IMPLEMENTATION OF KIRCHOFF-LOVE PLATE MODEL WITH THE FE METHOD

12/13/2024

Solal ESPIC and Douri JEONG

ÉCOLE POLYTECHNIQUE

IP PARIS

ABSTRACT

This reports shows our progress during the implementation of the Kirchoff-Love plate model with LIB552. We star by recalling the key assumptions of Kirchoff-Love plate model and the weak formulation that arises from it. We then choose a certain discretization strategy that required some new functions to be implemented in LIB552. After proofing them on a unit triangle mesh, this proofing allowed us to realize that linear shape functions could not account for the plate bending. Hence we explored two ways of increasing the order of the shape functions : using P2 triangles elements or choosing a specific family of shape functions over the 3 interpolation point triangle. We tried to briefly compare both methods, highlighting problems with both. We take a brief step backwards at the end of this report to mention our first try with the FeNICs library.

CONTENTS

1 BUILDING THE NUMERICAL MODEL

1.1 Kirchoff-Love Plate model

The Kirchhoff-Love theory is used to analyze thin plates and is based on the following assumptions:

- The middle plane is initially flat, meaning it does not have any curvature.
- Sections perpendicular to the middle sheet remain perpendicular during deformation, meaning shear effects can be ignored.
- The plate has a small thickness, meaning deformation in the thickness direction is negligible, which implies that stresses in this direction can be negliged.
- The analysis is conducted under the assumption of small deformations.

Hence the kinematic description adopted is the following:

$$\begin{cases} u(x,y,z) = u_0(x,y) + z\,\theta_y(x,y), \\ v(x,y,z) = v_0(x,y) - z\,\theta_x(x,y), \\ w(x,y,z) = w_0(x,y), \end{cases}$$

But in addition, with some geometrical considerations from [1], the Kirchhoff-Love assumptions give:

$$\theta_y = -\frac{\partial w_0}{\partial x}, \quad \theta_x = \frac{\partial w_0}{\partial y}.$$

Hence, we can compute the symmetric gradient coming from this form of displacement field, again, see [1] for a full explanation of the reduction to only a 3×1 vector for ϵ. After calculations, we have, with Voigt notation:

$$\epsilon^g = \begin{pmatrix} \epsilon^g_{xx} \\ \epsilon^g_{yy} \\ \epsilon^g_{xy} \end{pmatrix} = \begin{pmatrix} e_{xx} \\ e_{yy} \\ e_{xy} \end{pmatrix} + z \begin{pmatrix} \kappa_{xx} \\ \kappa_{yy} \\ \kappa_{xy} \end{pmatrix} = \epsilon_m + z\,\epsilon_b.$$

Where ϵ_m can be interpreted as the membrane strain tensor and ϵ_b as the bending strain tensor. With components:

$$\begin{cases} \epsilon^g_{xx} = -\frac{\partial u_0(x,y)}{\partial x} + z\frac{\partial \theta_y(x,y)}{\partial x} \\ \epsilon^g_{yy} = \frac{\partial v_0(x,y)}{\partial y} - z\frac{\partial \theta_x(x,y)}{\partial y} \\ \epsilon^g_{xy} = \frac{1}{2}\left(\frac{\partial u_0(x,y)}{\partial y} + \frac{\partial v_0(x,y)}{\partial x}\right) + \frac{1}{2}z\left(\frac{\partial \theta_y(x,y)}{\partial y} - \frac{\partial \theta_x(x,y)}{\partial x}\right) \end{cases}$$

So, with the linear isotropic elasticity assumption, the stress tensor becomes:

$$\underbrace{\begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{pmatrix}}_{\underline{\sigma}} = \underbrace{\frac{E}{1-\nu^2}\begin{pmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & 1-\nu \end{pmatrix}}_{\underline{\underline{C}}} \underbrace{\begin{pmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{xy} \end{pmatrix}}_{\underline{\epsilon}} \tag{1}$$

We then proceed to write the weak formulation of the problem using the Virtual Works Principle:

$$\forall \underline{U}^*, W_{int}(\underline{U};\underline{U}^*) = W_{ext}(\underline{U}^*)$$

But we choose the trial functions to be the same form as $\underline{U}$, so:

$$\underline{U}^*(x,y,z) = \begin{pmatrix} u^* \\ v^* \\ w^* \end{pmatrix} + z\begin{pmatrix} \theta_y \\ -\theta_x \\ 0 \end{pmatrix}.$$

