How We Built a 2D FEM Program in Python from Scratch

Victor Lüddemann, Emile Breyer

August 7, 2025

Contents

Abstract

This document is part of a student project in which a 2D FEM tool in Python has been developed, aiming to make FEM theory more accessible and practical for students, engineers, and anyone looking to dive deeper into computational mechanics.

The document presents the development and verification of the finite element library `femlearn.py` that can be accessed via https://github.com/VictorLued/femlearn.py:

- The software's structure, including class hierarchies, information flow, and class declarations are described. Subsequently, the usage of the library is outlined, covering the stages of preparation, pre-processing, solution, and post-processing, as well as the visualization of results through plotting functionalities.

- The results and verification of the implemented algorithms are also taken into account. Benchmark load cases, such as tension and bending and line loads, are modeled and simulated for different element types and different plane conditions. The results obtained with `femlearn.py` are compared with those of the commercial FEM software ADINA. The close agreement in displacements and stress values reinforces the correctness of the implementation for the given use cases. However, the Python tool should still be used with caution as undetected errors and bugs might exist.

Due to its open-source nature and modular structure, `femlearn.py` is particularly suited for academic environments and educational purposes in the field of the two dimensional finite element method.

1 Introduction

1.1 Motivation

Finally understand the Finite Element Method (FEM) properly! What better way to achieve this than by independently implementing a functional FEM code? And in which degree program would such a project be better placed than in the Master's program *Computation and Simulation in Mechanical Engineering* at University of Applied Sciences (HAW) Hamburg, where the focus lies on the many facets of the FEM?

The result of this work is an openly accessible Python library that can be imported for the analysis of two-dimensional models and extended as desired. This gives users access to all information and processes that are part of solving problems using FEM. *Shape functions, stiffness matrices, equivalent nodal forces, Gauss integration* – here, theory becomes practice. This documentation was written especially with the intention of helping other students understand how commercial Finite Elements (FE) software works.

1.2 Methodology

The Python implementation named `femlearn.py` comprises 24 classes and nearly 3000 lines of code (including comments). In the academic context, this represents a larger project that requires some planning and coordination for division of labor. This section is intended to provide a brief insight into these processes.

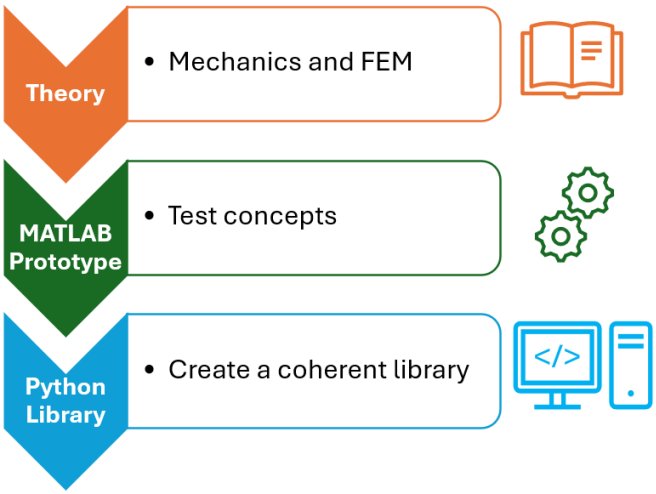


Figure 1.1: Project steps

The project can essentially be divided into three steps (see Fig. 1.1). The study of the theory and its implementation in the form of smaller code snippets is initially carried out independently by both students. The code snippets include first attempts at implementing shape functions or creating the element stiffness matrix. The result is an initial prototype developed in the commercial software *MATLAB*, which proves advantageous due to its convenient handling of matrices for rapid prototyping. Finally, the code is implemented

in the programming language Python, which is well suited for the project due to its free availability and extensive libraries. To enable parallel work on the Python code, agreement is reached in advance on class names and their dependencies. This allows the programming work to be divided between two people:

1. Student:
    - Pre-processing: Creating geometries, automatic meshing, and importing external files,
    - Visualization: Displaying geometries, elements, boundary conditions, and result quantities.
2. Student:
    - Data structure: General implementation of class structures,
    - Solver: Processing the mesh and boundary conditions, setting up equations, and outputting solution variables.

2 Structure

2.1 Hierarchy

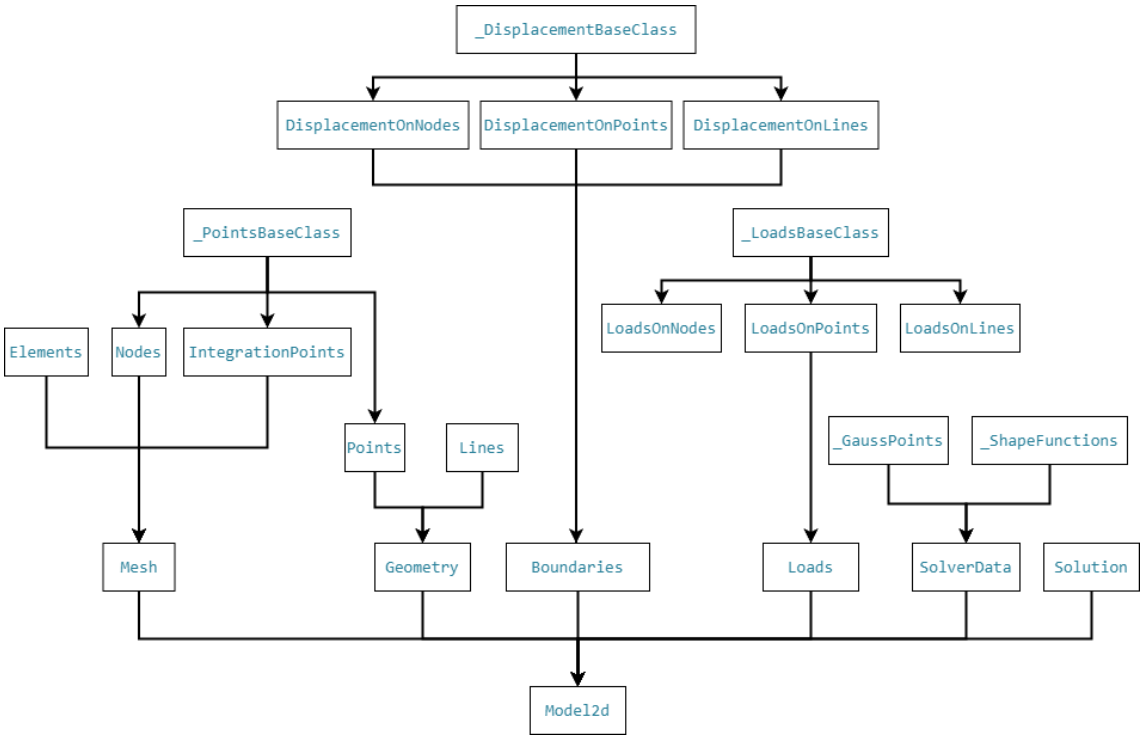


Figure 2.1: Hierarchy of the implemented classes

Figure 2.1 shows the hierarchy of the implemented classes. The class `Model2D()` contains all information that is defined for or generated within the FE model. The subclasses represent intermediate steps of a typical simulation process:

1. Pre-processing: `Mesh()`, `Geometry()`, `Boundaries()`, `Loads()`,
2. Solving: `SolverData()`,
3. Post-processing: `Solution()`.

