Numerical Prediction of Eigenmodes in the Chladni Experiment

10 points

Wave problems are very important in engineering. A wave equation may govern physical phenomena such as string or drum vibration in musical instruments and earthquake wave propagation. When the external force (excitation) equal to some frequencies, the response (vibration) may exhibit patterns: part of the domain does not vibrate with velocity equal to zero in time. Such patterns are called **eigenmodes**. The eigenmodes can be calculated by solving an eigenvalue problem.

For this problem, you will compute the **eigenmodes** of vibration for a flat plate.The plate is excited using a frequency generator which results in a vibration pattern characteristic of the frequency. More on this is explained in the following videos:

- Introduction to the Chladni Experiment (https://www.youtube.com/watch?v=wYoxOJDrZzw)
- Formation of Nodal lines (https://www.youtube.com/watch?v=wvJAgrUBF4w)

The plate vibration can be modeled by the wave equation. The plate is placed in x and y direction. x is width and y is height. The plate vibrates in z direction.

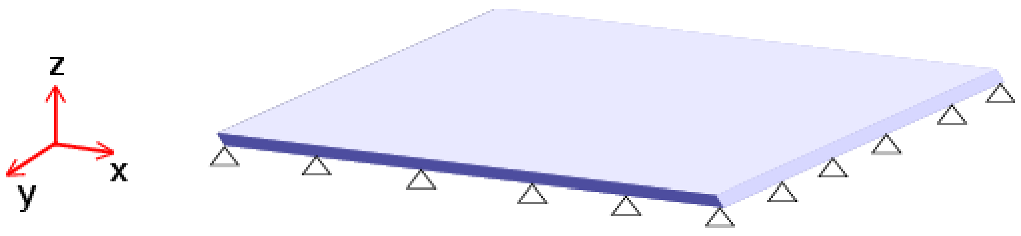


Figure 1 Side view of the plate

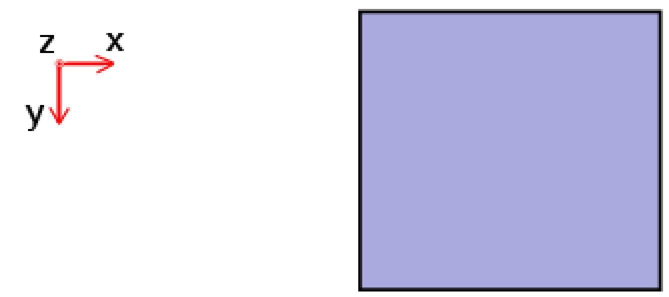


Figure 2 Top view of the plate

Derivation of the Wave Equation (click to view)

Let $\vec{x}$, where $\vec{x} = (x, y)$, be any point on the plate such that $U(\vec{x}, t)$ represents the displacement in the z direction at point $\vec{x}$ at a given time t. The plate resists bending with a restoring force F, which can be approximated by the second derivative of u. This may be expressed as

$$F = c(x, y)(U_{xx} + U_{yy}),$$

where $c(x, y)$ is a function representing how much the plate resists bending, roughly a "stiffness".

From Newton's second law of motion, we have $F = ma$, where F is the force, $a = \partial^2 U/\partial t^2$ is acceleration, and m is a localized mass, i.e. a mass density. Hence

$$U_{tt} = \frac{c}{m}(U_{xx} + U_{yy}),$$

which is called the *wave equation*.

Assuming the motion is sinusoidal in time (or, "time-harmonic"), the eigenmode solution can be written as:

$$U(\vec{x}, t) = u(\vec{x})e^{i\omega t}.$$

Plugging this into the the wave equation, we find:

$$-\omega^2 u = \frac{c(x, y)}{m}(u_{xx} + u_{yy}), \qquad (1)$$

which is an eigenvalue problem for the differential operator $u_{xx} + u_{yy}$. So what we will do is use finite differences to make a discrete version of that operator and then feed that to an Arnoldi-based eigensolver to find the vibrational modes.

What Do I Need To Do? (click to view)

We will start by creating matrices that take x and y derivatives on a Cartesian grid. To do so, we will follow these step.

1. You are given `n` points equispaced on $(0, 1)$ in `nodes_1d`. First, use any `nstencil` consecutive nodes from `nodes_1d` to obtain a differentiation matrix `D2` that, using polynomial interpolation, maps function values at these nodes to the approximate values of the second derivative.

