SVV Structures Assignment 2021 – 2022

AE3212-II Simulation, Verification and Validation course
Written by dr. ir. J.M.J.F. van Campen
February 2022

Background

The design of spacecraft structures is an engineering challenge. Not only are spacecraft structures subjected to an environment that is difficult to simulate on earth, but a spacecraft structure must perform as predicted within small tolerances. The recently launched James Webb telescope provides an example. The thermal deformation of the composite structure supporting the mirrors (fig. 1) is allowed to contribute up to a few nanometers to the so-called Wave Front Error of the telescope infrared observations [1]. For general spacecraft structure, further complications are extreme structural requirements in terms of lightweight and foldable design that must withstand harsh vibrational loads during launch, and significant thermal loads once in orbit. Proper understanding of simulation, verification and validation (SVV) is therefore paramount for economically viable spaceflight.

The above is the background for the AE3212-II SVV structures assignment in front of you. For the purpose of this assignment, we pretend that you are a group of engineers running your own company in spacecraft structures, which is a subcontractor for Spacecraft structures Project (SpacePro). SpacePro is developing a new satellite. Your company is asked to design the structure of an antenna dish. The antenna dish will be connected to the spacecraft which is designed by SpacePro. Note, that the structures, loads and dimensions used in this assignment are fictional.

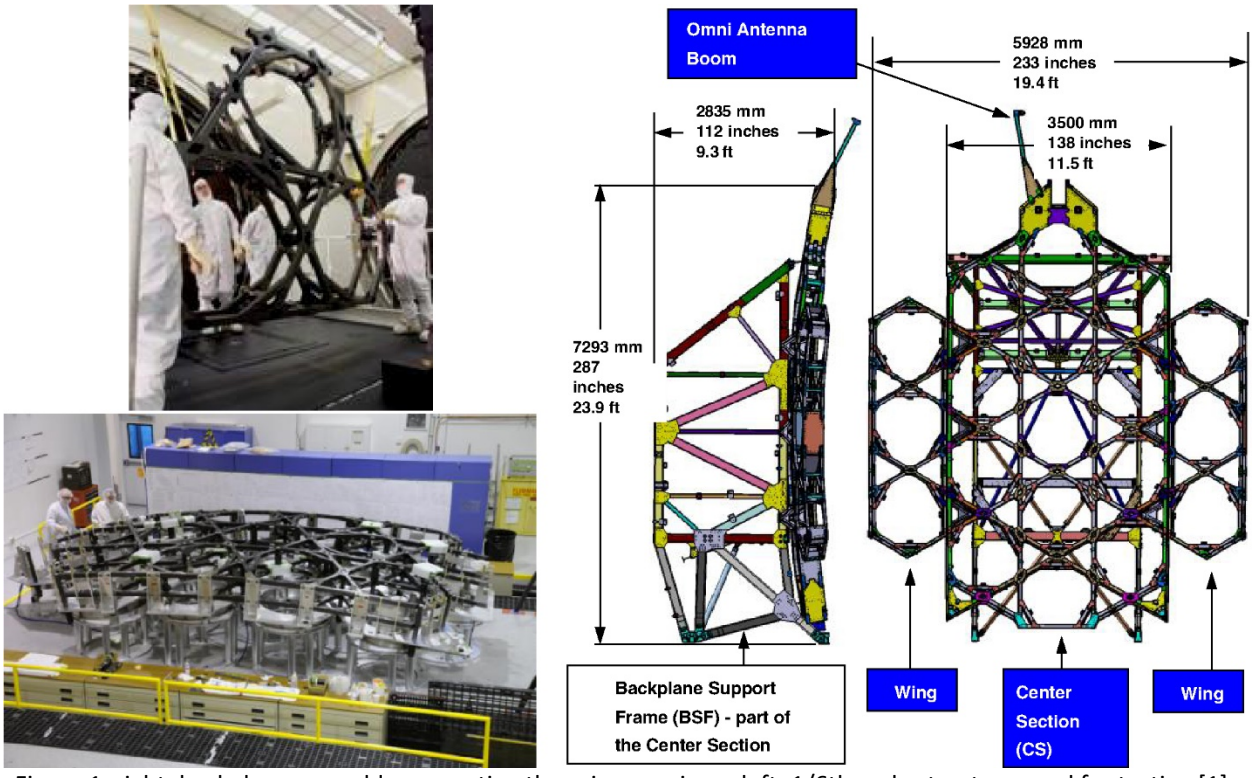

Figure 1: right: backplane assembly supporting the primary mirror; left: 1/6th scale structure used for testing [1].