These in turn partially contain subordinate classes that must be defined by the user.

1. `Mesh()`:
   - `Elements()`,
   - `Nodes()`,
   - `IntegrationPoints()`,
2. `Geometry()`:
   - `Points()`,

- `Lines()`,

3. `Boundaries()`:
   - `DispalcementOnNodes()`,
   - `DispalcementOnPoints()`,
   - `DisplacementOnLines()`,
4. `Loads()`:
   - `LoadsOnNodes()`,
   - `LoadsOnPoints()`,
   - `LoadsOnLines()`,
5. `SolverData()`,
6. `Solution()`.

2.2 Information Flow

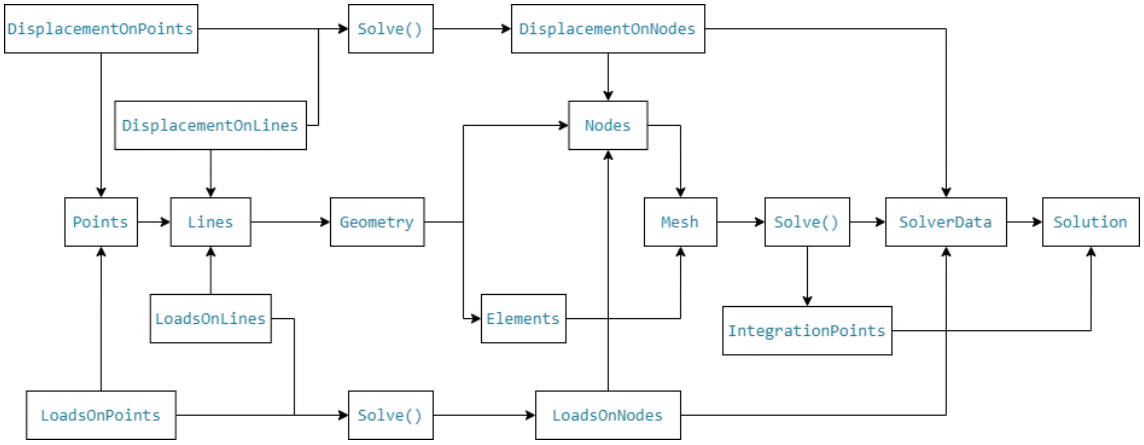


Figure 2.2: Information flow during model creation and solution

Figure 2.2 shows the essential information flow between the classes. A model can generally be created in two ways. In the direct method, nodes and elements are manually generated. Boundary conditions are applied directly to the nodes by the user, allowing the equation system to be built immediately from this method. However, the model can also be defined indirectly using geometric points. In this case, only the geometry is specified. The mesh is then generated automatically afterward. Therefore, the boundary conditions applied to the geometry are transferred to the nodes before assembling the equation system when using the `solve()` function.

2.3 Class Declarations

Base Classes

Code blocks 2.1 to 2.3 show the declarations of the base classes. These inherit fundamental structures on which later classes are built.

```
class _PointsBaseClass():
    def __init__(self, type="default", coordinates=[[]], ids=[]):...
    @property
        def x(self):...
    @property
        def y(self):...
    def setCoordinates(self, coordinates):...
    def setIds(self, ids):...
    def _findIndexByNodeIds(self, ids):...
    def findCoordinatesByNodeIds(self, ids):...
    def printParameters(self):...
    def add(self, coordinate, id=None):...
    def plot(self,color = 'red'):...
```

Python Code 2.1: Declaration: _PointsBaseClass()

```
class _LoadsBaseClass():
    def __init__(self, type="default", loads=[], ids=[]):...
    @property
        def x(self):...
    @property
        def y(self):...
    def setLoads(self, loads):...
    def setIds(self, ids):...
```

Python Code 2.2: Declaration: _LoadsBaseClass()

```
class _DisplacementBaseClass():
    def __init__(self, type="default", displacements=[], ids=[]):...
    @property
        def x(self):...
    @property
        def y(self):...
    def setDisplacements(self, displacements):...
    def setIds(self, ids):...
    def printParameters(self):...
```

Python Code 2.3: Declaration: _DisplacementBaseClass()

Meshing

Code blocks 2.4 to 2.7 show the class declarations relevant for building the FE mesh.

```
class Nodes(_PointsBaseClass):...
    def __init__(self, coordinates=[], ids=[]):...
```

Python Code 2.4: Declaration: Nodes()

```
class IntegrationPoints(_PointsBaseClass):...
    def __init__(self, coordinates=[], ids=[]):...
```

Python Code 2.5: Declaration: IntegrationPoints()

```
class Elements():
    def __init__(self, nodeIds=[[]], integrationpointIds=[[]],integrationOrder=[], thickness=[],
     youngsModulus=[], poissonsRatio=[], planarAssumption=[], ids=[]):...
    def setIds(self, ids):...
    def setNodeIds(self, nodeIds):...
    def findIndexByElementIds(self, ids):...
    def findNodeIdsByElementId(self, id):...
    def setIntegrationpointIds(self, integrationpointIds):...
    def findIntegrationpointIdsByElementId(self, id):...
    def setThickness(self, thickness):...
    def setYoungsModulus(self, youngsModulus):...
    def setPoissonsRatio(self, poissonsRatio):...
```

```
    def setPlanarAssumption(self, planarAssumption):...
    def printParameters(self):...
    def add(self, nodeId, integrationPointId, thickness, youngsModulus, poissonsRatio,
     planarAssumption, id=None):...
```

Python Code 2.6: Declaration: Elements()

```
class Mesh():
    def __init__(self, nodes=Nodes(), elements=Elements()):...
    @property
    def numberOfElements(self):...
    @property
    def numberOfNodes(self):...
    def setNodes(self, nodes):...
    def getNodes(self):...
    def getNodeSelection(self,ids):...
    def setElements(self, elements):...
    def getElements(self):...
    def getElementSelection(self,ids):...
    def getNodesOfElementId(self,id):...
    def setIntegrationPoints(self, integrationPoints):...
    def getIntegrationPoints(self):...
    def plotMesh(self,show_ids = False):...
    def printParameters(self):...
```

Python Code 2.7: Declaration: Mesh()

Geometry

Code blocks 2.8 to 2.10 show the class declarations that contain geometric information.

```
class Points(_PointsBaseClass):
    def __init__(self, nodes=Nodes(), def __init__(self, coordinates=[], ids=[]):...
```

Python Code 2.8: Declaration: Points()

```
class Lines():
    def __init__(self, pointIds=[], ids=[]):...
    def setPoints(self, pointIds):...
    def setIds(self, ids):...
    def _findIndexByLineIds(self, ids):...
    def printParameters(self):...
    def add(self, referencePointId, id=None):...
```

Python Code 2.9: Declaration: Lines()

```
class Geometry():
    def __init__(self, points=Points(), lines=Lines()):...
    def setPoints(self, points):...
    def setLines(self, lines):...
    def _findIndexByLineIds(self, ids):...
    def add(self, referencePointId, id=None):...
    def makePolygon(self, coordinates):...
    def plot(self):...
    def printParameters(self):...
```

Python Code 2.10: Declaration: Geometry()

