Compas_fea2_opensees Development

By

Moosa Saboor,

Prof. Aryan Rezaei Rad

Department of Civil and Mineral Engineering
University of Toronto

Chapter 1
Overview of compas_fea2_opensees

The compas_fea2_opensees project is a Python-based interface bridging the COMPAS framework with the OpenSees finite element solver. The development of this plugin is ongoing, with the aim of extending COMPAS' parametric and geometry-aware capabilities into structural analysis through OpenSees' finite element analysis (FEA) tools. This document outlines my development experience working with compas_fea2_opensees, the core ideas behind its architecture, and instructions for effective use, as well as instruction and information for individuals who may seek to add onto the development.

The plugin allows users to define structural models and analysis steps in a structured, object-oriented way using Python. It translates these models into OpenSees-compatible formats and executes simulations via automatically generated Tcl scripts. The results are then post-processed and can be visualized or further analyzed within the COMPAS ecosystem using tools such as compas_veiwer.

The primary motivation behind this project is to advance the development of the compas_fea2_opensees backend, with the goal of providing engineers with a free, accessible, and modular tool for structural analysis. By supporting and extending this backend, the project contributes to a more complete and practical integration of OpenSees within the COMPAS ecosystem. In parallel, improving the documentation of compas_fea2 plays a key role in making the framework more approachable to new users, enabling engineers to take full advantage of the broader COMPAS suite—including tools like compas_viewer, compas_gmsh—to visualize and interact with their structural models in a cohesive workflow.

Chapter 2
Development cycle and Challenges

2.1 How COMPAS_FEA2 and the OpenSees backend function

Before delving into development, it is essential to first understand how the compas_fea2 framework operates and how it interfaces with OpenSees. At its core, compas_fea2 provides a structured Python-based framework for defining structural models. It includes the necessary tools to create elements such as beams, trusses, nodes, loads, and boundary conditions—essentially everything required to construct a finite element model.

While the framework includes functions related to analysis, these are not functional by default. If a backend is not specified in the user script, calling these functions will result in a NotImplementedError. To enable analysis, the user must explicitly assign a backend using the set_backend("[backend_name]") function, where the string corresponds to a supported solver (e.g., opensees, abaqus, calculix, etc.). The details of this scripting workflow are further discussed in Chapter 3.

Calling set_backend dynamically maps the high-level compas_fea2 functions to their corresponding implementations in the selected backend. As a result, switching between different solvers is as simple as changing the backend name in the set_backend command—provided the script only uses functionality supported by the selected backend.

The OpenSees backend in particular works by leveraging the OpenSees executable, which runs Tcl-based input scripts. Under the hood, the backend constructs the Tcl script by concatenating Python strings derived from the model definitions in the user's script. These strings are compiled into a single Tcl file, which is then passed to OpenSees.exe for execution. The analysis output—such as nodal displacements and element deformations—is written into data files. These results can then be visualized using tools like compas_viewer, allowing users to review the structural behavior graphically. Examples and explanations on how to develop more functions into compas_fea2 and how to actually use compas to model and analyze a structure will be explored in Chapter 3.

A summary of the code execution path can be seen through the flowchart in figure 1.

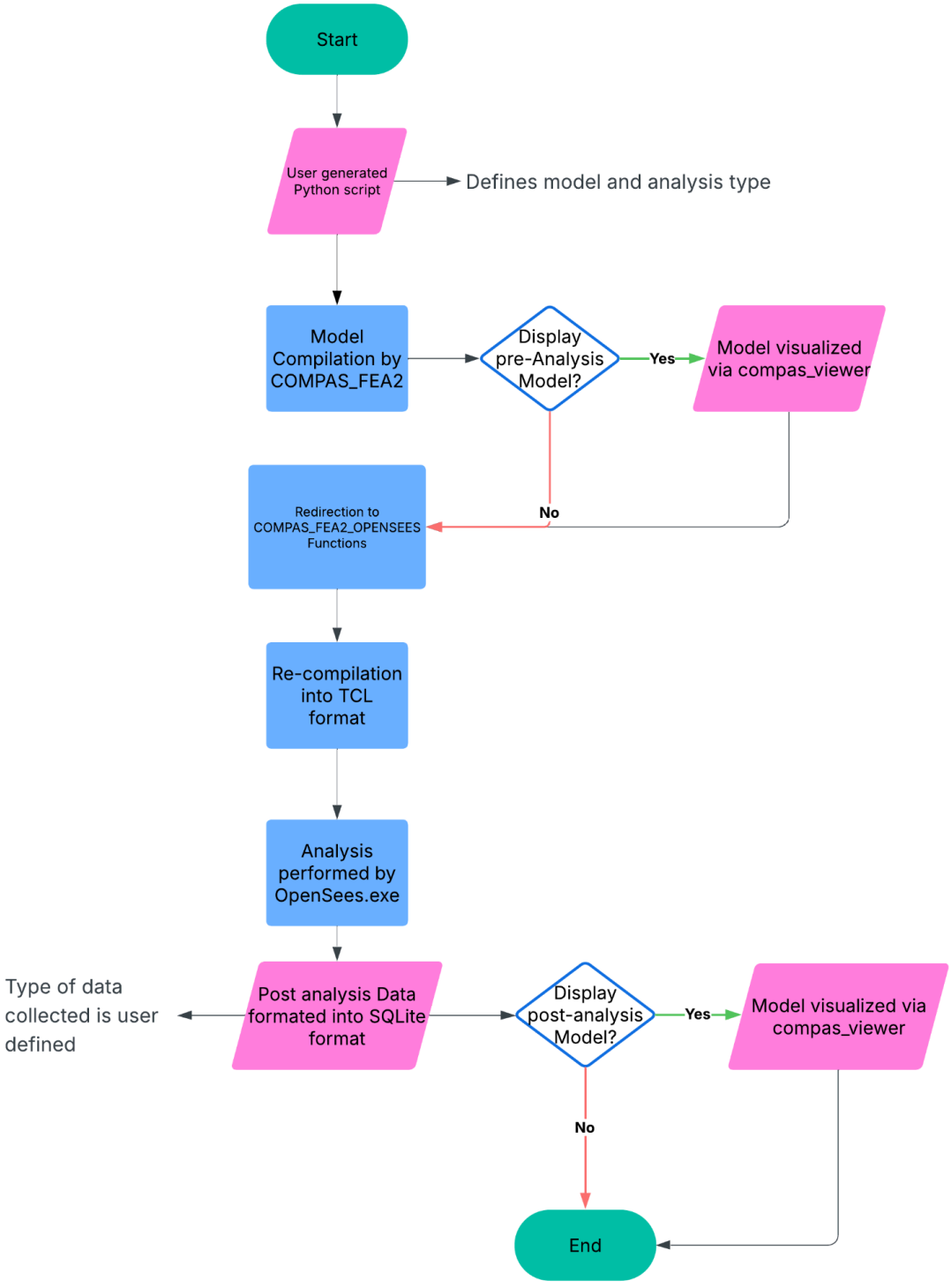

Figure 1: Simplified summary of execution process for the average compas_fea2_opensees script

2.2 My development process

Development began on May 5th, 2025, with the initial objective of gaining a comprehensive understanding of the compas_fea2 framework and how the OpenSees backend integrates into its development pipeline. To establish this foundation, I started by rewriting official OpenSees examples—specifically the basic truss and portal frame models [1]—in the current Compas_fea2-compatible format. These particular examples were selected because they had previously been implemented within compas_fea2, albeit for an older version of the framework that was already outdated and no longer functional.

Once familiar with the structure and logic of compas_fea2, I was provided with a prioritized list of OpenSees functionalities to implement, document, and ensure usability within the project timeline. This list included: zeroLength elements, full support for truss and beam elements, twoNodeLink elements, shell elements, and the geomTransf transformation commands. Among these, I identified that zeroLength elements, beam-related commands (such as beamWithHinges and sectional definitions), geomTransf commands, and twoNodeLink functionality were either entirely unimplemented or nonfunctional (as was the case with the sectional commands).

On May 29th, 2025, a significant update was pushed to both the compas_fea2 and compas_fea2_opensees repositories. This update consolidated approximately six months' worth of development that had not yet been merged into the main branches. It is important to note that the updated repositories were not used at the outset of the project, as development was occurring independently and the primary compas_fea2 repository had not been receiving updates on a regular basis.

The integration of this update introduced a number of unforeseen challenges. Notably, scripts that had previously functioned correctly—such as the basic truss and portal frame examples—no longer executed as expected. In a follow-up meeting with Inès Champagne, one of the original developers from ETH Zurich, we determined that many of the issues stemmed from fundamental changes in the order of operations within the framework introduced by the update.

For example, in earlier versions, nodal elements were assigned a unique key identifier at the time of their creation. Post-update, however, nodes only received these keys during the analysis phase. This change posed a problem for legacy scripts that relied on the keys to assign constraints and establish connectivity. Two solutions were implemented to address this issue: the first involved identifying nodes based on their coordinate data using an existing COMPAS utility; the second involved storing node objects in a dictionary, allowing them to be referenced by user-defined keys.

