MAT 560 Final Report

Alexander Jansing, Aaron Gregory

May 5, 2018

1 Introduction

Numerical analysis provides us with a variety of tools that enable the discovery of approximate solutions for problems that are prohibitively complex to solve analytically. Here we consider the motion of an unforced supported Kirchhoff-Love plate lying flat on a two dimensional domain Ω. We form a general solution out of the plate's eigenmodes, and are able to recreate arbitrary initial conditions.

A note on notation: we use bold symbols to represent vectors and tensors (e.g. $\boldsymbol{r}$, $\boldsymbol{u}$, $\boldsymbol{\sigma}$) and subscripts without boldface to represent components (e.g. r_1, u_i, σ_{jk}).

2 Overview of Kirchhoff-Love Plate Theory

Kirchhoff-Love plate theory describes the motion of plates (elastic solids resistant to both bending and stretching) that are of negligible thickness. We consider a plate that is isotropic and homogeneous, which means it is equally resistant to deformation in all directions at any point in the domain, and of uniform density. These two restrictions greatly simplify the plate's equations of motion.

The deformation of a plate is defined as $\boldsymbol{u} = \boldsymbol{r}' - \boldsymbol{r}$, where $\boldsymbol{r}$ is the original position of a point in the plate, and $\boldsymbol{r}'$ is the position the point is moved to after being deformed. Consider deformation as the sum of mid-surface and out-of-plane components, $\boldsymbol{u} = \boldsymbol{u}_\parallel + \boldsymbol{u}_\perp$. A very thin plate has the useful property that far less energy is required to produce significant deformations along the surface normal than within the plane of the surface, so it is reasonable to assume that $|\boldsymbol{u}_\parallel| \ll |\boldsymbol{u}_\perp|$. Therefore in some cases we will be able to ignore the lateral components of the plate's deformation, writing $\boldsymbol{u} \approx w(\boldsymbol{x})\hat{\boldsymbol{n}}$, where $\hat{\boldsymbol{n}}$ is the normal vector to the surface of our plate.

One consideration that we avoid by making our plate thin is displacement due to torsion. Here we will treat our plate as though it is a plane (on which restorative forces have no torque component), and attempt to solve for how that plane is deformed. The plane we use as an abstraction of our plate is called the plate's neutral surface: it lies halfway between the two faces of our plate, and is therefore mostly unaffected by any actual twisting forces which exist in the plate.

The determining equations for our system arise form the interaction of two tensors: stress and strain. Stress (denoted $\boldsymbol{\sigma}$) is the distribution of force within the plate due to both internal and external causes. We derive a formula for $\boldsymbol{\sigma}$ by noting that the total force acting on a physical object can be calculated with an integral across the object's surface. Surface and volume integrals are only interchangeable when the integrand can be written as divergence,

and therefore we have $\boldsymbol{F} = \text{div}\ \boldsymbol{\sigma}$, where $\boldsymbol{F}(\boldsymbol{x})$ is the force acting on point $\boldsymbol{x}$ in our plane. We will not give the proof here, but it is known that stress is symmetric: $\sigma_{ij} = \sigma_{ji}$.

The strain tensor (denoted $\boldsymbol{\varepsilon}$) represents how much our plate "feels" its deformation:

$$2\varepsilon_{ij} = \partial_j u_i + \partial_i u_j + \sum_k (\partial_i u_k)(\partial_j u_k)$$

where ∂_i denotes a partial derivative with respect to the ith component of $\boldsymbol{x}$. Notice that strain, like stress, is symmetric. When $\partial_i u_j$ is small for all i and j (e.g. when our plate is not bent sharply), we can approximate strain with $2\varepsilon_{ij} = \partial_i u_j + \partial_j u_i$. This is the value we will use for the rest of our derivations.

The elastic properties of an object are usually defined by the interaction of stress and strain within it, that is, between the forces acting on the object and the object's reaction. To understand the difference between strain and deformation, consider a plate that has been folded. Deformation on the crease is very small, since those points were not moved far when the plate was bent. However, the crease is the region where strain is largest, and accordingly where the majority of the restorative forces originate. We are interested in balancing internal forces $\boldsymbol{F}$, and strain $\boldsymbol{\varepsilon}$ captures more important information about force than does deformation $\boldsymbol{u}$.

