Eigenvalue Analysis

Saran Ahluwalia and Mountain Chan

October 9, 2020

Eigenvalue and Eigenvector Analysis for Damped Systems

In a mass spring system, ideally when there are no damping involved, then the mass spring system will have no energy loss, but when damping is within the system, we can turn the problem into an ODE problem. From Hooke's Law, we get that,

$$F = -kx$$

and we also know that $F = ma$, then

$$ma = -kx$$

from Physics, we know that acceleration is the second derivative of position, thus

$$m\frac{dx^2}{d^2t} = -kx$$

This is sufficient for a no damp mass spring system, but for a damped mass spring system, we need a damp constant which is proportional to velocity and it opposes the force and again from physics, velocity is the first derivative of position, thus we obtain

$$m\frac{dx^2}{d^2t} + c\frac{dx}{dt} + kx = 0$$

With the given physical parameters, we obtain a complex roots for the natural frequency which is the eigenvalues which showed that this is an underdamping system and the system should be damped by a factor of $e^{-ct/2m}$, if we take a limit $t \to \infty$, $e^{-ct/2m} \to 0$. Hence we are expected to see the solution from the ODE goes to 0 as $t \to \infty$.

Superposition Principle with Matrix Differential Equations

The general solution for a linear differential system can be given by

$$\boldsymbol{x}(t) = \sum_{i=1}^{n} \gamma_i e^{\lambda_i t} \boldsymbol{v}_i \tag{1}$$

Equation (1) can be expanded in terms of sines and cosines using the Euler formula, resulting in

$$\boldsymbol{x}(t) = \sum_{i=1}^{n} \left(\alpha_i \cos\left(\sqrt{\mu_i}t\right) + \beta_i \sin\left(\sqrt{\mu_i}t\right)\right) \boldsymbol{v}_i \tag{2}$$

where $\lambda = -i\sqrt{\mu}$ and the coefficients α_i and β_i are determinted by initial conditions

$$\boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{3a}$$
$$\dot{\boldsymbol{x}}(0) = \boldsymbol{z}_0 \tag{3b}$$

Cosine and sine are easily evaluated at 0, resulting in the following relations for eqs. (3a)–(3b)

$$\boldsymbol{x}(0) = \sum_{i=1}^{n} \alpha_i \boldsymbol{v}_i \tag{4a}$$

$$\dot{\boldsymbol{x}}(0) = \sum_{i=1}^{n} \beta_i \sqrt{\mu_i} \boldsymbol{v}_i \tag{4b}$$

By taking the eigenvectors $\{\boldsymbol{v}_i\}$ to be column vectors in a matrix V, we have

$$V := \begin{bmatrix} v_1^1 & v_2^1 & \cdots & v_n^1 \\ \vdots & \vdots & & \vdots \\ v_1^m & v_2^m & \cdots & v_n^m \end{bmatrix}$$

i.e., $V_i^j = v_i^j$ where v_i^j is the j^{th} component of the i^{th} eigenvector or $V := [\boldsymbol{v}_1 \ ... \ \boldsymbol{v}_n]$. Taking $\boldsymbol{\alpha} = [\alpha_1 \ ... \ \alpha_n]^T$ and $\boldsymbol{b} = \left[\beta_1\sqrt{\mu_1} \ ... \ \beta_n\sqrt{\mu_n}\right]^T$, we are able to rewrite eqs. (4a)–(4b) as the following matrix equations

$$V\boldsymbol{\alpha} = \boldsymbol{x}_0 \tag{5a}$$
$$V\boldsymbol{b} = \boldsymbol{z}_0 \tag{5b}$$

Using matrix techniques, the solution vectors α and $\boldsymbol{b}$ may be obtained. Using the definitions of these solution vectors, we have the coefficients α_i as the i^{th} component of $\boldsymbol{\alpha}$ and $\beta_i = \frac{b_i}{\sqrt{\mu_i}}$ [6].

Inverse Eigenvalue Problem

In addition to the regular eigenvalue problem discussed in the first section, there is also the inverse eigenvalue problem. The general inverse eigenvalue problem is constructing the state-space matrix [2] that will lead to a desired set of eigenvalues; in the context of the damped mass-spring eigenvalue problem, it is permuting on the masses (M), stiffness (K), and C damping such that the spring should be made out of to have a certain set of frequencies.