Another major issue arose with the geomTransf functionality, which is essential in OpenSees for defining the geometric transformation type required for structural analysis (e.g., linear, corotational, or P-Delta). One of the parameters of this function requires specifying an orthogonal direction—used to define the “up” direction for 3D models. However, in 2D models, this parameter is not only unnecessary but can also cause the analysis to fail. To resolve this, a conditional check was introduced to determine the dimensionality of the model before assigning the transformation type, ensuring correct behavior in both 2D and 3D contexts (Figure 2).

```
 def jobdata(self):
        return "\n".join(
            [
                # f"geomTransf Linear {self.key}",
                "geomTransf Corotational {} {}".format(self.key, "
".join([str(i) for i in self.frame.zaxis])),
                self._job_data(),
            ]
        )
```

Figure 2a: Original problematic script

```
def jobdata(self):
        if self.part.ndm == 2:
            return "\n".join(
                [
                    f"geomTransf Corotational {self.key}", #2D
                    self._job_data(),
                ])
        elif self.part.ndm == 3:
            return "\n".join(
                [
                    "geomTransf Corotational {} {}".format(self.key,
" ".join([str(i) for i in self.frame.zaxis])), #3D
                    self._job_data(),
                ])
        else:
            raise ValueError("Beam elements are only supported in 2D
and 3D models, not {}".format(self.part.ndm))
```

Figure 2b: Fixed conditional script

Another issue presented itself in the member section declaration scripts, which are essentially the functions responsible for defining a beam's shape and calculating values such as the second moments of inertia, area, volume etc. Essentially, for these functions, some attributes (i.e; shape, material, area) were lost between the process in which compas_fea2 passes the function off to the backend counterpart. In order to remedy this, each shape-dependent function was explicitly passed through the main OpenseesBeamSection function (of which these functions were derivative) in order to ensure proper transfer of data (Figure 3).

```
class OpenseesBoxSection(BoxSection):
    """"""

    __doc__ += BoxSection.__doc__

    def __init__(self, w, h, t, material, **kwargs):
        super(OpenseesBoxSection, self).__init__(self, w, h, t, material,
**kwargs)
        self.shape = Rectangle(w=self.w, h=self.h)
        OpenseesBeamSection(name=self.name, A=self.A, Ixx=self.Ixx,
Iyy=self.Iyy, Ixy=self.Ixy, Avx=self.Avx, Avy=self.Avy, J=self.J, g0=0, gw=0,
material=material, shape=self.shape)

    def jobdata(self):
        return beam_jobdata(self)


class OpenseesCircularSection(CircularSection):
    """"""

    __doc__ += CircularSection.__doc__

    def __init__(self, r, material, **kwargs):
        super(OpenseesCircularSection, self).__init__(r, material, **kwargs)
        self.r = r
        self.shape = Circle(radius=self.r)
        OpenseesBeamSection(name=self.name, A=self.A, Ixx=self.Ixx,
Iyy=self.Iyy, Ixy=self.Ixy, Avx=self.Avx, Avy=self.Avy, J=self.J, g0=0, gw=0,
material=material, shape=self.shape)

    def jobdata(self):
        return beam_jobdata(self)
```

Figure 3: New implementation of shaped sections; each section (i.e OpenSeesCircularSection) is a derivative of the main OpenseesBeamSection.

Once the plugin was fully functional, I implemented functions for zeroLengthMember under the name OpenseesSpringElement, twoNodeLink under the name OpenseesLinkElement and equalDOF under the name OpenseesTieConstraint. I also added in the beamWithHinges command under the same name. The implementation of these functions, and how development of additional functions into compas_fea2_opensees is done, will be discussed in Chapter 3. Once these functions were added, I began work on test scripts to showcase functionality and aid documentation.

Chapter 3
Example scripts and developed functions

3.1 Initial Challenges

To ensure I had reference results for comparison—thereby validating the accuracy of my own calculations—I chose to recreate a set of test scripts originally written for the compas_fea framework. These scripts, developed by Andrew Liew, were designed to be executed within Rhino and are publicly available in the main compas_fea webpage [2].

However, it's important to note that substantial architectural and functional changes have occurred since the original compas_fea, and as a result, the usefulness of these examples was limited beyond providing general structural concepts. At the time of development, comprehensive documentation for compas_fea2—particularly regarding backend integration and script authoring—was lacking. Consequently, the scripts used in this project were developed independently, based on a detailed examination of how commands propagate through the compas_fea2 and compas_fea2_opensees codebases.

In total, six of Andrew Liew's example scripts were reimplemented within the OpenSees backend: beam_bathe, beam_frame, beam_simple, mesh_plate, truss_frame, and truss_tower. These constituted the only viable cases for adaptation, as the remaining examples were tailored specifically for the Abaqus backend and relied on functions that were either incompatible with or unavailable in OpenSees. To reconstruct the selected models, the corresponding 3dm files were obtained from the official compas_fea repository [2]. Using IronPython within Rhino3D, the nodal coordinates and element data were extracted and reformatted into Python dictionaries and lists for subsequent processing. Furthermore, I also developed two 2-Dimensional examples found on the official Opensees examples page [1] into compas_fea2 scripts; The Elastic Frame and Truss examples.

3.2 Truss_Tower.py

The goal of this script is to create a steel truss tower which will support a load applied directly to its topmost node.

```
from compas_fea2.model import Model, Part, Node, TrussElement,
ElasticIsotropic, TrussSection
from compas_fea2.problem import Problem, StaticStep, LoadCombination
from compas_fea2.results import DisplacementFieldResults
from compas_fea2_opensees import TEMP
import compas_fea2
import os
# Set the backend
compas_fea2.set_backend("compas_fea2_opensees")
mdl = Model(name="TrussTower")
prt = mdl.add_part(Part(name="Truss-3"))
prt.ndm = 3
prt.ndf = 3

# Materials
steel = ElasticIsotropic(name="Steel", E=200000000000, v=0.3, density=7850)

# Sections
mainTruss = TrussSection(name='sec_main', A=0.0001, material=steel)
```

Firstly it is necessary to import all the functions necessary in order to write this script, and to set the backend to OpenSees. Model and part names are optional modifiers but can help track down files where issues may be occurring. prt.ndm describes which dimension this structure will be in, this particular structure being 3-dimensional, the value is 3. For a 2-dimensional structure it would be 2. Prt.ndf describes the degrees of freedom allowed for the structure. It is important to note that in OpenSees, a truss structure only has 3 degrees of freedom. The steel variable holds the material which will be used for this structure, providing properties such as the young's modulus, poisson's ratio and density. While compas_fea2 does come with a built-in library to define units, it is not explicitly necessary, but one must be careful to keep all their units the same if they do not want to use it. MainTruss is created using the TrussSection command, which is circular by default.

```
nodes_data ={0: (0.250000, 0.250000, 2.500000),
  1: (0.750000, 0.250000, 2.500000),
  2: (0.750000, 0.750000, 2.500000),
```

```
  3: (0.250000, 0.750000, 2.500000),
  4: (0.375000, 0.375000, 3.750000),
  5: (0.625000, 0.375000, 3.750000),
  6: (0.625000, 0.625000, 3.750000),
  7: (0.375000, 0.625000, 3.750000),
  8: (0.500000, 0.500000, 5.000000),
  9: (0.000000, 0.000000, 0.000000),
  10: (0.125000, 0.125000, 1.250000),
  11: (0.000000, 1.000000, 0.000000),
  12: (0.125000, 0.875000, 1.250000),
  13: (1.000000, 0.000000, 0.000000),
  14: (0.875000, 0.125000, 1.250000),
  15: (1.000000, 1.000000, 0.000000),
  16: (0.875000, 0.875000, 1.250000),
  17: (0.125000, 0.500000, 1.250000),
  18: (0.500000, 0.125000, 1.250000),
  19: (0.875000, 0.500000, 1.250000),
  20: (0.500000, 0.875000, 1.250000),}

lines = [(0, 1), (1, 2), (2, 3), (3, 0), (4, 5), (5, 6), (6, 7), (7, 4), (0,
4), (4, 8), (3, 7), (7, 8), (1, 5), (5, 8), (2, 6), (6, 8), (9, 10), (10, 0),
(11, 12), (12, 3), (13, 14), (14, 1), (15, 16), (16, 2), (11, 17), (17, 9), (9,
18), (18, 13), (19, 13), (19, 15), (20, 15), (20, 11), (10, 18), (18, 14), (14,
19), (19, 16), (12, 20), (20, 16), (12, 17), (17, 10), (19, 2), (19, 1), (2,
20), (20, 3), (0, 17), (3, 17), (0, 18), (1, 18), (2, 0), (7, 5), (7, 0), (7,
2), (5, 0), (5, 2), (17, 18), (18, 19), (19, 20), (20, 17), (20, 18)]
```

These are the nodes and lines, the nodes being a dictionary assigning a key and coordinate and the lines being a list of tuples consisting of the IDs of both nodes the tuple connects to. These values were pulled from a .3dm file associated with the model Andrew Liew created.

```
constrained_nodes_coordinates = [(0.000000, 0.000000, 0.000000), (0.000000,
1.000000, 0.000000), (1.000000, 1.000000, 0.000000), (1.000000, 0.000000,
0.000000),]
loaded_nodes_coordinates = [(0.500000, 0.500000, 5.000000)]

nodes = {}
for nid, xyz in nodes_data.items():
    nodes[nid] = prt.add_node(Node(name=nid, xyz=xyz))

for node_ids in lines:
    n1 = nodes[node_ids[0]]
    n2 = nodes[node_ids[1]]
```

```
    prt.add_element(TrussElement(nodes=[n1, n2], section=mainTruss))
for coords in constrained_nodes_coordinates:
    n = prt.find_closest_nodes_to_point(coords, single=True)
    mdl.add_pin_bc(n)
```

Here the nodes are actually added to the model by storing nodal elements in a dictionary and assigning them a key. When adding constraints to a node, it is possible to again use a key from the dictionary, however it was just more convenient in my case to use coordinates and the find_closest_nodes_to_point command due to the way I extracted data from the .3dm file. Generally, this function stores a cluster of nodes, which you cannot easily work with, which is why the parameter “single” is set to true, to ensure only a singular nodal element is stored within n.

```
prb = mdl.add_problem(Problem(name="TrussTowerAnalysis"))
stp = prb.add_step(StaticStep(name="StaticStep"))

stp.combination = LoadCombination.SLS()
for coords in loaded_nodes_coordinates:
    n = prt.find_closest_nodes_to_point(coords, single=True)
    stp.add_uniform_node_load(nodes=n, load_case="LL", x=2000.0, y=-1000,
z=-100000, xx=0.0, yy=0.0, zz=0.0)

stp.add_outputs([DisplacementFieldResults])
prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
Verbose=True)

stp.show_deformed(scale_results=10, show_original=0.1, show_bcs=0.0003,
show_loads=0.000005)
```

The problem (which needs to be analyzed) is now created. The Add_step command determines what kind of analysis is to be performed on the model. In this case we are doing a singular static loading test, however, OpenSees has the capacity to do much more, including Modal analysis and buckling tests. For this model we are applying a force to a node, but it should be mentioned that using the add_uniform_point_load command, you can also apply loads away from nodes. We can also add many different types of outputs. For this model, we are only obtaining the DisplacementFieldResults but we can also obtain stress and strain results. The final deformation of the structure, which is printed by the show_deformed command by compas_viewer, can be seen in Figure 4.