Let us assume that our plate obeys Hooke's law, which states that stress and strain are linearly related. Using our previous knowledge that $\boldsymbol{u} \approx w(\boldsymbol{x})\hat{\boldsymbol{n}}$ (which means that we can assume displacement to be uniform along $\hat{\boldsymbol{n}}$), and choosing our coordinate frame such that $\hat{\boldsymbol{n}} = \hat{\boldsymbol{z}}$, we can now write some components of stress in terms of x, y, and z:

$$\begin{aligned}\sigma_{zx} &= \mu\varepsilon_{zx} = 0\\ \sigma_{zy} &= \mu\varepsilon_{zy} = 0\\ \sigma_{zz} &= 0\end{aligned}$$

Where Hooke's law is used to represent stress as a multiple of strain. Applying our linearized approximation of strain, we learn that $\partial_z u_x = -\partial_x u_z$ and $\partial_z u_y = -\partial_y u_z$. Since (according to our choice of coordinates) $u_z = w(x, y)$, which is not a function of z, we must have $u_x = -z\partial_x w$ and $u_y = -z\partial_y w$.

Now we can find the deformation vector, and consequently the entire strain tensor, in terms of $w(x, y)$. We use this information to find a solution which minimizes the free energy of our plate while satisfying Newton's second law ($\text{div}\ \boldsymbol{\sigma} = \rho\ddot{\boldsymbol{w}}$, where ρ is density). This involves two lengthy manipulations, which we will not detail here, but can be found in other resources [LHLL06, LL86]. Our requirement that free energy be minimized reflects our wish to find physically applicable solutions (physical objects naturally take, or move toward, low energy states). The free energy of our plate is given with the formula

$$F_{pl} = \frac{Eh^2}{24(1-\nu^2)} \int_\Omega \left(\partial_x^2 w + \partial_y^2 w\right)^2 + 2(1-\sigma)\left((\partial_x\partial_y w)^2 - (\partial_x^2 w)(\partial_y^2 w)\right)\, d\boldsymbol{x}$$

Where E is Young's modulus (related to the stiffness of our plate), h is the thickness of the plate, and ν is Poisson's ratio (which relates compression along one axis with expansion along all other axes). In order to minimize this value, the deflection w of our plate must satisfy the plate equation:

$$\frac{Eh^3}{12(1-\nu^2)}\Delta^2 w + \rho\ddot{w} = 0, \qquad \forall \boldsymbol{x} \in \Omega$$

Where usually $\frac{Eh^3}{12(1-\nu^2)}$ is written as a single constant D, called the *bending rigidity* of our plate. This single equation is enough to determine a valid deformation of a plate, but it is underdetermined (its solutions are not unique). To solve it uniquely we must add boundary conditions.

3 Derivation of Model

Since our plate is supported, we add a simple boundary condition: $w = 0$, $\forall \boldsymbol{x} \in \partial\Omega$. For reasons that will become apparent during our numerical analysis, we also require that $\Delta w = 0$, $\forall \boldsymbol{x} \in \partial\Omega$. Our final set of determining equations for our unforced Kirchhoff-Love plate is:

$$\begin{aligned} \frac{\rho}{D}\ddot{w} + \Delta^2 w = 0, \qquad & \forall \boldsymbol{x} \in \Omega \\ w = \Delta w = 0, \qquad & \forall \boldsymbol{x} \in \partial\Omega \end{aligned}$$

Because these equations are linear, we can represent their general solution as a superposition of our plate's eigenmodes (e.g. if we solve for the eigenmodes, we have fully solved our problem). An eigenmode, also called an eigenfunction or vibrational mode, is a deformation of our plate where every point moves sinusoidally with the same phase. The simple harmonic motion that defines an eigenmode allows us to perform separation of variables on our determining equations with $w = \cos(\omega t)u(\boldsymbol{x})$. A short manipulation then brings us to $\Delta^2 u = \frac{\rho\omega^2}{D}u = \lambda^2 u$. Already we can see the plate equation taking the form of an eigenvalue problem. To be clear: we do not know which eigenvalues λ will produce a solvable equation. That's why we refer to $\Delta^2 u = \lambda^2 u$ as an eigen*value* problem.