Figure 2: SpacePro logo

They have been chosen to simplify the complexity of spacecraft structures to a level where the simulation, verification and validation of a structural design tool come into focus, rather than the complex structure itself.

Content

1. Introduction

SpacePro wants your team to complete a new Python code of a preliminary design tool and to perform the necessary verification and validation tests.

The preliminary design tool is called MechRes2D, which is short for Mechanical Response 2D. The design tool simulates the mechanical response of a 2D cross-section of the structure of an antenna dish.

MechRes2D can simulate two types of structures:

- Beam structures
- Truss structures

The shape of the antenna dish influences its performance. Therefore, deformations must remain within prescribed bounds. The code takes as input the geometrical design of the structure and materials selected from a user database. The code gives the following outputs:

- Nodal displacements
- Element axial stresses
- Element bending moment (if applicable)
- First 5 eigen frequencies of the structure

For validation, the test results of a simple mechanical experiment are available.

2. Specifications of the Antenna Dish Structure

There are two main types of structure for which SpacePro uses MechRes2D, (fig. 3). For the beam type structure (fig. 3.a), only the dish itself (circular part) is subjected to an equal thermal load, whereas the flexures (indicated in green) it is resting on are not. For the truss type structure (fig 3.b) the entire structure is subjected to an equal thermal load.

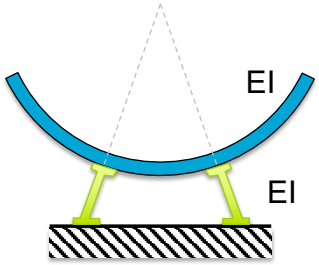

Figure 3.a: Beam type

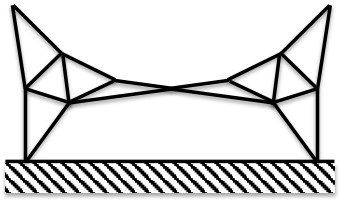

Figure 3.b: Truss Type

Figure 3: Two types of structure for which MechRes2D is used mainly

There is a fixed parametrization scheme (fig. 4). for the two types of structures discussed above. Typical parameter sets used for both types of structure are given in Table 1.

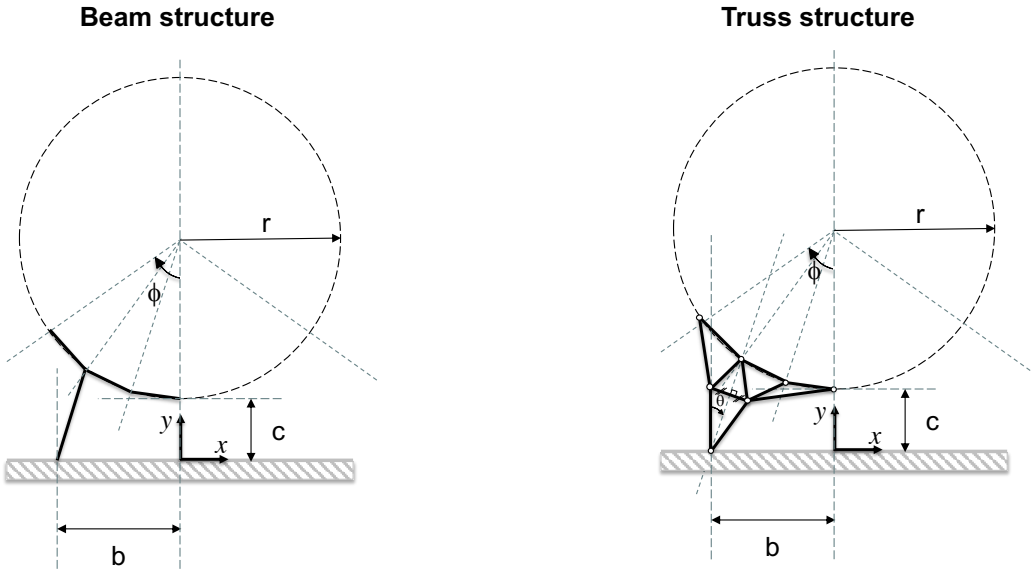

Figure 4: Discretization of the two types of antenna dish structure

Table 1: typical parameter values for antenna dish structure

	Set 1	Set 2	Set 3
b [mm]	700	850	350
c [mm]	600	300	350
r [mm]	2400	5000	1100
φ [°]	30	15	20
Material[1]	1	2	3

[1] The material database that is referred to in Table 1 is hardcoded in MechRes2D.