While the inverse eigenvalue problem is difficult to solve and is an active area of mathematical research, we will explore this idea by analyzing the effect of creating a system with variations in the parameter's for stiffness constants and damping constants. In addition to examining the configuration of masses on the spring, one may examine the effects of the values of C or K.

For our system, we defined the following in order to start our analysis:

For this system the kinetic energy is:

$$T = \frac{1}{2}[m_1\dot{q_0}(t)^2 + m_2\dot{q_1}(t)^2] = \frac{1}{2}\dot{\mathbf{q}}^T(t)\ M\ \dot{\mathbf{q}}(t) \quad (6)$$

[6]

where $\mathbf{q(t)} = [q_0(t)\ q_1(t)]^T$ is the configuration vector. The mass matrix for this system is given by:

$$M = \begin{bmatrix} m_3 & 0 \\ 0 & m_4 \end{bmatrix} \quad (7)$$

The stiffness matrix is:

$$K = \begin{bmatrix} k_3 + k_4 + k_5 & -k_4 \\ k_4 & -k_4 \end{bmatrix} \quad (8)$$

The damping matrix is:

$$C = \begin{bmatrix} c_2 + c_3 & -c_3 \\ -c_3 & c_3 \end{bmatrix} \quad (9)$$

The Python 3 code below details how we experimentally converged on a set of parameters by computing the normal modes of the state-space [4] for the case of the 2 -degree of freedom model. We created one variation for computing the natural frequencies, damping ratios, and mode shapes of the system [5].

One function will return the natural frequencies (w_n), the damped natural frequencies (w_d), the damping ratios (ζ), the right eigenvectors (X) and the left eigenvectors (Y) for a system defined by M, K and C. It should be noted that w_d and w_n are both in $rad\ /\ s$.

The other function returns the natural frequencies (w), eigenvectors (P), mode shapes (s) and the modal transformation matrix S [1] for an undamped system. Although this was not in the scope of this project, this addition did help elucidate certain properties for the eigenvectors and eigenvalues for the various modal types.

Briefly, we permuted on 51,300 permutations for values of the coordinates for m_1, m_2, c_2, c_3, k_4, k_5, and $k3$. We selected - 673 to be precise - only those values in which the real and imaginary parts of the eigenvalues which were within 1e$-$5 of the real and imaginary parts of the target eigenvalues. We further narrowed down the window of values by filtering out all imaginary values that were not within 0.00001 of the imaginary part for the target eigenvalue $-1\pm3i$. The resulting set contained 268 permutation set of parameters.

There is much variation in the eigenmodes and it can be hypothesized that using more masses could provide access to a broader range of frequencies. This case could be used to systematically place masses to achieve a set of desired frequencies [3].

From this exploration of the inverse eigenvalue problem, it seems that frequencies in general are quite arbitrary when the parameters of a system are able to be changed. However, for a given physical system, such as a guitar, the modes and frequencies are already determined. Furthermore, in a system such as a guitar, the masses on the spring would be represented by infinitesimal mass m. It should also be noted that while there are a given number of natural modes with their respective natural frequencies, superposition of these modes also happens as a system evolves in time.

For the given physical parameters, M, C, and K, we can use Matlab polyeig function to solve for the eigenvalues and corresponding eigenvectors.

Matlab Code

0.0.1 Part 1a

```
%Part a
m1 = 1;
m2 = 2;
m3 = 3;
m4 = 1;

c1 = 0.1;
c2 = 0.2;
c3 = 0.3;

k1 = 1;
k2 = 2;
k3 = 1;
k4 = 4;
k5 = 3;

M = diag([m1 m2 m3 m4]);
C = [c1+c2 0 -c2 0; 0 0 0 0; -c2 0 c2+c3 -c3; 0 0 -c3 c3];
K = [k1+k2+k5 -k2 -k5 0; -k2 k2+k3 -k3 0; -k5 -k3 k3+k4+k5 -k4; 0 0 -k4 k4
    ];

% X's column is eigenvector
% e is eigenvalue
% s conditional number for eignevalues
[X,e,s] = polyeig(K, C, M)

% Checking eigenvector
lambda = e(1);
x = X(:,1);
% Should =0 for verification
(M*lambda^2 + C*lambda + K)*x
```

