📦 fem_toolbox documentation

Finite Element Toolbox for 1D and 2D Frame Structures - Static and Modal Analysis

---

Version: 0.1
Author: Girardi Gabriele

Table of contents

Introduction

`fem_toolbox` is a collection of Python functions designed to perform finite element analysis (FEA) on structural systems. The package currently supports only **1D structural elements**:

- Bar elements
- Euler-Bernoulli beam elements
- 2D frame (beam) elements

The choice of writing an implementation from scratch instead of using well-known FEA solvers or packages comes from the fact that by designing and controlling every aspect of the code I hope to have ensured full flexibility for integrating future modules. The primary motivation for this is to **build a shape optimization module on top of this package**.

It's worth mentioning the open-source project *PyNite*, which stood out as one of the few user-friendly, accessible FEA packages that isn't heavily math-abstracted (unlike FEniCS, for example). However, I ultimately chose to develop my own implementation rather than invest time in learning and adapting an existing codebase.

That said, PyNite is a well-established project with support for 2D and 3D elements, P-Δ effects, and many other advanced features. For this reason, I anticipate eventually porting the shape optimization module developed here to the PyNite ecosystem, while leaving this package (fem_toolbox) primarily as a didactic exercise.

This package is not intended to replace professional FEA tools but it is rather a personal exercise. The goal was to have a minimal and extensible base for experimentation and further development in the context of structural optimization and custom analysis tools.

Installation

This section explains how to install the `fem_toolbox` package locally for development and usage.

Step 0: Place the root directory of the project ('fem_toolbox') wherever you like on your filesystem

Step 1: Navigate to the project root directory

This should be the root folder, containing the `setup.py` file.

```
cd path/to/fem_toolbox
```

Step 2: Install the package

Use pip to install the package. If you wish to make modifications to the library, use the flag '-e' so it can be edited live during development.

```
pip install .
```

```
pip install -e .
```

Step 3: Verify the installation

Launch Python or Jupyter and test that the package is accessible.

```
import fem_toolbox
```

Notes

- Make sure the correct virtual environment is activated (if used).
- Restart the kernel in Jupyter notebooks after installation.
- Restart the kernel in Jupyter notebooks after modifications (no need to re-install the library)
- If the import still fails, ensure you're running in the environment where the package was installed.

Dependencies

`numpy`, `scipy`, and `matplotlib` will be automatically installed with the package (if not present already). However, if you're using a conda environment or installing offline, you may need to manually install them:

```
conda install numpy scipy matplotlib
```

or

```
pip install numpy scipy matplotlib
```

`sympy` is used in some example notebooks to allow for pretty formatting of matrix expressions, but it is not required to use the package.

Package organization and tutorials

To support clarity and modularity, the package is organized into submodules such as geometry, mesh, elements etc, each reflecting a logical stage in the finite element workflow. This structure mirrors the typical progression of FEA, from geometric definition and mesh generation to system assembly and solution, and is also reflected in the structure of this document. By presenting the functions in this order, the documentation follows a natural pipeline that closely aligns with the practical steps of setting up and solving a FEA problem.

Before diving into the documentation, I recommend to have a look at the subdirectory "**Tutorials**". Here you may find some jupyter notebooks:

- prototype (1D and 2D): these notebooks were used to develop the library itself, through experimentation. *Useful only for getting a better understanding of the code in case you intend to modify or add functionality to the library*
- tutorials (1D and 2D): these notebooks were used primarily as a test for the library but also serve as tutorials since they show the complete FEA workflow, from importing the geometry to the plot of the results.
  - The 1D tutorial is heavily commented and its purpose is to explain in detail the different functions and how to use them.
  - The 2D tutorial goes straight to the point and shows how, in a quick way and with very few function calls, this package allows to analyze 2D structures. *Very useful*
- geometry&boundary-files: here you may find some .txt that define geometry and boundary conditions for the 2 tutorial cases. You can have a look at the file structure and use them as a reference

---

Library overview

- Module 1: geometry.py

  - read_structure('file_path'): reads geometry from file

- Module 2: mesh.py

  - discretize(...): discretizes geometry into finite elements.
  - plot_discretized_geometry_2D: plots the original frame geometry and the discretized FEM mesh.
  - plot_discretized_geometry_1D: plots the original 1D geometry and the discretized FEM mesh.

