2018

SEMESTER 2

ENGSCI 331 Eigen Problems Assignment

Connor McDowall
530913386
cmcd398

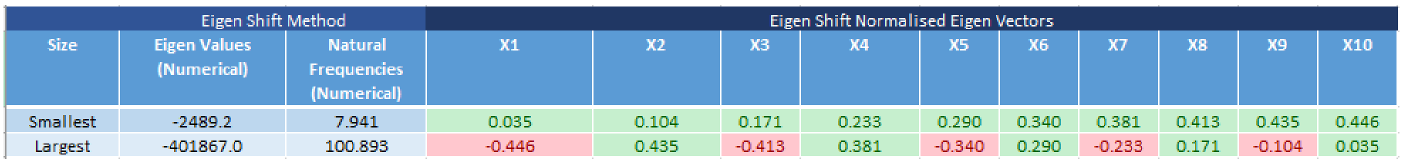
October 23, 2018

Contents

Listings

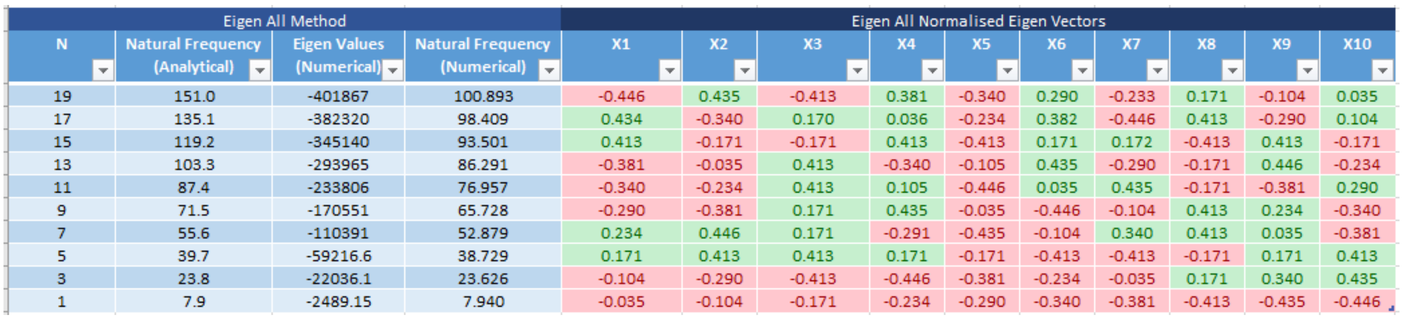
List of Figures

1 Report



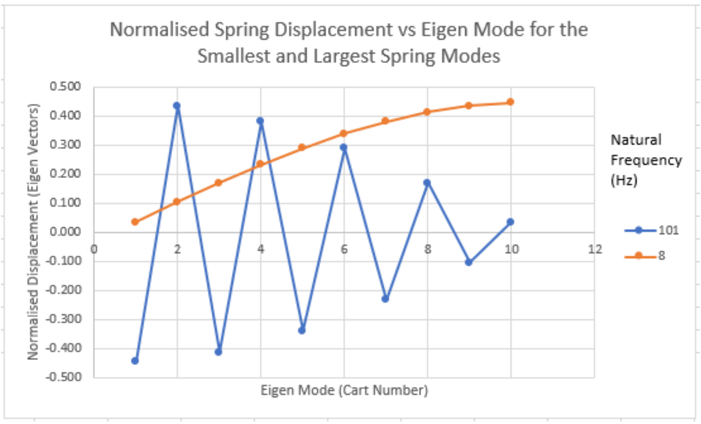
Eigen Shift Method			Eigen Shift Normalised Eigen Vectors									
Size	Eigen Values (Numerical)	Natural Frequencies (Numerical)	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
Smallest	-2489.2	7.941	0.035	0.104	0.171	0.233	0.290	0.340	0.381	0.413	0.435	0.446
Largest	-401867.0	100.893	-0.446	0.435	-0.413	0.381	-0.340	0.290	-0.233	0.171	-0.104	0.035

Figure 1: Eigen Shift Method Natural Frequencies and Eigen Vectors

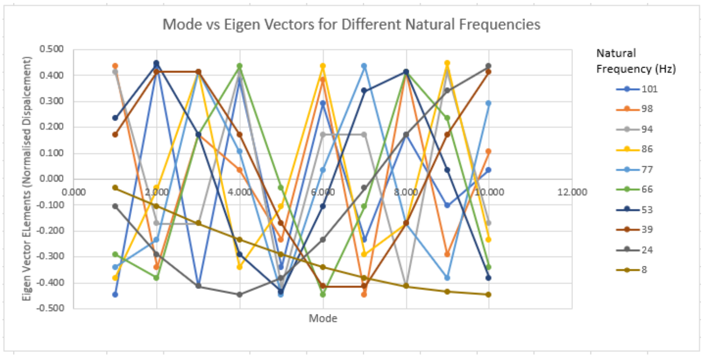


Eigen All Method				Eigen All Normalised Eigen Vectors									
N	Natural Frequency (Analytical)	Eigen Values (Numerical)	Natural Frequency (Numerical)	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
19	151.0	-401867	100.893	-0.446	0.435	-0.413	0.381	-0.340	0.290	-0.233	0.171	-0.104	0.035
17	135.1	-382320	98.409	0.434	-0.340	0.170	0.036	-0.234	0.382	-0.446	0.413	-0.290	0.104
15	119.2	-345140	93.501	0.413	-0.171	-0.171	0.413	-0.413	0.171	0.172	-0.413	0.413	-0.171
13	103.3	-293965	86.291	-0.381	-0.035	0.413	-0.340	-0.105	0.435	-0.290	-0.171	0.446	-0.234
11	87.4	-233806	76.957	-0.340	-0.234	0.413	0.105	-0.446	0.035	0.435	-0.171	-0.381	0.290
9	71.5	-170551	65.728	-0.290	-0.381	0.171	0.435	-0.035	-0.446	-0.104	0.413	0.234	-0.340
7	55.6	-110391	52.879	0.234	0.446	0.171	-0.291	-0.435	-0.104	0.340	0.413	0.035	-0.381
5	39.7	-59216.6	38.729	0.171	0.413	0.413	0.171	-0.171	-0.413	-0.413	-0.171	0.171	0.413
3	23.8	-22036.1	23.626	-0.104	-0.290	-0.413	-0.446	-0.381	-0.234	-0.035	0.171	0.340	0.435
1	7.9	-2489.15	7.940	-0.035	-0.104	-0.171	-0.234	-0.290	-0.340	-0.381	-0.413	-0.435	-0.446