Hence, we can write:

$$\begin{aligned} W_{int}(\underline{U};\underline{U}^*) &= \int_A \int_{-\frac{h}{2}}^{\frac{h}{2}} \underline{\epsilon}(\underline{U}^*) \cdot \underline{\sigma}(\underline{U}) \\ &= \int_A \int_{-\frac{h}{2}}^{\frac{h}{2}} (\underline{\epsilon}_m + z\,\underline{\epsilon}_b) \cdot \underline{\underline{C}} \cdot (\underline{\epsilon}_m + z\,\underline{\epsilon}_b) \\ &= \int_A \underline{\epsilon}_m \cdot \underline{\underline{A}} \cdot \underline{\epsilon}_m + \int_A \underline{\epsilon}_b \cdot \underline{\underline{D}} \cdot \underline{\epsilon}_b + \int_A (\dots)(x,y) \underbrace{\int_{-\frac{h}{2}}^{\frac{h}{2}} z}_{0} \\ &= \int_A \underline{\epsilon}_m(\underline{U}^*) \cdot \underline{\underline{A}} \cdot \underline{\epsilon}_m(\underline{U}) + \int_A \underline{\epsilon}_b(\underline{U}^*) \cdot \underline{\underline{D}} \cdot \underline{\epsilon}_b(\underline{U}) \end{aligned}$$

where:

$$\underline{\underline{A}} = \int_{-\frac{h}{2}}^{\frac{h}{2}} \underline{\underline{C}} \quad \text{and} \quad \underline{\underline{D}} = \int_{-\frac{h}{2}}^{\frac{h}{2}} z^2 \underline{\underline{C}}$$

And for the virtual work of external forces :

$$W_{ext}(\underline{U^*}) = h\int_A \begin{pmatrix} u^* \\ v^* \\ w^* \end{pmatrix} \cdot \underline{F} + \underbrace{\int_{-\frac{h}{2}}^{\frac{h}{2}} z}_{0} \int_A \begin{pmatrix} \partial_x u^* \\ \partial_y v^* \\ 0 \end{pmatrix} \cdot \underline{F}$$

Because $\underline{F}$ does not depend on z.

1.2 Discretization

We will use point-wise ordering of degrees of freedom throughout this work.

So we discretize $\underline{U} = \underline{N}^T \cdot \underline{\mathbb{U}}$ and $\underline{U} = \underline{N}^T \cdot \underline{\mathbb{U}^*}$, and $\underline{\epsilon_m} = \underline{\underline{B}}^T \cdot \underline{\mathbb{U}}$, $\underline{\epsilon_b} = \underline{\underline{P}}^T \cdot \underline{\mathbb{U}}$.

With the $\underline{\underline{\mathrm{P}}}$ and $\underline{\underline{\mathrm{B}}}$ defined as follows :

$$\underline{\underline{B}}^T = \sum_e \underline{\underline{L_{l2g}}} \cdot \underline{\underline{B_e}}^T \cdot \underline{\underline{L_{l2g}}}^T = \sum_e \underline{\underline{L_{l2g}}} \cdot \begin{pmatrix} \partial_x \phi_{e,1} & 0 & 0 & \ldots\ldots \\ 0 & \partial_y \phi_{e,1} & 0 & \ldots\ldots \\ \frac{\sqrt{2}}{2}\partial_y \phi_{e,1} & \frac{\sqrt{2}}{2}\partial_x \phi_{e,1} & 0 & \ldots\ldots \end{pmatrix} \cdot \underline{\underline{L_{l2g}}}^T$$

and

$$\underline{\underline{P}}^T = \sum_e \underline{\underline{L_{l2g}}} \cdot \underline{\underline{P_e}}^T \cdot \underline{\underline{L_{l2g}}}^T = \sum_e \underline{\underline{L_{l2g}}} \cdot \begin{pmatrix} 0 & 0 & -\partial_{xx}\phi_{e,1} & \ldots\ldots \\ 0 & 0 & -\partial_{yy}\phi_{e,1} & \ldots\ldots \\ 0 & 0 & -2\sqrt{2}\partial_{xy}\phi_{e,1} & \ldots\ldots \end{pmatrix} \cdot \underline{\underline{L_{l2g}}}^T$$

