried the run the banded function again without running my method, and the results were similar.



I assumed that either there is a problem with the code, or my prediction was just wrong, but it is probable that the long computation time came from ‘bandifying’ the Matrix A, so that it could be used in the banded function. The bandify() function I had created was a suspect for this computation time, so after checking the computation time for running this function, the following result was shown.



The strange computation time was actually from ‘bandifying’ the Matrix A. The actual computation time for solving the matrix was 225.130454 - 224.938846 = 0.191608. The computation time for the banded function was 0.191608, which is exponentially faster than my method, and this agrees with the hypothesis. Therefore, the results agree with the hypothesis and it seems to behave as it should.