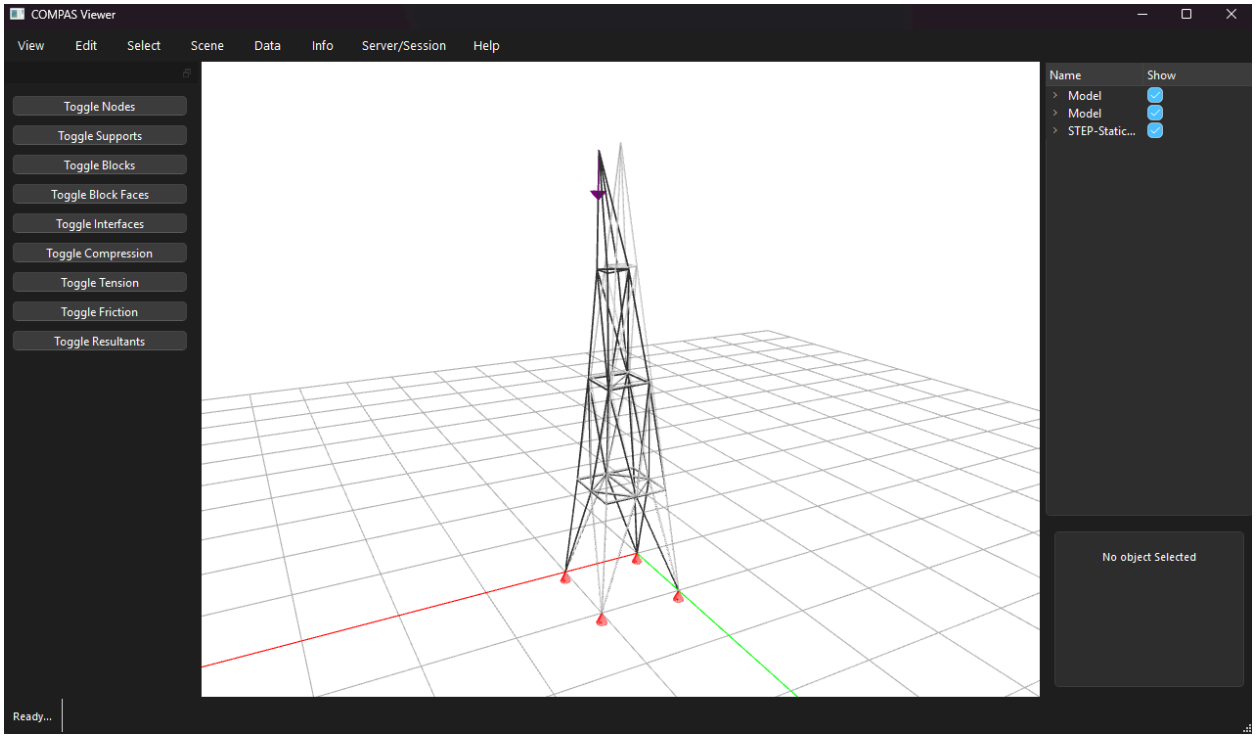

Figure 4: Light gray represents undeformed structure while dark represents deformed.

3.3 plate_mesh.py

A shell or plate element represents a two-dimensional structural component, in contrast to one-dimensional elements like trusses or beams. To obtain meaningful results from an analysis, the shell must first be discretized into finite elements. Manually performing this discretization can be tedious and error-prone, but the process is significantly simplified using the shell_from_compas_mesh command, which automates mesh-based shell generation from geometric input.

```
import os
from compas.datastructures import Mesh

import compas_fea2
from compas_fea2.model import Model, Part
from compas_fea2.model import ElasticIsotropic, ShellSection
from compas_fea2.problem import LoadCombination
from compas_fea2.results import DisplacementFieldResults
from compas_fea2_opensees import TEMP
compas_fea2.set_backend("compas_fea2_opensees")
lx = 5
ly = 1
```

```
nx = 50
ny = 10
plate = Mesh.from_meshgrid(lx, nx, ly, ny)
thk = 0.02
mdl = Model(name="mesh_plate")
mat = ElasticIsotropic(name='mat_elastic', E=75*10**9, v=0.3, density=2700)
sec = ShellSection(t=thk, material=mat)
prt = Part.shell_from_compas_mesh(mesh=plate, section=sec, name="shell")
mdl.add_part(prt)
```

In this snippet, the backend is first established and then a compas_mesh is created, lx and ly describing length values while nx and ny describe number of squares in the x and y directions. Then the ShellSection is declared by providing a thickness and material, then a shell element is created by utilizing the previously mentioned shell_from_compas_mesh command.

```
# Set boundary conditions at both ends of the shell
fixed_coords = [
  (0.200000, 0.000000, 0.000000),
  (0.100000, 0.000000, 0.000000),
  (0.000000, 0.000000, 0.000000),
  (0.000000, 0.100000, 0.000000),
  (0.000000, 0.200000, 0.000000),
  (0.000059, 0.800007, 0.000000),
  (0.000068, 0.900007, 0.000000),
  (0.000077, 1.000007, 0.000000),
  (0.100077, 0.999998, 0.000000),
  (0.200077, 0.999989, 0.000000),
  (4.800000, 0.000000, 0.000000),
  (4.900000, 0.000000, 0.000000),
  (5.000000, 0.200000, 0.000000),
  (5.000000, 0.100000, 0.000000),
  (5.000000, 0.000000, 0.000000),
  (4.999990, 1.000005, 0.000000),
  (5.000000, 0.900006, 0.000000),
  (5.000009, 0.800006, 0.000000),
  (4.899991, 0.999996, 0.000000),
  (4.799991, 0.999987, 0.000000)
]

fixed_nodes = []
for coord in fixed_coords:
    nid = prt.find_closest_nodes_to_point(coord, single=True)
    fixed_nodes.append(nid)
```

```
mdl.add_fix_bc(nodes=fixed_nodes)
```

We use a list of coordinates which correlate with corner nodes which we must apply a fixed boundary condition to. This is more convenient to do with coordinates as opposed to a dictionary of nodes as we did not manually create nodes this time; they were automatically created by the shell_from_compas_mesh function.

```
prb = mdl.add_problem(name="mid_load")
stp = prb.add_static_step()
stp.combination = LoadCombination.SLS()

# Add a load in the middle of the grid
loaded_node_coords = [
  (3.000000, 1.000000, 0.000000),
  (2.900000, 1.000000, 0.000000),
  (2.800000, 1.000000, 0.000000),
  (2.700000, 1.000000, 0.000000),
  (2.600000, 1.000000, 0.000000),
  (2.500000, 1.000000, 0.000000),
  (2.400000, 1.000000, 0.000000),
  (2.300000, 1.000000, 0.000000),
  (2.200000, 1.000000, 0.000000),
  (2.100000, 1.000000, 0.000000),
  (2.000000, 1.000000, 0.000000),
  (2.000048, 0.899841, 0.000000),
  (2.100048, 0.899841, 0.000000),
  (2.200048, 0.899841, 0.000000),
  (2.300048, 0.899841, 0.000000),
  (2.400048, 0.899841, 0.000000),
  (2.500048, 0.899841, 0.000000),
  (2.600048, 0.899841, 0.000000),
  (2.700048, 0.899841, 0.000000),
  (2.800048, 0.899841, 0.000000),
  (2.900048, 0.899841, 0.000000),
  (3.000048, 0.899841, 0.000000),
  (2.000044, 0.799841, 0.000000),
  (2.100044, 0.799841, 0.000000),
  (2.200044, 0.799841, 0.000000),
  (2.300044, 0.799841, 0.000000),
  (2.400044, 0.799841, 0.000000),
  (2.500044, 0.799841, 0.000000),
  (2.600044, 0.799841, 0.000000),
  (2.700044, 0.799841, 0.000000),
  (2.800044, 0.799841, 0.000000),
```

```
  (2.900044, 0.799841, 0.000000),
  (3.000044, 0.799841, 0.000000),
  (2.000041, 0.699841, 0.000000),
  (2.100040, 0.699841, 0.000000),
  (2.200041, 0.699841, 0.000000),
  (2.300040, 0.699841, 0.000000),
  (2.400041, 0.699841, 0.000000),
  (2.500041, 0.699841, 0.000000),
  (2.600040, 0.699841, 0.000000),
  (2.700041, 0.699841, 0.000000),
  (2.800040, 0.699841, 0.000000),
  (2.900041, 0.699841, 0.000000),
  (3.000041, 0.699841, 0.000000),
  (2.000037, 0.599841, 0.000000),
  (2.100037, 0.599841, 0.000000),
  (2.200037, 0.599841, 0.000000),
  (2.300037, 0.599841, 0.000000),
  (2.400037, 0.599841, 0.000000),
  (2.500037, 0.599841, 0.000000),
  (2.600037, 0.599841, 0.000000),
  (2.700037, 0.599841, 0.000000),
  (2.800037, 0.599841, 0.000000),
  (2.900037, 0.599841, 0.000000),
  (3.000037, 0.599841, 0.000000),
  (2.000034, 0.499841, 0.000000),
  (2.100034, 0.499841, 0.000000),
  (2.200034, 0.499841, 0.000000),
  (2.300034, 0.499841, 0.000000),
  (2.400034, 0.499841, 0.000000),
  (2.500034, 0.499841, 0.000000),
  (2.600034, 0.499841, 0.000000),
  (2.700034, 0.499841, 0.000000),
  (2.800034, 0.499841, 0.000000),
  (2.900034, 0.499841, 0.000000),
  (3.000034, 0.499841, 0.000000)
]
loaded_nodes = []
for coord in loaded_node_coords:
    nid = prt.find_closest_nodes_to_point(coord, single=True)
    loaded_nodes.append(nid)

stp.add_uniform_node_load(nodes=loaded_nodes, load_case="LL", x=0.0, y=-300,
z=-100.0, xx=0.0, yy=0.0, zz=0.0)

# Define field outputs
```

```
stp.add_output(DisplacementFieldResults)
prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
Verbose=True)
# Show deformed shape
stp.show_deformed(scale_results=1, show_original=0.2, show_bcs=0.00003,
show_loads=0.05)
```

Similarly, we now gather a list of coordinates corresponding to nodes which are experiencing a force/load, and then perform a static step analysis on the model as done previously in 3.3. The final deformed shape can then be visualized (Figure 5)