Then weak formulation becomes, being valid for all $\underline{\underline{U^*}}$:

$$\Big(\int_A \underline{\underline{B}} \cdot \underline{\underline{A}} \cdot \underline{\underline{B}}^T + \underline{\underline{P}} \cdot \underline{\underline{D}} \cdot \underline{\underline{P}}^T\Big) \cdot \underline{\mathbb{U}} = h\int_A \underline{\underline{N}}^T \cdot \underline{F}$$

i.e.

$$(\underline{\underline{K_m}} + \underline{\underline{K_b}}) \cdot \underline{\mathbb{U}} = \underline{F}$$

2 Extension of LIB552

The first challenge when working with LIB552 was its vector element structure that only allowed for 2D vector field computations over 2D mesh.

Hence, after reaching out to Pr. Genet about this issue he replied to us with a beta version of LIB552 with some nDD vector field extension. Our work builded upon this version of LIB552.

2.1 New functions

In order to implement the $\underline{\underline{B_e}}$ and $\underline{\underline{P_e}}$ tensors, we had to follow the same procedure as already followed in LIB552 to compute the $\underline{\underline{B_e}}$ tensor for a regular problem.

Following the existing logic, new functions were implemented in `VectorElement.py`:

- `get_B_B_and_P_P_int`
- `init_get_B_B_and_P_P_int`
- `_init_sym_B_B_and_P_P`

- `_init_sym_B_B_and_P_P_int`
- `_init_sym_B_and_P`
- `_init_only_sym_P`
- `_init_only_sym_B`

You can find the code for those in Appendix A.

They all follow the same logic as existing functions with almost identical names.

Other functions were extended to the 3D vector field case : `field_to_ugrid` from `MeshUtils.py` can handle 3 degrees of freedom by nodes field.

2.2 Testing the functions on an unit triangle mesh

You can find the results of the computation on the attached `Kirchoff-Love_plate_P1_linear` notebook.

Before starting the computation on a complete mesh, it was necessary to make sure that the functions were working on a simple case : the unit triangle mesh.

To make calculations tractable, we set $E = 1, \nu = 0, h = 1$ making $\underline{\underline{A}}$ and $\underline{\underline{D}}$ simpler to handle.

As the finite element is here a P1 triangle, the shape function are polynoms of degree one :

$$\underline{N} = \begin{pmatrix} 1-\xi-\eta \\ \xi \\ \eta \end{pmatrix} \quad ; \quad \underline{\nabla N} = \begin{pmatrix} -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad ; \quad \underline{\nabla^2 N} = \underline{\underline{0}}$$

So,

$$\underline{\underline{B}}^T = \begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0 & 0 \end{pmatrix} \quad ; \quad \underline{\underline{P}}^T = \underline{\underline{0}} \tag{*}$$

$\underline{\underline{P}}$ is second order by nature so it is null here with linear shape functions.

Hence, the stiffness matrix will only be composed by the $\underline{\underline{K_m}}$ term. Before imposing any boundary conditions on certains dofs, in this unit triangle - mono cell case, $\underline{\underline{K_m}} = \frac{1}{2}\underline{\underline{B}} \cdot \underline{\underline{A}} \cdot \underline{\underline{B}}^T$.

But from the column in red in (*), it causes $\underline{\underline{K_m}}$ to be singular as you can see the sparsity pattern :

The system would be solvable here if considered in 2D. The linear shape functions do not permit to account for the bending. As seen, in its literal expression, $\underline{\underline{P}}$ only depends on w, the z-component of the displacement.

We show this at the end of the attached`Kirchoff-Love_plate_P1_linear` notebook where we constrain all z-degrees of freedom and clamp one node to prevent translation in the plane and impose an in-plane force. We even change the mesh and it work as intended, this time we could verify the computation with a specific plane stress model like the one in E7 but we choose to move on and try to describe bending first.

Here are some deformed in-plane configurations comming from the model described in this section and all z-degrees of freedom constrained :

To account for bending with the Kirchoff-Love model, it is sufficient to increase only the order of the z shape function.

2.3 Increase of the shape functions' order

As just concluded it is sufficient to increase only the order of the z shape function. But the current architecture of LIB552's `FiniteElement.py` plan on using only one shape function for each interpolation point of the finite element i.e. the same shape function is used for all vector field degrees of freedom at one given point.