Figure 2: Eigen All Method Natural Frequencies and Eigen Vectors



(a) Low vs High Natural Frequency (Hz)



(b) All 10 Modes

Figure 3: Comparisons to describe the specific patterns of motion

The carts model an eigenmode, a natural vibration of a system where all parts of the system move at the same frequency, moving sinusoidally with amplitudes changing proportionally to eachother. Using the analytical parameters, we observe this pattern in figures 3 (a) and (b).

The largest eigen value (highest natural frequency) has an ocsillating pattern across the 10 coil spring system. The associated eigen vector has both positive and negative elements in the vector. The varying signs of the eigen vector's elements show the carts are moving in different directions with the coils extending or compressing depending on the sign. This oscillating movement and pattern of the system is expected at high frequencies. Surging does not occur in the high spring system. Resonant behaviour is not observed with the surging frequencies much higher than the engine vibration frequencies.

The smallest eigen value has spring modes all moving in the same direction, as shown by the upward trend on figure 3 (a). All carts are moving in the same direction with coils all extending. The frequency is not large enough to cause the spring coils and carts to move in different directions. We observe resonant behaviour. We expect this from the low frequency and the given parameter combination. The extent the system oscillates decreases for each mode as the natural frequency decreases until all carts move in the same direction. This observed in figure 3 (b), leading to surging behaviour.

2 Appendices

All functions were tested using the test functions in the myEigenFunctions file. The output was printed and compared to the analytical solutions in the notes. The eigen vectors in the notes are not normalised but the outputs from my functions are. I converted the eigen vectors in the notes to a normalised form by hand. See 2.4 for screenshots of the test and implementation output.

After the assignment reduction, I assumed we no longer had to use a function to construct the A matrix as this was not specified in the new handout. The A matrix is constructed manually using the parameters specified in the original handout. I also assumed we didn't need to have a matrix constructed based on user inputs.

The signs of the eigen vector's elements inform the direction you are looking at the eigen vector. The smallest eigen value's vector is the same for both the Eigen All and Eigen Shift methods. This is the case as the signs for eigen all's vector elements are the exact inverse for eigen shift's vector elements. Both methods find the same eigen vector but look at it from different directions.

There is a difference in the analytical and numerical methods. This is attributable to numerical inaccuracies in working with large numbers at high frequencies. Small differences in large numbers make big impacts, as seen with higher frequencies becoming increasingly inaccurate. See figure 4 for the visualisation.

Figures 3 (a) and (b) are plotted using straight lines as the normalised eigen vectors and natural frequencies are discrete values.