4 Discretization

Now we take our eigenvalue problem and perform reduction of order:

$$\begin{bmatrix} 0 & \Delta \\ \Delta & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \lambda \begin{bmatrix} u \\ v \end{bmatrix}$$

But perhaps this seems unjustified - by adding a new variable to solve for, have we not increased the complexity of our problem? Well, yes and no. It is possible for reduction of order to make our problem harder to solve analytically. But we are considering a fourth order PDE defined on an arbitrarily complex domain. The eigenvalue problem as it stood was already prohibitively complex for symbolic discovery of solutions. Our goal, therefore, is not to find a perfect analytical description of a plate's motion given any domain, but only to approximate its motion. This approximation will be done with a computer, and as a result our chief concern is how well our model lends itself to computation (not symbolic manipulation).

4.1 Finite Difference Method

One of the simplest tools of numerical analysis is the method of finite differences (FDM). The backing concept here is that if it is too difficult to find a full solution to a problem (e.g. the value of u at every point in Ω), then perhaps we can lessen the problem's complexity by restricting our domain to an easier to handle subdomain. Instead of being able to sample our solutions anywhere, we only look for their values at specific points (our subdomain is defined by the finite differences between the sampling points). By using discretizations of continuous

operators (converting our differential equations into difference equations), FDM constructs a matrix of the relationships between points in our subdomain. Solving the matrix problem $A\boldsymbol{x} = \boldsymbol{b}$ will produce an approximation of the solution to our PDE.

FDM has conceptual simplicity going for it, but it has few other benefits. One of its significant drawbacks is its inability to adapt well to complex domains. Suppose our domain is very complicated in one region and very simple in another. In order to achieve a reasonable accuracy we would have to lower the distance between points across the entire mesh, wasting computational resources on the simple areas where a coarse mesh would suffice. Finite differences also require special consideration at the boundaries of our domain, which needlessly complicates our computations. Due to this lack of flexibility, we chose not to use finite differences in our numerical analysis, opting instead to use the method of finite elements.

4.2 Finite Element Method

The method of finite elements (FEM) borrows a few core ideas from FDM, but applies them in a very different way. Whereas finite differences operates by constricting our domain, finite elements finds a result by assuming that our solution lives in a finite dimensional function space. This solves the problems presented by FDM, because our function space can be constructed in a manner that gives high accuracy in one region of our domain, and low accuracy in another. Since the boundary of our domain is enforced by the very choice of the function space, we can completely avoid our boundary value requirements after deciding on our function space.

FEM splits the problem of approximating a differential equation into two computational steps, which when handled separately are more efficient than when handled together. First we discretize our domain by defining a mesh on it, and we use that mesh to define our function space. Secondly we solve for our solution as a member of our function space, constructing a matrix of the dependencies that have to be satisfied, and solving just like in FDM. This allows us to deal separately with domain-related matters and solution-related matters.

We do not have many constraints in our choice of function space, but there are some limitations. For example, we must show that as the dimension of our space approaches infinity, our approximate solution converges to the true solution. There are certain common function spaces where this is guaranteed, and we choose ours from among them. We can, however, safely define our function space with emphasis on arbitrary regions in our domain, which allows us to focus computational resources where they are needed.

4.2.1 Weak Formulation

Consider our eigenvalue problem: $\begin{bmatrix} 0 & \Delta \\ \Delta & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \lambda \begin{bmatrix} u \\ v \end{bmatrix}$. In its current state, this equation requires that u and v both be twice continuously differentiable. By finding what is called a weak formulation, we can reduce this requirement. First we integrate across our domain and multiply our eigenproblem by two arbitrary test functions:

$$\int_\Omega \begin{bmatrix} \phi \\ \varphi \end{bmatrix}^T \begin{bmatrix} 0 & \Delta \\ \Delta & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \, d\boldsymbol{x} = \int_\Omega \lambda \begin{bmatrix} \phi \\ \varphi \end{bmatrix}^T \begin{bmatrix} u \\ v \end{bmatrix} \, d\boldsymbol{x}$$