   Note that `nstencil` will always be an odd number.

   You may verify that the matrix does what it is expected to do by using it to take a second derivative you know, e.g. of $\cos(x)$.

2. Next, by using the middle row of `D2` as a "sliding" stencil and the top and bottom halves of `D2` near the boundaries, create a matrix `d2x_mat_1d` that maps a vector of function values at `nodes_1d` to approximate second derviatives at the same points.

   You may verify that the matrix does what it is expected to do by using it to take a second derivative you know, e.g. of $\cos(x)$.

3. Use `d2x_mat_1d` to build matrices that take second derivatives in the x and y directions on an $n \times n$ grid in two dimensions. Create two matrices `d2x_mat` and `d2y_mat` that map vectors of length $n \cdot n$ representing function values on the 2D grid to vectors containing approximate second derivatives along the x and y directions respectively.

The given array `numbering` determines how the two-dimensional grid is mapped to the entries of a (one-dimensional) vector.

You may verify that the matrices do what they are expected to do by using them to take a second derivative you know, e.g. of $\cos(x)$ and $\cos(y)$. To help with that, you are given vectors `x` and `y` that represent the x and y coordinates of each point in the vector.

4. Use `d2x_mat` and `d2y_mat` to compute a matrix representing the right-hand side of equation (1), where the pointwise value of $c(x, y)/m$ is provided in the variable `stiffness`. Plot the stiffness to see which of the (randomly chosen) stiffness functions is being used.

   To enforce the clamped-plate boundary conditions, the unknowns corresponding to values of u on the boundary must be set to zero. Equivalently, to ensure that they contribute nothing to the relevant finite difference calculations, we can simply eliminate these unknowns from the matrix, reducing the size of the system from $n \cdot n$ to $(n-2) \cdot (n-2)$. Use `numbering` as a guide for which unknowns to eliminate. Apply this elimination process to the right-hand side operator and store the result in `interior_operator`.

   Hint: Indexing rows and columns with an array of indices corresponding to interior points will achieve this far more efficiently than, e.g. deleting boundary points one-by-one.

5. To compute the 36 smallest-magnitude eigenvalues of `interior_operator` along with their eigenvectors, we will use the software ARPACK (https://www.caam.rice.edu/software/ARPACK/) (for "ARnoldi PACKAGE") through an interface in `scipy`, namely `scipy.sparse.linalg.eigs`. This uses the Arnoldi procedure we discussed in class to compute a subset of the eigenvalues of a matrix. (For entertainment, compare its runtime with `numpy.linalg.eig`.) Tell `eigs` to compute the smallest eigenvalues by using "shift-invert" mode, by setting its `sigma` keyword argument to 0.

   Note: While `interior_operator` is a sparse matrix, and while treating it as such would save memory and computational cost, for this problem, please produce `interior_operator` as a dense matrix. Despite the name, `scipy.sparse.linalg.eigs` is perfectly happy with dense matrices.

   Store the resulting eigenvalues in `eigvals` and the resulting eigenvectors in `eigvecs`. Use the supplied plotting code to show the Chladni figures for the given `stiffness`. Observe how the patterns change depending on the given `stiffness`. You may also experiment with your own `stiffness` functions.

INPUT:

- `nodes_1d`: `numpy` array of shape `(n,)`. Node coordinates for the grid in each of the x and y directions.
- `nstencil`: an odd integer. Number of nodes to use when constructing the differentiation matrix.
- `numbering`: `numpy` array of shape `(n,n)` containing integers indicating the index of each point in the 2D grid in the solution-vector.
- `x` : `numpy` array of shape `(n*n,)`. x coordinates of the points in the 2D grid.
- `y` : `numpy` array of shape `(n*n,)`. x coordinates of the points in the 2D grid.
- `stiffness`: `numpy` array of shape `(n*n,)`. Stiffness parameter at each point in the 2D grid.

OUTPUT:

- `D2`: `numpy` array of shape `(nstencil, nstencil)`. A second-derivative matrix on an interval discretized with `nstencil` points spaced like the nodes in `nodes_1d`
- `d2x_mat_1d`: `numpy` array of shape `(n, n)`. A second-derivative matrix on `nodes_1d`.