Boundary Conditions

Code blocks 2.11 to 2.14 show the class declarations containing information about boundary conditions.

```
class DisplacementOnNodes(_DisplacementBaseClass):
    def __init__(self, displacements=[[]], nodeIds=[], ids=[]):...
    def setNodeIds(self, nodeIds):...
    def _getNodeIdsWithFixationX(self):...
    def _getNodeIdsWithFixationY(self):...
    def _getPrescribedNodalDispalcementX(self):...
    def _getPrescribedNodalDispalcementY(self):...
    def add(self, displacement, nodeId, id=None):...
    def printParameters(self):...
    def remove(self, id):...
```

Python Code 2.11: Declaration: DisplacementOnNodes()

```
class DisplacementOnPoints(_DisplacementBaseClass):
    def __init__(self, displacements=[[]], pointIds =[], ids=[]):...
    def setpointIds(self, pointIds):...
    def add(self, displacement, referencePointId, id=None):...
    def printParameters(self):...
```

Python Code 2.12: Declaration: DisplacementOnPoints()

```
class DisplacementOnLines(_DisplacementBaseClass):
    def __init__(self, displacements=[[]], lineIds=[], ids=[]):...
    def setLineIds(self, lineIds):...
    def add(self, displacement, lineId, id=None):...
    def printParameters(self):...
```

Python Code 2.13: Declaration: DisplacementOnLines()

```
class Boundaries():
    def __init__(self, displacementOnNodes=DisplacementOnNodes(), displacementOnPoints=
     DisplacementOnPoints(), def setDisplacementOnNodes(self, displacementOnNodes):...
    def getDisplacementOnNodes(self):
    def setDisplacementOnPoints(self, displacementOnPoints):...
    def getDisplacementOnPoints(self):...
    def setDisplacementOnLines(self, displacementOnLines):...
    def getDisplacementOnLines(self):...
    def printParameters(self):...
```

Python Code 2.14: Declaration: Boundaries()

Boundary Loads

Code blocks 2.15 to 2.18 show the class declarations containing information about boundary loads.

```
class LoadsOnNodes(_LoadsBaseClass):
    def __init__(self, loads=[[]], nodeIds=[], ids=[]):...
    def setNodeIds(self, nodeIds):
    def printParameters(self):...
    def add(self, load, nodeId, id=None):...
```

Python Code 2.15: Declaration: LoadsOnNodes()

```
class LoadsOnPoints(_LoadsBaseClass):
    def __init__(self, loads=[[]], pointIds=[], ids=[]):...
    def setpointIds(self, pointIds):
    def printParameters(self):...
    def add(self, load, referencePointId, id=None):...
```

Python Code 2.16: Declaration: LoadsOnPoints()

```
class LoadsOnLines(_LoadsBaseClass):
    def __init__(self, loads=[[]], lineIds=[], ids=[]):...
    def setLineIds(self, lineIds):...
    def printParameters(self):...
    def add(self, load, lineId, id=None):...
```

Python Code 2.17: Declaration: LoadsOnLines()

```
class Loads():
    def __init__(self, loadsOnNodes=LoadsOnNodes(), loadsOnPoints=LoadsOnPoints(), loadsOnLines=
     LoadsOnLines()):...
    def setLoadsOnNodes(self, loadsOnNodes):...
    def getLoadsOnNodes(self):...
    def setLoadsOnPoints(self, loadsOnPoints):...
    def getLoadsOnPoints(self):...
    def setLoadsOnLines(self, loadsOnLines):...
    def getLoadsOnLines(self):...
    def printParameters(self):...
```

Python Code 2.18: Declaration: Loads()

Solver

Code blocks 2.19 to 2.21 show the class declarations containing information about the solver.

```
class _ShapeFunctions():
    def __init__(self):...
    def _getShapeFunctionCoefficients(self, xi=0.0, eta=0.0, eType=0):...
    def _getShapeFunctionDerivateCoefficients(self, xi=0.0, eta=0.0, eType=0):...
    def _getLineLoadCoefficients(self, eType=0):...
```

Python Code 2.19: Declaration: _ShapeFunctions()

```
class _GaussPoints():
    def __init__(self):...
    def _getTRIAGaussPoint(self, order=0):...
    def _getQuadGaussPoint(self, order=0):...
    def _getTRIAGaussWeights(self, order=0):...
    def _getQuadGaussWeights(self, order=0):...
```

Python Code 2.20: Declaration: _GaussPoints()

```
class SolverData(_GaussPoints, _ShapeFunctions):
    def __init__(self, mesh=Mesh(), meshDeformed=Mesh()):...
    def _getIntegrationPointsCoordinates(self,nodeCoords=np.empty([1, 2]), xi=0.0, eta=0.0):
    def _getElementDMatrix(self, E=0.0, nu=0.0, behaviour=""):...
    def _getElementBMatrixandJacobiDeterminant(self, nodeCoords=np.empty([1, 2]), xi=0.0, eta=0.0)
     :...
    def _getElementKMatrix(self, nodeCoords=np.empty([1, 2]), order=0, E=.0, nu=.0, t=.0, behaviour=
     ""):...
    def _getMatrixGuideVector(self, nodeIndex=np.empty([1, 1])):...
    def _findNearestNeighbour(self, coordinates, refCoordinates):...
    def _findNodesBetweenTwoPoints(self, nodeCoords, pointCoords, tolerance=1E-10):...
    def _resolveDuplicateDisplacements(self):...
    def _resolveDuplicateLoads(self):...
    def _combineBoundaries(self, geometry=Geometry(), boundaries=Boundaries()):...
    def _combineLoads(self,geometry=Geometry(), loads = Loads()):...
    def _buildStiffnessMatrix(self):...
    def _buildLoadVector(self):...
    def _applyPrescribedDisplacements(self):...
    def _reduceStiffnesMatrixAndLoadVector(self):...
    def _solveLinearEquation(self):...
    def _getNodalSolution(self):...
    def _getFullSolution(self):...
```

Python Code 2.21: Declaration: SolverData()

Solution

Code block 2.22 shows the declaration of the solution class containing the solution results.

```
class Solution():
    def __init__(self, mesh=Mesh(), meshDeformed=Mesh(),
                     dispX=[], dispY=[], dispTotal=[],
                     strainX=[], strainY=[], strainXY=[],
                    stressX=[], stressY=[], stressXY=[],
                    stress11=[], stress22=[], stress33=[],
                    stressMises=[], stressZ=[]):...
    def _getAveragedResult(self,variableName):...
```

Python Code 2.22: Declaration: Solution()

Model

Code block 2.23 shows the declaration of the model class which contains all the information about the model.

```
class Model2d():
    def __init__(self, geometry=Geometry(), mesh=Mesh(), boundaries = Boundaries(), loads=Loads(),
     solverData = SolverData(), solution = Solution()):...
    def printParameters(self):...
    def _checkParameters(self):...
    def generateQuadMesh(self, nx=1, ny=1, type = 4, specified_nodes=None, integrationOrder=4,
     thickness=[], poissonRation=[], youngsModulus=[], planarAssumption=[]):...
    def generateTRIAngleMesh(self,size, type=3, integrationOrder=1, thickness=[], poissonRation=[],
     youngsModulus=[], planarAssumption=[]):...
    def importNasFile(self,filename,integrationOrder=2):...
    def solve(self):...
    def plotSolution(self,variableName,averaged=False,deformed=True):...
    def plotBoundaries(self, deformed=False):...
    def plotLoads(self, deformed=False):...
    def plotMesh(self,deformed=False, show_ids = False):...
```