Since ϕ and φ are arbitrary, we can impose whatever conditions on them we want. In particular, let's assume that ϕ and φ are both continuously differentiable. That means that we can do integration by parts (which is why we required earlier that $\Delta w = 0$, since otherwise our

boundary conditions would not vanish):

$$-\int_\Omega \nabla v\cdot\nabla\phi+\nabla u\cdot\nabla\varphi\ d\boldsymbol{x}=\int_\Omega \lambda u\phi+\lambda v\varphi\ d\boldsymbol{x}$$

Now we have a form of our eigenproblem that only requires u and v to be once continuously differentiable. This will give us greater freedom when choosing our function space.

4.2.2 Mesh Creation

Geometries and meshes are easily created with pygmsh[Sch]. The code below (also seen in appendix 8.2.1) creates a circular plate with no deformations.

```
geom = pygmsh.opencascade.Geometry(
  characteristic_length_min=0.1,
  characteristic_length_max=0.1,
  )
outerDisk = geom.add_disk([ 0.0, 0.0, 0.0], 0.5)
print(geom.get_code())
points, cells, point_data, cell_data, field_data = pygmsh.generate_mesh(geom,
    ↪ geo_filename = "plate.geo")
```

Modifications to a mesh are generally easy to implement (see appendices 8.2.2, 8.2.3). We do not have any requirements for our plate geometry in our code, so different meshes can be used interchangeably.

4.2.3 Choice of Function Space

Now that we have a mesh of our domain and we know that our test functions must be once continuously differentiable, we can pick what our function space will be. We chose a set of Continuous Galerkin functions, which are members of the first order (linear) Lagrange family of interpolation polynomials. Our final solution will be constructed out of a set of piecewise linear functions - they can be thought of as pyramid functions for every point in our mesh.

Now we can apply our function space (which has a finite number of elements) to our weak formulation. This is done in two ways: first (as we have said), we assume that our solutions will fall in our function space:

$$(u,v)=\sum_i(a_i\phi_i,a_i\varphi_i)\in\mathcal{F}=\text{span}\{(\phi_i,\varphi_i)\}$$

Second, we only require that our weak formulation holds when our test functions (ϕ,φ) are in our function space. This is where the benefits of using FEM rather than FDM are clearly visible. In the method of finite elements, discretization does not mean restricting our domain, but only restricting our test functions to our mesh–defined function space, which is arbitrarily close to the exact solution.

$$-\int_\Omega \nabla v\cdot\nabla\phi_i+\nabla u\cdot\nabla\varphi_i\ d\boldsymbol{x}=\int_\Omega \lambda u\phi_i+\lambda v\varphi_i\ d\boldsymbol{x}$$

Notice that our weak formulation is linear in terms of u, v, ϕ and φ, and therefore is also linear in terms of a_i. This means that we can write our full discretized eigenproblem as a set of linear equations, which will determine the weights a_i of the linear combination of

basis functions that defines u and v. Substituting in $u = \sum_i a_i\phi_i$ and $v = \sum_i a_i\varphi_i$, our weak formulation becomes

$$-\int_\Omega \sum_i a_i\nabla\varphi_i\nabla\varphi_j + a_i\nabla\phi_i\nabla\phi_j \; d\boldsymbol{x} = \int_\Omega \lambda \sum_i a_i\varphi_i\phi_j + a_i\phi_i\varphi_j \; d\boldsymbol{x} = \int_\Omega \lambda \sum_i a_i C_{ij} \; d\boldsymbol{x}$$

Both sides are still linear in a_i, so we have a solvable system of equations. After putting this system into matrix form we solve for the eigenvalues λ_i and the eigenvectors e_i. Both of these have physical interpretations: the eigenvectors define vibrational forms of the plate we're modeling, and the eigenvalues tell us at what frequency the associated forms vibrate.

5 Computation

To begin our computational work, we used gmsh to create discretizations of several 2–dimensional plates. This was implemented with the pygmsh python package (see Section 4.2.2).

