ENGSCI 331:

Due on Thursday, September 26, 2013

Ben Goodger

ID: 1822049

Task 1: Program Output

The following output was obtained when running the program for the provided application. (Some minor formatting has been done not affecting the results)

The implementation was tested using the test code provided in the appendix. The correct output was provided in the notes.

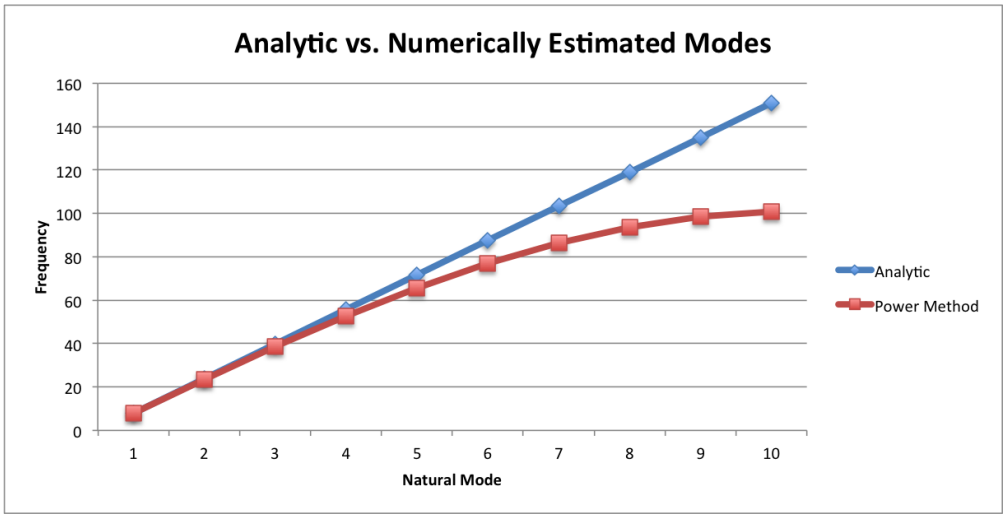
Shifting
Natural Frequency : 100.893
y: 0.445853 -0.434871 0.413179 -0.381311 0.340054 -0.290426 0.23365 -0.171124 0.104388 -0.0350837
Natural Frequency : 7.94046
y: -0.0350924 -0.104412 -0.171159 -0.233687 -0.290458 -0.340074 -0.381314 -0.413164 -0.434843 -0.445817

All
y1: 0.445853 -0.434871 0.413179 -0.381311 0.340054 -0.290426 0.23365 -0.171124 0.104388 -0.0350837
Natural Frequency : 100.893
y2: 0.434845 -0.340048 0.17112 0.0351121 -0.233691 0.381328 -0.445842 0.413171 -0.290439 0.104398
Natural Frequency : 98.4088
y3: 0.413167 -0.171134 -0.171149 0.413174 -0.413166 0.171129 0.171153 -0.413178 0.413173 -0.171141
Natural Frequency : 93.5013
y4: 0.38131 0.0350918 -0.413173 0.340058 0.104408 -0.434859 0.290437 0.171148 -0.445837 0.233668
Natural Frequency : 86.2915
y5: 0.340061 0.23367 -0.413168 -0.104405 0.445834 -0.0350821 -0.434859 0.171138 0.381316 -0.290443
Natural Frequency : 76.957
y6: 0.290441 0.381312 -0.171139 -0.434857 0.0350844 0.445835 0.104403 -0.413171 -0.23367 0.340064
Natural Frequency : 65.7274
y7: 0.233668 0.445834 0.171143 -0.29044 -0.434857 -0.104402 0.340063 0.413173 0.0350892 -0.381313
Natural Frequency : 52.8795
y8: 0.171141 0.41317 0.413171 0.171142 -0.17114 -0.413171 -0.413172 -0.171143 0.171141 0.413172
Natural Frequency : 38.7295
y9: 0.1044 0.290441 0.413171 0.445835 0.381313 0.233669 0.0350886 -0.171141 -0.340064 -0.434858
Natural Frequency : 23.6259
y10: 0.035088 0.1044 0.171141 0.233668 0.290442 0.340064 0.381312 0.413172 0.434857 0.445835
Natural Frequency : 7.94046

Analytical Modes:
7.94863 23.8459 39.7431 55.6404 71.5377 87.4349 103.332 119.229 135.127 151.024