- `d2x_mat` : `numpy` array of shape `(n*n, n*n)` . A second-derivative matrix in the x direction on the 2D grid.
- `d2y_mat` : `numpy` array of shape `(n*n, n*n)` . A second-derivative matrix in the y direction on the 2D grid.
- `interior_operator` : `numpy` array of shape `((n-2)**2, (n-2)**2)` . Differential operator for eigenvalue finding.
- `eigvals` : `numpy` array of shape `(36,)` , containing the smallest 36 eigenvalues of `interior_operator` , order by magnitude.
- `eigvecs` : `numpy` array of shape `((n-2)**2, 36)` , containing the corresponding eigenvectors.

Problem set-up code (click to view)

Starter code (click to view)

Answer*

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import scipy.sparse.linalg as ssla
n = nodes_1d.shape[0]
V = np.zeros((nstencil,nstencil))
for i in range(nstencil):
    V[:,i] = nodes_1d[:nstencil]**i
Vpp = np.zeros((nstencil,nstencil))
for i in range(nstencil):
    if i - 2 <0:
        Vpp[:,i] = np.zeros((nstencil))
    else:
        Vpp[:,i] = nodes_1d[:nstencil]**(i-2)*(i-1)*i
D2 = Vpp@la.inv(V)
Vd = np.zeros((n,n))
print(nstencil//2)
Vd[:nstencil//2,:nstencil] = D2[:nstencil//2,:]
Vd[-nstencil//2:,-nstencil:] = D2[-nstencil//2:,:]
for i in range(n - 2*nstencil//2):
    j = i + nstencil//2
    Vd[j,i:i+nstencil] = D2[nstencil//2,:]
```

Press F9 to toggle full-screen mode. Set editor mode in user profile (/profile/).

Your answer is correct.

Here is some feedback on your code:

- 'D2' looks good
- 'd2x_mat_1d' looks good
- 'd2x_mat' looks good
- 'd2y_mat' looks good

- 'interior_operator' looks good
- 'eigvals' looks good
- 'eigvecs' looks good
- Nice job! Everything looks good.
- Execution time: 9.0 s -- Time limit: 20.0 s

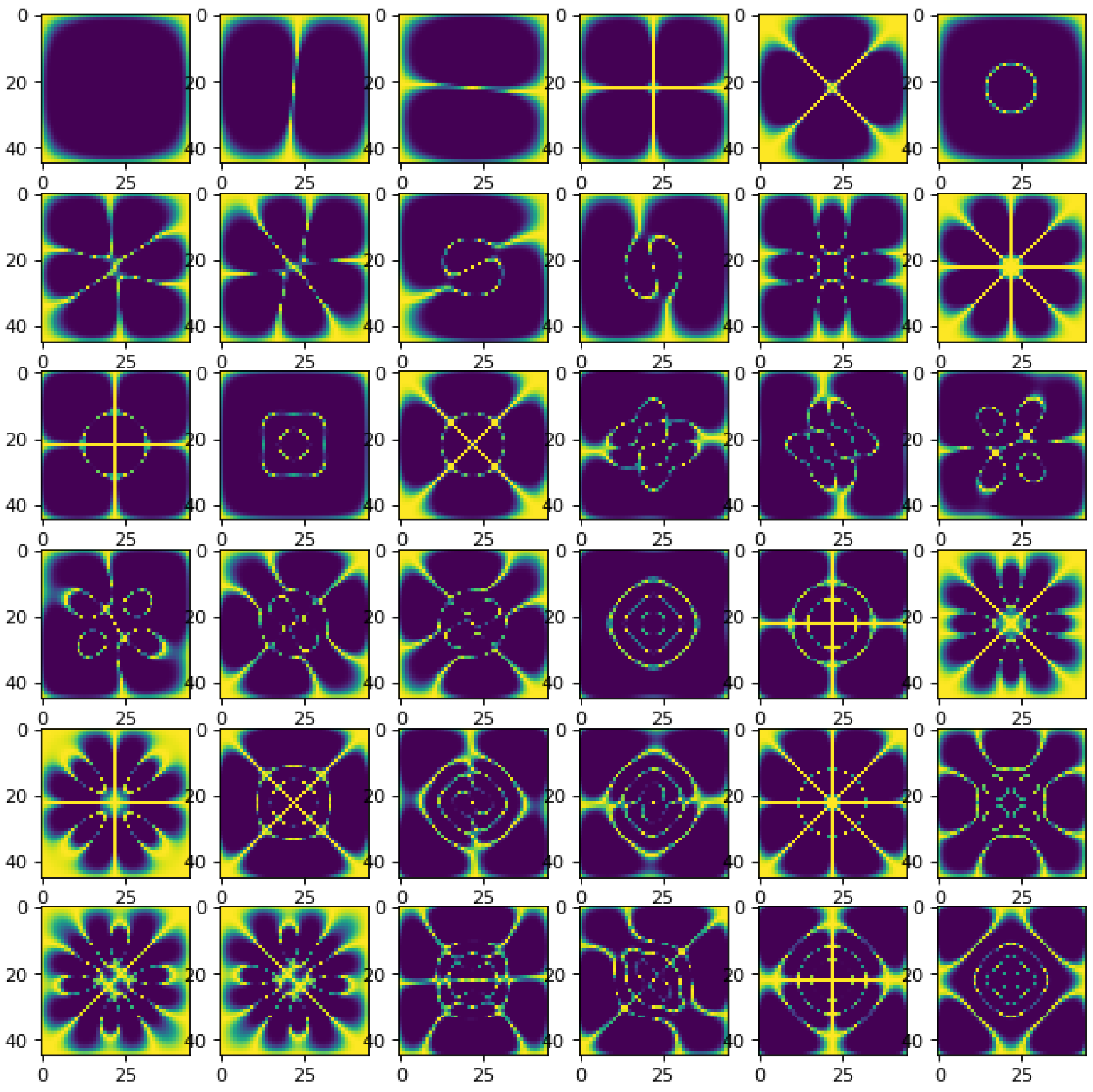
Your code ran on relate-01.cs.illinois.edu.

Your code printed the following output:

```
3
```

Your code produced the following plots:

Figure1



The following code is a valid answer:

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import scipy.sparse.linalg as ssla

nswidth = nstencil//2

fd_nodes = nodes_1d[:nstencil]

V = np.array([
    fd_nodes**i for i in range(len(fd_nodes))
]).T
Vprime = np.array([
    i*fd_nodes**(i-1) if i else 0*fd_nodes
    for i in range(len(fd_nodes))
]).T

D = Vprime @ la.inv(V)
D2 = D@D

n = len(nodes_1d)

d2x_mat_1d = np.zeros((n, n))
for i in range(nswidth, n-nswidth):
    d2x_mat_1d[i, i-nswidth:i+nswidth+1] = D2[nswidth]
d2x_mat_1d[:nswidth, :nstencil] = D2[:nswidth]
d2x_mat_1d[-nswidth:, -nstencil:] = D2[-nswidth:]

d2x_mat = np.zeros((n*n, n*n))
for i in range(n):
    for j in range(n):
        idx = i
        start = min(n-nstencil, max(0, idx-nswidth))
        end = max(nstencil, min(n, idx+nswidth+1))
        assert end-start == nstencil, (idx, start, end)

        d2x_mat[numbering[i,j], numbering[start:end,j]] = d2x_mat_1d[idx, start:enc

d2y_mat = np.zeros((n*n, n*n))
for i in range(n):
    for j in range(n):
        idx = j
        start = min(n-nstencil, max(0, idx-nswidth))
        end = max(nstencil, min(n, idx+nswidth+1))
        assert end-start == nstencil, (idx, start, end)

        d2y_mat[numbering[i,j], numbering[i, start:end]] = d2x_mat_1d[idx, start:er

operator = stiffness.reshape(-1, 1)*(d2x_mat + d2y_mat)

interior = numbering[1:-1, 1:-1].reshape(-1)
interior_operator = operator[interior][:, interior]

eigvals, eigvecs = ssla.eigs(interior_operator, k=36, sigma=0)

sort_idx = np.argsort(np.abs(eigvals))
eigvals = eigvals[sort_idx]
```

```
eigvecs = eigvecs[:, sort_idx]
u_0 = eigvecs[:,0]

try:
    # prevent plotting if this variable is defined
    # (autograder logistics)
    prohibit_plot
except NameError:
    plt.imshow(stiffness.reshape(n, n))
    plt.colorbar()
    plt.show()

    plt.figure(figsize=(10, 10))
    for iplot in range(36):
        i = iplot
        plt.subplot(6, 6, iplot+1)

        d = eigvecs[:, i].reshape(n-2, n-2).copy()
        d = d/np.max(np.abs(d))
        nodes = np.exp(-300*np.abs(d)**2)
        plt.imshow(nodes)
    plt.show()
```