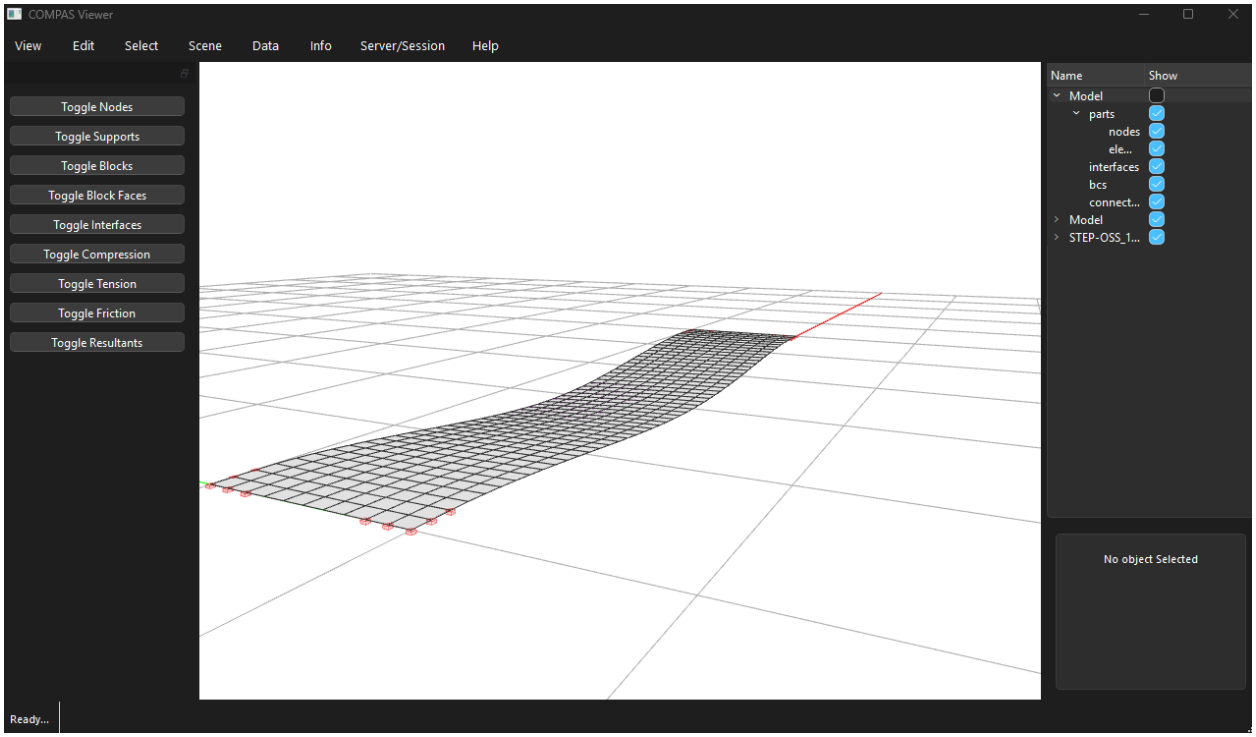

Figure 5: Deformed shell model visualized through compas_viewer

3.4 Beam_frame.py

This beam frame structure was discretized manually via a Rhino3D script, and contains 90 nodal elements and 120 beam elements, for this reason the dictionaries containing all of their coordinate data will be truncated significantly for the sake of brevity.

```
from compas_fea2.model import Model, Part, Node, BeamElement, ElasticIsotropic,
BeamSection, RectangularSection, PipeSection
from compas_fea2.model.shapes import Circle
from compas_fea2.problem import Problem, StaticStep, LoadCombination
```

```
from compas_fea2.results import DisplacementFieldResults
from compas_fea2_opensees import TEMP
import compas_fea2
import os
compas_fea2.set_backend("compas_fea2_opensees")
mdl = Model(name="Beam_Frame")
prt = mdl.add_part(Part(name="Beam_Frame_analysis"))

prt.ndm = 3
prt.ndf = 6
steel = ElasticIsotropic(name="Steel", E=200000000000, v=0.3, density=7850)
section_beam1 = PipeSection(r=0.1, t=0.005, material=steel)
Nodes_data = {}
elements_data = {}
```

We first establish the backend and prepare the model space. Unlike the Truss model in 3.2, this beam frame model has 6 degrees of freedom, therefore prt.ndf is set to 6. As previously mentioned, the dictionaries containing nodal and beam element data have been truncated.

```
nodes = {}
for nid, xyz in nodes_data.items():
    nodes[nid] = prt.add_node(Node(name=nid, xyz=xyz))

for eid, nodes in elements_data.items():
    n1 = prt.find_closest_nodes_to_point(nodes_data[nodes[0]], single = True)
    n2 = prt.find_closest_nodes_to_point(nodes_data[nodes[1]], single = True)
    prt.add_element(BeamElement(nodes=[n1, n2], section=section_beam1,
frame=[1,1,1]))

pinned = prt.find_closest_nodes_to_point((0.0, 0.0, 0.0), single = True)
roller = prt.find_closest_nodes_to_point((4.0, 0.0, 0.0), single = True)

h_loaded_node = prt.find_closest_nodes_to_point((0.0, 0.0, 2.0), single = True)
v_loaded_node = prt.find_closest_nodes_to_point((2.0, 0.0, 2.0), single = True)

mdl.add_pin_bc(pinned)
mdl.add_pin_bc(roller)

#mdl.show()

prb = mdl.add_problem(Problem(name="BeamFrameAnalysis"))
stp = prb.add_step(StaticStep(name="StaticStep"))
```

```
stp.combination = LoadCombination.SLS()
stp.add_uniform_node_load(nodes=h_loaded_node, load_case="LL", x=4000.0, y=0.0,
z=0.0, xx=0.0, yy=0.0, zz=0.0)
stp.add_uniform_node_load(nodes=v_loaded_node, load_case="LL", x=0.0, y=0.0,
z=-6000, xx=0.0, yy=0.0, zz=0.0)

stp.add_outputs([DisplacementFieldResults])
prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
Verbose=True)

stp.show_deformed(scale_results=50, show_original=0.1, show_bcs=0.0003,
show_loads=0.0001)
```

The rest of the analysis is performed identically to 3.2, deformed structure can be seen in Figure 6

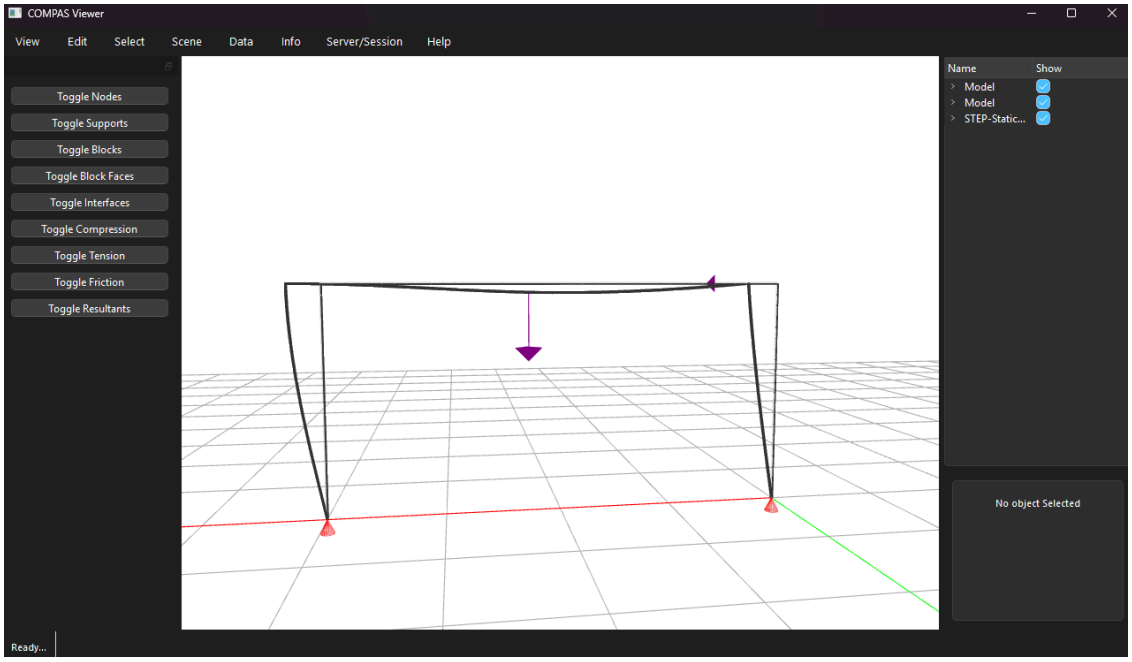

Figure 6: Discretized beam frame deformation seen through compas_viewer. Lighter shade represents undeformed structure.

3.5 Beam_bathe.py

Beam_bathe depicts a curved cantilever beam with a point load attached to its free end. Functionally it is nearly identical to Beam_frame.py from section 3.4.

```
from compas_fea2.model import Model, Part, Node, BeamElement, ElasticIsotropic,
BeamSection, RectangularSection, PipeSection
```

```
from compas_fea2.problem import Problem, StaticStep, LoadCombination
from compas_fea2.results import DisplacementFieldResults, ReactionFieldResults,
StressFieldResults
from compas_fea2_opensees import TEMP
import compas_fea2
import os
compas_fea2.set_backend("compas_fea2_opensees")
mdl = Model(name="Beam_bathe")
prt = mdl.add_part(Part(name="Beam_bathe_analysis"))

prt.ndm = 3
prt.ndf = 6
steel = ElasticIsotropic(name="Steel", E=100000000, v=0.3, density=7850)

section_beam1 = RectangularSection(w=1, h=1, material=steel)

nodes_data = {0: (70.710678, -29.289322, 0.000000),
  1: (0.000000, 0.000000, 0.000000),
  2: (9.801714, -0.481527, 0.000000),
  3: (19.509032, -1.921472, 0.000000),
  4: (29.028468, -4.305966, 0.000000),
  5: (38.268343, -7.612047, 0.000000),
  6: (47.139674, -11.807874, 0.000000),
  7: (55.557023, -16.853039, 0.000000),
  8: (63.439328, -22.698955, 0.000000),}

elements_data = [
  (1, 2),
  (2, 3),
  (3, 4),
  (4, 5),
  (5, 6),
  (6, 7),
  (7, 8),
  (8, 0),]

nodes = {}
for nid, xyz in nodes_data.items():
    nodes[nid] = prt.add_node(Node(name=nid, xyz=xyz))

for nodes_ids in elements_data:
    n1 = nodes[nodes_ids[0]]
    n2 = nodes[nodes_ids[1]]
    prt.add_element(BeamElement(nodes=[n1, n2], section=section_beam1,
frame=[1,1,1]))
```

```
support_node = prt.find_closest_nodes_to_point((0.0, 0.0, 0.0), single=True)
loaded_node = prt.find_closest_nodes_to_point((70.710678, -29.289322,
0.000000), single=True)
mdl.add_fix_bc(support_node)

# mdl.show(show_bcs=0.005)

prb = mdl.add_problem(Problem(name="BeamBatheAnalysis"))
stp = prb.add_step(StaticStep(name="StaticStep"))

stp.combination = LoadCombination.SLS()
stp.add_uniform_node_load(nodes=loaded_node, load_case="LL", x=0.0, y=0.0,
z=600, xx=0.0, yy=0.0, zz=0.0)

stp.add_outputs([DisplacementFieldResults])
prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
Verbose=True)

stp.show_deformed(scale_results=1, show_original=0.1, show_bcs=0.003,
show_loads=0.001)
```

The final deformed shape can be seen in figure 7.