Python Code 2.23: Declaration: Model2d()

3 Using the Library

3.1 Preparation

The finite element tool `femlearn.py` that can be accessed via https://github.com/VictorLued/femlearn.py. Since the code is implemented in the Python programming language, Python must be installed beforehand to use it. This can be done, for example, via the official website at https://www.python.org/downloads/windows/. It is also recommended to use an integrated development environment such as *Visual Studio Code* or *Spyder*. Since `femlearn.py` relies on several external packages, these need to be installed separately. It is recommended to set up a virtual environment for this purpose. To do so, open Windows Powershell or Command Prompt and enter the following commands to create and activate a new environment:

```
py -m venv environment_name

environment_name/Scripts/activate
```

Then, the required packages can be installed using the following command:

```
pip install -r path/to/requirements.txt
```

The file `requirements.txt` is located in the project directory. Alternatively, the packages can be installed individually.

3.2 Pre-Processing

In general, the pre-processing follows these steps:

1. Creating a model with `Model2d()`,
2. Creating the geometry (optional),
3. Generating the finite element mesh and defining element parameters,
4. Applying boundary conditions and loads,
5. Assigning everything to the model.

Model Setup via Explicit Definition

There are several ways to set up the model. With the *direct method*, the FE mesh is explicitly defined. This means that the coordinates of each node and their connectivity into elements are manually specified by the user. Nodes and elements are defined using the initialization functions `Nodes()` and `Elements()` and added to the model using the methods `mesh.setNodes()` and `mesh.setElements()`. Element thickness, material properties (Poisson's ratio and Young's modulus), and the stress or strain assumptions must then be defined. These values can be set individually for each element or globally for the entire model, depending on whether a single value or a list is passed. Next, the boundary conditions are defined. Since no geometric points or lines exist in the direct definition, supports

and loads must be applied directly to the nodes. For this, they are first created as objects using `DisplacementOnNodes` and `LoadsOnNodes` and then assigned to the model via the methods `boundaries.setDisplacementOnNodes` and `loads.setLoadsOnNodes`. An example setup is shown in code block 3.1.

```
from femlearn import *

# Create a model
model = Model2d()

# Create nodes
nodes = Nodes(
        coordinates = [
                [0,0],
                [1,0],
                [2,0],
                [2,1],
                [1,1],
                [0,1]
                ],
        ids = [1,2,3,4,5,6]
)
model.mesh.setNodes(nodes)

# Create elements
elements = Elements(
        nodeIds = [
                [1,2,5,6], # Element 1 consists of nodes 1,2,5,6
                [2,3,4,5]
                ],
        integrationOrder=[4] # Four integration points
)

# Set element properties
model.mesh.elements.setPlanarAssumption("plane stress")
model.mesh.elements.setThickness(1)
model.mesh.elements.setPoissonsRatio(.3)
model.mesh.elements.setYoungsModulus(1)

model.mesh.setElements(elements)

# Define boundary conditions
disp = DisplacementOnNodes(
        displacements = [
                [0,0],
                [0,"free"]
                ],
        nodeIds = [1,6]
)
model.boundaries.setDisplacementOnNodes(disp)

# Define loads
force = LoadsOnNodes(
        loads = [[0,-0.01]],
        nodeIds= [2]
)
model.loads.setLoadsOnNodes(force)

# Solve model
model.solve()
```

Python Code 3.1: Example: Direct node-based definition.

Model Setup via Geometry

As a second option, users can define the model using geometry. Instead of manually defining nodes and elements, geometric points and lines can be created and then meshed automatically. Rectangular QUAD4 meshes can be generated via a custom implemented

mesher. Arbitrary polygon shapes can be meshed using an external algorithm. Alternatively, meshes can be imported directly from commercial software like *ADINA* (refer to next section). An advantage of using geometry is that loads and boundary conditions are independent of the mesh.

Unlike the previous section, geometry must now be defined. This can be done using the `geometry.makePolygon()` function, which automatically creates the points and lines of a polygon. Alternatively, geometric points and lines can be manually defined and added. Once the geometry is created, it can be meshed. An example is shown in code block 3.1.

```
from femlearn import *

model = Model2d()

# Create geometry
model.geometry.makePolygon(
        coordinates=[
                [0,0],
                [20,0],
                [20,4],
                [0,10]
                ])

model.geometry.plot()
plt.show()

# Meshing
model.generateTRIAngleMesh(
        size = 2,
        type = 6,
        integrationOrder = 3,
        thickness = 1,
        poissonRation = 0.3,
        youngsModulus = 100,
        planarAssumption = "plane stress")

# Boundary conditions
disp = DisplacementOnPoints(
        displacements = [
                [0,0],
                [0,"free"]
                ],
        pointIds=[1,4])
model.boundaries.setDisplacementOnPoints(disp)

# Loads
loads = LoadsOnPoints(
        loads = [[0,-10]],
        pointIds = [2])
model.loads.setLoadsOnPoints(loads)

model.solve()
```

Python Code 3.2: Example: Model setup via geometry.

Importing External Models

Meshes can also be imported as Nastran files. Exporting a mesh from commercial software like *ADINA* is possible via *File* → *Export NASTRAN...*. The `.nas` file can then be imported into the model using the function `importNasFile()` (see code block 3.3). The ADINA import only works for quadrilateral elements within a single *Element Group*. After import, the remaining model parameters like geometry, material data, boundary conditions, and loads must still be defined.

```
from femlearn import *

model = Model2d()
```

```
model.geometry.makePolygon(
        coordinates=[[0,0],
                [20,0],
                [20,4],
                [0,10]])

model.importNasFile(
        filename='filename.nas',
        integrationOrder=1,
        thickness=1,
        poissonRation=0.3,
        youngsModulus=1,
        planarAssumption="plane stress")

disp = DisplacementOnPoints(
        displacements=[[0,0],
        [0,"free"]],
        pointIds=[1,4])

model.boundaries.setDisplacementOnPoints(disp)
#...
```

Python Code 3.3: Import from ADINA files.

3.3 Solver and Post-Processing

Once the model has been set up in the pre-processing phase, it can be solved. The results can then be printed and visualized.

Solving the Model

To solve the model, the function `model.solve()` is called. Internally, all boundary conditions are converted and the stiffness matrix is assembled. Once the system of equations is solved, the variables are available as the solution.

Output Solver Data

Various data are generated during the solution process, such as the system matrices or computation time. Users can access and print these values as shown in code block 3.4.

```
from femlearn import *

model = Model2d()

# model setup with geometry, mesh, boundaries, etc.
#...

model.solve()

# Output solver data
print(model.solverData.stiffnessMatrix) # Stiffness matrix (without boundaries)
print(model.solverData.reducedStiffnessMatrix) # Stiffness matrix (with boundaries)
print(model.solverData.elapsedTime) # Elapsed time of the solver
print(model.solverData.loadVector) # Load vector
print(model.solverData.displacementVector) # Displacement vector
print(model.solverData.numberDOF) # Number of degrees of freedom
```

Python Code 3.4: Example: Available solver data.