3. Specifications MechRes2D

Most of the theory used in MechRes2D can be found in Aircraft Structures for Engineering Students by Megson, fifth or sixth edition [2]. Especially chapter 6 is relevant for the assignment. Knowledge of the AE2135-II Vibrations course will be useful too. In this section the element types and the solution procedures that have been implemented in MechRes2D will be described.

Please note that, in this document all matrices will be indicated with square brackets and all vectors will be indicated with curly brackets. Furthermore, please note, that the equations for the beam element in this document have been taken from Appendix A of Gan [3]. The description of the matrix method in Megson [2], is more than sufficient to use these equations.

Element Types

The user of MechRes2D can choose between two element types: beam elements or rod elements.

Euler-Bernoulli Straight Beam

A graphical description of the Euler-Bernoulli straight beam is given (fig. 5).

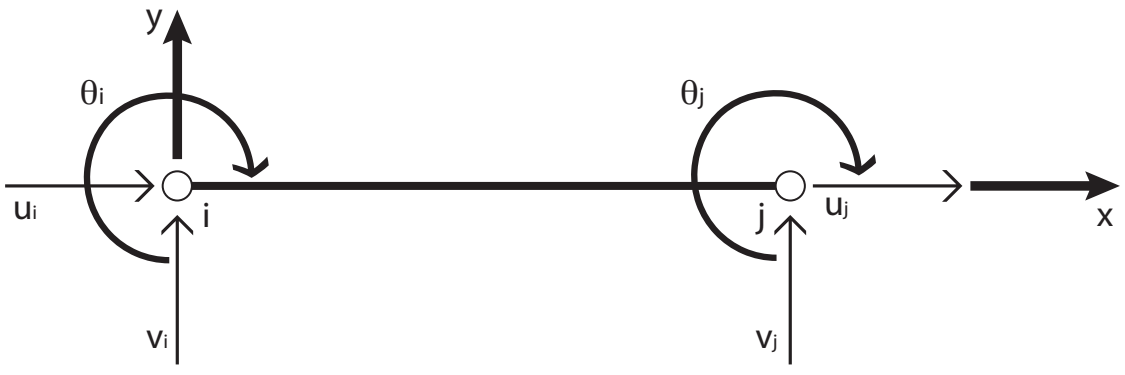

Figure 5: Euler-Bernoulli Straight Beam, modified from Megson [2]

Assumptions made in the description of the Euler-Bernoulli straight beam are the following:

- Plane sections of the cross-section of the beam remain plane
- The slopes of the deformed beams are small, i.e. the angles remain small

The stiffness matrix of the Euler-Bernoulli straight beam is given in equation 1. Where E, A, I and L refer to the properties of the element.

$$\left[\overline{K}_{ij}\right] = \begin{bmatrix} \frac{EA}{L} & 0 & 0 & -\frac{EA}{L} & 0 & 0 \\ 0 & 12EI/L^3 & 6EI/L^2 & 0 & -12EI/L^3 & 6EI/L^2 \\ 0 & 6EI/L^2 & 4EI/L & 0 & -6EI/L^2 & 2EI/L \\ -\frac{EA}{L} & 0 & 0 & \frac{EA}{L} & 0 & 0 \\ 0 & -12EI/L^3 & -6EI/L^2 & 0 & 12EI/L^3 & -6EI/L^2 \\ 0 & 6EI/L^2 & 2EI/L & 0 & -6EI/L^2 & 4EI/L \end{bmatrix} \quad (1)$$

The above stiffness matrix is given in the local coordinate system of the element and must be rotated to the global coordinate system of the structure. The rotation matrix is given in equation 2. Where $\lambda = \cos\theta$ and $\mu = \sin\theta$. Angle θ is the rotation angle of the element w.r.t the x-axis of the global coordinate system. This angle should not be confused with the rotational degree of freedom θ_i (fig. 5).