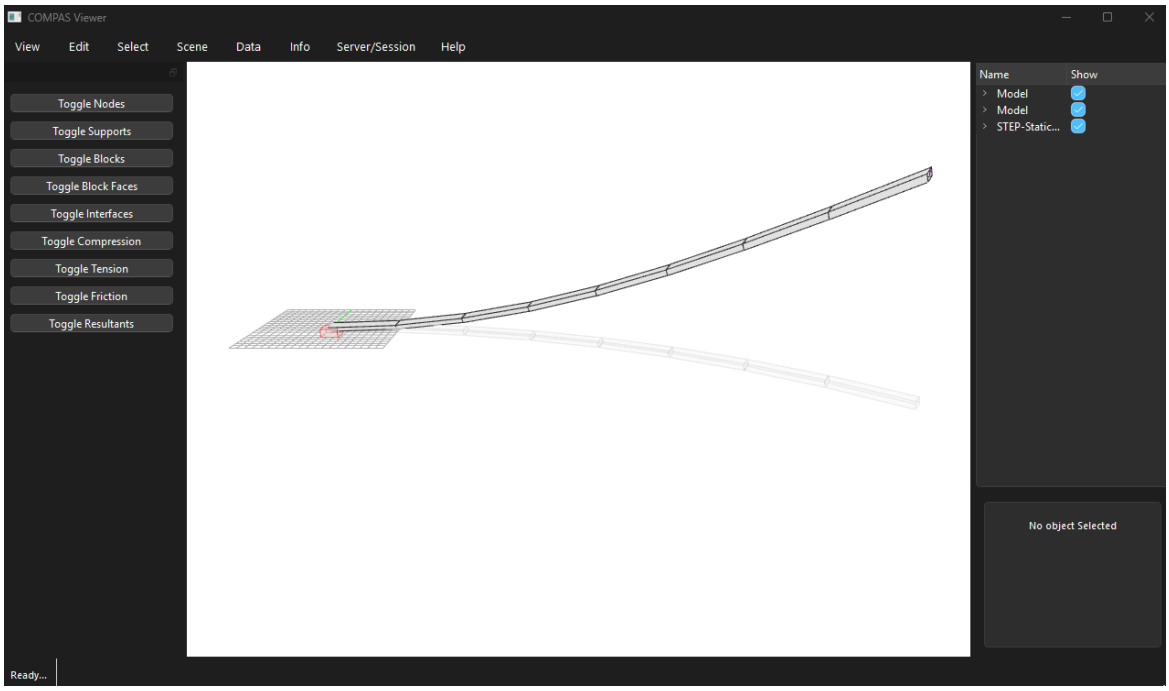

Figure 7: The lighter shade represents the undeformed shape.

3.6 Beam_simple.py

Beam_simple depicts a simple beam with two fixed ends and load applied in the center. Similar to the beam_frame from section 3.4, the model is manually discretized, therefore nodes and element data will be truncated for simplicity.

```
from compas_fea2.model import Model, Part, Node, BeamElement, ElasticIsotropic,
BeamSection, RectangularSection, PipeSection, CircularSection
from compas_fea2.model.shapes import Rectangle, Circle
from compas_fea2.problem import Problem, StaticStep, LoadCombination
from compas_fea2.results import DisplacementFieldResults, ReactionFieldResults,
StressFieldResults
from compas_fea2_opensees import TEMP
import compas_fea2
import os
compas_fea2.set_backend("compas_fea2_opensees")
mdl = Model(name="Beam_simple")
prt = mdl.add_part(Part(name="Beam_simple_analysis"))

prt.ndm = 3
prt.ndf = 6
steel = ElasticIsotropic(name='mat_elastic', E=20*10**9, v=0.3, density=1500)

section_beam1 = CircularSection(r=0.005, material=steel)

nodes_data = {0: (1.000000, 0.000000, 0.000000),
  1: (0.950000, 0.000000, 0.000000),
  2: (0.900000, 0.000000, 0.000000),
  3: (0.850000, 0.000000, 0.000000),
       .
       .
       .
  99: (0.020000, 0.000000, 0.000000),
  100: (0.010000, 0.000000, 0.000000),}

elements_data = [
  (21, 0),
  (22, 21),
       .
       .
       .
  (99, 98),
  (100, 99),
  (20, 100),]
```

```
nodes = {}
for nid, xyz in nodes_data.items():
    nodes[nid] = prt.add_node(Node(name=nid, xyz=xyz))

for node_ids in elements_data:
    n1 = nodes[node_ids[0]]
    n2 = nodes[node_ids[1]]
    prt.add_element(BeamElement(nodes=[n1, n2], section=section_beam1,
frame=[1,1,1]))

loaded_nodes_coords =   [(0.050000, 0.000000, 0.000000),
.
.
.
  (0.950000, 0.000000, 0.000000),]

loaded_nodes = []
for coord in loaded_nodes_coords:
    n = prt.find_closest_nodes_to_point(coord, single=True)
    loaded_nodes.append(n)

left_n = prt.find_closest_nodes_to_point((0.0,0.0,0.0),single = True)
right_n = prt.find_closest_nodes_to_point((1.0,0.0,0.0),single = True)
mdl.add_fix_bc(nodes=[left_n,right_n])

# mdl.show(show_bcs=0.0001)

prb = mdl.add_problem(Problem(name="BeamSimpleAnalysis"))
stp = prb.add_step(StaticStep(name="StaticStep"))

stp.combination = LoadCombination.SLS()
stp.add_uniform_node_load(nodes=loaded_nodes, load_case="LL", x=0.0, y=0.0,
z=1.0, xx=0.0, yy=0.0, zz=0.0)

stp.add_outputs([DisplacementFieldResults])
prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
Verbose=True)

stp.show_deformed(scale_results=10, show_original=0.1, show_bcs=0.00003,
show_loads=0.001)
```

The final deformed shape can be seen in Figure 8.

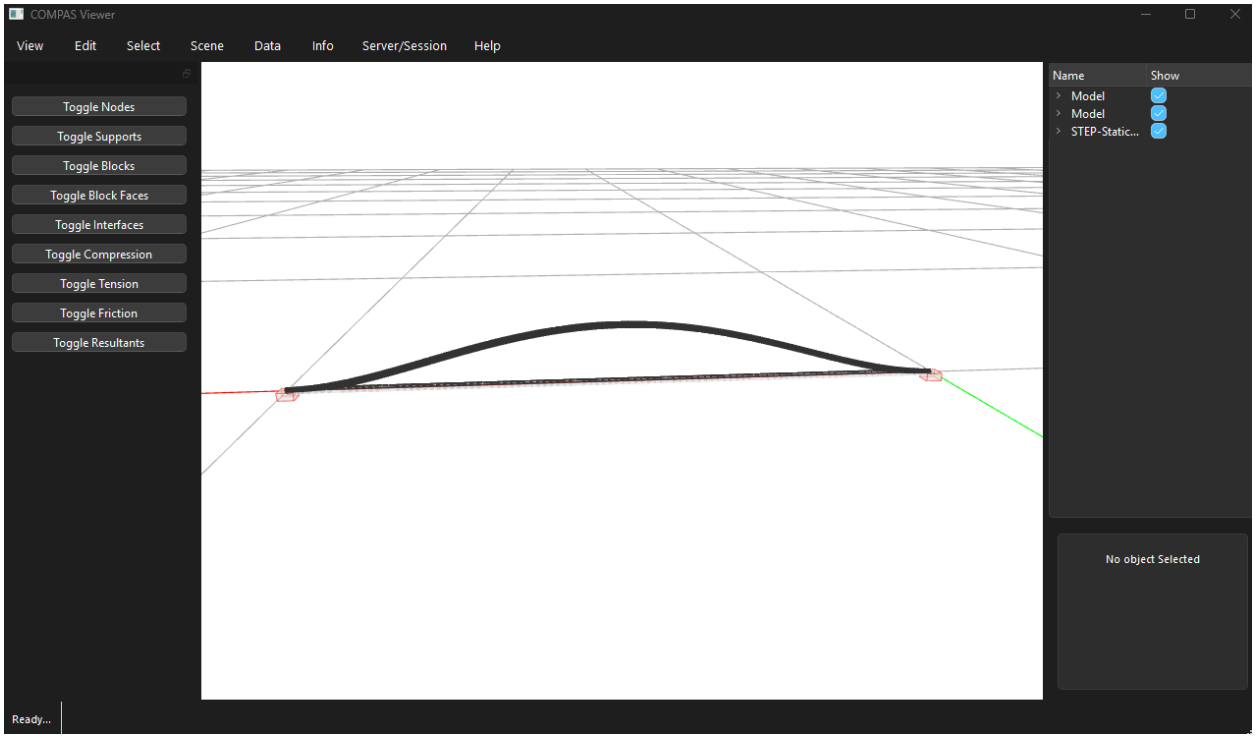

Figure 8: Lighter shade represents the undeformed model.