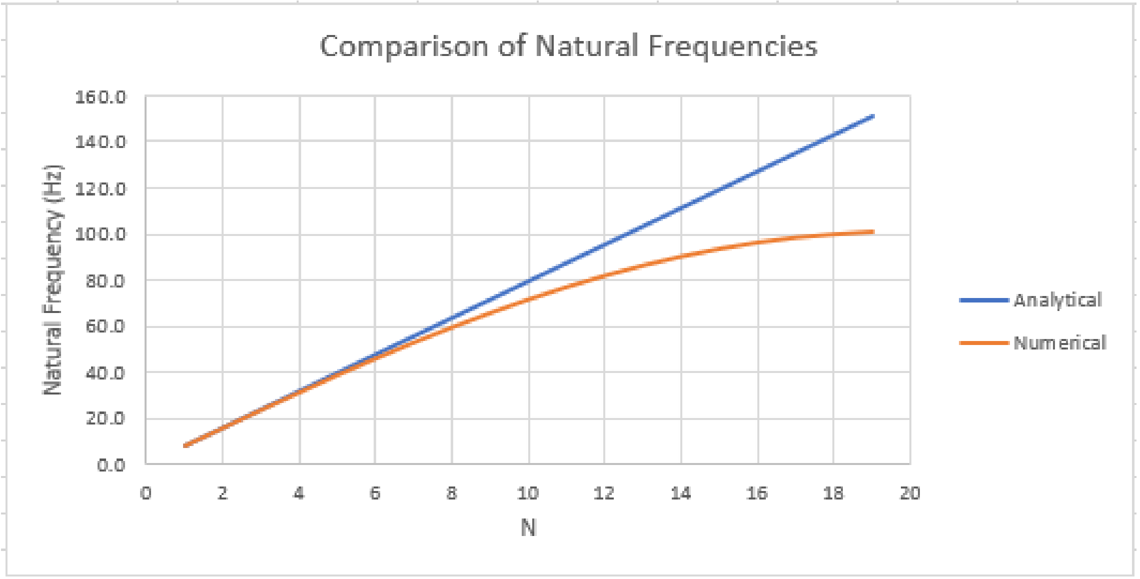


Figure 4: Numerical Solution compared against the Analytical

2.1 myEigenFunctions.h

```
/***************************************************************************************
        myEigenFunctions.h

        This file is where you'll put the header information about the functions you write.  This is just
        the argument list (as it is in the first line of your function declarations in myEigenFunctions.cpp)
        followed by a semi-colon.  Two example functions have been done for you.

==============================================================================================*/
#include <iostream>
#include <math.h>




// Note that these two functions have the same name, but different argument lists.  This is called
// "function overloading" and is allowed by C++ compilers.

double DotProduct(double *A, double *B, int n);

double* DotProduct(double **A, double *v, int n);

double DotProduct(double **A, double **B, int n,int m);

void powermethodanddeflatetest();

struct Eigenpair {
  double value; //Eigenvalue
  double *vector; //Eigenvector
  int length; //Length of eigenvector
  void normalize() {
    // Set eigenvalue to norm of vector and normalize eigenvector
    value = sqrt(DotProduct(vector, vector, length));
    for (int i=0; i<length; i++)
      vector[i]/= value;
  }; //

  void print() {
    std::cout << value << ": \t";
    for (int i=0; i<length; i++)
      std::cout << vector[i] << "\t";
    std::cout << "\n";
  }
  // Constructor
  // Attribute value is set to 0.0 and attribute vector to an array of doubles with lenght n
Eigenpair(const int n) : value(0.0), length(n), vector(new double[n]) {} //Constructor

};
```

```
Eigenpair power_method(double **A, double *v, int n, double tol);

void deflate(double **A, Eigenpair eigenpair);

void print_matrix(double **A, int n, int m);

void print_vector(double *v, int n);

Eigenpair eigenshift(double **A, double *v, int n, double tol);

void eigenall(double **A, double *v, int n, int tol);

void eigenalltest();

void eigenshifttest();
```