Task 2/3: Interpretation of Model

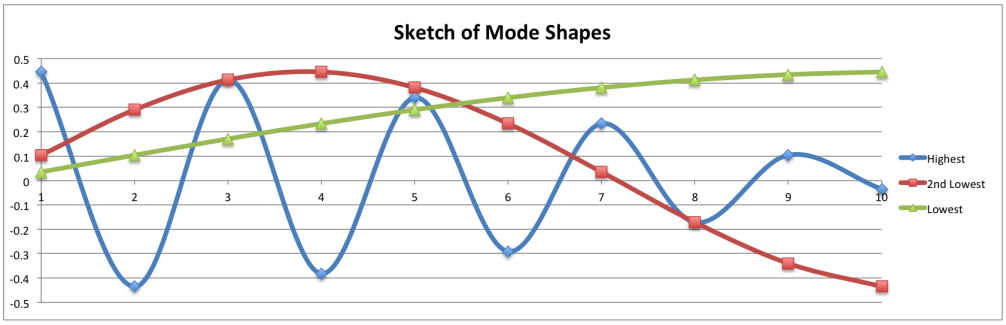
Difference between Numeric and Analytic solutions



The difference between the analytic and numerically calculated frequencies is due to discretisation error. Our eigen solution breaks the continuous mass into elements which results in an error.

This is why the largest error occurs on the 10th mode, as it contains the error from the full ten discretisations.

If the error was due to computational error (only minimal in this case) the largest error would be on the first mode as that is the last one we found.



A normal mode for a system represents the displacement when oscillating at the natural frequency.

To sketch the Mode Shapes, we plotted the Eigen vector corresponding to the desired Modal frequency. This represents the relative displacement of the masses.

The relative sign of the eigen vector components corresponds to phase of the mass' oscillation. I.e. if two components have the same sign, they are in phase. The sign of a component in isollation is meaningless as it depends on the signs of the other components.

Appendices

Code Files

```
/*
    APPLICATION OF EIGEN FUNCTIONS TO SOLVE MASS SYSTEM
    Ben Goodger
    1822049
*/

#define _CRT_SECURE_NO_DEPRECATE

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

#include "myEigenFunctions.h"
using namespace std;

#define PI 3.14159265358979323846
#define eps 0.00000000000000000000001

int main(void)
{
    // Defining local variables to be used:

    double *M = new double [n];          // Mass vector
           *K = new double [n];          // Spring vector

    double **A= new double* [n];         // System Matrix

    double G = 7.929*pow(10.0,10);       // Shear modulus
    double rho = 7751;                   // Density
    double D = 0.005;                    // Wire diameter
    double R = 0.0532;                   // Mean coil radius
    double Na = 10;                      // number of active coils

    n =(int) Na;                         // number of coils and masses are the same in this discretisation

    for(int i = 0; i < n; i++)
        A[i] = new double [n];

    // Values given for problem
    for(int i = 0; i < n; i++)  {
        M[i] = PI*PI*D*D*rho*R/2;
        K[i] = (G*pow(D,4))/(64*pow(R,3));
    }

    K[0] = 2*K[0]; // Correct first value

    // Construct the A matrix
    A[0][0] = ( -K[0] -K[1]) / M[0];

    A[n-1][n-1] = -K[n-1] / M[n-1];
    A[n-1][n-2] = K[n-1] / M[n-1];
    A[n-2][n-1] = K[n-1] / M[n-2];

    for(int i = 1; i < n-1; i++) {
```

```
        A[i][i] = (-K[i] -K[i+1]) / M[i];
        A[i][i-1] = K[i] / M[i];
        A[i-1][i] = K[i] / M[i-1];
    }

    // Allocating/Initial values needed
    double* y = new double [n];
    double lambda = 0;
    double delta = eps;

    cout << "\n" << "Shifting" << "\n";
    eigen_shift (A, y, n, lambda, delta);

    cout << "\n\nAll" << "\n";
    eigen_all (A, y, n, lambda, delta);


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

    delete [] A;
    delete [] M;
    delete [] K;

}
```

```
/*
    EIGEN FUNCTION HEADER FILE
    Ben Goodger
    1822049
*/

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

#define PI 3.14159265358979323846
#define eps 0.00000000000000000001

void power_method(double **A, double *y, int n, double &lamdba, double delta);
void eigen_all (double **A, double *y, int n, double &lambda, double delta);
double** deflate (double **A, int n,  double *y, double lambda);
void eigen_shift (double **A, double *y, int n, double &lambda, double delta);

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

```
/*
    EIGEN FUNCTION IMPLEMENTATION
    Ben Goodger
    1822049
*/

#include "myEigenFunctions.h"

void eigen_shift (double **A, double *y, int n, double &lambda, double delta) {
```

```
    // Calculates the largest and smallest eigen value.

    double largeLambda;