3.7 Truss_frame.py

Truss_frame is a model depicting a frame made out of truss elements. Similar to the Truss_tower model, it relies on prt.ndf being equal to 3.

```
from compas_fea2.model import Model, Part, Node, TrussElement,
ElasticIsotropic, TrussSection
from compas_fea2.problem import Problem, StaticStep, LoadCombination
from compas_fea2.results import DisplacementFieldResults, ReactionFieldResults,
StressFieldResults
from compas_fea2_opensees import TEMP
import compas_fea2
import os
compas_fea2.set_backend("compas_fea2_opensees")
mdl = Model(name="TrussFrame")
prt = mdl.add_part(Part(name="Truss-2"))
prt.ndm = 3
prt.ndf = 3

p = 7850
A1 = 0.0008
```

```
# Materials
steel = ElasticIsotropic(name="Steel", E=200000000000, v=0.3, density=p)

# Sections
mainTruss = TrussSection(name='sec_main', A=A1, material=steel)

nodes_data = {
    0: (0.243596, 2.624894, 5.002196),
    1: (-4.312031, 8.337478, 0.0),
       .
       .
       .
    54: (5.243596, 1.917787, 4.29509),
    55: (3.243596, 1.917787, 4.29509),
}

lines = [
    (0, 1),
    (2, 3),
       .
       .
       .
    (42, 49),
    (12, 20),
]

constrained_nodes_coordinates = [(-4.31203138797, -4.50190454813, 0.0),
(-4.31203138797, 8.33747825318, 0.0),
(0.243596487824, 1.21068007134, 0.00219637817533),
(0.243596487824, 2.62489363371, 0.00219637817533),
(1.24359648782, 1.91778685253, 0.00451802597678),
(15.2435964878, 1.91778685253, 0.00451802597678),
(16.2435964878, 1.21068007134, 0.00219637817533),
(16.2435964878, 2.62489363371, 0.00219637817533)]

loaded_nodes_coordinates = [(8.24359648782, 2.62489363371, 5.00219637818),
(8.24359648782, 1.21068007134, 5.00219637818),
(6.24359648782, 1.21068007134, 5.00219637818),
(6.24359648782, 2.62489363371, 5.00219637818),
(10.2435964878, 2.62489363371, 5.00219637818),
(10.2435964878, 1.21068007134, 5.00219637818)]

nodes = {}
for nid, xyz in nodes_data.items():
    nodes[nid] = prt.add_node(Node(name=nid, xyz=xyz))
```

```
for _nodes in lines:
    n1 = nodes[_nodes[0]]
    n2 = nodes[_nodes[1]]
    prt.add_element(TrussElement(nodes=[n1, n2], section=mainTruss))

for coords in constrained_nodes_coordinates:
    n = prt.find_closest_nodes_to_point(coords, single=True)
    mdl.add_fix_bc(n)

#mdl.show(show_bcs=0.0003)

prb = mdl.add_problem(Problem(name="TrussFrameAnalysis"))
stp = prb.add_step(StaticStep(name="StaticStep"))

stp.combination = LoadCombination.SLS()
for coords in loaded_nodes_coordinates:
    n = prt.find_closest_nodes_to_point(coords, single=True)
    stp.add_uniform_node_load(nodes=n, load_case="LL", x=0.0, y=0.0, z=-15500,
xx=0.0, yy=0.0, zz=0.0)

stp.add_outputs([DisplacementFieldResults])
prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
Verbose=True)

stp.show_deformed(scale_results=10, show_original=0.1, show_bcs=0.0003,
show_loads=0.0001)
```

The final deformed shape can be seen in figure 9.

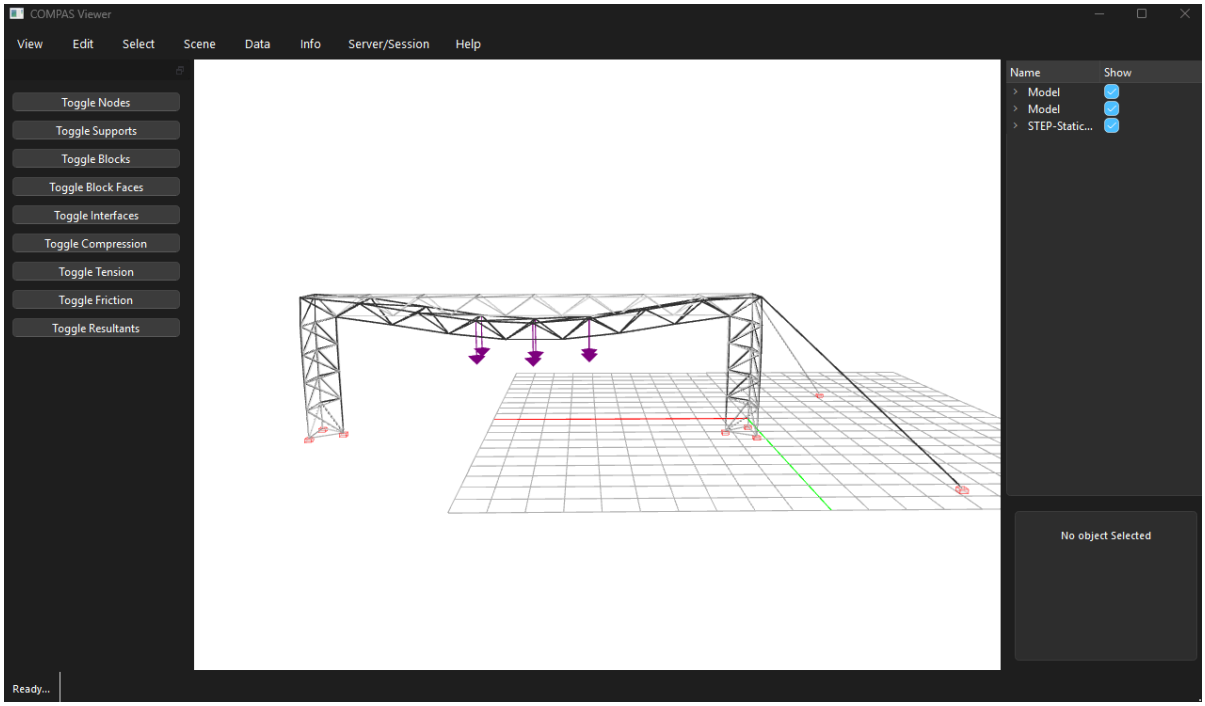

Figure 9: the lighter shade represents the undeformed shape.

3.8 Truss.py

This is one of the models adapted from the official Opensees examples. It depicts a 2-Dimensional truss with 3 members. The primary difference with this model is that prt.ndm is set to 2, as the model is 2-dimensional.

```
from compas_fea2.model import Model, Part, Node, TrussElement,
ElasticIsotropic, TrussSection
from compas_fea2.problem import Problem, StaticStep, LoadCombination
from compas_fea2.results import DisplacementFieldResults, ReactionFieldResults,
StressFieldResults
from compas_fea2_opensees import TEMP
import compas_fea2
import os

# Set the backend
compas_fea2.set_backend("compas_fea2_opensees")

mdl = Model(name="TrussModel")
prt = mdl.add_part(Part(name="Truss-1"))
prt.ndm = 2
prt.ndf = 3
```

```
#Materials
steel = ElasticIsotropic(name="Steel", E=3000.0, v=0.0, density=0.0)
section_large = TrussSection(name="TrussSec", A=10.0, material=steel)
section_small = TrussSection(name="TrussSecSmall", A=5.0, material=steel)

#nodal coords
nodes_data = {
    1: (0.0, 0.0, 0.0),
    2: (144.0, 0.0, 0.0),
    3: (168.0, 0.0, 0.0),
    4: (72.0, 96.0, 0.0),
}

nodes = {}
for nid, xyz in nodes_data.items():
    nodes[nid] = prt.add_node(Node(name=nid, xyz=xyz))

for nid in [1, 2, 3]:
    rid = nodes_data[nid]
    n = prt.find_closest_nodes_to_point(rid, single=True)
    mdl.add_pin_bc(n)


for nid in [4]:
    rid = nodes_data[nid]
    n = prt.find_closest_nodes_to_point(rid, single=True)
    mdl.add_specific_bc(n)


elements = [
    (1, 1, 4),
    (2, 2, 4),
    (3, 3, 4),
]

prt.add_element(TrussElement(name=str(1), nodes=[nodes[1], nodes[4]],
section=section_large))
prt.add_element(TrussElement(name=str(2), nodes=[nodes[2], nodes[4]],
section=section_small))
prt.add_element(TrussElement(name=str(3), nodes=[nodes[3], nodes[4]],
section=section_small))

prb = mdl.add_problem(Problem(name="TrussAnalysis"))
stp = prb.add_step(StaticStep(name="StaticStep"))
```

```
loaded_node = prt.find_closest_nodes_to_point(nodes_data[4], single = True)
stp.combination = LoadCombination.SLS()
stp.add_uniform_node_load(nodes=loaded_node, load_case="LL", x=100.0, y=-50.0,
z=0.0, xx = 0.0, yy = 0.0, zz = 0.0)
stp.add_outputs([DisplacementFieldResults])

#mdl.show(show_bcs = 0.03, show_loads = 10)

prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
Verbose=True)

stp.show_deformed(scale_results=10, show_original=0.1, show_bcs=0.01)
```

The deformed shape can be seen in figure 10.