2.2 myEigenFunctions.cpp

```
/*****************************************************************************************************

 This file is where you'll put the source information for the functions you write.  Make sure it's
 included in your project (shift-alt-A) and that the functions you add are declared in the header
 file, myEigenFunctions.h.  You can add any extra comments or checks that could be relevant to these
 functions as you need to.

====================================================================================================*/
#include "myEigenFunctions.h"

double DotProduct(double *A, double *B, int n)
{
        //
        //      This is a function that takes two vectors A and B of identical length (n) and
        //  calculates and returns their dot product.
        //

        double dot = 0.0;

        for (int i = 0; i < n; i++) {
                dot += A[i] * B[i];
        }
        return dot;
}
double DotProduct(double **A, double **B, int n, int m)
```

```
{
        //
        // This is a function that takes two matrices A and B of identical dimensions (n*m) and
        // returns their dot product.
        //

        double dot = 0.0;

        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < m; j++)
                {
                        dot += A[i][j] * B[i][j];
                }
        }
        return dot;
}
double* DotProduct(double **A, double *v, int n)
{
        //
        //      This is a function that takes a nxn A matrix and n dimensional B vector and
        //  stores the product A.V at the original location of v
        //
        double *result = new double[n]; //point to the result vector

        for (int i = 0; i < n; i++)
        {
                result[i] = 0.0; // Initialize ith element of result v
                for (int j = 0; j < n; j++)
                {
                        result[i] += A[i][j] * v[j];
                }
        }
        return result;
}
// Write the power_method
Eigenpair power_method(double **A, double *v, int n, double tol)
{
        // This function computes the largest eigenvalue and the corresponding normalised eigen vector
        // Do all the initial set up
        Eigenpair eigenpair(n);
        double *vector_hat = new double[n];
        double value_hat;
        int istore;

        // Setting the initial eigenpair values as those from the inputs
        for (int i = 0; i < n; i++)
        {
                eigenpair.vector[i] = v[i];
        }
        // Normalise the original eigen value estimate
        eigenpair.normalize();

        do {
                //Set the initial eigen value for convergence
                value_hat = eigenpair.value;
                eigenpair.vector = DotProduct(A, eigenpair.vector, eigenpair.length);

                // Find the index of the largest element
                double vstore = 0;
                for (int i = 0; i < n; i++)
                {
                        if (abs(eigenpair.vector[i]) > vstore)
                        {
                                istore = i;
                                vstore = (abs(eigenpair.vector[i]));
                        }
                }

                // Set eigenvalue to the norm of the vector and normalise the vector
                eigenpair.normalize();

                // Condition to break the loop
        } while (abs(eigenpair.value - value_hat) / abs(eigenpair.value) > tol);

        // If condition to assess if eigen value is going in the opposite direction.
        if (DotProduct(A, eigenpair.vector, n)[istore] / eigenpair.vector[istore] < 0)
        {
                eigenpair.value = -1 * eigenpair.value;
        }
        // Convert the eigen vector back to an unnormalised form
        // eigenpair.vector = eigenpair.vector*
        return eigenpair;
}

// deflate method
```

```
void deflate(double **A, Eigenpair eigenpair)
//
// This function removes the largest eigenvalue from a matrix so the power method
// can find the next largest value.
// Input A matrix, normalised eigenvector and largest eigen value
//
{       // Eigen vector already normalised
        //Apply the deflation method in a for loop
        //Create a new matrix
        double **C;
        C = new double*[eigenpair.length];
        for (int i = 0; i < eigenpair.length; i++)
        {
                C[i] = new double[eigenpair.length];
        }

        // Calculate the C matrix values as you go
        //Create a double matrix
        for (int i = 0; i < eigenpair.length; i++)
        {
                for (int j = 0; j < eigenpair.length; j++)
                {
                        C[i][j] = eigenpair.value*eigenpair.vector[i] * eigenpair.vector[j];
                }
        }
        // Perform the calculation
        for (int i = 0; i < eigenpair.length; i++)
        {
                for (int j = 0; j < eigenpair.length; j++)
                {
                        A[i][j] = A[i][j] - C[i][j];
                }
        }
}

// eigen_shift (Insert the function here).
Eigenpair eigenshift(double **A, double *v, int n, double tol)
{
        // Use the power method to find the largest eigenvalue
        Eigenpair a = power_method(A, v, n, tol);
        // Store the largest eigen value
        double eigenlarge = a.value;
        // Create the identity matrix
```

```
double **I;
I = new double*[a.length];
for (int i = 0; i < a.length; i++)
{
        I[i] = new double[a.length];
}

// Assign a value of zero to the entire matrix
for (int i = 0; i < a.length; i++)
{
        for (int j = 0; j < a.length; j++)
        {
                I[i][j] = 0;
        }
}
// Assign values of 1 the trace elements
for (int i = 0; i < a.length; i++)
{
        I[i][i] = eigenlarge;
}

// Use a loop to Perform the shift
for (int i = 0; i < a.length; i++)
{
        for (int j = 0; j < a.length; j++)
        {
                A[i][j] = A[i][j] - I[i][j];
        }
}
// Use the power method again to shift the matrix
Eigenpair b = power_method(A, v, n, tol);

// Shift the eigen value to return the smallest
double eigensmall = b.value + eigenlarge;

//Print Eigen Small outputs
std::cout << "Eigensmall";
std::cout << "\n";
std::cout << eigensmall;
std::cout << "\n";
std::cout << "Eigenlarge";
std::cout << "\n";
std::cout << eigenlarge;
```

```
        std::cout << "\n";
        //Print Eigen Large outputs
        std::cout << "Eigensmall␣Vector";
        std::cout << "\n";
        print_vector(b.vector, b.length);
        std::cout << "\n";
        std::cout << "Eigenlarge␣Vector";
        std::cout << "\n";
        print_vector(a.vector, a.length);
        std::cout << "\n";

        // This assumes eigen values are all the correct sign
        return a , b ;
}

// eigen_all (Assumes all functions given will be symmetric)
void eigenall(double **A, double *v, int n, int tol)
{
        // Use the power method and deflation in an iterative scheme
        for (int i = 0; i < n; i++)
        {
                // Use the iterative scheme
                //Test the function
                Eigenpair test = power_method(A, v, n, 1e-8);
                std::cout << "Eigenvalue";
                std::cout << "\n";
                std::cout << test.value;
                std::cout << "\n";
                std::cout << "Eigen␣Vector␣After␣Power␣Method";
                std::cout << "\n";
                print_vector(test.vector, test.length);
                std::cout << "\n";

                // Print the matrix before deflating
                std::cout << "Matrix␣Before␣Deflating";
                std::cout << "\n";
                print_matrix(A, test.length, test.length);
                std::cout << "\n";

                // Test the deflate method
                deflate(A, test);

                //Print the output matrix after deflating
                std::cout << "Matrix␣After␣Deflating";
                std::cout << "\n";
                print_matrix(A, test.length, test.length);
                std::cout << "\n";
        }
}

// Create a power method testong function
void powermethodanddeflatetest()
{
        // Create a matrix
        int n = 3;
        double**A;
        A = new double*[n];
        for (int i = 0; i < n; i++)
        {
                A[i] = new double[n];
        }
        //Set up the matrix
        A[0][0] = 1;
        A[0][1] = 2;
        A[0][2] = 0;
        A[1][0] = 2;
        A[1][1] = 1;
        A[1][2] = 0;
        A[2][0] = 0;
        A[2][1] = 0;
        A[2][2] = 2;

        // Set up vector
        double *v;
        v = new double[n];
        v[0] = 2;
        v[1] = 1;
        v[2] = 3;

        //Test the function
        Eigenpair test = power_method(A, v, n, 1e-8);
        std::cout << test.value;
        std::cout << "\n";
        print_vector(test.vector, test.length);
        std::cout << "\n";
        // Test the deflate method
        deflate(A, test);

        //Print the output matrix
        print_matrix(A, test.length, test.length);
}
void print_matrix(double **A, int n, int m)
{
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        std::cout << A[i][j];
                        std::cout << "␣";
                }
                std::cout << "\n";
        }
}
void print_vector(double *v, int n)
{
        for (int i = 0; i < n; i++)
        {
                std::cout << v[i];
                std::cout << "␣";
        }
}
void eigenalltest()
{       // Test used the numerical soultion in the notes and printed all the outputs.
        // These were prepared and came out correct in the system.
        // Create a matrix
        int n = 3;
        double**A;
        A = new double*[n];
        for (int i = 0; i < n; i++)
        {
                A[i] = new double[n];
        }
        //Set up the matrix
        A[0][0] = 1;
        A[0][1] = 2;
        A[0][2] = 0;
        A[1][0] = 2;
        A[1][1] = 1;
        A[1][2] = 0;
        A[2][0] = 0;
        A[2][1] = 0;
        A[2][2] = 2;

        // Set up vector
        double *v;
        v = new double[n];
        v[0] = 1;
        v[1] = 2;
        v[2] = 3;

        //Test the function
        eigenall(A, v, n, 1e-8);
}
void eigenshifttest()
{
        int n = 3;
        double**A;
        A = new double*[n];
        for (int i = 0; i < n; i++)
        {
                A[i] = new double[n];
        }
        //Set up the matrix
        A[0][0] = 1;
        A[0][1] = 2;
        A[0][2] = 0;
        A[1][0] = 2;
        A[1][1] = 1;
        A[1][2] = 0;
        A[2][0] = 0;
        A[2][1] = 0;
        A[2][2] = 2;

        // Set up vector
        double *v;
        v = new double[n];
        v[0] = 1;
        v[1] = 2;
        v[2] = 3;

        // Test the method
        eigenshift(A, v, n, 1e-8);
}
```