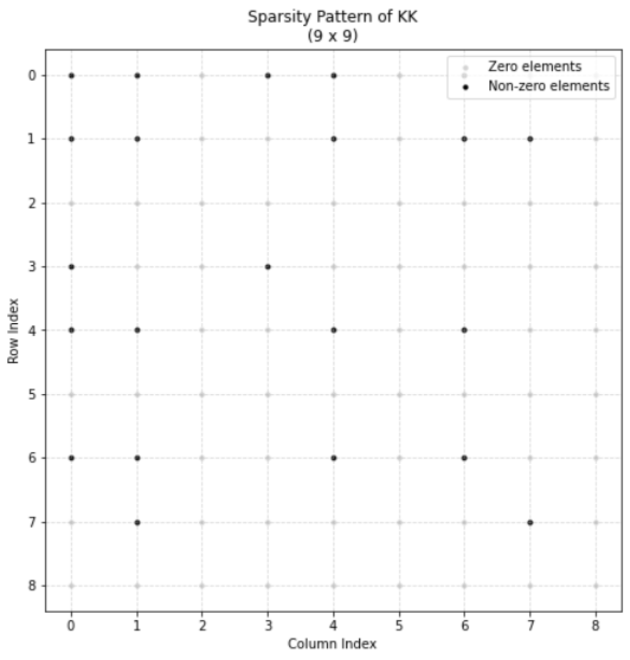


Figure 1: Sparsity pattern of the stiffness matrix on a unit triangle without any boundary conditions

We then thought about increasing the order of all the shape functions. Two possibilies arose :

- switching from the P1 triangle finite element to the P2 triangle finite element;
- keeping the same finite element interpolation points but increasing the order of the shape functions used.

2.3.1 • P2 triangle element

We first tried to switch from the P1 triangle finite element to the P2 triangle finite element as it seemed more "standard". **You can find a computation on the attached `unit_triangle_P2_quadratic.ipynb`**.

We faced multiples problems :

- the `Mesh` class had to be extended to compute edges midpoints coordinates, *we did not manage to make it work with multiple cells*, hence the computation is only limited to an unit triangular mesh;
- all the visualization functions from `MeshUtils.py` were thought for degrees of freedom only attached to nodes of the mesh, hence we had to only plot the first n_{nodes} components of the edge, *effectively plotting only half of the computed points*;
- the compilation of the symbolic functions from Sympy for this case were really long (≈15 min) for only 153 functions;

Even with all those challenges, we managed to go through an entire computation but we have nothing to confirm or not that our simulations are any good so we switched to the second possibility.

Here is an example deformed configurations with one node clamped and the force $\underline{F}$ applied on one edge:

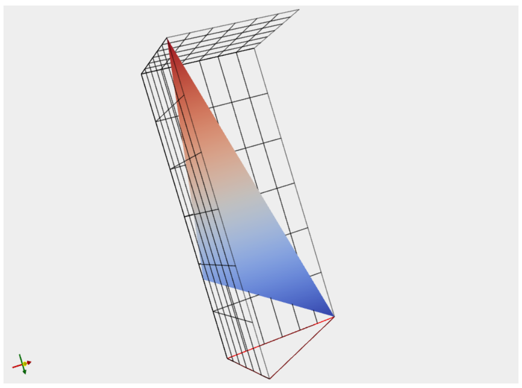


Figure 2: Deformed configuration on an unit triangle mesh with P2 shape functions initial mesh with parameters $h = 1\,m$, $E = 100\,Pa$, $\nu = 0.3$ and $\underline{F} = (0,\,0,\,10^{-12})^T\,N$

2.3.2 • Quadratic interpolation on triangle with 3 degrees of freedom

We then tried to increase the order of the shape function used along the 3 points interpolation triangle.

We defined a new `FiniteElement_Triangle_P1_quadratic` class and new `compute_quadratic_pseudo_-Lagrange_shape_functions_through_linear_system` function in `FiniteElement.py` to create a finite element with quadratic interpolations functions over it and only 3 interpolation points.
You can find the code for those in Appendix B.