Output Solution Variables

The results of the model are stored in `model.solution`. The available variables are listed in code block 3.5.

```
from femlearn import *

model = Model2d()

# model setup with geometry, mesh, boundaries, etc.
#...

model.solve()

# Displacements
print(model.solution.dispX)
print(model.solution.dispY)
print(model.solution.dispTotal)

# Strains
print(model.solution.strainX)
print(model.solution.strainY)
print(model.solution.strainXY)

# Stresses
print(model.solution.stress11)
print(model.solution.stress22)
print(model.solution.stress33)
print(model.solution.stressX)
print(model.solution.stressY)
print(model.solution.stressZ)
print(model.solution.stressXY)
print(model.solution.stressMises)
```

Python Code 3.5: Example: Available solution variables.

3.4 Working with Plots

Users have a wide range of options for creating plots. Geometry, various types of points (nodes, integration points, and geometric points), the FE mesh, loads, boundary conditions, and solutions can all be visualized and combined. An example of plotting during pre-processing is shown in code block 3.6.

```
from femlearn import *

model = Model2d()

model.geometry.makePolygon(
coordinates=[[0,0],
[20,0],
[20,4],
[0,10]])

# Plot geometry
model.geometry.plot()
plt.show()

model.importNasFile(
filename='filename.nas',
integrationOrder=1,
thickness=1,
poissonRation=0.3,
youngsModulus=1,
planarAssumption="plane stress")

disp = DisplacementOnPoints(
displacements=[[0,0],
[0,"free"]],
pointIds=[1,4])
model.boundaries.setDisplacementOnPoints(disp)

# Define loads
loads = LoadsOnPoints(loads=[[0,-10]],
```

```
pointIds=[2])
model.loads.setLoadsOnPoints(loads)

# Plot mesh
model.plotMesh()
# Plot loads
model.plotLoads()
# Plot boundary conditions
model.plotBoundaries()
plt.show()

#...
```

Python Code 3.6: Example: Plotting in Pre-Processing.

After the model is solved, solution variables like displacements or stresses can also be visualized. All variables listed in Section 3.3 can be plotted using the function `plotSolution()`. To also display boundary conditions, the functions `plotLoads()` and `plotBoundaries()` can be combined. `plotSolution()` accepts up to three arguments:

1. the name of the variable to be plotted,
2. whether to average values per element or plot them at nodes or integration points,
3. whether to display the mesh in its deformed and undeformed state.

An example is shown in code block 3.7. The resulting plots are shown in Figs. 4.4 to 4.5.

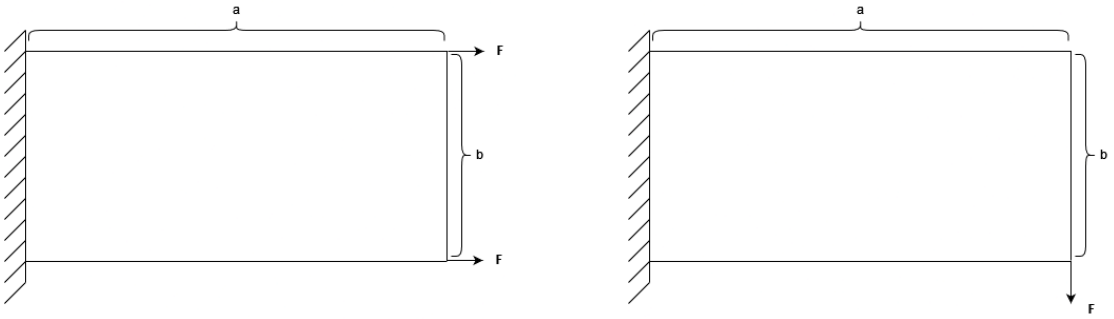
```
from femlearn import *

model = Model2d()

# model setup with geometry, mesh, boundaries, etc.
#...

model.solve()

# Y-displacement in deformed mesh with loads and BCs
model.plotSolution("dispY", averaged=True, deformed=True)
model.plotBoundaries(deformed=True)
model.plotLoads(deformed=True)
plt.show()

# Mises stress in undeformed mesh
model.plotSolution("stressMises", averaged=False, deformed=False)
model.plotMesh(deformed=False)
model.plotBoundaries(deformed=False)
model.plotLoads(deformed=False)
plt.show()
```

Python Code 3.7: Example: Plotting solution variables.

4 Use Cases and Verification

4.1 Model for Tension and Bending Load Cases

To ensure the correctness of the implemented code, it is verified using the commercial FEM software. For this purpose, the results for the geometry shown in Figure 4.1 are compared for two different load cases (tension and bending) and for all implemented element types. The corresponding parameters are listed in Table 4.1.



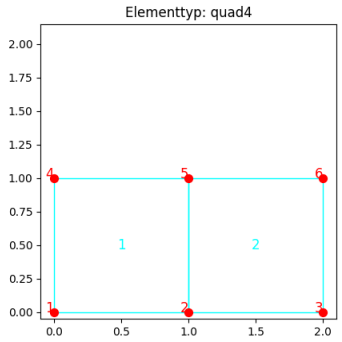
(a) Tension load case (b) Bending load case

Figure 4.1: Load cases to be compared

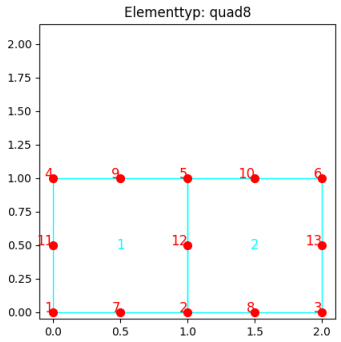
Table 4.1: Parameters for the verification models

Parameter	**Value**
Force F	0.01
Length a	2
Length b	1
Young's modulus	1
Poisson's ratio	0.3
Thickness	1
Assumption	plane stress

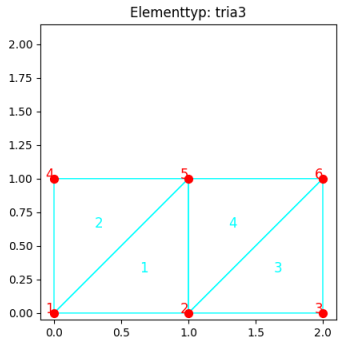
The meshes for the four different element types are shown in Figure 4.2. The meshes were created in ADINA, exported, and then imported into `femlearn.py`, ensuring identical nodes and elements for both programs. The plots were done using `femlearn.py` functions.