2.3 myEigenMain.cpp

```
/***************************************************************************************************

        This is a template main file for the ENGSCI331 Eigenvectors module.  It demonstrates some new C++
        syntax and functions, as described in the accompanying document ENGSCI331_Eigenstuff.pdf.

        *** There are some examples of "bad" programming in here (bits missing etc) that you will need to
        find and fix, though this file should compile without errors straight away. ***

        You should use this file to get you started on the assignment.  You're welcome to change whatever
        you'd like to as you go, this is only a starting point.

===================================================================================================*/

#define _CRT_SECURE_NO_DEPRECATE

#include <iostream>
#include <fstream>
#include <string>
#include <cmath>

// This is the header file for your functions.  Usual programming practise would be to use a *.cpp
// file that has the same name (ie: myEigenFunctions.cpp) and include it as normal in your project.
// Inside this file you'll see some ideas for functions that you could use during this project.  I
// suggest you plan out your code first to see what kind of functions you'll use repeatedly and then
// write them.
#include "myEigenFunctions.h"

using namespace std;

#define PI 3.14159265358979323846

int main(void)
{
        // ------------------------------------------------------------------------------------------
        //
        // PART 1: Initialisation
        //
        // ------------------------------------------------------------------------------------------
// Defining local variables to be used:

// n is the dimensions of the matrix, will be square for the eigen problems
// the option whether to read the matrix from a file or to construct from
// user-entered values of k and m

int n = 0,
        option = 0;


double *M = NULL,
        *K = NULL;

double **A = NULL;
double **B = NULL;

ifstream infile;
ofstream outfile;
string filename;

// Test all the methods
powermethodanddeflatetest();
eigenalltest();
eigenshifttest();

// Prompt and read number of masses in system
cout << "␣Enter␣the␣number␣of␣masses␣in␣system:␣";
cin >> n;
cout << endl;

// Allocating memory for the 1D arrays - these are the number of masses, n, long:
M = new double[n];
K = new double[n];

// Length for iterating
int length = 10;

// Allocating memory for the 2D arrays - these have dimensions of n*n:
A = new double*[n];
for (int i = 0; i < n; i++)
        A[i] = new double[n];
```

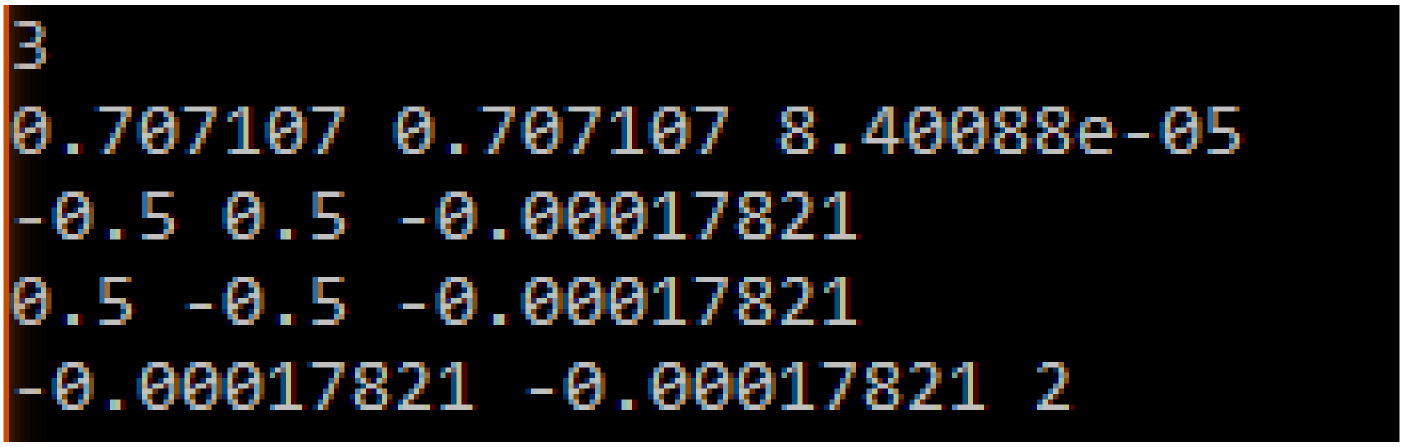
```
B = new double*[n];
for (int i = 0; i < n; i++)
        B[i] = new double[n];

// ------------------------------------------------------------------------------------------------
//
// PART 2: Populating matrices from user variables OR from a file
//
// ------------------------------------------------------------------------------------------------

cout << "A␣matrix␣built␣in␣the␣code␣(option␣1):";
cin >> option;
cout << endl;

switch (option) {
case 1:

        // Reading in the A matrix from a file
        //
        //                              You get to do this bit!
        //

        break;
default:
        cout << "ERROR:␣Option␣" << option << "␣is␣unrecognised.␣Enter␣1." << endl;
        break;
}
// Set in all the values
double G = 7.929e10;
double rho = 7751;
double D = 0.005;
double R = 0.0532;
double Na = 10;

// Set all values in the matrix to 0 first
for (int i = 0; i < length; i++)
{
        for (int j = 0; j < length; j++)
        {
                A[i][j] = 0;
        }
}
```

```
// Assign the mass and k values to the values to the 1D arrays
for (int i = 0; i < length; i++)
{
        M[i] = (PI*PI*D*D*rho*R) / 2;
        K[i] = (G*D*D*D*D / (64 * R*R*R));
}
K[0] = 2 * K[0];

// Assign the values to the A matrix and solve

//First Cart
A[0][0] = (-K[0] - K[1]) / M[0];
A[0][1] = K[1] / M[0];

// Second Cart
A[1][0] = K[1] / M[1];
A[1][1] = (-K[1] - K[2]) / M[1];
A[1][2] = K[2] / M[1];

// Third Cart
A[2][1] = K[2] / M[2];
A[2][2] = (-K[2] - K[3]) / M[2];
A[2][3] = K[3] / M[2];

// Forth Cart
A[3][2] = K[3] / M[3];
A[3][3] = (-K[3] - K[4]) / M[3];
A[3][4] = K[4] / M[3];

// Fifth Cart
A[4][3] = K[4] / M[4];
A[4][4] = (-K[4] - K[5]) / M[4];
A[4][5] = K[5] / M[4];

// Sixth Cart
A[5][4] = K[5] / M[5];
A[5][5] = (-K[5] - K[6]) / M[5];
A[5][6] = K[6] / M[5];

// Seventh Cart
A[6][5] = K[6] / M[6];
A[6][6] = (-K[6] - K[7]) / M[6];
A[6][7] = K[7] / M[6];
```

```
// Eighth Cart
A[7][6] = K[7] / M[7];
A[7][7] = (-K[7] - K[8]) / M[7];
A[7][8] = K[8] / M[7];

// Ninth Cart
A[8][7] = K[8] / M[8];
A[8][8] = (-K[8] - K[9]) / M[8];
A[8][9] = K[9] / M[8];

// Tenth Cart
A[9][8] = K[9] / M[9];
A[9][9] = -K[9] / M[9];

// Define the initial guess for the eigen vector
double *x;
x = new double[length];
x[0] = 1;
x[1] = 2;
x[2] = 3;
x[3] = 4;
x[4] = 5;
x[5] = 6;
x[6] = 7;
x[7] = 8;
x[8] = 9;
x[9] = 10;

// Assign a values for a copy of the A matrix
for (int i = 0; i < length; i++)
{
        for (int j = 0; j < length; j++)
        {
                B[i][j] = A[i][j];
        }
}
// -------------------------------------------------------------------------------------------
//
//      PART 3: Solving the eigen problem
//
// -------------------------------------------------------------------------------------------
```