The quadratic function used were pretty basic : $\phi(x, y) = a_1 + a_2x^2 + a_3y^2$. Those came naturally when we thought about increasing the order and respecting the "partition of unity" requirements for any shape functions over an element but we didn't investigate further to see if we could increase the order even more to get more precision.
As we thought it wasn't a "standard" approach we limited ourselves to this case which happened to enable the computations to go through but we reflected afterward and thought it would be possible to increase order but we would be forced to increase the number of interpolation points if we wanted to increase the number of coefficients present in the shape function.
This is because to get the values for (a_1, a_2, a_3), we need to solve the 3 equations, linear system : $\underline{\underline{A}} \cdot \underline{a} = \underline{b}$ with $\underline{a^T} = (a_1, a_2, a_3)$ and $b = (\delta_{k0}, \delta_{k1}, \delta_{k2})$.
We only came to this explanation later and had the luck to see it work before understanding that it was reason that they were 6 interpolation points on the P2 triangle elements for the general 2 variable quadratic shape function to be defined :

$$\phi(x, y) = a_1 + a_2x + a_3y + a_4xy + a_5x^2 + a_6y^2$$

Those 6 coefficients are determined with 6 linear equations : so 6 interpolation points are needed in a general case. **You can find a computation example in the attached notebook `Kirchoff-Love_plate_-P1_quadratic_interpolation.ipynb`**.

Here is a an example of deformed and initial configuration with one clamped and $\underline{F}$ applied along the opposite edge:
As expected and experienced with cards for instance, it requires much more force to deform the plate in it's plane rather than bending it

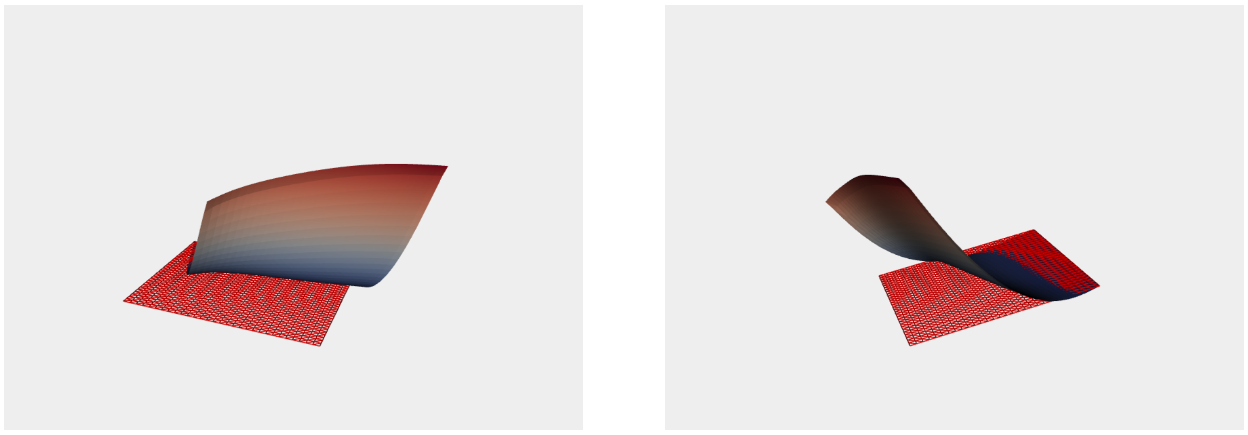


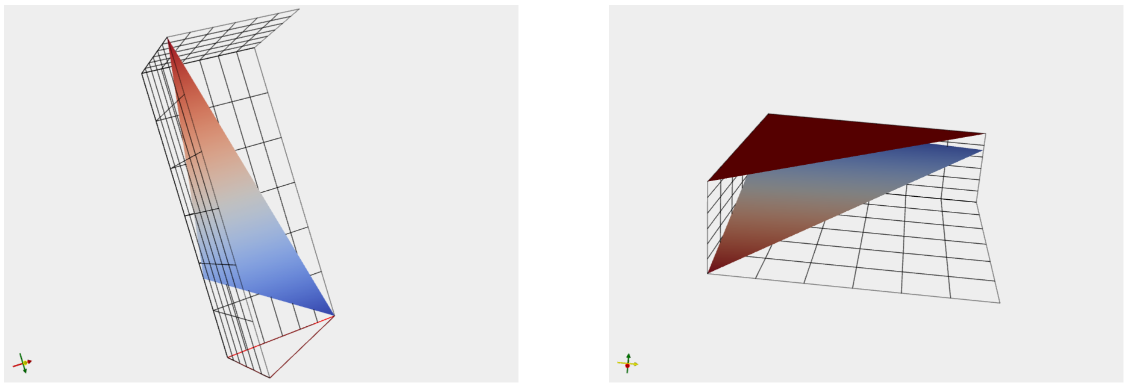
Figure 3: Deformed configuration where a x10 warp factor was applied and initial mesh with parameters $L_x = L_y = 10\,m$, $h = 1\,m$, $E = 100\,Pa$, $\nu = 0.3$ and $\underline{F} = (300,\ -300,\ 15)^T\,mN$

2.3.3 • Comparison on the unit triangle mesh between the P2 element and the quadratic P1 element

As the things we had working with the P2 element was a unit triangle mesh, it was the the only case for which we were able to compare both methods.