From the above code, it outputs the eigenvalues and eigenvectors

```
X =

  Columns 1 through 6

  -0.8643 - 0.1592i  -0.8643 + 0.1592i   0.4609 + 0.2304i   0.4609 - 0.2304i  -0.0248 - 0.1141i  -0.0248 + 0.1141i
   0.1281 - 0.0048i   0.1281 + 0.0048i  -0.1654 - 0.0346i  -0.1654 + 0.0346i   0.0250 - 0.7564i   0.0250 + 0.7564i
   0.2702 + 0.0977i   0.2702 - 0.0977i   0.2423 + 0.0378i   0.2423 - 0.0378i  -0.0125 + 0.3346i  -0.0125 - 0.3346i
  -0.3205 - 0.1608i  -0.3205 + 0.1608i  -0.7722 - 0.2227i  -0.7722 + 0.2227i   0.0039 + 0.5490i   0.0039 - 0.5490i

  Columns 7 through 8

  -0.4364 - 0.0006i  -0.4364 + 0.0006i
  -0.5052 - 0.0027i  -0.5052 + 0.0027i
  -0.5181 - 0.0021i  -0.5181 + 0.0021i
  -0.5346 - 0.0022i  -0.5346 + 0.0022i


e =

  -0.2000 + 2.6815i
  -0.2000 - 2.6815i
  -0.1622 + 2.2804i
  -0.1622 - 2.2804i
  -0.0155 + 1.2537i
  -0.0155 - 1.2537i
  -0.0057 + 0.3513i
  -0.0057 - 0.3513i
```

where each of the columns in X matrix is a eigenvector and each e is a eigenvalues.
Next, we wrote a function that will take in the physical parameters and initial conditions as a parameter and output the solution $x(t)$, we then plotted the solution function $x(t)$ with 100 points within the interval $[0, 10]$.