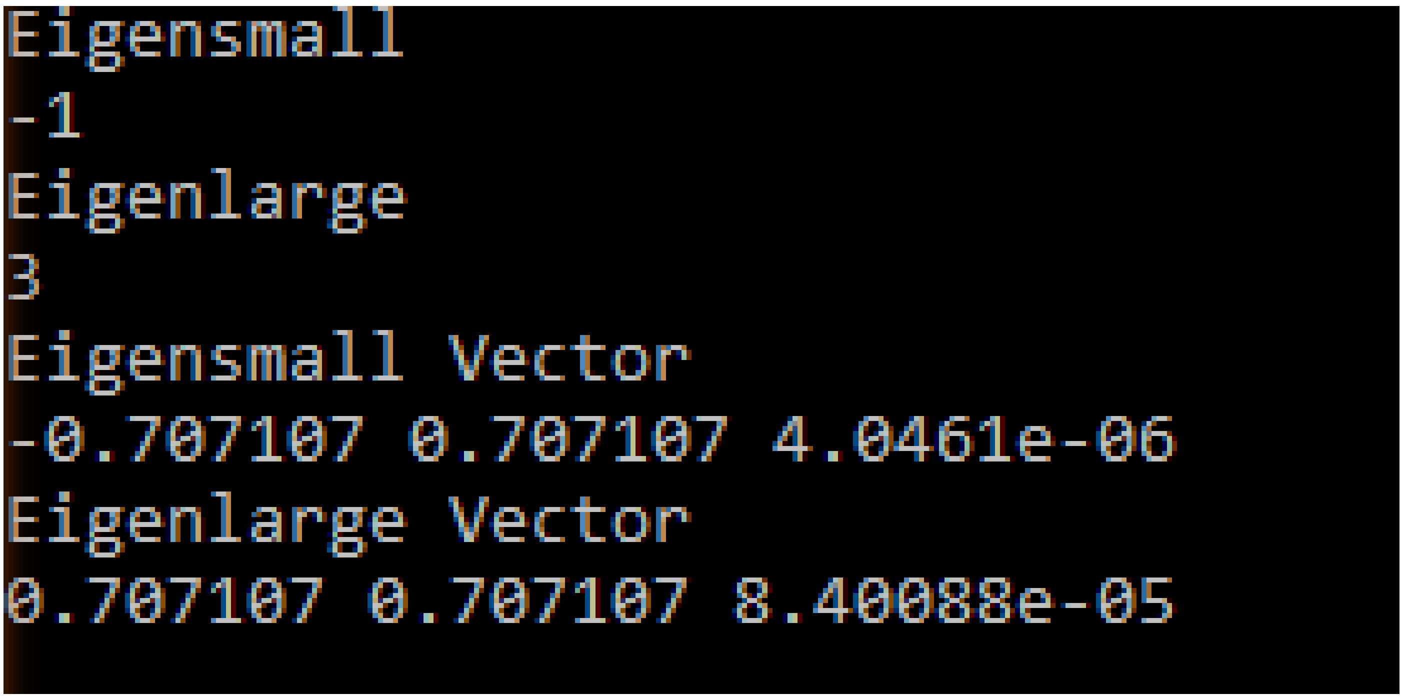
```
        // Use eigenshift to get the natural frequencies
        // and eigenvectors for the lowest and highest
        // calculated spring nodes
        eigenshift(A, x, length, 1e-8);
        // Reset the matrix
        for (int i = 0; i < length; i++)
        {
                for (int j = 0; j < length; j++)
                {
                        A[i][j] = B[i][j];
                }
        }
        // Use the eigenall to get the natural frequencies and eigen vectors for all 10 modes
        eigenall(A, x, length, 1e-8);

        // ------------------------------------------------------------------------------------------
        //
        //      PART 4: Displaying/writing the results
        //
        // ------------------------------------------------------------------------------------------

        // Printed to the command line and copied into an
        // excel spreadsheet. See the report for screenshots and tables.

        // ------------------------------------------------------------------------------------------
        //
        //      PART 5: Housekeeping
        //
        // ------------------------------------------------------------------------------------------

        for(int i = 0; i < n; i++) {
                delete [] A[i];
        }
        delete [] A;

        cout << "I'm␣finished!"<<endl;
}
```