- Module 3: elements.py

  - k_rod(E, A, L): computes local stiffness matrix for rod element.
  - k_beam(E, I, L): computes local stiffness matrix for beam element.
  - k_beam2d(E, A, I, L): computes local stiffness matrix for 2d beam element.
  - m_rod(rho, A, L): computes local mass matrix for rod element.
  - m_beam(rho, A, L): computes local mass matrix for beam element.
  - m_beam2d(rho, A, L): computes local mass matrix for 2d beam element.
  - rotation_2d(x1, y1, x2, y2): computes rotation matrix between local and global reference system

- Module 4: BC_loads.py

  - read_bc_and_forces('file_path'): reads boundary conditions and forces from a structured file.
  - validate_constraints(...): validates that the structure is not under-constrained.
  - plot_fem_model(...): plots structure, BC and forces. Provides visual validation of the inputs

- Module 5: femsolver.py

  - dof_map(ndofs_per_node, n1, n2): Returns the list of global degrees of freedom (DOF) indices for an element defined by its two node indices.
  - assembleK(...): assembles the global stiffness matrix.
  - assembleM(...): assembles the global mass matrix.
  - build_force_vector(...): assembles the global external force vector.
  - static_analysis: solves the global system K u = f for the displacement u
  - modal_analysis(...): solves the eigenvalue problem K * u = lambda * M *u for a specified number of natural frequencies

- Module 6: postprocessing.py -eval_stress(...): computes internal actions and stresses (axial, bending, shear, von Mises) in each element.

  - compute_reaction_forces(...): computes the reaction forces at constrained DOFs.
  - plot_internal_actions_2D(...): plots internal action diagrams (Axial, Shear and Moment) for each beam of the structure.
  - plot_internal_actions_1D(...): plots internal action diagrams (Axial, Shear and Moment) for 1D horizontal geometries.
  - plot_2D_loaded_structure(...): plots the undeformed and deformed frame structure with stress (stress type selectable) color map.
  - plot_1d_loaded_structure(...): plots a 1D structure (beam or bar) with deformed shape and stress (stress type selectable) color map.
  - animate_mode_shape_2D(...): animates a specified mode shape for a 2D frame.

  - animate_mode_shape_1D(...): animate a specified mode shape for 1D bar or beam element.

Module 1: geometry.py (input parser)

This module provides functionality for reading a beam structure definition from a file. The file includes material properties and beam geometry with associated cross-sectional properties. It processes this data into structured arrays for use in a finite element model.

Function: `read_structure(file_path)`

Parses a text file describing a 2D frame structure into node coordinates, element connectivity, cross-section properties, and material properties.

Function parameters:

```
file_path (str):
Path to the input file.
```

Input file structure:

```
The first line should define material properties: E rho 0 0 0 0.
Each subsequent line defines a single beam as: x1 y1 x2 y2 A I.
```

Returns

```
node_coords_beam (ndarray, shape (N, 2)):
Unique coordinates of all nodes (2D).

connectivity_beam (ndarray, shape (E, 2)):
Element-to-node connectivity (index pairs referencing node_coords_beam).

beam_crossSections (list of dict):
List of dictionaries with cross-sectional data for each beam:
    "A": Area
    "I": Moment of inertia

mat_properties (dict):
Material properties:
    "E": Young's modulus
    "rho": Density
```

Module 2: mesh.py (discretization and visualization)

This module handles the conversion of a high-level frame geometry into a finite element mesh by discretizing beams into smaller elements. It also includes a visualization tool for plotting the discretized structure, making it possible to visually verify the input data.

Function: `discretize(node_coords_beam, beam_connectivity, beam_crossSsections, elements_per_beam=5, strategy="uniform")`

Subdivides each beam into multiple finite elements with preserved geometric and cross-sectional properties.