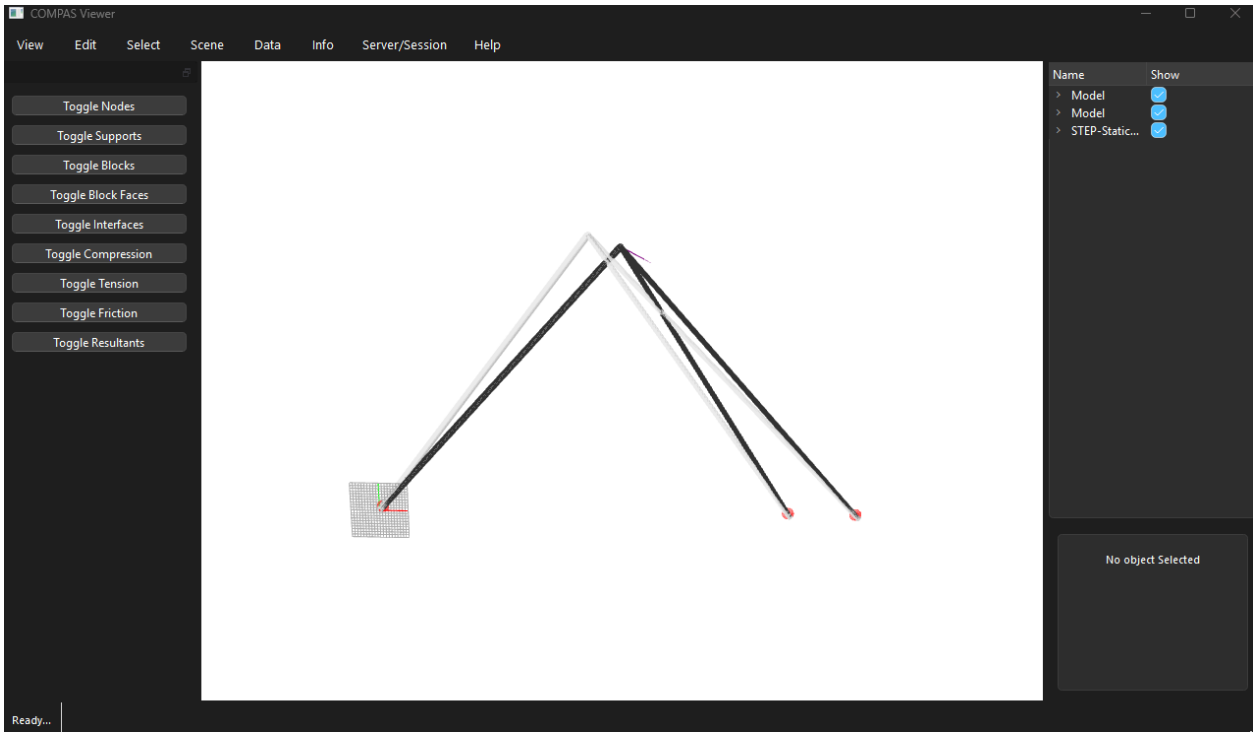

Figure 10: Lighter shade depicts undeformed shape.

3.9 ElasticFrame.py

This is the second model adapted from an official example given by Opensees. It depicts modal analysis of a 2-Dimensional frame structure. The initial setup is typical, however, this time gravity and mass values are used to assign weight to nodes. Similar to the truss model from 3.8, prt.ndm is 2 as the model is 2-dimensional.

```
from compas_fea2.model import Model, Part, Node, BeamElement, ElasticIsotropic,
BeamSection, GenericBeamSection
from compas_fea2.model.shapes import Circle
from compas_fea2.problem import Problem, ModalAnalysis
from compas_fea2_opensees import TEMP
import os
import compas_fea2

# Set the backend implementation
compas_fea2.set_backend("compas_fea2_opensees")

# Variables
g = 386.4
ft = 12.0
Load1 = 1185.0
Load2 = 1185.0
Load3 = 970.0

# set floor masses
m1 = Load1 / (4 * g)  # 4 nodes per floor
m2 = Load2 / (4 * g)
m3 = Load3 / (4 * g)

# set floor distributed loads
w1 = Load1 / (90 * ft)  # frame 90 ft long
w2 = Load2 / (90 * ft)
w3 = Load3 / (90 * ft)

# Initialize the model
mdl = Model(name="ElasticFrame")
prt = mdl.add_part(Part(name="ElasticFrame-1"))
prt.ndm = 2
prt.ndf = 3
#Define shape of beam sections
face = Circle(radius=5)

# Define material properties (Steel)
steel = ElasticIsotropic(name="Steel", E=29000.0, v=0.3, density=0.0)

# Define cross-sectional properties
section_col_small = BeamSection(name="W14X257", A=75.6, Ixx=3400.0, Iyy=0,
Ixy=0, Avx=0, Avy=0, J=0, g0=0, gw=0, material=steel, shape=face)
section_col_large = BeamSection(name="W14X311", A=91.4, Ixx=4330.0, Iyy=0,
Ixy=0, Avx=0, Avy=0, J=0, g0=0, gw=0, material=steel, shape=face)
section_beam1 = BeamSection(name="W33X118", A=34.7, Ixx=5900.0, Iyy=0, Ixy=0,
```

```
Avx=0, Avy=0, J=0, g0=0, gw=0, material=steel, shape=face)
section_beam2 = BeamSection(name="W30X116", A=34.2, Ixx=4930.0, Iyy=0, Ixy=0,
Avx=0, Avy=0, J=0, g0=0, gw=0, material=steel, shape=face)
section_beam3 = BeamSection(name="W24X68", A=20.1, Ixx=1830.0, Iyy=0, Ixy=0,
Avx=0, Avy=0, J=0, g0=0, gw=0, material=steel, shape=face)

# Create nodes
nodes_data = {
    1: [(0.0, 0.0, 0.0), (0.0, 0.0, 0.0)],
    2: [(360.0, 0.0, 0.0), (0.0, 0.0, 0.0)],
    3: [(720.0, 0.0, 0.0), (0.0, 0.0, 0.0)],
    4: [(1080.0, 0.0, 0.0), (0.0, 0.0, 0.0)],
    5: [(0.0, 162.0, 0.0), (m1, m1, 0.0)],
    6: [(360.0, 162.0, 0.0), (m1, m1, 0.0)],
    7: [(720.0, 162.0, 0.0), (m1, m1, 0.0)],
    8: [(1080.0, 162.0, 0.0), (m1, m1, 0.0)],
    9: [(0.0, 324.0, 0.0), (m2, m2, 0.0)],
    10: [(360.0, 324.0, 0.0), (m2, m2, 0.0)],
    11: [(720.0, 324.0, 0.0), (m2, m2, 0.0)],
    12: [(1080.0, 324.0, 0.0), (m2, m2, 0.0)],
    13: [(0.0, 486.0, 0.0), (m3, m3, 0.0)],
    14: [(360.0, 486.0, 0.0), (m3, m3, 0.0)],
    15: [(720.0, 486.0, 0.0), (m3, m3, 0.0)],
    16: [(1080.0, 486.0, 0.0), (m3, m3, 0.0)],
}

for nid, data in nodes_data.items():
    x = prt.add_node(Node(name=nid, xyz=data[0], mass=data[1]))

# Apply boundary conditions (fixed supports at base nodes)
for nid in [1, 2, 3, 4]:
    rid = nodes_data[nid][0]
    n = prt.find_closest_nodes_to_point(rid, single=True)
    mdl.add_pin_bc(n)

# Define elements
columns_info = [
    (1, 1, 5, section_col_small),
    (2, 5, 9, section_col_small),
    (3, 9, 13, section_col_small),
    (4, 2, 6, section_col_large),
    (5, 6, 10, section_col_large),
    (6, 10, 14, section_col_large),
    (7, 3, 7, section_col_large),
    (8, 7, 11, section_col_large),
```

```
    (9, 11, 15, section_col_large),
    (10, 4, 8, section_col_small),
    (11, 8, 12, section_col_small),
    (12, 12, 16, section_col_small),
]

beams_info = [
    (13, 5, 6, section_beam1),
    (14, 6, 7, section_beam1),
    (15, 7, 8, section_beam1),
    (16, 9, 10, section_beam2),
    (17, 10, 11, section_beam2),
    (18, 11, 12, section_beam2),
    (19, 13, 14, section_beam3),
    (20, 14, 15, section_beam3),
    (21, 15, 16, section_beam3),
]

for eid, n1, n2, section in columns_info + beams_info:
    nid1 = nodes_data[n1][0]
    n1 = prt.find_closest_nodes_to_point(nid1, single = True)
    nid2 = nodes_data[n2][0]
    n2 = prt.find_closest_nodes_to_point(nid2, single = True)
    prt.add_element(BeamElement(nodes=[n1, n2], section=section, frame=[0, 0,
1]))
```

In Modal Analysis, outputs do not have to be explicitly defined, and are included within the analysis itself.

```
prb = mdl.add_problem(Problem(name="ModalAnalysis"))
stp = prb.add_step(ModalAnalysis(modes=5))

prb.analyse_and_extract(problems=[prb], path=os.path.join(TEMP, prb.name),
verbose=True)

stp.show_mode_shape(step=stp, mode=1, scale_results=100, show_bcs=0.03)
```

The Final deformed shape can be seen in Figure 11.

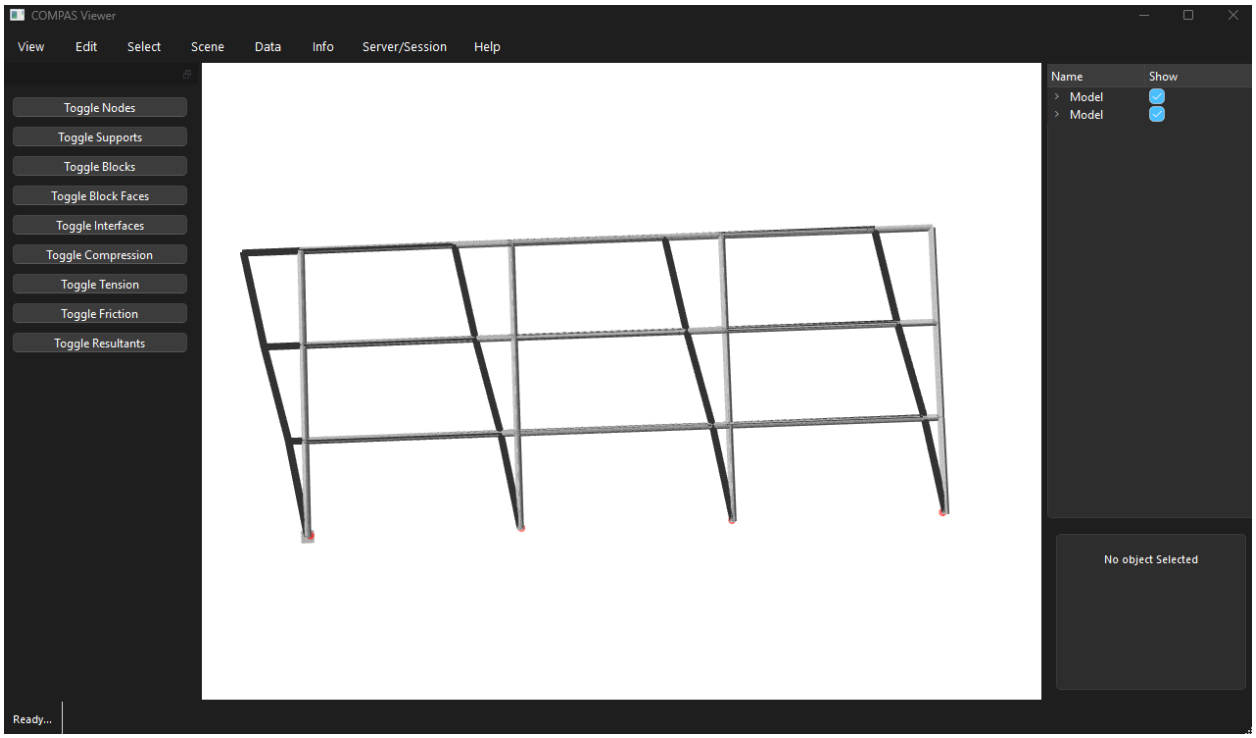

Figure 11: Lighter shade depicts undeformed structure.