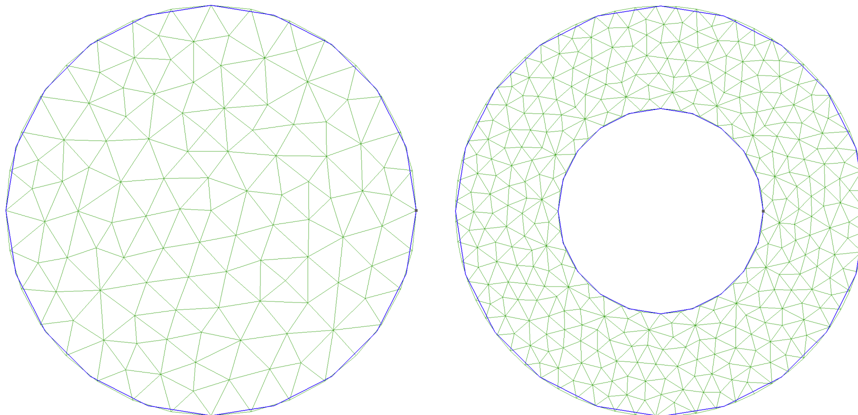

Figure 1: Meshes created with gmsh.

In conjunction with gmsh, we used FEniCS's python package, dolfin, to convert the `msh` files to an `xml` format.

Once the `xml` files were created, the number of vertices was recorded (manually) in relation to how many eigenvalues and vectors were found from the mesh. This data allowed us to predict how demanding the computation would be for a given mesh, and how much accuracy could be reasonably attained, given the hardware and time available.

The `xml` file is read in and converted to a dolfin Mesh object, then provided to the EigenSolver class (see 8.2.4) along with two FiniteElement objects. From here, we provide a method call that prints minimal output and returns two arrays of eigenvectors and eigenvalues (E and λ).

Eigenvectors:

$$E = \begin{bmatrix} e_0 & e_1 & e_2 & \cdots & e_{n-1} \end{bmatrix}$$

Individual Eigenvector:

$$e_i = \begin{bmatrix} e_{i0} & e_{i1} & e_{i2} & \cdots & e_{i,n-1} \end{bmatrix}$$

Eigenvalues:

$$\lambda = \begin{bmatrix} \lambda_0 & \lambda_1 & \lambda_2 & \cdots & \lambda_{n-1} \end{bmatrix}$$

Then the mesh, one of the FiniteElement objects, E, and λ are passed to the Simulator object (see 8.2.5). The Simulator automatically:

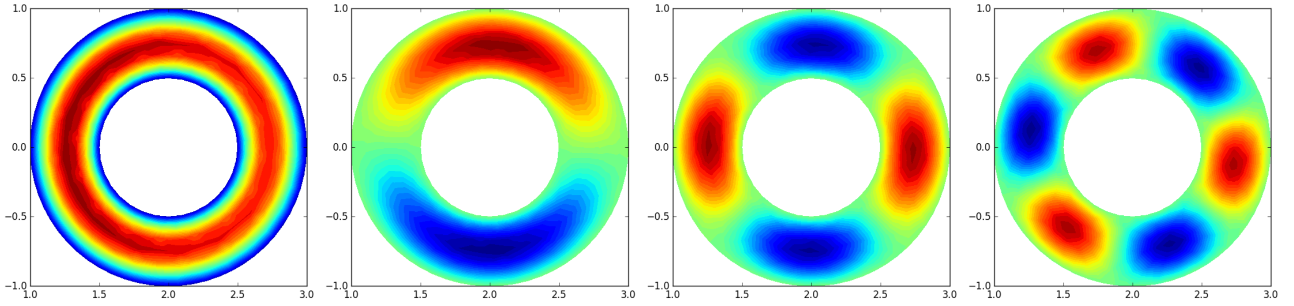

Figure 2: Some eigenfunctions found with FEniCS.

1. creates a FunctionSpace based on the mesh and the FiniteElement object,
2. interpolates a function from the given Expression within the FunctionSpace,
3. and the eigenbasis vectors ($\frac{e_i \cdot \vec{x}}{e_i \cdot e_i}$ portions of computation of w in Section 6).

After the constructor finishes the steps above, a call of the evaluate method with a time t and Point x (optional) will return the displacement at time t and at Point x (or a function if x is not supplied).