Function parameters:

```
node_coords_beam (ndarray, shape (N, 2)):
Coordinates of the input geometry’s nodes.

beam_connectivity (ndarray, shape (B, 2)):
Node indices defining each beam.

beam_sections (list of dicts):
Cross-section properties for each beam ({"A": ..., "I": ...}).

elements_per_beam (int, default=5):
Number of FEM elements to create per beam.

strategy (str, default="uniform"):
Currently only "uniform" subdivision is supported.
```

Returns:

```
fem_nodes (ndarray, shape (M, 2)):
Unique coordinates of the generated FEM nodes.

fem_elements (ndarray, shape (E, 2)):
Element connectivity using node indices.

element_crossSection (list of dicts):
Cross-section properties for each FEM element.
```

Function: `plot_discretized_geometry_2D(original_nodes, original_beams, fem_nodes, fem_elements, show_elements=True, show_nodes=True, node_label_offset=0.02)`

Visualizes both the original beam geometry and the corresponding discretized FEM mesh for 2D frames.
Function parameters

```
original_nodes (ndarray, shape (N, 2)):
Coordinates of the original geometry nodes.

original_beams (ndarray, shape (B, 2)):
Beam connectivity as pairs of node indices.

fem_nodes (ndarray, shape (M, 2)):
Coordinates of the FEM nodes created through discretization.

fem_elements (ndarray, shape (E, 2)):
FEM element connectivity using FEM node indices.

show_elements (bool, default=True):
If True, labels each FEM element with its index at the midpoint.

show_nodes (bool, default=True):
If True, labels each FEM node with its index.

node_label_offset (float, default=0.02):
Offset used to avoid overlapping node labels with node markers.
```

Returns:

```
(None):
Generates and displays a matplotlib plot showing the discretized geometry,
original beam lines, and optional node/element IDs.
```

Function: `plot_discretized_geometry_1D(original_nodes, original_beams, fem_nodes, fem_elements, show_elements=True, show_nodes=True, node_label_offset=0.02)`

Visualizes both the original beam geometry and the corresponding discretized FEM mesh for 1D geometries.

Function parameters

```
original_nodes (ndarray, shape (N, 2)):
Coordinates of the original geometry nodes.

original_beams (ndarray, shape (B, 2)):
Beam connectivity as pairs of node indices.

fem_nodes (ndarray, shape (M, 2)):
Coordinates of the FEM nodes created through discretization.

fem_elements (ndarray, shape (E, 2)):
FEM element connectivity using FEM node indices.
```

```
show_elements (bool, default=True):
If True, labels each FEM element with its index at the midpoint.

show_nodes (bool, default=True):
If True, labels each FEM node with its index.

node_label_offset (float, default=0.02):
Offset used to avoid overlapping node labels with node markers.
```

Returns:

```
(None):
Generates and displays a matplotlib plot showing the discretized geometry,
original beam lines, and optional node/element IDs.
```

Module 3: `elements.py` (Element stiffness, mass matrices, and transformations)

This module defines local stiffness and mass matrices for various types of 1D structural elements used in finite element analysis. It includes functionality for:

- Rod (axial) elements
- Euler-Bernoulli beam elements
- 2D frame elements (axial + bending behavior)
- Local-to-global coordinate transformations

Function: `k_rod(E, A, L)`

Computes the local stiffness matrix for a 2-node axial (rod) element.

Function parameters:

```
E (float): Young’s modulus
A (float): Cross-sectional area
L (float): Element length
```

Returns:

```
k_local (ndarray, shape (2, 2)): Local stiffness matrix
```

Function: `k_beam(E, I, L)`

Returns the **local stiffness matrix** for a 2-node **Euler-Bernoulli beam element** (no axial DOFs, only bending).

Function parameters:

```
E (float): Young’s modulus
I (float): Moment of inertia
L (float): Element length
```

Returns:

```
k_local (ndarray, shape (4, 4)): Local bending stiffness matrix
```

Function: `k_beam2d(E, A, I, L)`

Computes the **local stiffness matrix** for a 2D beam (frame) element with 6 DOFs: `[u1, v1, θ1, u2, v2, θ2]`.

Function parameters:

```
E (float): Young’s modulus
A (float): Cross-sectional area
I (float): Moment of inertia
L (float): Element length
```

Returns:

```
k_local (ndarray, shape (6, 6)): Local stiffness matrix including axial and
bending behavior
```

Function: `m_rod(rho, A, L)`

Returns the **consistent** mass matrix for a 2-node rod (axial) element.