In both case, we clamped one node and load an edge. Here are the deformed configurations for both numerical approaches for the parameter set:

$$h = 1\ m,\ E = 100\ Pa,\ \nu = 0.3$$



(a) P2 element, $\underline{F} = (0,\ 0,\ -10^{-12})^T\ N$ (b) 3 points quadratic element, $\underline{F} = (0,\ 0,\ -1)^T\ N$

Figure 4: Deformed configurations under different loads

We twisted the representation but both go the same direction but there are two main things to point out:

- For the same physical parameters, there is a 10^{12} factor between the load necessary to have a strain around the unit : there is a clear numerical instability on the P2 element that we cannot account for;
- All nodes of the 3 point quadratic element are moving although we supposedfly clamped one node : even if the bigger mesh case depicted in Figure 4 seem physically viable, there is a problem somewhere with the computation but we didn't investigate further.

3
WITH FENICS

This section is dedicated to show our work with FeNICs, quickly abandoned in favor of LIB552.

Our approach relied on the bending-only case where the weak formulation simplify to give

$$\Delta^2 W = \frac{q(x,y)}{D}.$$

Here, W represents the transverse deflection, $q(x,y)$ is the distributed load applied to the plate, and D is the bending stiffness of the plate, defined as:

$$D = \frac{Et^3}{12(1-\nu^2)}.$$

Our goal was to implement tetrahedral meshing and apply loads to simulate the plate's deflection.

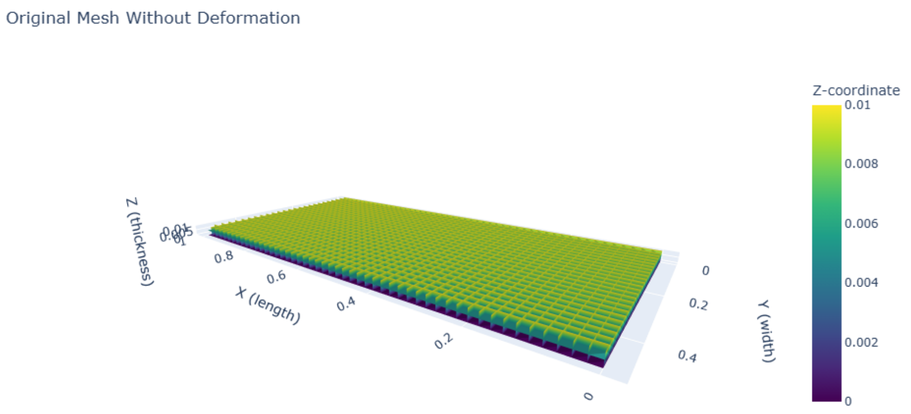


Figure 5: Mesh using FeniCs

However, the complexity of FEniCS, coupled with significant challenges in visualization and debugging, hindered our progress. Notably, the results produced by FEniCS exhibited unexpected behavior, such as wave-like deflection patterns (Figure 4) that suggested multiple force applications on the plate.

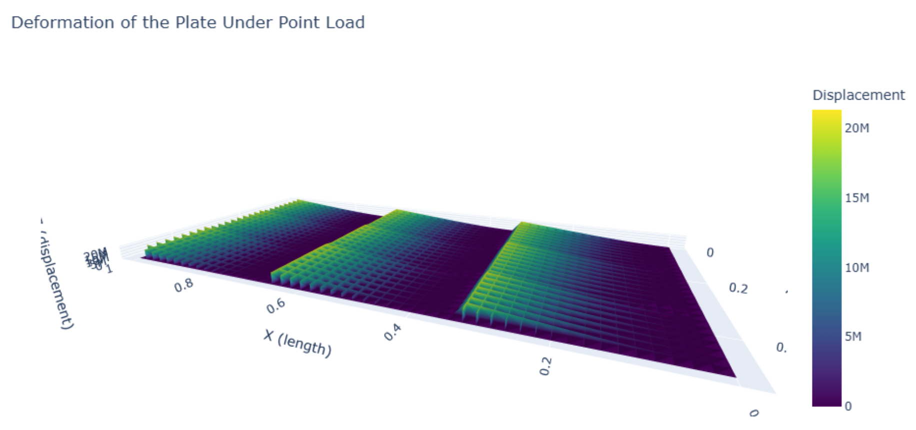


Figure 6: Attempt at applying point load on the plate using FeniCs

We choose to dive in the code of LIB552 instead of continuing to work with FeNICs.