6 Recreation of Initial Conditions

After we have computed the set of our plate's eigenmodes, recreating initial conditions is a relatively simple matter. Suppose we want initial displacement $p(\boldsymbol{x})$ and initial velocity $v(\boldsymbol{x})$. With eigenfunctions $e_i(\boldsymbol{x})$ and vibrational frequencies $\omega_i = \sqrt{\frac{D}{\rho}\lambda_i}$, we can write our final solution as

$$w(\boldsymbol{x}, t) = \sum_i \left(\cos(\omega_i t) \frac{\langle e_i, p \rangle}{\langle e_i, e_i \rangle} e_i(\boldsymbol{x}) + \sin(\omega_i t) \frac{\langle e_i, v \rangle}{\langle e_i, e_i \rangle} \frac{e_i(\boldsymbol{x})}{\omega_i} \right)$$

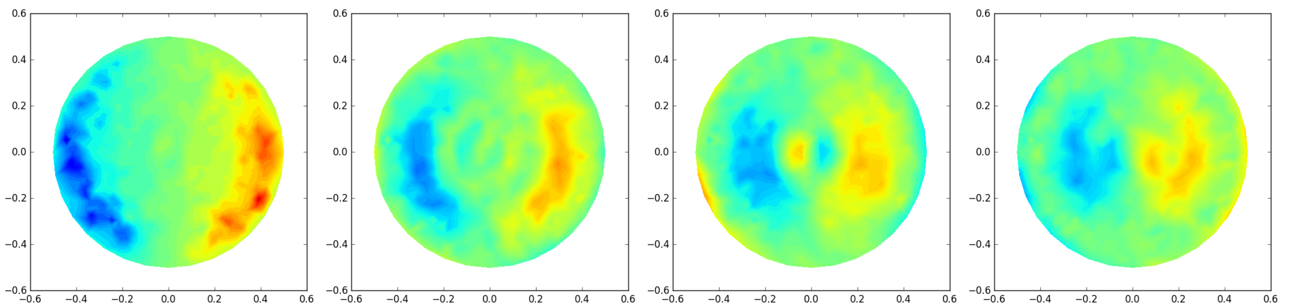

Figure 3: Time evolution (moving rightward) from initial position and velocity.

7 Results

Through experimentation, creating meshes of varying sizes and finding the number of eigenvalues and vectors associated with each mesh, there seems to be a strictly linear (1 : 2) correlation between the number of vertices and the number of eigenvalues found from each mesh.

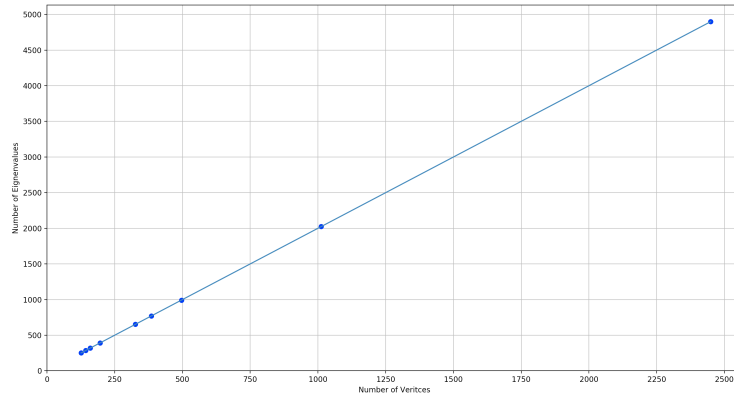

Figure 4: Number of Vertices to Number of Eigenvalues. **??**

This two-to-one relationship between the number of eigenvalues and vertices of the mesh shows that we are finding all available eigenvalues, since our matrix stores information for both u and v, each of which has one element per mesh point.

8 Appendix

8.1 Data Table: Mesh Vertices vs. Eigenvalues

Data was generated from the left mesh of Figure 1 as the base geometry. Smaller *Element size factors* were used ($[1, 0.9, 0.8, \cdots, 0.2]$) to create finer meshes (more vertices).

vertices	eigenvalues
126	252
143	286
160	320
196	392
326	652
385	770
496	992
1012	2024
2449	4898

8.2 Code Listings

Below is some of the core code we developed. Some pieces of code which are not relevant to the numerical work being done have been removed for brevity. The full code can be accessed at https://github.com/gregory2718/SUNY_MAT560 or through the links provided above each code snippet.