$$[T] = \begin{bmatrix} \lambda & \mu & 0 & 0 & 0 & 0 \\ -\mu & \lambda & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda & \mu & 0 \\ 0 & 0 & 0 & -\mu & \lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (2)$$

The contribution to the global stiffness matrix of the element follows from the transformation described in equation 3.

$$\left[K_{ij}\right] = [T]^{\mathrm{T}}\left[\overline{K}_{ij}\right][T] \quad (3)$$

The mass matrix of the beam element is given in equation 4.

$$[m] = \left[m_{\rho A}\right] + \left[m_{\rho I}\right] \quad (4)$$

Where:

$$\left[m_{\rho A}\right] = \frac{\rho AL}{420}\begin{bmatrix} 140 & 0 & 0 & 70 & 0 & 0 \\ 0 & 156 & 22L & 0 & 54 & -13L \\ 0 & 22L & 4L^2 & 0 & 13L & -3L^2 \\ 70 & 0 & 0 & 140 & 0 & 0 \\ 0 & 54 & 13L & 0 & 156 & -22L \\ 0 & -13L & -3L^2 & 0 & -22L & 4L^2 \end{bmatrix}$$

and

$$\left[m_{\rho I}\right] = \frac{\rho I}{30L}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 36 & 3L & 0 & -36 & 3L \\ 0 & 3L & 4L^2 & 0 & -3L & -L^2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -36 & -3L & 0 & 36 & -3L \\ 0 & 3L & -L^2 & 0 & -3L & 4L^2 \end{bmatrix}$$

Rod Element

A rod element as can be found in a pin-jointed framework or truss structure is shown (fig. 6). The element stiffness matrix of a rod in a pin-jointed framework is given in equation 5. The main assumption behind the rod element is that it can only transfer axial loads.

Figure 6: Rod element, modified from Megson [2]

$$[\overline{K}_{ij}] = \frac{EA}{L}\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{5}$$

The rotation matrix for the rod element is given in equation 5. Where $\lambda = \cos\theta$ and $\mu = \sin\theta$. Angle θ is of course the rotation angle of the element. The rotation itself is performed using equation 3, which results in the addition of the element to the global stiffness matrix of the structure.

$$[T] = \begin{bmatrix} \lambda & \mu & 0 & 0 \\ -\mu & \lambda & 0 & 0 \\ 0 & 0 & \lambda & \mu \\ 0 & 0 & -\mu & \lambda \end{bmatrix} \tag{5}$$

The mass matrix of the rod element is given in equation 6.

$$[m] = \frac{\rho AL}{6}\begin{bmatrix} 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix} \tag{6}$$

Alternatively, the mass for rod elements can be lumped in the nodes. The lumped mass matrix is given in equation 7. Typically, it is used to reduce the computational effort required to evaluate

the eigenfrequencies of the system, however this does not consider some mass inertia effects. Better results will be obtained by dividing a structure into more elements.

$$[\bar{m}] = \frac{\rho AL}{2}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{7}$$

Loading Vectors

The MechRes2D programme allows for two types of load vectors, both of which can only be applied at the nodes of the model.

First of all, the nodal loads due to thermal expansion, given in equation 8 for the Euler-Bernoulli beam element, where α is the thermal expansion coefficient. For the rod element the third entry for each of the two nodes is simply removed.

$$\{\vec{Q}\} = EA\alpha\Delta T\begin{Bmatrix}1\\ 0\\ 0\\ -1\\ 0\\ 0\end{Bmatrix} \tag{8}$$

Next to the thermal load vector, nodal loads and moments can be applied as well.