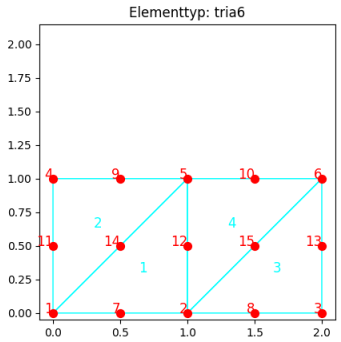
(a) QUAD4



(b) QUAD8



(c) TRIA3



(d) TRIA6

Figure 4.2: Models with different meshes

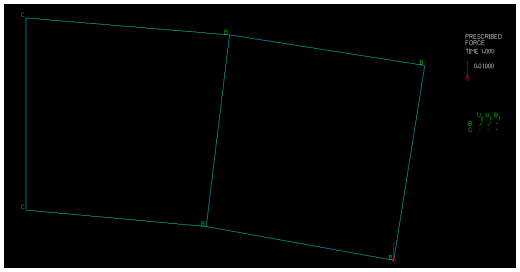
4.2 Results for Bending Load Cases

Displacement

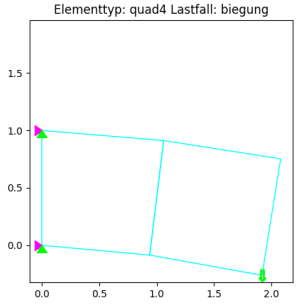
Figure 4.3 shows a comparison of the deformed meshes for the different element types under bending. As an example, Table 4.2 shows the displacement values for QUAD4 elements. Note that ADINA uses a different global coordinate system (ADINA Y represents `femlearn.py` dispX). The node positions can be found in Figure 4.2a. It is evident that the values match exactly within the representable digits. Tables with all results can be found in Appendix 5. They show that the results for the other element types and for the tension load case also match those from ADINA. This indicates that ADINA uses the same methodology as `femlearn.py`.

ADINA			`femlearn.py`	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	-0.0603066	-0.0854664	-0.060306590970	-0.085466414344
Node 3	-0.0822979	-0.261319	-0.082297880746	-0.261319366390
Node 4	0	0	0	0
Node 5	0.0610267	-0.0878669	0.061026742363	-0.087866918989
Node 6	0.0794799	-0.247125	0.079479897032	-0.247125078054

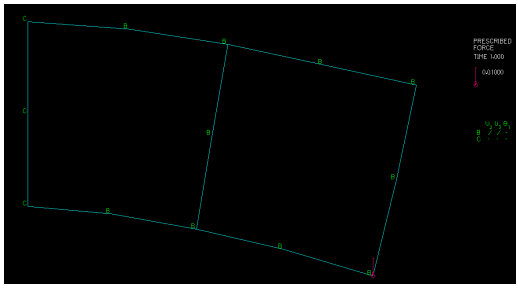
Table 4.2: Comparison of displacements for QUAD4 elements in the bending load case



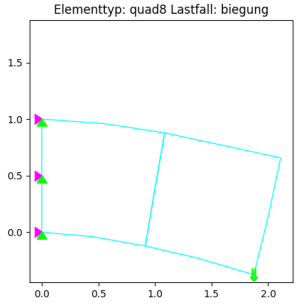
(a) QUAD4 ADINA



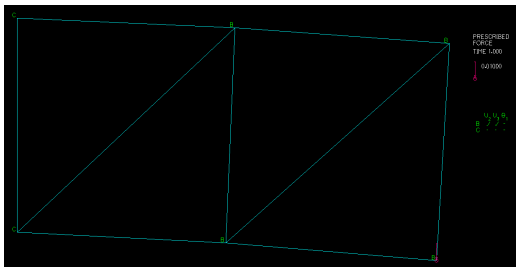
(b) QUAD4 `feamlearn.py`



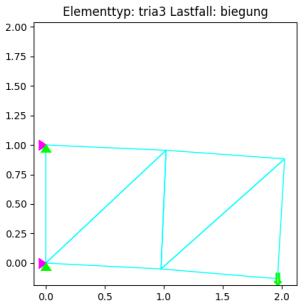
(c) QUAD8 ADINA



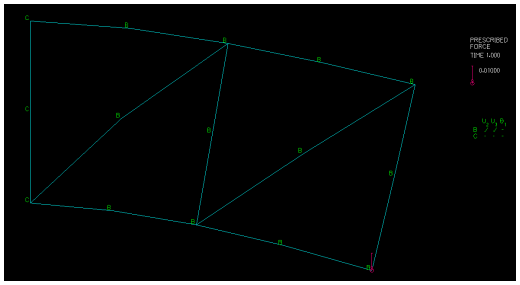
(d) QUAD8 `feamlearn.py`



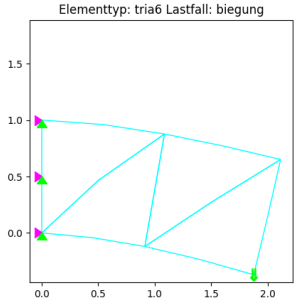
(e) TRIA3 ADINA



(f) TRIA3 `feamlearn.py`



(g) TRIA6 ADINA

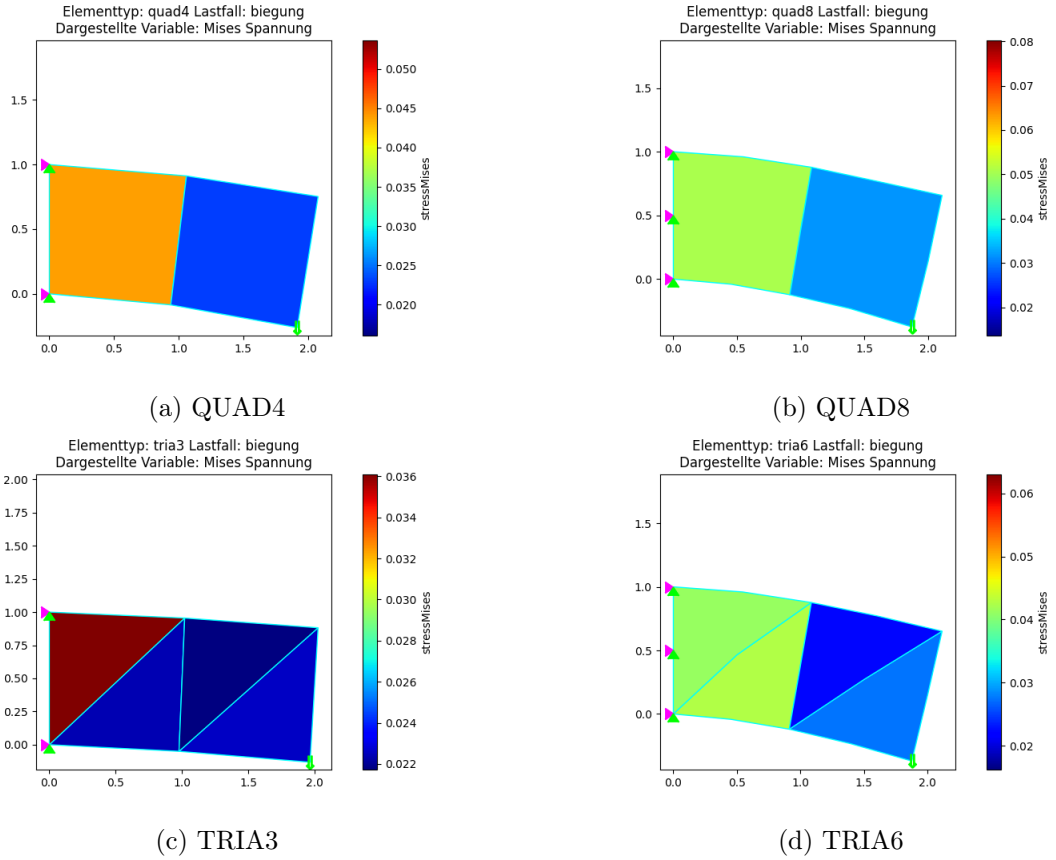


(h) TRIA6 `feamlearn.py`

Figure 4.3: Graphical comparison of the deformed mesh of both programs for the bending load case

Stress

The von Mises equivalent stresses of the `femlearn.py` model for the bending load case are shown in Figures 4.4 and 4.5.