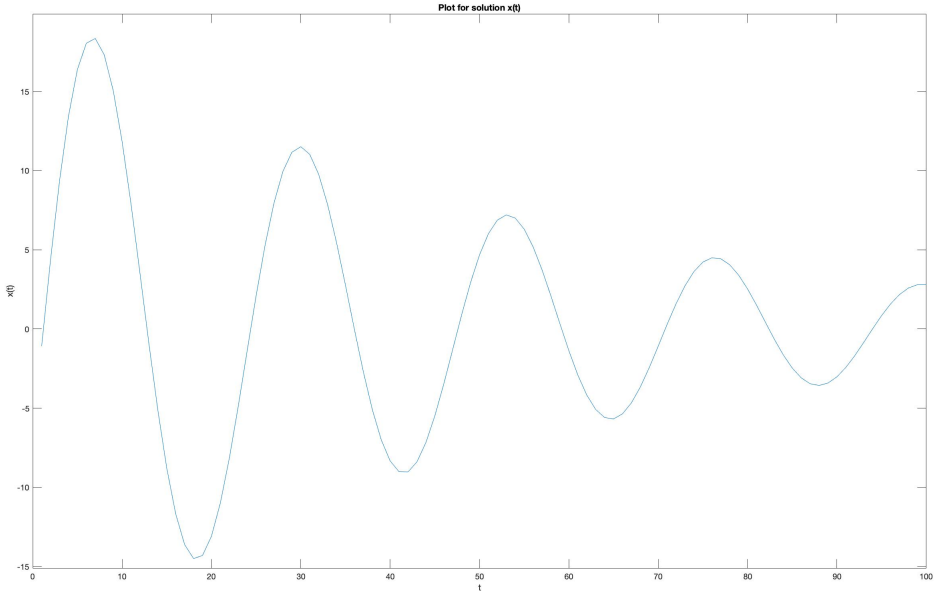
Matlab Code

0.0.2 Part 1b

```
% Input as vector
function GenSolver(m,c,k,initial)

% Initializing the value in M matrix
m1 = m(1);
m2 = m(2);
m3 = m(3);
m4 = m(4);

% Initializing the value in C matrix
c1 = c(1);
c2 = c(2);
c3 = c(3);

% Initializing the value in K matrix
k1 = k(1);
k2 = k(2);
k3 = k(3);
k4 = k(4);
k5 = k(5);

% Initializing M matrix
M = diag([m1 m2 m3 m4]);

% Initializing C matrix
C = [c1+c2 0 -c2 0; 0 0 0 0; -c2 0 c2+c3 -c3; 0 0 -c3 c3];

% Initializing K matrix
```

```
K = [k1+k2+k5 -k2 -k5 0; -k2 k2+k3 -k3 0; -k5 -k3 k3+k4+k5 -k4; 0 0 -k4 k4
    ];

% Computing eigenpairs with polynomial
[eigVec, eigVal, CondNum] = polyeig(K, C, M);

% Setting up the linear system
A = zeros(size(initial,1), 2);
B = [initial(1); initial(2); initial(3); initial(4)];
coeff = [];

% Calculating the coefficient using Ax = b
% A is the real and imaginary value of eigenvector
% B is the initial values
for i = 1: 2: size(eigVal)
    A(:,1) = real(eigVec(:,i));
    A(:,2) = imag(eigVec(:,i));
    sol = linsolve(A,B);

    % Concatinating coefficients
    coeff = [coeff; sol];
end

% Plotting
t = linspace(0,10);
y1 = exp(real(eigVal(1) * t));
y2 = coeff(1,1) * (real(eigVec(1,1)) * cos(imag(eigVal(1) * t)) - imag(
    eigVec(1,1)) * sin(imag(eigVal(1) * t)));
y3 = coeff(2,1) * (real(eigVec(1,1)) * sin(imag(eigVal(1) * t)) + imag(
    eigVec(1,1)) * cos(imag(eigVal(1) * t)));
% Elements to elements multiplicatio2
y = times(y1, (y2 + y3))

figure
plot(y)
end
```

Using the given physical parameters M, C, and K from part 1a, we plotted the first solution and obtain the following figure:



which this agrees with the physical constraint of the problem. When damping is introduced into the mass-spring system, there are now energy loss within the system and the force should decrease over time. In this case, it is represented by the position of the spring $x(t)$ and we see that the amplitude of the figure decrease over time.