Analysis

MechRes2D performs a linear elastic analysis, described in equation 9. This equation expresses the static equilibrium of the system. Vector $\{\vec{P}\}$ contains all applied nodal loads, vector $\{\vec{Q}\}$ contains the thermal nodal loads, and vector $\{\vec{R}\}$ contains the nodal reaction forces, and vector $\{\vec{U}\}$ contains all nodal displacements.

$$[K]\{\vec{U}\} = \{\vec{P}\} - \{\vec{Q}\} + \{\vec{R}\} \tag{9}$$

The equilibrium equation can be arranged in the following way:

$$\begin{bmatrix}K_r & K_{rs}\\ K_{sr} & K_s\end{bmatrix}\begin{Bmatrix}\overrightarrow{U_r}\\ \overrightarrow{U_s}\end{Bmatrix} = \begin{Bmatrix}\overrightarrow{P_r}\\ \overrightarrow{P_s}\end{Bmatrix} - \begin{Bmatrix}\overrightarrow{Q_r}\\ \overrightarrow{Q_s}\end{Bmatrix} + \begin{Bmatrix}\overrightarrow{R_r}\\ \overrightarrow{R_s}\end{Bmatrix} \tag{10}$$

Where r indicates the reduced equation, $\{\overrightarrow{R_s}\}$ is the vector with all reaction forces. Reaction forces only exist where boundary conditions are specified, because of the third law of Newtown. In MechRes2D boundary conditions fully constrain a degree of freedom. As a result, $\{\overrightarrow{U_s}\} = \{\vec{0}\}$. The reduced displacement vector $\{\overrightarrow{U_r}\}$ contains the displacements of all nodes that have not been constrained.

The reaction forces and displacements are calculated by equations 11 and 12.

$$\{\overrightarrow{R_s}\} = [K_{sr}]\{\overrightarrow{U_r}\} - (\{\overrightarrow{P_s}\} - \{\overrightarrow{Q_s}\}) \quad (11)$$

$$\{\overrightarrow{U_r}\} = [K_r]^{-1}(\{\overrightarrow{P_r}\} - \{\overrightarrow{Q_r}\}) \quad (12)$$

The equations of motion for the constrained system are given in equation 13.

$$[m]\{\ddot{\vec{U}}\} + [K]\{\vec{U}\} = \{\vec{P}\} - \{\vec{Q}\} + \{\vec{R}\} \quad (13)$$

The eigenfrequencies of the constrained system follow from solving equation 14.

$$\left|\frac{1}{\omega^2}[I_r] - [K_r]^{-1}[m_r]\right| = 0 \quad (14)$$

Where $[I_r]$ is an identity matrix with the size of the reduced system. MechRes2D will return the first 5 solutions.

Element Strains

Using the nodal displacement vector $\{\vec{U}\}$ the length of each element in deformed state can be calculated. Using equation 15 the strain of each element is calculated in MechRes2D , where x refers to the local coordinate system of the element. This equation is known as engineering strain.

$$\varepsilon_x = \frac{\Delta L}{L} \quad (15)$$

Element Stresses

Using the stress strain relationship described in equation 16 the element stresses are calculated in MechRes2D, where x refers to the local coordinate system of the element. In case beam elements are being used the stresses that occur due to bending are ignored.

$$\sigma_x = E\varepsilon_x \quad (16)$$

Code Structure

The numbering used in the flow-diagram of MechRes2D (fig. 7) is identical to the numbering used in the comments of MechRes2D, see appendix A. Please use this numbering the navigate through MechRes2D more easily.