APPENDIX A : NEW FUNCTIONS IMPLEMENTED IN VECTORELEMENT.PY

```python

    def get_B_B_and_P_P_int(self,mesh, k_cell, loc_mat):
        """(Efficient) computation of shape functions derivatives products element
        ↪  integral."""
        loc_mat[:,:] = self._get_B_B_and_P_P_int(*mesh.get_cell_nodes_coords(k_cell))

    def init_get_B_B_and_P_P_int(self, coeff_B, coeff_P, n=0):
        """Initializes the (efficient) computation of the shape functions symmetric
        ↪  gradients products element integral."""
        self._init_sym_B_and_P() #Compute the elementary B and P
        print("B and P initialized")
        self._init_sym_B_B_and_P_P(coeff_B,coeff_P) #Initializes the products
        print("BB and PP initialized")
        self._init_sym_B_B_and_P_P_int(n=n) #Initializes the integral of the array
        print("BB and PP int initialized")

        self._get_B_B_and_P_P_int = sympy.lambdify(
            args=self.sym_nodes.tolist(),
            expr=self.sym_B_B_and_P_P_int,
            modules="numpy")
        print("B_B_and_P_P_int initialized")

    def _init_sym_B_B_and_P_P(self,coeff_B,coeff_P):

        """Computes the (symbolic) products of shape functions symmetric gradients,
        and stores them as a (n_dofs x n_dofs) sympy Array."""

        # For 2D mesh, 3D vector case
        if self.dim == 2 and self.n_components == 3:
            # assert (self.sym_B.shape == (self.n_dofs, self.n_components))
            # assert (coeff_B.shape == (self.n_components, self.n_components))

            #La somme des deux termes
            self.sym_B_B_and_P_P = sympy.MutableDenseNDimArray.zeros(self.n_dofs,
            ↪  self.n_dofs)

            # B*A*B.T
            t_sym_B = sympy.transpose(self.sym_B)
            coeff_B = sympy.Array(coeff_B)


            AtB = sympy.tensorcontraction(
                                        sympy.tensorproduct(
                                            coeff_B,
                                            t_sym_B),
                                        (1, 2)
                                    )
            # print("Atb shape", AtB.shape)
            # print("sym_B shape:", self.sym_B.shape)

            self.sym_B_B = sympy.tensorcontraction(
                                sympy.tensorproduct(
                                    self.sym_B,
                                    AtB
                                ),
                                (1, 2)
                            )

            # P*D*P.T
            t_sym_P = sympy.transpose(self.sym_P)
            coeff_P = sympy.Array(coeff_P)


            DtP = sympy.tensorcontraction(
                                        sympy.tensorproduct(
                                            coeff_P,
                                            t_sym_P),
                                        (1, 2)
                                    )
            # print("Atb shape", DtP.shape)
            # print("sym_B shape:", self.sym_P.shape)

            self.sym_P_P = sympy.tensorcontraction(
                                sympy.tensorproduct(
                                    self.sym_P,
                                    DtP
                                ),
                                (1, 2)
                            )

            #On les somme
            self.sym_B_B_and_P_P = self.sym_B_B + self.sym_P_P

    def _init_sym_B_B_and_P_P_int(self, n):
        """Computes the (symbolic) integrals over the element of the of shape functions
        ↪ symmetric gradients products (stiffness matrix)."""
        # for i in range(self.sym_B_B_and_P_P.shape[0]):
        #     for j in range(self.sym_B_B_and_P_P.shape[1]):
        #         self.sym_B_B_and_P_P[i,j].
        self.sym_B_B_and_P_P_int =
        ↪ self.finite_element._integrate_array(array=self.sym_B_B_and_P_P, coeff=1, n=n)

    def _init_sym_B_and_P(self):
        #Initializes the shape functions derivatives
        self.finite_element._init_sym_dphi()
        self.finite_element._init_sym_ddphi()

        #Initializes arrays
        self.sym_B = sympy.MutableDenseNDimArray.zeros(self.n_dofs, self.n_components)
        self.sym_P = sympy.MutableDenseNDimArray.zeros(self.n_dofs, self.n_components)

        #Fill the arrays
        self._init_only_sym_P()
        self._init_only_sym_B()

    def _init_only_sym_P(self):
        assert self.dim == 2 and self.n_components == 3
        if (self.ordering == "point-wise"):
            #self.finite_element.sym_ddphi
            for k_dof in range(self.finite_element.n_dofs): #self.finite_element.n_dofs :
            ↪  they're at the 3 vertices for a P1 triangle element
                self.sym_P[3*k_dof+2,0] = - self.finite_element.sym_ddphi[k_dof ,0,0]
                self.sym_P[3*k_dof+2,1] = - self.finite_element.sym_ddphi[k_dof,1,1]
                self.sym_P[3*k_dof+2,2] = - (self.finite_element.sym_ddphi[k_dof,0,1] +
                ↪  self.finite_element.sym_ddphi[k_dof,1,0] )*numpy.sqrt(2) #the two terms
                ↪  are supposed to be equal


    def _init_only_sym_B(self):
        assert self.dim == 2 and self.n_components == 3
        if (self.ordering == "point-wise"):
            for k_dof in range(self.finite_element.n_dofs): #self.finite_element.n_dofs :
            ↪  they're at the 3 vertices for a P1 triangle element
                self.sym_B[3*k_dof , 0] = self.finite_element.sym_dphi[k_dof,0]
                self.sym_B[3*k_dof,2] = self.finite_element.sym_dphi[k_dof,1]/numpy.sqrt(2)

                self.sym_B[3*k_dof+1,1] = self.finite_element.sym_dphi[k_dof,1]
                self.sym_B[3*k_dof+1,2] =
                ↪  self.finite_element.sym_dphi[k_dof,0]/numpy.sqrt(2)
```