Function parameters:

```
rho (float): Material density
A   (float): Cross-sectional area
L   (float): Element length
```

Returns:

```
m_local (ndarray, shape (2, 2)): Local mass matrix
```

Function: `m_beam(rho, A, L)`

Returns the **consistent** mass matrix for a 2-node Euler-Bernoulli beam element (4 DOFs).

Function parameters:

```
rho (float): Material density
A   (float): Cross-sectional area
L   (float): Element length
```

Returns:

```
m_local (ndarray, shape (4, 4)): Local mass matrix for bending DOFs
```

Function: `m_beam2d(rho, A, L)`

Returns the **consistent** mass matrix for a 2D frame (beam) element with both axial and bending DOFs.

Function parameters:

```
rho (float): Material density
A   (float): Cross-sectional area
L   (float): Element length
```

Returns:

```
m_local (ndarray, shape (6, 6)): Full local mass matrix for a frame element
```

Function: `rotation_2d(x1, y1, x2, y2)`

Constructs the 6×6 rotation matrix to transform quantities between local and global coordinates for 2D beam elements.

Function parameters:

```
x1, y1 (float): Coordinates of the first node
x2, y2 (float): Coordinates of the second node
```

Returns:

```
R (ndarray, shape (6, 6)): Local-to-global rotation matrix
```

Module 4: BC_loads.py (assigning boundary conditions and loads)

Handles reading, validating, and visualizing boundary conditions (BCs) and external forces for the FEM model.

Function: `read_bc_and_forces(file_path)`

Reads BCs and forces from a structured text file with a compact syntax.

Function parameters:

```
file_path (str):
Path to the BC/force input file.

    BC lines start with "BC node_id" followed by (dof value) pairs, e.g.,
BC 3 (0 0.0) (1 0.0).

    Force lines start with "FORCE node_id dof value", e.g., FORCE 2 1 -500.
```

Returns:

```
constrained_nodes (list[int]):
Sorted unique nodes with BCs.

bc_nodes (list[int]):
Nodes where BCs are applied (may contain duplicates).

bc_dofs (list[int]):
Degrees of freedom constrained (0=u, 1=v, 2=theta).

bc_values (list[float]):
Values prescribed at constrained DOFs.

f_nodes (list[int]):
Nodes with applied forces.

f_dofs (list[int]):
```

```
DOFs where forces are applied.

f_values (list[float]):
Force values.
```

Function: `validate_constraints(num_nodes, bc_nodes, bc_dofs)`

Checks whether the structure is properly constrained and the BC DOFs are valid.

Parameters:

```
num_nodes (int):
Total number of FEM nodes.

num_dofs (int):
Number of dofs per node, element dependent (bar=1dof, beam=2dofs, 2dbeam =
3dofs)

bc_nodes (list[int]):
Nodes with applied BCs.

bc_dofs (list[int]):
Corresponding DOFs constrained.
```

Raises:

```
ValueError if:

    - DOFs referenced exceed available DOFs.

    - The system is under-constrained (<3 constrained DOFs).

    - The system is over-constrained (all DOFs fixed).
```

Function: `plot_fem_model(node_coords, fem_elements, bc_nodes, bc_dofs, bc_vals, force_nodes, force_dofs, force_vals, scale_force=0.1, scale_moment=0.1)`

Plots the FEM mesh with boundary conditions and applied forces.

Function parameters:

```
node_coords (ndarray, shape (N, 2)):
Coordinates of FEM nodes.
```

```
fem_elements (ndarray, shape (E, 2)):
Connectivity of FEM elements.

bc_nodes (list[int]):
Nodes with BCs.

bc_dofs (list[int]):
DOFs constrained.

bc_vals (list[float]):
Values of BCs.

force_nodes (list[int]):
Nodes with forces.

force_dofs (list[int]):
DOFs with applied forces.

force_vals (list[float]):
Force magnitudes.

scale_force (float, default=0.1):
Scaling factor for force arrow length.

scale_moment (float, default=0.1):
Scaling factor for moment visualization.
```

Returns:

```
(None): Displays a matplotlib plot with:

    FEM elements (black lines).

    Nodes (red dots with blue labels).

    BCs as colored squares (blue, green, magenta).

    Forces as arrows or moment arcs.
```

Module 5: femsolver.py (core FEM functionality)

Contains core functions to assemble global FEM matrices, perform static and modal analysis, and build the global force vector for 2D frame structures.