8.2.1 plate.py

This code creates a circular plate with no deformations.

```
import pygmsh

geom = pygmsh.opencascade.Geometry(
  characteristic_length_min=0.1,
  characteristic_length_max=0.1,
  )
outerDisk = geom.add_disk([ 0.0, 0.0, 0.0], 0.5)
print(geom.get_code())
points, cells, point_data, cell_data, field_data = pygmsh.generate_mesh(geom,
    ↪ geo_filename = "plate.geo")
```

8.2.2 donut.py

This code creates a circular plate with a hole in the middle.

```
import pygmsh

geom = pygmsh.opencascade.Geometry(
  characteristic_length_min=0.1,
  characteristic_length_max=0.1,
```

```
  )

outerDisk = geom.add_disk([ 2.0, 0.0, 0.0], 1.0)
innerDisk = geom.add_disk([ 2.0, 0.0, 0.0], 0.5)
flat = geom.boolean_difference([outerDisk], [innerDisk])
# geom.extrude(flat, [0, 0, 0.3])

points, cells, point_data, cell_data, field_data = pygmsh.generate_mesh(geom,
    ↪ geo_filename = "donut.geo")

print(geom.get_code())
```

8.2.3 swiss_cheese.py

This code creates a circular plate with a number of random holes punched in it to provide a non-trivial case of a plate.

```
import pygmsh
import random
import time

geom = pygmsh.opencascade.Geometry(
  characteristic_length_min=0.1,
  characteristic_length_max=0.1,
  )

random.seed(int(time.time()))

outerDisk = geom.add_disk([ 0.0, 0.0, 0.0], 2.0)
for i in range(random.randint(10,20)):
        x = random.random()*2.
        y = random.random()*2.
        quadrant = random.randint(1,4)
        if quadrant == 1:
                quadrant = [1,1]
        elif quadrant == 2:
                quadrant = [1,-1]
        elif quadrant == 3:
                quadrant = [-1,-1]
        elif quadrant == 4:
                quadrant = [-1,1]
        size = random.random()
        innerDisk = geom.add_disk([ x*quadrant[0], y*quadrant[1], 0.0], size
            ↪ *.75)
        outerDisk = geom.boolean_difference([outerDisk], [innerDisk])
# geom.extrude(outerDisk, [0, 0, 0.3])

```

```
points, cells, point_data, cell_data, field_data = pygmsh.generate_mesh(geom,
    ↪ geo_filename = "swiss_cheese.geo")
```

8.2.4 polyplate package – EigenSolver.py

Class that takes two sets of FiniteElements and a mesh. And from that, it can generate the eigenvectors and eigenvalues.

```
from dolfin import *
from dolfin.cpp.mesh import *
from mshr import *
import pylab
import numpy as np

class EigenSolver:
        def __init__(self, U, V, mesh):
                self.U = U
                self.V = V
                self.mesh = mesh

        def getEigenVectorValue(self, saveData = False, saveDataDir = None,
            ↪ saveDateName = "output"):
                if(self.mesh == None):
                        print('Mesh␣not␣present.')
                        exit()

                if(self.U == None):
                        self.U = FiniteElement('CG', triangle, 1)
                if(self.V == None):
                        self.V = FiniteElement('CG', triangle, 1)
                W = FunctionSpace(self.mesh, self.V * self.U)

                class DirichletBoundary(SubDomain):
                        def inside(self, x, on_boundary):
                                return on_boundary

                # Define boundary condition
                u0 = Constant(0.0)
                boundaryCondition1 = DirichletBC(W.sub(0), u0,
                    ↪ DirichletBoundary())
                boundaryCondition2 = DirichletBC(W.sub(1), u0,
                    ↪ DirichletBoundary())

                # Define the bilinear form
                (u, v) = TrialFunction(W)
                (f1, f2) = TestFunction(W)
                a = -(dot(grad(u), grad(f2)) + dot(grad(v), grad(f1)))*dx
                L = (u*f1 + v*f2)*dx
```

```
        # Create the matrices
        A = PETScMatrix()
        b = PETScMatrix()
        assemble(a, tensor = A)
        assemble(L, tensor = b)
        boundaryCondition1.apply(A)
        boundaryCondition2.apply(A)

        # Create eigensolver
        eigensolver = SLEPcEigenSolver(A, b)

        # Compute all eigenvalues of A x = \lambda x
        eigensolver.solve()

        eigenVectors = []
        eigenValues = []

        # If we're saving the data, make the directories to save the
           ↪ data to.
        if saveData:
                if saveDataDir == None:
                        saveDataDir = 'eigen'
                mkdirs(saveDataDir)

        u = Function(W)
        for i in reversed(range(eigensolver.get_number_converged())):
                r, c, rx, cx = eigensolver.get_eigenpair(i)
                eigenVectors += [ rx.vec().getArray() ]
                eigenValues += [ r ]
                if saveData:
                        u.vector()[:] = rx
                        plot(u.sub(0))
                        # save images
                        pylab.savefig('%s/images/%s%d_%04d.png' % (
                           ↪ saveDateName, saveDateName, m, i),
                                bbox_inches='tight')

        eigenVectors = np.array(eigenVectors)
        eigenValues = np.array(eigenValues)

        return eigenVectors, eigenValues
```