0.0.3 Part 2

0.0.4 Computing Eigenmodes

```
# Here is Number 2

import matplotlib as mpl
import numpy as np
import scipy.linalg as la

mpl.rcParams["lines.linewidth"] = 2
mpl.rcParams["figure.figsize"] = (10, 6)


def compute_eigen_values(A, B=None):
    """Return sorted eigenvector/eigenvalue pairs.
    e.g. for a given system linalg.eig will return eingenvalues as:
    (array([ 0. +89.4j,  0. -89.4j,  0. +89.4j,  0. -89.4j,  0.+983.2j,
             0.-983.2j,  0. +40.7j,  0. -40.7j])
    This function will sort this eigenvalues as:
    (array([ 0. +40.7j,  0. +89.4j,  0. +89.4j,  0.+983.2j,  0. -40.7j,
             0. -89.4j,  0. -89.4j,  0.-983.2j])

    Correspondent eigenvectors will follow the same order.
    Note: Works fine for moderately sized models. Does not leverage the
    full set of constraints to optimize the solution.
    Parameters
    ----------
    A: array
```

```
        A complex or real matrix whose eigenvalues and eigenvectors
        will be computed.
    B: float or str
        Right-hand side matrix in a generalized eigenvalue problem.
        Default is None, identity matrix is assumed.
    Returns
    -------
    evalues: array
        Sorted eigenvalues
    evectors: array
        Sorted eigenvalues
    Examples
    --------
    >>> L = np.array([[2, -1, 0],
    ...               [-4, 8, -4],
    ...               [0, -4, 4]])
    >>> lam, P = compute_eigen_values(L)
    >>> lam
    array([  0.56+0.j,   2.63+0.j,  10.81+0.j])
    """
    if B is None:
        evalues, evectors = la.eig(A)
    else:
        evalues, evectors = la.eig(A, B)

    if all(eigs == 0 for eigs in evalues.imag):
        if all(eigs > 0 for eigs in evalues.real):
            idxp = evalues.real.argsort()  # positive in increasing order
            idxn = np.array([], dtype=int)
        else:
            # positive in increasing order
            idxp = evalues.real.argsort()[int(len(evalues) / 2):]
            # negative in decreasing order
            idxn = evalues.real.argsort()[int(len(evalues) / 2) - 1:: -1]

    else:
        # positive in increasing order
        idxp = evalues.imag.argsort()[int(len(evalues) / 2):]
        # negative in decreasing order
        idxn = evalues.imag.argsort()[int(len(evalues) / 2) - 1:: -1]

    idx = np.hstack([idxp, idxn])

    return evalues[idx], evectors[:, idx]


def normalize(X, Y):
    """
    Return normalized left eigenvectors.
    This function is used to normalize vectors of the matrix
    Y with respect to X so that Y.T @ X = I (identity).
    This is used to normalize the matrix with the left eigenvectors.
    Parameters
    ----------
    X: array
        A complex or real matrix
```

```
    Y: array
        A complex or real matrix to be normalized
    Returns
    -------
    Yn: array
        Normalized matrix
    Examples
    --------
    >>>
    >>> X = np.array([[ 0.84+0.j  ,  0.14-0.j  ,  0.84-0.j  ,  0.14+0.j  ],
    ...               [ 0.01-0.3j ,  0.00+0.15j,  0.01+0.3j ,  0.00-0.15j],
    ...               [-0.09+0.42j, -0.01+0.65j, -0.09-0.42j, -0.01-0.65j],
    ...               [ 0.15+0.04j, -0.74+0.j  ,  0.15-0.04j, -0.74-0.j  ]])
    >>> Y = np.array([[-0.03-0.41j,  0.04+0.1j , -0.03+0.41j,  0.04-0.1j ],
    ...               [ 0.88+0.j  ,  0.68+0.j  ,  0.88-0.j  ,  0.68-0.j  ],
    ...               [-0.21-0.j  ,  0.47+0.05j, -0.21+0.j  ,  0.47-0.05j],
    ...               [ 0.00-0.08j,  0.05-0.54j,  0.00+0.08j,  0.05+0.54j]])
    >>> Yn = normalize(X, Y)
    >>> Yn
    array([[ 0.58-0.05j,  0.12-0.06j,  0.58+0.05j,  0.12+0.06j],
           [ 0.01+1.24j, -0.07-0.82j,  0.01-1.24j, -0.07+0.82j],
           [-0.  -0.3j ,  0.01-0.57j, -0.  +0.3j ,  0.01+0.57j],
           [ 0.11-0.j  , -0.66-0.01j,  0.11+0.j  , -0.66+0.01j]])
    """
    Yn = np.zeros_like(X)
    YTX = Y.T @ X  # normalize y so that Y.T @ X will return I
    factors = [1 / a for a in np.diag(YTX)]
    # multiply each column in y by a factor in 'factors'
    for col in enumerate(Y.T):
        Yn[col[0]] = col[1] * factors[col[0]]
    Yn = Yn.T

    return Yn


def undamped_modes_system(M, K):
    r"""Return eigensolution of multiple DOF system.
    Returns the natural frequencies (w),
    eigenvectors (P), mode shapes (S) and the modal transformation
    matrix S for an undamped system.
    See Writeup for explanation of the underlying math.
    Parameters
    ----------
    M: float array
        Mass matrix
    K: float array
        Stiffness matrix
    Returns
    -------
    w: float array
        The natural frequencies of the system
    P: float array
        The eigenvectors of the system.
    S: float array
        The mass-normalized mode shapes of the system.
    Sinv: float array
```

```
        The modal transformation matrix S^-1(takes x -> r(modal coordinates))
    Notes
    -----
    Given :math:`M\ddot{x}(t)+Kx(t)=0`, with mode shapes :math:`u`, the matrix
    of mode shapes :math:`S=[u_1 u_1 \ldots]` can be created. If the modal
    coordinates are the vector :math:`r(t)`. The modal transformation separates