3.10 Development of Additional Functions

As introduced in Section 2.2, four functions were implemented: OpenseesLinkElement, OpenseesSpringElement, beamWithHinges, and OpenseesTieConstraint. This section outlines how these functions were integrated into the backend and what enables them to operate correctly.

The first three—OpenseesLinkElement, OpenseesSpringElement, and OpenseesTieConstraint—each include dedicated Jobdata functions (Figure 12). Within the OpenSees backend, these routines are critical because they assemble the information required to generate valid Tcl commands. For instance, the Jobdata for OpenseesSpringElement collects the keys of the connected nodes, while that for OpenseesTieConstraint identifies the master and slave nodes. This data is then combined into a properly formatted string containing the corresponding OpenSees command name (e.g., equalDOF for OpenseesLinkElement). Ultimately, these strings are collated and written into the Tcl file that is executed by OpenSees.exe.

```
class OpenseesSpringElement(SpringElement):
    """OpenSees implementation of
:class:`compas_fea2.model.SpringElement`.\n"""
```

```
    __doc__ += SpringElement.__doc__

    def __init__(self, *, nodes, section, **kwargs):
        super(OpenseesSpringElement, self).__init__(nodes=[nodes],
section=section, **kwargs)

    def jobdata(self):
        """Return the OpenSees command for spring element."""
        return f"element zeroLength {self.key} {self.nodes[0].key}
{self.nodes[1].key} -mat {self.section.material.key}
{self.section.material.key} -dir 1" #NOTE: adds axial stiffness

class OpenseesLinkElement(LinkElement):
    """
    OpenSees implementation of :class:`compas_fea2.model.LinkElement`.\n

    Additional Parameters
    ---------------------
    materials:
        List input of material objects. list should contain between 1 to 6
elements. Each material's position within the list determines which direction
it corresponds to

    directions:
        List input of directions. List should be a list of numbers between 1
and 6, an input of [1 2 3 4 5 6] means all directions between two nodes are
linked
        1 = axial direction
        2 = shear y direction
        3 = shear z direction
        4 = torsion x direction
        5 = moment y direction
        6 = moment z direction

    Note
    ----------------------
    Both materials and directions MUST be equal length lists. Even if only one
material is used for all 6 directions, it must be repeated all 6 times.
    """

    __doc__ += LinkElement.__doc__

    def __init__(self, nodes, materials: list, directions: list, **kwargs):
        super(OpenseesLinkElement, self).__init__(nodes=nodes, **kwargs)
```

```
        self.nodes = nodes
        self.materials = " ".join(str(mat.key) for mat in materials)
        self.directions = " ".join(str(dir) for dir in directions)
```

Figure 12a: The OpenseesLinkElement and OpenseesSpringElement, located in compas_fea2_opensees/model/elements.py

```
class OpenseesTieConstraint(TieConstraint):
    def __init__(self, master, slave, **kwargs):
        super(OpenseesTieConstraint, self).__init__(master, slave, tol=None,
**kwargs)
        self.master = master
        self.slave = slave

        _freedoms = kwargs.get("freedoms") if "freedoms" in kwargs else [1, 2,
3, 4, 5, 6]
        self.freedoms = " ".join(str(freedom) for freedom in _freedoms)

    def jobdata(self):
        """Return the OpenSees command for equal DOF constraint.  """
        return f"equalDOF {self.master} {self.slave} {self.freedoms}"
```

Figure 12b: The OpenseesTieConstraint, located in compas_fea2_opensees/model/constraints.py

The function beamWithHinges is unique because it is actually a function of the OpenseesBeamElement class, as opposed to being a class itself, similar to the elasticBeam function and only exists in the opensees backend. Therefore, it lacks a jobdata function (Figure 13)

```
class OpenseesBeamElement(BeamElement):
    """OpenSees implementation of :class:`compas_fea2.model.BeamElement`.\n"""

    __doc__ += BeamElement.__doc__

    def __init__(self, nodes, section, implementation="elasticBeamColumn",
frame=[0.0, 0.0, -1.0], **kwargs):
        if not implementation:
            implementation = "elasticBeamColumn"
        super(OpenseesBeamElement, self).__init__(nodes=nodes, section=section,
frame=frame, implementation=implementation, **kwargs)

        try:
```

```
            self._job_data = getattr(self, "_" + implementation)
        except AttributeError:
            raise ValueError("{} is not a valid implementation
model".format(implementation))

    def jobdata(self):
        if self.part.ndm == 2:
            return "\n".join(
                [
                    f"geomTransf Corotational {self.key}", #2D
                    self._job_data(),
                ])
        elif self.part.ndm == 3:
            return "\n".join(
                [
                    "geomTransf Corotational {} {}".format(self.key, "
".join([str(i) for i in self.frame.zaxis])), #3D
                    self._job_data(),
                ])
        else:
            raise ValueError("Beam elements are only supported in 2D and 3D
models, not {}".format(self.part.ndm))

    def _elasticBeamColumn(self):
        """Construct an elasticBeamColumn element object.

        For more information about this element in OpenSees check
        `here
<https://opensees.github.io/OpenSeesDocumentation/user/manual/model/elements/gr
adientInelasticBeamColumn.html>`_
        """
        if self.part.ndm == 2:
            return "element elasticBeamColumn {} {} {} {} {} {}".format(
                self.key,
                " ".join(str(node.key) for node in self.nodes),
                self.section.A,
                self.section.material.E,
                self.section.Ixx,
                self.key,
            )
        else:
            return "element {} {} {} {} {} {} {} {} {} {}".format(
                self._implementation,
                self.key,
                " ".join(str(node.key) for node in self.nodes),
```

```
                self.section.A,
                self.section.material.E,
                self.section.material.G,
                self.section.J,
                self.section.Ixx,
                self.section.Iyy,
                self.key,
            )

    def _beamWithHinges(self):
        """Construct a beamWithHinges element object.

        For more information about this element in OpenSees check
        `here
<https://opensees.github.io/OpenSeesDocumentation/user/manual/model/elements/be
amWithHinges.html>`_

        """
        return "element beamWithHinges {} {} {} {} {} {}".format(
            self.key,
            " ".join(str(node.key) for node in self.nodes),
            self.section.A,
            self.section.Ixx,
            self.section.Iyy,
            self.key,
        )
```

Figure 13: The OpenseesBeamElement, containing the _beamWithHinges command. Located in compas_fea2_opensees/model/elements.py

In addition, the super command is used to map the attributes of high-level compas_fea2 objects to their OpenSees counterparts. In some cases, technical limitations require extending the base implementation with additional attributes. For example, OpenseesLinkElement requires explicit lists for directions and materials to ensure correct behavior.

Finally, for these functions to be accessible within the main library, they must be registered properly in the backend. This involves two steps: (1) adding the backend function in compas_fea2_opensees/__init__.py (Figure 14), and (2) declaring it within the __all__ variable and importing its main compas_fea2 counterpart in the appropriate module-level __init__.py file (e.g., compas_fea2_opensees/model/__init__.py for the three functions here) (Figure 15).

```
  def _register_backend():
        backend = compas_fea2.BACKENDS["compas_fea2_opensees"]

        backend[Model] = OpenseesModel
        backend[Part] = OpenseesPart
        backend[Node] = OpenseesNode

        backend[MassElement] = OpenseesMassElement
        backend[SpringElement] = OpenseesSpringElement
        backend[LinkElement] = OpenseesLinkElement
        backend[TieConstraint] = OpenseesTieConstraint
```

Figure 14: The register backend function, located in compas_fea2_opensees/__init__.py

```
__all__ = [
    "OpenseesModel",
    "OpenseesPart",
    "OpenseesNode",
    "OpenseesMassElement",
    "OpenseesLinkElement",
    "OpenseesBeamElement",
    "OpenseesTrussElement",
    "OpenseesSpringElement",
    "OpenseesShellElement",
    "_OpenseesElement3D",
    "OpenseesBeamSection",
    "OpenseesGenericBeamSection",
```

Figure 15: A snippet of the __all__ list located in compas_fea2_opensees/model/__init__.py

Chapter 4
Results and remaining tasks

As a result of this project, compas_fea2 has become significantly more accessible and functional for engineers aiming to use it in their structural analysis workflows or contribute to its continued development. The addition of key OpenSees functionalities—including support for zeroLength elements, beamWithHinges, twoNodeLink, shell elements, and geometric transformation commands (geomTransf)—has expanded the backend's analytical capabilities and made it suitable for a wider range of structural applications. These enhancements, coupled with improvements to documentation and example scripts, reduce the learning curve for new users while providing a more stable and extensible foundation for advanced users and developers. Engineers can now more confidently adopt compas_fea2 in real-world projects or build upon it to extend its solver support, model types, or automation features.

Still, some functionality which does exist for base OpenSees has not yet been incorporated into Compas_fea2_Opensees. For instance, compas_fea2_Opensees currently has not implemented Pressure loads, Prestress loads, Tributary loads, harmonic point loads and harmonic pressure loads. Therefore there still remains room for further development following the conclusion of this project.

References

[1]“Basic Examples Manual - OpenSeesWiki,” *Berkeley.edu*, 2025. https://opensees.berkeley.edu/wiki/index.php?title=Basic_Examples_Manual (accessed Aug. 28, 2025).

[2]Andrew Liew, “compas_fea - Examples,” *Compas.dev*, 2017. https://compas.dev/compas_fea/latest/examples.html (accessed Aug. 28, 2025).

[3] Zhu, M., McKenna, F., & Scott, M. H. (2018). OpenSeesPy: Python library for the OpenSees finite element framework. *SoftwareX*, 7, 6–11.