8.2.5 polyplate package – Simulator.py

```
from dolfin import *
from dolfin.cpp.mesh import *
from mshr import *
```

```
from math import sin, cos
import numpy as np


class Simulator():
	"""docstring for Simulator"""
	def __init__(self, mesh, U, eigenVectors, eigenValues,
	 pExpr = Expression('sin(x[0])', degree=1),
	 vExpr = Expression('cos(x[0])', degree=1), C = 1):
		# Test for PETSc and SLEPc
		if not has_linear_algebra_backend("PETSc"):
			print("DOLFIN has not been configured with PETSc. 
			    ↪ Exiting.")
			exit()

		if not has_slepc():
			print("DOLFIN has not been configured with SLEPc. 
			    ↪ Exiting.")
			exit()

		parameters['reorder_dofs_serial'] = False
		self.timeScale = 1.0 / C
		self.fSpace = FunctionSpace(mesh, U)
		self.eVecs = eigenVectors
		self.eVals = eigenValues
		self.pVec = fem.interpolation.interpolate(pExpr, self.fSpace).
		    ↪ vector()
		self.vVec = fem.interpolation.interpolate(vExpr, self.fSpace).
		    ↪ vector()
		self.pEig = self.inEigenBasis(self.pVec)
		self.vEig = self.inEigenBasis(self.vVec)
		for i in range(len(self.eVals)):
			self.vEig[i] /= self.eVals[i]

	def inEigenBasis(self, vec):
		return np.array([np.dot(vec, e) / np.dot(e, e) for e in self.
		    ↪ eVecs])

	def vecAtTime(self, t):
		S = np.zeros(self.eVecs[0].shape)
		for i in range(len(self.eVecs)):
			# position component
			S += cos(self.eVals[i]*t)*self.pEig[i]*self.eVecs[i]
			# velocity component
			S += sin(self.eVals[i]*t)*self.vEig[i]*self.eVecs[i]
		return S

	def upperBound(self):
```

```
        m = np.zeros(self.eVecs[0].shape)
        for i in range(len(self.eVecs)):
            m += np.abs(self.pEig[i]*self.eVecs[i])
            m += np.abs(self.vEig[i]*self.eVecs[i])
        return np.max(m)

    def evaluate(self, t, x = None):
        f = Function(self.fSpace)
        f.vector().set_local(self.vecAtTime(t * self.timeScale))
        f.update()
        if x is None:
            return f
        else:
            return f(Point(x[0], x[1]))
```

References

[LHLL06] P Lu, LH He, HP Lee, and C Lu. Thin plate theory including surface effects. *International Journal of Solids and Structures*, 43(16):4631–4647, 2006.

[LL86] LD Landau and E Lifschitz. Theory of elasticity, 1986.

[Sch] Nico Schlömer. `pygmsh`. 2018. pypi, pypi.org/project/pygmsh/.