2.4 Screenshots from Testing and Output



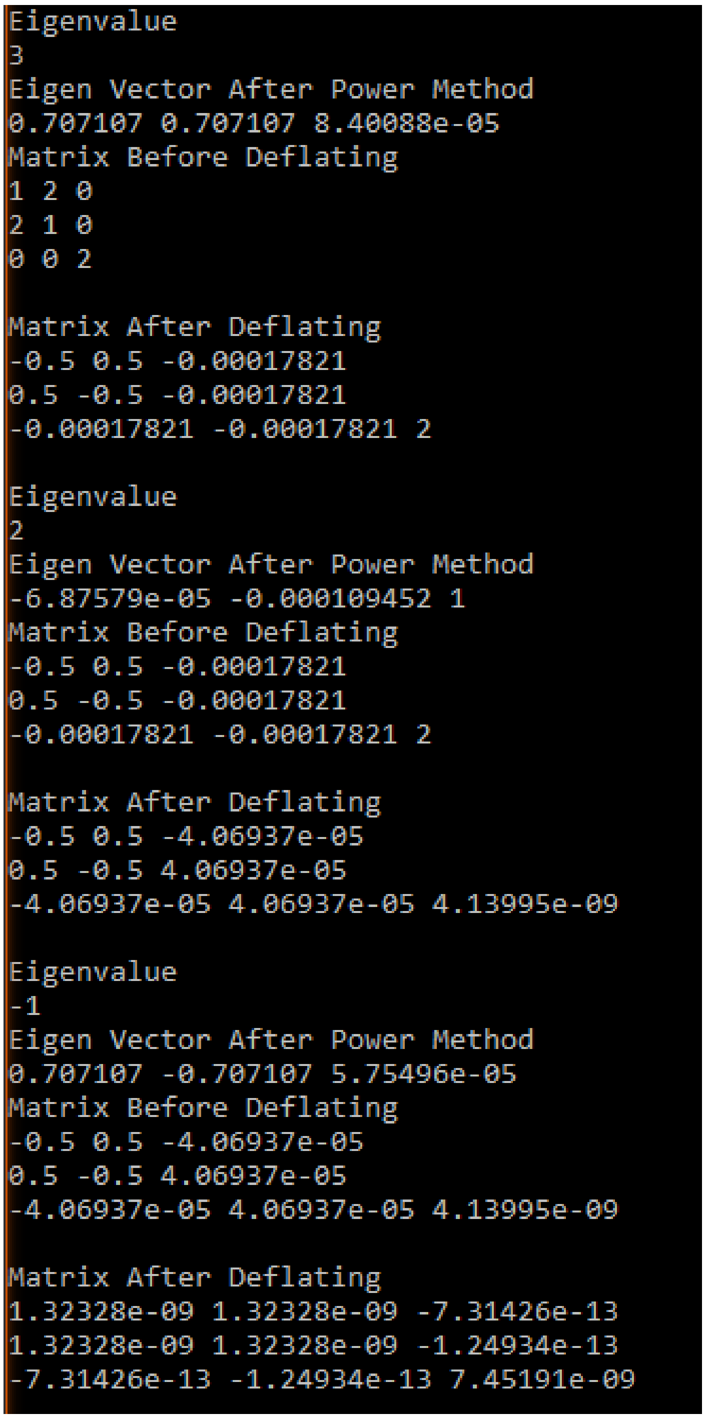
```
3
0.707107 0.707107 8.40088e-05
-0.5 0.5 -0.00017821
0.5 -0.5 -0.00017821
-0.00017821 -0.00017821 2
```

Figure 5: Power Method and Deflation Test



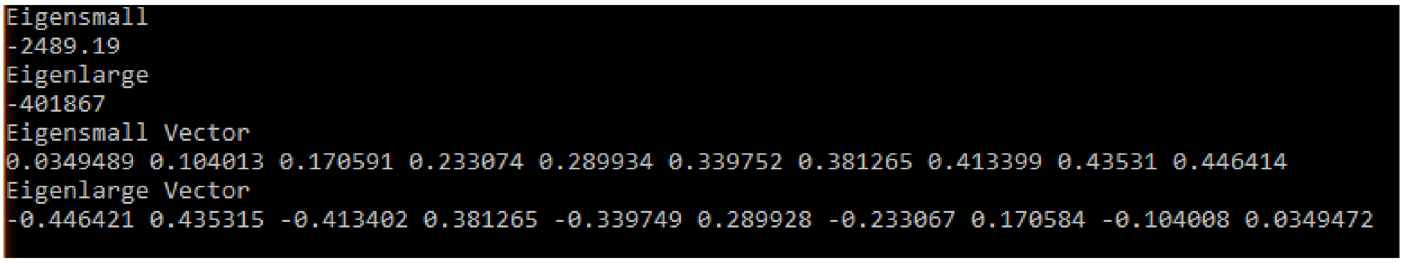
```
Eigensmall
-1
Eigenlarge
3
Eigensmall Vector
-0.707107 0.707107 4.0461e-06
Eigenlarge Vector
0.707107 0.707107 8.40088e-05
```