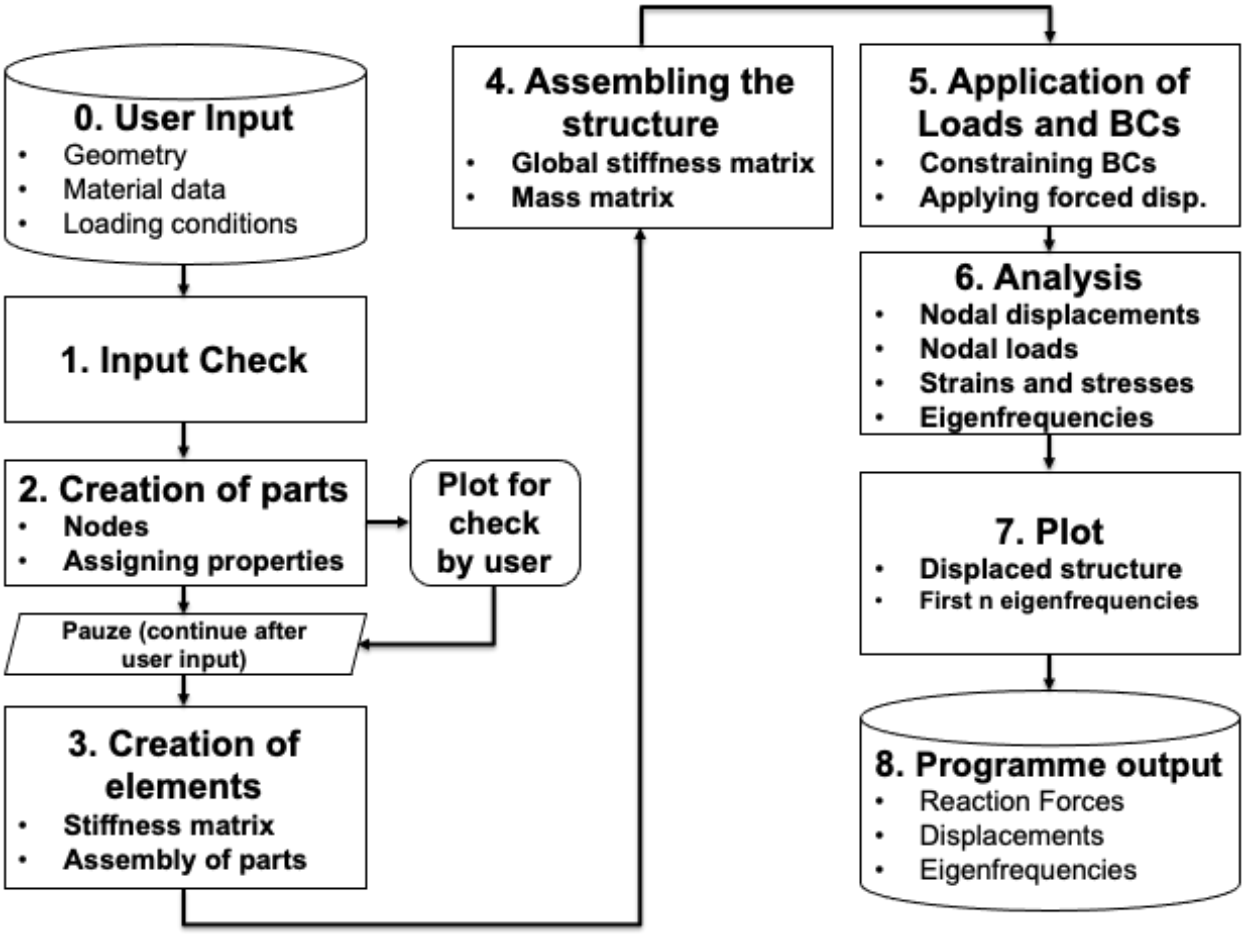

Figure 7: Flow diagram MechRes2D

0. User Input

Your team will be provided with the code MechRes2D.py. This script contains two functions MechRes2D and ConstGeom. ConstGeom constructs an input geometry from the variable given in Table 1. It is not necessary to use the function ConstGeom. The user can also input a structure manually, or using a self-written function similar to ConstGeom. More on this will be explained below.

Running MechRes2D

To run the code compile the file MechRes2D.py. The last line in the file, see appendix A, will run the programme MechRes2D:

```
inp = []; out = MechRes2D(inp)
```

Please note that the provided code has been tested to work correctly in Spyder version 4.1.5 and PyCharm version 2021.1.3 and 20212.2. When using PyCharm make sure that the *Run with Python console* option is checked.

Input of variables

The user of MechRes2D can specify the following inputs:

- **Points**

  2D coordinates of points in the global coordinate system

  2xn matrix, where n is the number of points specified. In the matrix x-coordinates are placed on the first row, y-coordinates on the second row.

- **Parts**

  Lines connecting specified points

  2xn matrix, where n is the number of parts specified. In the matrix the index of the first point defining a part (referring to the matrix with points) is placed on the first row. The second point is placed on the second row. A line will be constructed between the two points.

- **Type**

  Element type assigned to part

  1xn vector, where n is the number of parts specified. The entries in the matrix can be either 0 or 1. Rod elements are specified with 0. Beam elements are specified with 1.

- **Seed**

  Number of nodes between the two points defining an element.

  1xn vector, where n is the number of parts specified. The integer number m for each part in the vector specifies the number of nodes between the two end points for that part. The points defined are automatically converted to nodes. Therefore, the total number of nodes for a part is m + 2. The seeded nodes have equal distances from one another. For each part in the structure a different seed number can be specified.