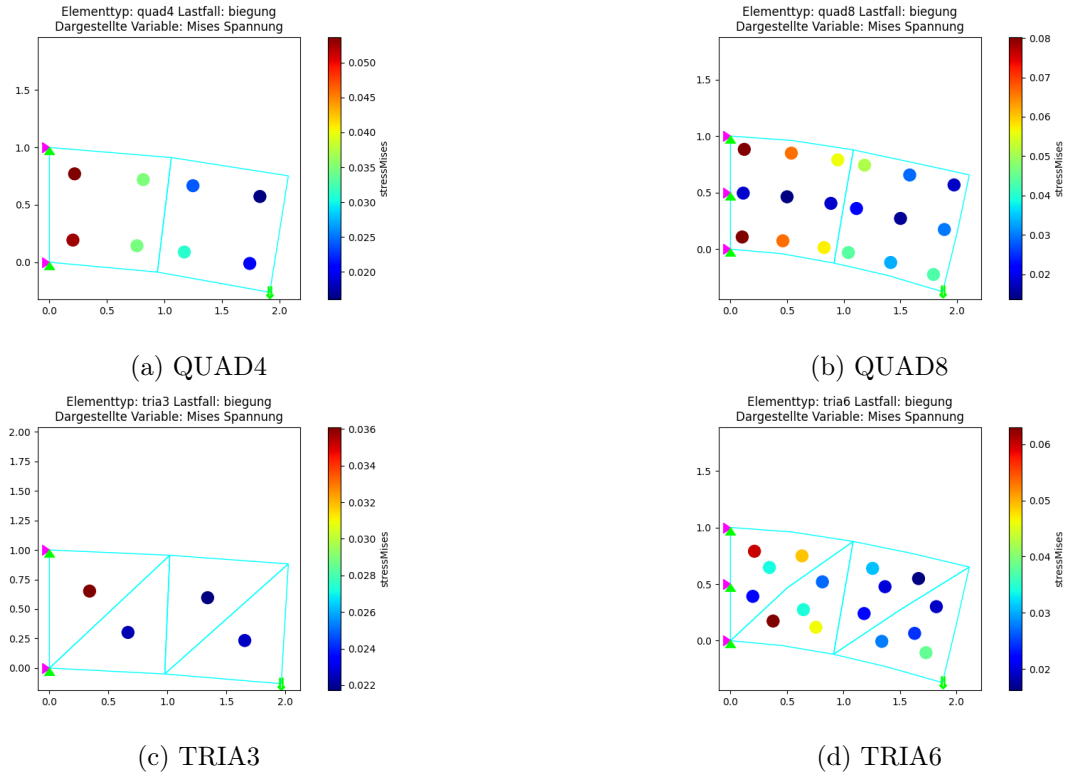
(a) QUAD4

(b) QUAD8

(c) TRIA3

(d) TRIA6

Figure 4.4: Averaged von Mises stresses for QUAD4, QUAD8, TRIA3, and TRIA6 elements in the bending load case



(a) QUAD4

(b) QUAD8

(c) TRIA3

(d) TRIA6

Figure 4.5: Unaveraged von Mises stresses at the integration points for QUAD4, QUAD8, TRIA3, and TRIA6 elements in the bending load case

4.3 Model for Line Load

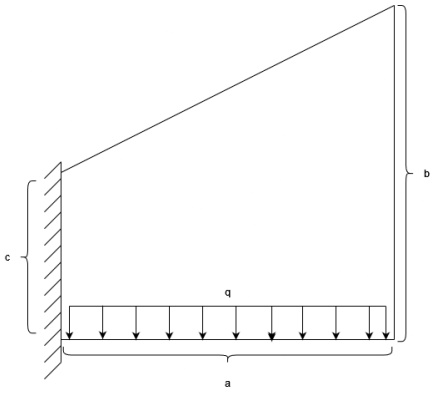


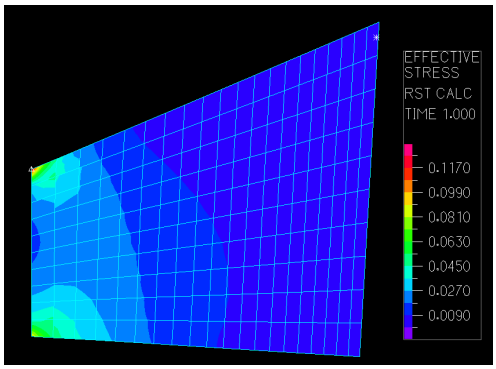
Figure 4.6: Geometry for the line load model

As an additional load case, a line load is calculated using a QUAD8 mesh and compared with ADINA. The geometry shown in Figure 4.6 also features inclined edges and varying element sizes, so that distorted elements are considered in this load case. The model is computed using a line load of $q = 0.005$ and dimensions $a = b = 2$ and $c = 1$ instead of a single force.

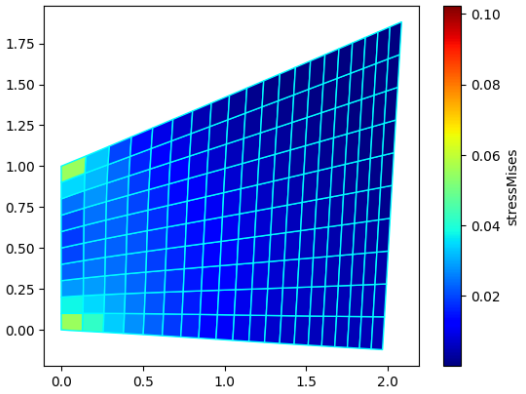
4.4 Results for Line Load

The results of the load case are summarized in Table 4.3 and visualized as a contour plot in Figure 4.7. Due to the high number of nodes, Table 4.3 does not list the values for all

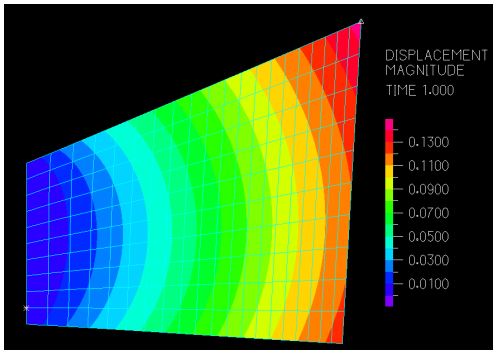
nodes for the sake of clarity. Here as well, the displacements and stresses match between `femlearn.py` and ADINA for both the plane stress and plane strain conditions.