Figure 6: Eigen Shift Test



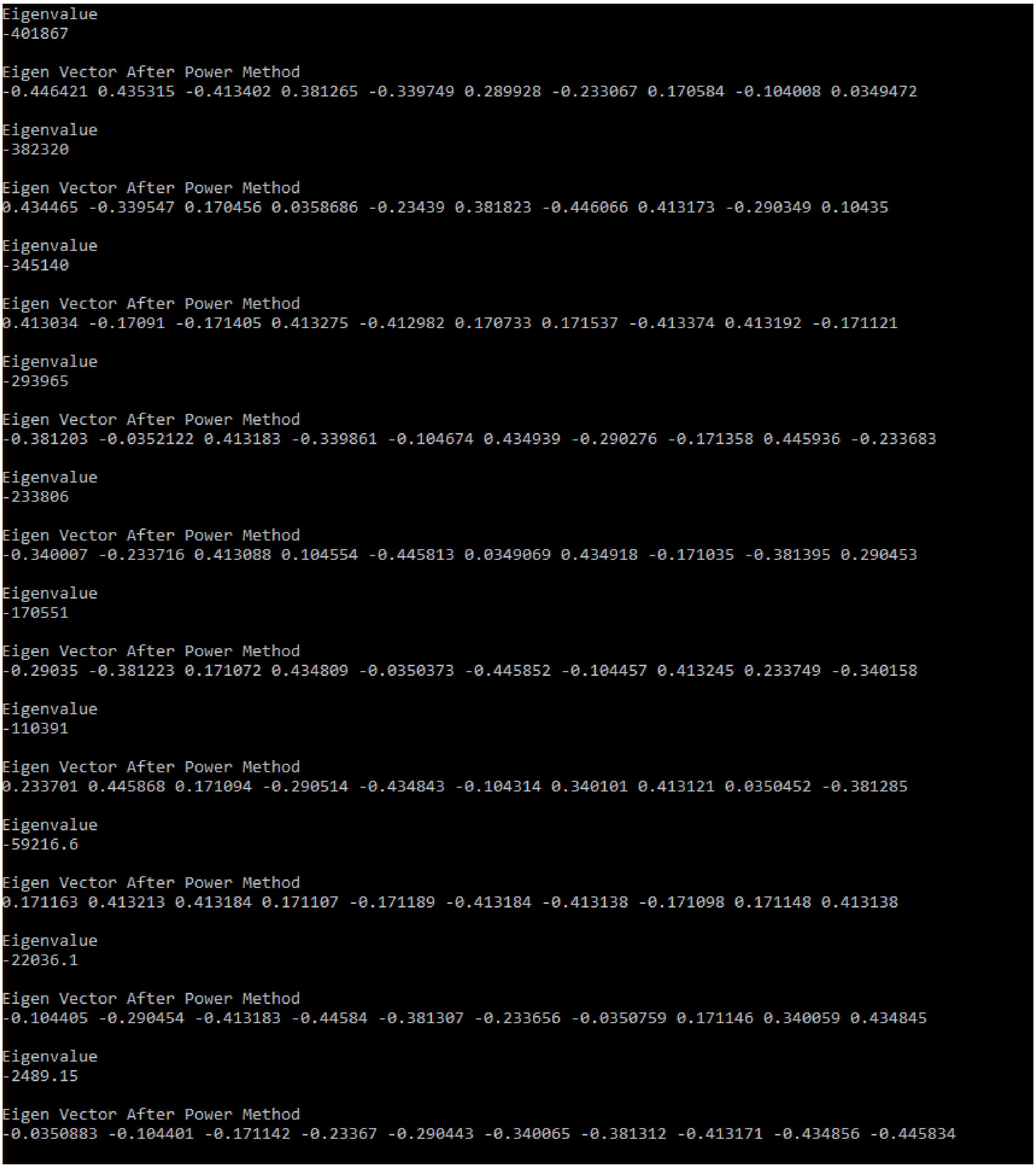
```
Eigenvalue
3
Eigen Vector After Power Method
0.707107 0.707107 8.40088e-05
Matrix Before Deflating
1 2 0
2 1 0
0 0 2

Matrix After Deflating
-0.5 0.5 -0.00017821
0.5 -0.5 -0.00017821
-0.00017821 -0.00017821 2

Eigenvalue
2
Eigen Vector After Power Method
-6.87579e-05 -0.000109452 1
Matrix Before Deflating
-0.5 0.5 -0.00017821
0.5 -0.5 -0.00017821
-0.00017821 -0.00017821 2

Matrix After Deflating
-0.5 0.5 -4.06937e-05
0.5 -0.5 4.06937e-05
-4.06937e-05 4.06937e-05 4.13995e-09

Eigenvalue
-1
Eigen Vector After Power Method
0.707107 -0.707107 5.75496e-05
Matrix Before Deflating
-0.5 0.5 -4.06937e-05
0.5 -0.5 4.06937e-05
-4.06937e-05 4.06937e-05 4.13995e-09

Matrix After Deflating
1.32328e-09 1.32328e-09 -7.31426e-13
1.32328e-09 1.32328e-09 -1.24934e-13
-7.31426e-13 -1.24934e-13 7.45191e-09
```

Figure 7: Eigen All Test



```
Eigensmall
-2489.19
Eigenlarge
-401867
Eigensmall Vector
0.0349489 0.104013 0.170591 0.233074 0.289934 0.339752 0.381265 0.413399 0.43531 0.446414
Eigenlarge Vector
-0.446421 0.435315 -0.413402 0.381265 -0.339749 0.289928 -0.233067 0.170584 -0.104008 0.0349472
```

Figure 8: Eigen Shift Output



```
Eigenvalue
-401867

Eigen Vector After Power Method
-0.446421 0.435315 -0.413402 0.381265 -0.339749 0.289928 -0.233067 0.170584 -0.104008 0.0349472

Eigenvalue
-382320

Eigen Vector After Power Method
0.434465 -0.339547 0.170456 0.0358686 -0.23439 0.381823 -0.446066 0.413173 -0.290349 0.10435

Eigenvalue
-345140

Eigen Vector After Power Method
0.413034 -0.17091 -0.171405 0.413275 -0.412982 0.170733 0.171537 -0.413374 0.413192 -0.171121

Eigenvalue
-293965

Eigen Vector After Power Method
-0.381203 -0.0352122 0.413183 -0.339861 -0.104674 0.434939 -0.290276 -0.171358 0.445936 -0.233683

Eigenvalue
-233806

Eigen Vector After Power Method
-0.340007 -0.233716 0.413088 0.104554 -0.445813 0.0349069 0.434918 -0.171035 -0.381395 0.290453

Eigenvalue
-170551

Eigen Vector After Power Method
-0.29035 -0.381223 0.171072 0.434809 -0.0350373 -0.445852 -0.104457 0.413245 0.233749 -0.340158

Eigenvalue
-110391

Eigen Vector After Power Method
0.233701 0.445868 0.171094 -0.290514 -0.434843 -0.104314 0.340101 0.413121 0.0350452 -0.381285

Eigenvalue
-59216.6

Eigen Vector After Power Method
0.171163 0.413213 0.413184 0.171107 -0.171189 -0.413184 -0.413138 -0.171098 0.171148 0.413138

Eigenvalue
-22036.1

Eigen Vector After Power Method
-0.104405 -0.290454 -0.413183 -0.44584 -0.381307 -0.233656 -0.0350759 0.171146 0.340059 0.434845

Eigenvalue
-2489.15

Eigen Vector After Power Method
-0.0350883 -0.104401 -0.171142 -0.23367 -0.290443 -0.340065 -0.381312 -0.413171 -0.434856 -0.445834
```

Figure 9: Eigen All Output