Function: `dof_map(ndofs_per_node, n1, n2)`

Returns the list of global degrees of freedom (DOF) indices for an element defined by its two node indices.

The function adapts the mapping depending on the number of DOFs per node, supporting:

- 1 DOF per node: axial bar element
- 2 DOFs per node: Euler-Bernoulli beam element (v, θ)
- 3 DOFs per node: 2D frame element (u, v, θ)

Function parameters:

```
ndofs_per_node (int):
Number of DOFs per node (1, 2, or 3).

n1, n2 (int):
Indices of the two nodes forming the element.
```

Returns:

```
dof_map (list of int):
List of global DOF indices corresponding to the element's local DOFs.
```

Function: `assembleK(k_local_func, rotation_func, fem_nodes, fem_elements, element_crossSections, mat_properties, ndof_per_node)`

Assembles the global stiffness matrix for a 2D frame FEM structure.

Function parameters:

```
k_local_func (callable):
Function returning the local element stiffness matrix, signature (E, A, I,
L) -> ndarray.

rotation_func (callable):
Function returning the rotation matrix transforming local to global coords.

fem_nodes (ndarray, shape (N, 2)):
Node coordinates.

fem_elements (list of tuples):
Connectivity list of node pairs per element.

element_crossSections (list of dicts):
Each with 'A' (area) and 'I' (moment of inertia).

mat_properties (dict):
Material properties, e.g., { 'E': Young's modulus }.

ndof_per_node (int):
Degrees of freedom per node (e.g., 3 for 2D beam).
```

Returns:

```
    K_global (ndarray):
    Global stiffness matrix of shape (ndof, ndof).
```

Function: `assembleM(m_local_func, rotation_func, fem_nodes, fem_elements, element_crossSections, mat_properties, ndof_per_node)`

Assembles the global mass matrix for a 2D frame FEM structure.

Function parameters:

```
m_local_func (callable):
Function returning the local element mass matrix, signature (rho, A, L) ->
ndarray.

rotation_func (callable):
Rotation matrix function.

fem_nodes (ndarray):
Node coordinates.

fem_elements (list of tuples):
Connectivity.

element_crossSections (list of dicts): E
ach with 'A', 'I'.

mat_properties (dict):
Material properties with density 'rho'.

ndof_per_node (int):
DOFs per node.
```

Returns:

```
M_global (ndarray):
Global mass matrix.
```

Function: `static_analysis(K_global, f_ext, bc_nodes, bc_dofs, bc_values, ndof_per_node)`

Solves static displacement problem $ Ku=f $ considering prescribed boundary conditions.

Function parameters:

```
K_global (ndarray):
Global stiffness matrix.

f_ext (ndarray):
Global external force vector.

bc_nodes (list[int]):
Nodes with prescribed BCs.

bc_dofs (list[int]):
DOFs constrained at those nodes.

bc_values (list[float]):
Prescribed displacement values.

ndof_per_node (int):
DOFs per node.
```

Returns:

```
u (ndarray):
Full displacement vector including constrained DOFs.
```

Function: `modal_analysis(K_global, M_global, bc_nodes, bc_dofs, ndof_per_node, num_modes=5)`

Computes natural frequencies and mode shapes by solving generalized eigenvalue problem.

Function parameters:

```
K_global (ndarray):
Global stiffness matrix.

M_global (ndarray):
Global mass matrix.

bc_nodes (list[int]):
Nodes with constraints.

bc_dofs (list[int]):
DOFs constrained.

ndof_per_node (int):
DOFs per node.
```

```
num_modes (int):
Number of modes to compute.
```

Returns:

```
frequencies_hz (ndarray):
Natural frequencies (Hz).

mode_shapes (ndarray):
Mode shapes (columns = modes).

free_dofs (ndarray):
Indices of unconstrained DOFs.
```

Function: `build_force_vector(f_nodes, f_dofs, f_values, num_dofs, dofs_per_node)`

Constructs the global force vector from nodal forces/moments.