APPENDIX B : NEW CLASS AND SHAPE FUNCTION COEFFICIENT COMPUTING METHOD IN FINITEELEMENT.PY

```
    def compute_quadratic_pseudo_Lagrange_shape_functions_through_linear_system(sym_x,
    ↪  sym_points):
    n_dofs = sym_points.shape[0]
    sym_phis = []
    for k_dof in range(n_dofs):
        # print (k_dof)
        ak = sympy.symbols('a:{}'.format(n_dofs))
        # print (ak)
        if   (n_dofs == 3):
            sym_phi = ak[0]              \
                    + ak[1] * sym_x[0]**2 \
                    + ak[2] * sym_x[1]**2
        else:
            assert (0), "Not implemented. Aborting."
        # print (sym_phi)
        ak_sol = sympy.solve(
            [sym_phi.subs({sym_x[0]:sym_points[l_dof,0], sym_x[1]:sym_points[l_dof,1]}) -
            ↪  float(l_dof == k_dof) for l_dof in range(n_dofs)],
            ak)
        # print (ak_sol)
        sym_phi = sym_phi.subs(ak_sol)
        # print (sym_phi)
        sym_phis.append(sym_phi)
    return sym_phis

    class FiniteElement_Triangle_P1_quadratic(FiniteElement_Triangle):
    def __init__(self):
        super().__init__() # FiniteElement_Triangle.__init__()
        self.interpolation = "P1"
        self.n_points = 3
        self.sym_points = sympy.Array(
            [self.sym_nodes[0],
             self.sym_nodes[1],
             self.sym_nodes[2]])
        self.n_dofs = 3
        self.dofs_attachement = ["node"]*3
        self.dofs_attachement_idx = [0, 1, 2]
        self.sym_phi = sympy.Array(

            ↪  compute_quadratic_pseudo_Lagrange_shape_functions_through_linear_system(self.sym_x,
            ↪  self.sym_points))

```

REFERENCE SECTION

1. Florence Zara. Modèle mécanique d'une plaque mince. Doctorat. France. 2017, pp.20 cel-01520287
2. Eléments finis de Structures Cours de Plaques `http://lafamilledurefuge.free.fr/doc/S8/Calcul%20de%20structure/Slides_Plaques-1x2.pdf`