    space and time from :math:`x(t)` such that :math:`x(t)=S r(t)`.
    Substituting into the governing equation:
    :math:`MS\ddot{r}(t)+KSr(t)=0`
    Premultiplying by :math:`S^T`
    :math:`S^TMS\ddot{r}(t)+S^TKSr(t)=0`

    The matrices :math:`S^TMS` and :math:`S^TKS` will be diagonalized by this
    process (:math:`u_i` are the eigenvectors of :math:`M^{-1}K`).
    If scaled properly (mass normalized so :math:`u_i^TMu_i=1`) then
    :math:`S^TMS=I` and :math:`S^TKS=\Omega^2` where :math:`\Omega^2` is a
    diagonal matrix of the natural frequencies squared in radians per second.

    Further, inverses are unstable so the better way to solve linear equations is with
    Gauss elimination.
    :math:`AB=C` given known :math:`A` and :math:`C`
    is solved using `la.solve(A, C, assume_a='pos')`.
    :math:`BA=C` given known :math:`A` and :math:`C` is solved by first
    transposing the equation to :math:`A^TB^T=C^T`, then solving for
    :math:`C^T`. The resulting command is
    `la.solve(A.T, C.T, assume_a='pos').T`
    Examples
    --------
    >>> M = np.array([[4, 0, 0],
    ...               [0, 4, 0],
    ...               [0, 0, 4]])
    >>> K = np.array([[8, -4, 0],
    ...               [-4, 8, -4],
    ...               [0, -4, 4]])
    >>> w, P, S, Sinv = modes_system_undamped(M, K)
    >>> w
    array([0.45, 1.25, 1.8 ])
    >>> S
    array([[ 0.16, -0.37, -0.3 ],
           [ 0.3 , -0.16,  0.37],
           [ 0.37,  0.3 , -0.16]])
    """
    L = la.cholesky(M)
    lam, P = compute_eigen_values(la.solve(L, la.solve(L, K, assume_a='pos').T,
                                           assume_a='pos').T)
    w = np.real(np.sqrt(lam))
    S = la.solve(L, P, assume_a="pos")
    Sinv = la.solve(L.T, P, assume_a="pos").T

    return w, P, S, Sinv


def damped_modes_system(M, K, C=None):
    """Natural frequencies, damping ratios, and mode shapes of the system.
    This function will return the natural frequencies (wn), the
    damped natural frequencies (wd), the damping ratios (zeta),
```

```
the right eigenvectors (X) and the left eigenvectors (Y) for a
system defined by M, K and C.
If the dampind matrix 'C' is none or if the damping is proportional,
wd and zeta will be none and X and Y will be equal.
Parameters
----------
M: array
    Mass matrix
K: array
    Stiffness matrix
C: array
    Damping matrix
Returns
-------
wn: array
    The natural frequencies of the system
wd: array
    The damped natural frequencies of the system
zeta: array
    The damping ratios
X: array
    The right eigenvectors
Y: array
    The left eigenvectors
Examples
--------
>>> M = np.array([[1, 0],
...               [0, 1]])
>>> K = np.array([[2, -1],
...               [-1, 6]])
>>> C = np.array([[0.3, -0.02],
...               [-0.02, 0.1]])
>>> wn, wd, zeta, X, Y = modes_system(M, K, C)
Damping is non-proportional, eigenvectors are complex.
>>> wn
array([1.33, 2.5 , 1.33, 2.5 ])
>>> wd
array([1.32, 2.5 , 1.32, 2.5 ])
>>> zeta
array([0.11, 0.02, 0.11, 0.02])
>>> X
array([[-0.06-0.58j, -0.01+0.08j, -0.06+0.58j, -0.01-0.08j],
       [-0.  -0.14j, -0.01-0.36j, -0.  +0.14j, -0.01+0.36j],
       [ 0.78+0.j  , -0.21-0.03j,  0.78-0.j  , -0.21+0.03j],
       [ 0.18+0.01j,  0.9 +0.j  ,  0.18-0.01j,  0.9 -0.j  ]])
>>> Y
array([[ 0.02+0.82j,  0.01-0.31j,  0.02-0.82j,  0.01+0.31j],
       [-0.05+0.18j,  0.01+1.31j, -0.05-0.18j,  0.01-1.31j],
       [ 0.61+0.06j, -0.12-0.02j,  0.61-0.06j, -0.12+0.02j],
       [ 0.14+0.03j,  0.53+0.j  ,  0.14-0.03j,  0.53-0.j  ]])
>>> C = 0.2*K # with proportional damping
>>> wn, wd, zeta, X, Y = modes_system(M, K, C)
Damping is proportional or zero, eigenvectors are real
>>> X
array([[-0.97,  0.23],
       [-0.23, -0.97]])
```

```
    """

    n = len(M)

    Z = np.zeros((n, n))
    I = np.eye(n)

    if (
                        C is None or
                    np.all(C == 0) or
                    la.norm(  # check if C has only zero entries
                                la.solve(M, C, assume_a="pos") @ K - \
                                    la.solve(M, K, assume_a="pos") @ C, 2
                    ) <
                        1e-8 * la.norm(la.solve(M, K, assume_a="pos") @ C, 2)
    ):
        w, P, S, Sinv = modes_system_undamped(M, K)
        wn = w
        wd = w
        # zeta = None
        zeta = np.diag(S.T @ C @ S) / 2 / wn
        wd = wn * np.sqrt(1 - zeta ** 2)
        X = P
        Y = P
        print("Damping is proportional or zero, eigenvectors are real")
        return wn, wd, zeta, X, Y

    Z = np.zeros((n, n))
    I = np.eye(n)

    # creates the state space matrix
    A = np.vstack(
        [
            np.hstack([Z, I]),
            np.hstack(
                [-la.solve(M, K, assume_a="pos"),
                 - la.solve(M, C, assume_a="pos")]
            ),
        ]
    )

    w, X = compute_eigen_values(A)
    _, Y = compute_eigen_values(A.T)

    wd = abs(np.imag(w))
    wn = np.absolute(w)
    zeta = -np.real(w) / np.absolute(w)

    Y = normalize(X, Y)

    print("Damping is non-proportional, eigenvectors are complex.")

    return wn, wd, zeta, X, Y
```