Function parameters:

```
f_nodes (list[int]):
Nodes where forces/moments applied.

f_dofs (list[int]):
DOFs at which forces applied.

f_values (list[float]):
Force/moment magnitudes.

num_dofs (int):
Total DOFs in system.

dofs_per_node (int):
DOFs per node.
```

Returns:

```
f_ext (ndarray):
Global force vector.
```

Module 6: postprocessing.py (post-processing and plotting)

Function: `eval_stress(k_local_func, R_func, u, fem_elements, fem_nodes, element_sections, material, section_shape)`

Computes axial, bending, shear, and von Mises stresses for each frame element.

Function parameters:

```
k_local_func (function):
Returns the 6x6 local stiffness matrix for the element given (E, A, I, L).

R_func (function):
Returns the rotation matrix from global to local coordinates using node
positions.

u (np.ndarray):
Global displacement vector [3 * n_nodes].

fem_elements (np.ndarray, shape (n_elements, 2)):
Connectivity matrix, each row [node1, node2].

fem_nodes (np.ndarray, shape (n_nodes, 2)):
Node coordinates [x, y].

element_sections (list/dict):
Per element properties including 'A' (area) and 'I' (moment of inertia).

material (dict):
Material properties with keys 'E' (Young’s modulus) and 'rho' (density).

section_shape (str):
Cross-section shape; "rectangle" or "circular".

ndof_per_node (int):
DOFs per node.
```

Returns:

```
stress_max (np.ndarray):
Maximum absolute normal stress per element (axial + bending).

stress_axial (np.ndarray):
Axial stress = N / A per element.

stress_bending (np.ndarray):
Bending stress = M * c / I per element.

stress_shear (np.ndarray):
Estimated max shear stress per element (section dependent).
```

```
von_mises_stress (np.ndarray):
Von Mises equivalent stress per element.
```

Function: `compute_reaction_forces(K_global, U, bc_nodes, bc_dofs)`

Computes reaction forces at constrained degrees of freedom.

Function parameters:

```
K_global (np.ndarray):
Global stiffness matrix.

U (np.ndarray):
Global displacement vector including constrained DOFs.

bc_nodes (list[int]):
Nodes with prescribed boundary conditions.

bc_dofs (list[int]):
DOFs per node with BCs (e.g., 0 for ux, 1 for uy, 2 for rotation).
```

Returns:

```
reactions (np.ndarray):
Reaction forces at constrained DOFs.

constrained_dof_ids (list[int]):
Indices of constrained DOFs.
```

Function: `plot_internal_actions_2D(beam_connectivity, fem_nodes, fem_elements, internal_actions, elements_per_beam=5)`

Plots internal action diagrams (Axial Force, Shear Force, Bending Moment) along each beam in a 2D structure.

Function parameters:

```
beam_connectivity (np.ndarray):
(n_beams, 2) array defining how beams are connected in the coarse model.

fem_nodes (np.ndarray):
(n_fem_nodes, 2) array of FEM node coordinates.

fem_elements (np.ndarray):
(n_elements, 2) array defining connectivity of FEM elements.
```

```
internal_actions (np.ndarray):
(n_elements, 3) array of internal actions [N, V, M] per element.

elements_per_beam (int, optional):
Number of FEM elements that make up each beam (default = 5).
```

Returns:
None – displays Matplotlib plots for each beam.

Function: `plot_internal_actions_1D(fem_nodes, fem_elements, internal_actions, dof_per_node)`

Plots internal forces or moments for 1D FEM models (bar or beam elements) along the X-axis.

Function parameters:

```
fem_nodes (np.ndarray):
(n_nodes, 2) array of node coordinates (only X is used).

fem_elements (np.ndarray):
(n_elements, 2) array of node indices for each element.

internal_actions (np.ndarray):
(n_elements, N) array of internal actions per element:
    - N = 1 for axial force (1 DOF)
    - N = 2 for shear force and bending moment (2 DOFs)

dof_per_node (int):
Either 1 (for bar elements) or 2 (for beam elements).
```

Returns:
None – displays Matplotlib plots of internal actions.

Function: `plot_2D_loaded_structure(fem_nodes, fem_elements, u, stress_values, stress_type, ndof_per_node, scale=1.0, title=None, show_labels=False)`

Plots the undeformed and deformed 2D structure with a color map representing stress (e.g., von Mises, axial, shear, or bending).