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

    power_method(A, y, n, lambda, delta);

    cout << "Largest lambda " << " : " << lambda << "\n";
    cout << "Natural Frequency " << " : " << sqrt(lambda)/(2*PI) << "\n";
    cout << "y"  << ":\n";
    for(int i = 0; i < n; i++)
        cout << y[i] << "  ";

    largeLambda = lambda; // Store it for later

    // Shift matrix
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            B[i][j] = A[i][j] -  ((i==j)? largeLambda : 0);
        }
    }

    cout << "\n";

    power_method(B, y, n, lambda, delta);

    // Lambda is transformed back to the original solution space
    cout << "\nsmallest lambda " << " : " << (largeLambda - lambda) << "\n";
    cout << "Natural Frequency " << " : " << sqrt(largeLambda - lambda)/(2*PI)<< "\n";
    cout << "y"  << ":\n";
    for(int i = 0; i < n; i++)
        cout << y[i] << "  ";

    cout << "\n";


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

    delete [] B;
}


void eigen_all (double **A, double *y, int n, double &lambda, double delta) {

    // Prints out all the eigen vectors, values and frequencies

    for(int j = 0; j < n; j++) {
        power_method(A, y, n, lambda, delta);
        cout << "y" << j+1  << ":\n";
        for(int i = 0; i < n; i++) {
            cout << y[i] << "  ";
        }
        cout << "\n" << "lambda " << j+1 << " : " << lambda << "\n";
        cout << "Natural Frequency " << " : " << sqrt(-lambda)/(2*PI) << "\n\n";

        A = deflate(A,n,y,lambda);
    }
```

```
}

double** deflate (double **A, int n,  double *y, double lambda) {

    // Creates a matrix without the specified eigen value/vector

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

    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            B[i][j] = A[i][j] - lambda*y[i]*y[j];
        }
    }

    return B; // Cant delete, it's needed later

}

void power_method (double **A, double *y, int n, double &lambda, double delta) {

    // Calculates the largest eigen vector and value for a square matrix A
    // The initial y vector is modified in place to be the normalised eigen vector
    // delta is the user defined tolerance
    // A must be a square matrix of dimension n, and y must be n long

    lambda = Norm(y ,n);
    double lambdaOld;
    double yNew[n];

    // Initial guess for the eigen vector
    for(int i = 0; i < n; i++)
        y[i] = 1/sqrt(n);

    while (true) {

        // A * y
        for(int i = 0; i < n; i++) {
            yNew[i] = 0;
            for(int j = 0; j < n; j++) {
                yNew[i] += A[i][j] * y[j];
            }
        }

        for(int i = 0; i < n; i++)
            y[i] = yNew[i];

        lambdaOld = lambda;
        lambda = Norm(y ,n);

        // The following checks for negative eigen values
        // If it is negative the eigen value needs to be adjusted
        for (int i = 0; i < n; i++) {
            if ((abs(y[i]) > eps) && (abs(yNew[i])>eps)) {
                if (y[i] + yNew[i] < y[i] + yNew[i]) {
                    lambda = -1*lambda;
                    break;
                }
            }
        }
```

```
        // y = y / lambda
        for(int i = 0; i < n; i++) {
            y[i] = y[i] / lambda;
        }

        if ((abs(lambda - lambdaOld) / abs(lambda)) < delta)  {
            // We have converged!
            return;
        }
    }


}

/*
    MATRIX OPERATIONS
*/
double Norm(double *A, int n) {

    // Returns the (pythagorean?) norm of a vector

    return  sqrt(DotProduct(A,A,n));

}

double DotProduct(double *A, double *B, int n) {
    // Dot product of two vectors

    double dot = 0.0;
    for(int i = 0; i < n; i++) {
        dot += A[i]*B[i];
    }
    return dot;

}
```

Test routine:

```
#include <iostream>
#include <cmath>
#include "myEigenFunctions.h"

using namespace std;

int main (void) {

    int n = 3;

    double *y = NULL;
    y = new double [n];

    double **A = NULL;
    A = new double* [n];

    for(int i = 0; i < n; i++)
        A[i] = new double [n];

    y[0] = 1;
    y[1] = 0;
    y[3] = 0;

    A[0][0] = 5;
    A[0][1] = -2;
    A[0][2] = 0;

    A[1][0] = -2;
    A[1][1] = 3;
    A[1][2] = -1;

    A[2][0] = 0;
    A[2][1] = -1;
    A[2][2] = 1;

    double delta = 0.0001;
    double lambda = 1;

    eigen_shift(A, y, n, lambda, delta);
    eigen_all(A, y, n, lambda, delta);
    return 0;
}
```