0.0.5 Computing Permutations of Parameters

```
import numpy as np
import itertools as it
import eigenmodes as em


if __name__ == "__main__":

    m1 = 0;
    m2 = 0;

    k1 = 0;
    k2 = 0;
    k3 = 0;

    c1 = 0;

    target = -2 + 2j

    M3 = list(np.arange(0.1, 0.7, 0.1))
    M4 = list(np.arange(0.0, 0.05, 0.01))

    C2 = list(np.arange(0.15, 0.60, 0.1))
    C3 = list(np.arange(0.015, 0.03, 0.01))

    K4 = list(np.arange(0.01, 0.2, 0.01))
    K5 = list(np.arange(0.1, 1, 0.1))

    l = [M3, M4, C2, C3, K4, K5]

    combinations = list(it.product(*l))

    collected_params = []

    for _, permutation in enumerate(combinations):

        m3 = permutation[0]
        m4 = permutation[1]

        c2 = permutation[2]
        c3 = permutation[3]

        k4 = permutation[4]
        k5 = permutation[5]

        M = np.array([[m3, 0],
                      [0, m4]])

        K = np.array([[k3 + k4 + k5, -k4],
                      [-k4, k4]])

        C = np.array([[c2 + c3, -c3],
                      [-c3, c3]])

        try:
```

```
        wn, wd, zeta, X, Y = em.modes_system(M, K, C)
    except:
        continue

    if target.imag - wd[0] < 1e-5:
        params = {
            "m3": m3,
            "m4": m4,
            "c2": c2,
            "c3": c3,
            "k4": k4,
            "k5": k5,
            "wd": wd,
            "wn": wn
        }

        collected_params.append(params)

    assert (len(collected_params) > 0)

    reduced_search_params = [item for item in collected_params
                             if item['wd'][1] - 3.0 <= 0.000001
                             ]

    print(reduced_search_params[0])
```

References

[1] Douglas E Adams. *ME 563: Multiple Degree of Freedom Modal Coordinate Transformation*. URL: https://engineering.purdue.edu/~deadams/ME563.

[2] Erik Cheever and Swarthmore College. URL: https://lpsa.swarthmore.edu/Representations/SysRepSS.html.

[3] Jean-Pierre Dedieu and Françoise Tisseur. "Perturbation theory for homogeneous polynomial eigenvalue problems". In: *Linear Algebra and its Applications* 358.1 (2003), pp. 71–94. ISSN: 0024-3795. DOI: https://doi.org/10.1016/S0024-3795(01)00423-2. URL: http://www.sciencedirect.com/science/article/pii/S0024379501004232.

[4] F. Tisseur and K. Meerbergen. "The Quadratic Eigenvalue Problem". In: *SIAM Review* 43.2 (2001), pp. 235–286. DOI: 10.1137/S0036144500381988. eprint: https://doi.org/10.1137/S0036144500381988. URL: https://doi.org/10.1137/S0036144500381988.

[5] Charles G. Torre. *How To Find Normal Modes*. Aug. 2014. URL: https://digitalcommons.usu.edu/foundation_wave/20.

[6] J. Kim Vandiver. *Vibration by Mode Superposition*. 2011.