Function parameters:

```
fem_nodes (np.ndarray):
(N, 2) array of node coordinates.

fem_elements (np.ndarray):
```

```
(E, 2) array of element connectivity.

u (np.ndarray):
Global displacement vector (flattened).

stress_values (np.ndarray):
(E,) array of stress values per element.

stress_type (str):
One of 'von Mises', 'axial', 'bending', 'shear'.

ndof_per_node (int):
Number of degrees of freedom per node (e.g., 2 for 2D: ux, uy).

scale (float, optional):
Scaling factor for visualizing deformation (default = 1.0).

title (str, optional):
Custom plot title (default is auto-generated).

show_labels (bool, optional):
If True, stress values are shown at element midpoints (default = False).
```

Returns:
None – displays a Matplotlib figure showing undeformed and deformed structure.

Function: `plot_1d_loaded_structure(fem_nodes, fem_elements, u, stress_values, stress_type, ndof_per_node, scale=1.0, title=None, show_labels=False)`

Plots a 1D structure (e.g., bar or beam) with its deformed shape and stress distribution along the X-axis.

Function parameters:

```
fem_nodes (np.ndarray):
(N, 2) array of node coordinates.

fem_elements (np.ndarray):
(E, 2) array of element connectivity.

u (np.ndarray):
Global displacement vector.

stress_values (np.ndarray):
(E,) array of stress values per element.

stress_type (str):
Type of stress shown (e.g., 'axial', 'bending', etc.).

ndof_per_node (int):
```

```
Number of degrees of freedom per node (e.g., 1 or 2).

scale (float, optional):
Magnification factor for visualizing displacements (default = 1.0).

title (str, optional):
Plot title (default is auto-generated).

show_labels (bool, optional):
If True, stress values are labeled at element midpoints (default = False).
```

Returns:
None – displays a Matplotlib figure of undeformed and deformed geometry with stress colors.

Function: `animate_mode_shape_2D(mode_index, eigenvecs, node_coords, free_dofs, K, elements, ndof_per_node, amplification=100, save_as=None)`

Animates a mode shape for a 2D frame structure using sine-based oscillation and displays or saves it.

Function parameters:

```
mode_index (int):
Index of the mode to animate (0-based).

eigenvecs (np.ndarray):
Matrix of eigenvectors (from reduced eigenvalue problem).

node_coords (np.ndarray):
(N, 2) array of nodal coordinates in 2D.

free_dofs (list[int]):
List of free DOF indices from the system.

K (np.ndarray):
Global stiffness matrix (used to infer total number of DOFs).

elements (np.ndarray):
(E, 2) array of element connectivity.

ndof_per_node (int):
Degrees of freedom per node (e.g., 3 for 2D frame: ux, uy, rotation).

amplification (float, optional):
Scaling factor for visual deformation (default = 100).

save_as (str, optional):
If provided, saves the animation to the given filename (e.g., "mode1.gif").
```

Returns:

```
animation (IPython.display.HTML):
HTML5 animation for use in Jupyter notebooks, unless saved to file.
```

Function: `animate_mode_shape_1D(mode_index, eigenvecs, node_coords, free_dofs, K, elements, ndof_per_node, amplification=100, save_as=None)`

Animates a mode shape for 1D structures (bars or beams), with optional saving and scaling.

Function parameters:

```
mode_index (int):
Index of the mode to animate (0-based).

eigenvecs (np.ndarray):
Array of eigenvectors from the reduced system.

node_coords (np.ndarray):
(N, 2) array of node positions (X, Y).

free_dofs (list[int]):
Indices of the free degrees of freedom.

K (np.ndarray):
Global stiffness matrix (used for total DOF count).

elements (np.ndarray):
(E, 2) array of node indices per element.

ndof_per_node (int):
Number of DOFs per node (1 for bar, 2 for beam).

amplification (float, optional):
Factor for visual magnification of mode shape (default = 100).

save_as (str, optional):
If set, saves the animation to a file (e.g., "mode1.gif").
```

Returns:

```
animation (IPython.display.HTML):
HTML animation for use in Jupyter notebooks (if not saved to file).
```