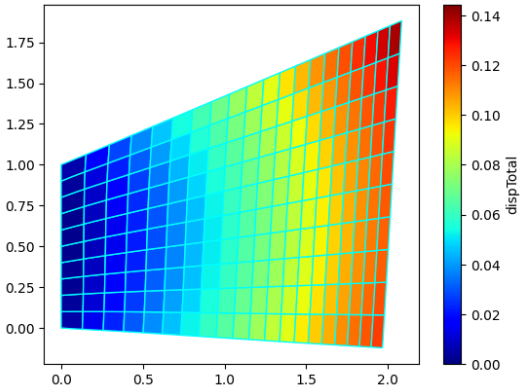
(a) Mises stress in ADINA



(b) Averaged Mises stress in `femlearn.py`



(c) Total displacement in ADINA



(d) Total displacement in `femlearn.py`

Figure 4.7: Graphical comparison of displacements and stresses from both programs under line load and plane stress condition

Variable	Plane Stress		Plane Strain	
	ADINA	`femlearn.py`	**ADINA**	`femlearn.py`
Max. Mises	0.102437	0.10243720151	0.0746501	0.074650115040
Min. Mises	7.81239E-07	7.811743384E-07	7.45169E-07	7.450848876E-07
Max. Displacement X	0.0833374	0.08333737427	0.0765012	0.076501149244
Max. Displacement Y	0	0	0	0
Min. Displacement X	-0.0338701	-0.03387011922	-0.0316896	-0.031689632543
Min. Displacement Y	-0.121512	-0.12151157152	-0.113299	-0.113298864467

Table 4.3: Comparison of stresses and displacements from both programs under line load

4.5 Summary

The comparison between the implemented FEM library and the commercial software ADINA shows that the results are consistent and can be applied in academic contexts. However, the Python tool should still be used with caution as undetected errors and bugs might exist. Due to its openly accessible and modifiable code, it is particularly suitable for further development and investigation of research questions in the field of FEM.

5 Appendix

Tension Load Case

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	0.0192614	0.00353814	0.019261440532	0.003538135107
Node 3	0.0393683	0.00281798	0.039368276178	0.002817983714
Node 4	0	0	0	0
Node 5	0.0192614	-0.00353814	0.019261440532	-0.003538135107
Node 6	0.0393683	-0.00281798	0.039368276178	-0.002817983714

Table 5.1: Comparison of displacements for element type QUAD4 and load case tension

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	0.0169696	0.00194342	0.016969618159	0.001943423820
Node 3	0.0575319	0.0157879	0.057531929910	0.015787916843
Node 4	0	0	0	0
Node 5	0.0169696	-0.00194342	0.016969618159	-0.001943423820
Node 6	0.0575319	-0.0157879	0.057531929910	-0.015787916843
Node 7	0.00833905	0.00336628	0.008339052293	0.003366275085
Node 8	0.0362124	-0.00034496	0.036212437058	-0.000344959750
Node 9	0.00833905	-0.00336628	0.008339052293	-0.003366275085
Node 10	0.0362124	0.00034496	0.036212437058	0.000344959750
Node 11	0	0	0	0
Node 12	0.0211265	-3.7858E-17	0.021126469544	0
Node 13	0.030598	4.57117E-18	0.030598038919	0

Table 5.2: Comparison of displacements for element type QUAD8 and load case tension

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	0.0204678	0.00429914	0.020467798092	0.004299143302
Node 3	0.0404806	0.00722941	0.040480596588	0.007229407223
Node 4	0	0	0	0
Node 5	0.0176734	-0.0015047	0.017673354945	-0.001504700156
Node 6	0.0375979	0.00124221	0.037597869938	0.001242205719

Table 5.3: Comparison of displacements for element type TRIA3 and load case tension

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	0.0197885	0.001654	0.019788516681	0.001653998695
Node 3	0.0519804	0.00987778	0.051980439416	0.009877775147
Node 4	0	0	0	0
Node 5	0.0202962	-0.00186785	0.020296153057	-0.001867853144
Node 6	0.0518756	-0.00710822	0.051875635463	-0.007108220714
Node 7	0.0102974	0.00262414	0.010297434529	0.002624139562
Node 8	0.0323284	0.00191567	0.032328366352	0.001915666443
Node 9	0.00957659	-0.00227286	0.009576591700	-0.002272863435
Node 10	0.0343938	-0.00317564	0.034393834810	-0.003175643675
Node 11	0	0	0	0
Node 12	0.0190121	0.000647397	0.019012072352	0.000647397347
Node 13	0.0334347	-0.00226463	0.033434702680	-0.002264631438
Node 14	0.00932605	0.000221506	0.009326051608	0.000221506092
Node 15	0.0278377	0.000307186	0.027837662288	0.000307186484

Table 5.4: Comparison of displacements for element type TRIA6 and load case tension

Bending Load Case

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	-0.0603066	-0.0854664	-0.060306590970	-0.085466414344
Node 3	-0.0822979	-0.261319	-0.082297880746	-0.261319366390
Node 4	0	0	0	0
Node 5	0.0610267	-0.0878669	0.061026742363	-0.087866918989
Node 6	0.0794799	-0.247125	0.079479897032	-0.247125078054

Table 5.5: Comparison of displacements for element type QUAD4 and load case bending

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	-0.0221615	-0.0498985	-0.022161469295	-0.049898519991
Node 3	-0.0339894	-0.132295	-0.033989449192	-0.132294793626
Node 4	0	0	0	0
Node 5	0.0206726	-0.0449355	0.020672568699	-0.044935518003
Node 6	0.02676	-0.118123	0.026760041969	-0.118122773523

Table 5.7: Comparison of displacements for element type TRIA3 and load case bending

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	-0.0849758	-0.123022	-0.084975830000	-0.123021950000
Node 3	-0.126047	-0.376174	-0.126046830000	-0.376174350000
Node 4	0	0	0	0
Node 5	0.085858	-0.122445	0.085858030000	-0.122445420000
Node 6	0.110259	-0.342887	0.110258910000	-0.342886620000
Node 7	-0.0480231	-0.0401138	-0.048023080000	-0.040113750000
Node 8	-0.111037	-0.232255	-0.111036510000	-0.232254960000
Node 9	0.0485074	-0.0399115	0.048507430000	-0.039911470000
Node 10	0.106037	-0.233641	0.106036990000	-0.233641240000
Node 11	0	0	0	0
Node 12	-0.000324478	-0.116872	-0.000324480000	-0.116872140000
Node 13	0.00171912	-0.356332	0.001719120000	-0.356332140000

Table 5.6: Comparison of displacements for element type QUAD8 and load case bending

ADINA			femlearn.py	
Node	**Y**	**Z**	**dispX**	**dispY**
Node 1	0	0	0	0
Node 2	-0.0877439	-0.119734	-0.087743893193	-0.119734409633
Node 3	-0.124023	-0.370241	-0.124023102093	-0.370240564941
Node 4	0	0	0	0
Node 5	0.0860126	-0.123402	0.086012635078	-0.123401999050
Node 6	0.114145	-0.349072	0.114145326946	-0.349072490570
Node 7	-0.0489718	-0.0418749	-0.048971789506	-0.041874879800
Node 8	-0.107671	-0.232997	-0.107671021303	-0.232996513988
Node 9	0.0466973	-0.0403072	0.046697308699	-0.040307223126
Node 10	0.104799	-0.229132	0.104798597888	-0.229132047808
Node 11	0	0	0	0
Node 12	0.000237587	-0.117324	0.000237586733	-0.117324128102
Node 13	-0.00019766	-0.352731	-0.000197660235	-0.352730824913
Node 14	0.00054815	-0.034916	0.000548150035	-0.034915982691
Node 15	0.000416591	-0.228473	0.000416591425	-0.228472586466

Table 5.8: Comparison of displacements for element type TRIA6 and load case bending