  Please note that it is only possible to seed nodes for parts that have been specified to consist of beam elements. The option has been disabled for rod elements, because rod elements can only carry axial loads. Therefore, seeding a rod element would lead to a structural instability, and an ill-defined system of equations.

- **Material**

  Material assigned to part

1xn vector, where n is the number of parts specified. The matrix contains integer numbers that refer to a material card, which has been hardcoded in the MechRes2D. For example, 2 refers to material card 2.

- **Section**

  Cross-section assigned to part

  1xn vector, where n is the number of parts specified. The matrix contains integer numbers that refer to a cross-section card, which has been hardcoded in the MechRes2D. For example, 2 refers to cross-section card 2. A cross-section card specifies the cross-sectional area and moment of inertia.

- **BC**

  Boundary Conditions

  1x3n vector, where n is the number of points specified. Can be used to specify the boundary conditions of the points that have been defined by the user. Every point is automatically converted to a node. Every node has three degrees of freedom: translation in global x-direction, translation in global y-direction and rotation about the global z-axis. Please note, that a right-handed Cartesian coordinate system is used. For every degree of freedom either a 0 or 1 can be specified. 0 means that a degree of freedom is not constrained. 1 means that a degree of freedom is fully constrained.

- **Sym**

  Symmetry condition

  This instruction is specific to ConstGeom. Can be specified as either 0 or 1. 0 means that the structure is not mirrored. 1 means that the structure is mirrored horizontally about the first point that has been specified. The symmetry condition dictates that the node of the first point can only move in y-direction, its to other degrees of freedom are constrained.

- **T**

  Temperature field

  1xn vector, where n is the number of points specified. The vector specifies the temperature elevation in K of each point. The thermal load vector is only calculated for parts where both end points have an equal elevated temperature. In other words, MechRes2D only can simulated parts having a constant elevated temperature, but no temperature gradients.

- **Name**

  String containing the name of the structure analysed

  The user can specify a name for the structure to be analyzed. The name will be used to save the output and the plots.

- **plotScale**

  Factor to scale plotted displacements

  Continues number that is used to scale the plotted deformations. When set to 1 the displacements will be plotted in their actual size. Selecting a higher number can be used to exaggerate deformations, such that way the structure deforms can be studied more easily.

1. Input Check

The first block of the programme checks the provided input. In case it is not complete, or the format is not correct, a warning will be issued to the user and the programme will be terminated.

2. Creation of Parts

The second block of the code creates the parts that have been specified by the user. All parts are straight lines and can only exist between user defined points. The resulting structure is plotted for inspection by the user.

3. Creation of Elements

The third block of the code creates either beam or rod elements based on the user defined input. A unit vector is calculated for the part, which is used to calculate the coordinates of each node in the part. The nodes are plotted for inspection by the user.

Next to the calculation of the coordinates of the nodes, also the properties of the elements are calculated, such as length and lumped nodal mass. Furthermore, the properties specified by the user are assigned to elements. The local stiffness matrix of each element is calculated and converted to the global coordinate system. In a similar fashion the mass matrix and thermal load vector for each element will be calculated.

4. Assembling the Structure

The fourth block of the code assembles all stiffness and mass matrices of all elements into a global stiffness and a global mass matrix. Furthermore, the thermal load vector is assembled into a thermal load vector for the global system.

5. Application of Loads and Boundary Conditions

In the fifth block of the code, the global nodal load vector is initialized. The loads applied to the user specified points are added to the global nodal load vector. Furthermore, the displacement vector is initialized, as well as the vector for nodal reaction forces. The user specified boundary conditions are then used to apply equation 10.

6. Analysis

In the sixth block of the code equations 11, 12 and 14 are applied. Equations 15 and 16 still have to be implemented

7. Plotting

In the seventh block of the code, the structure is plotted again in gray, on top of which the displacements that were calculated are plotted over the structure. This way the user can get an insight in the predicted deformation of the structure.

8. Programme Output

In the final block of the code the generated output and the generated figures should be stored. This has not been implemented yet.

4. Available Experimental Data

SpacePro provides your group with two sets of experimental results. One set of results for a beam structure and one set of results for a truss structure.

Beam Structure

The results of a simple mechanical experiment are available, a three-point bending test (fig. 8).

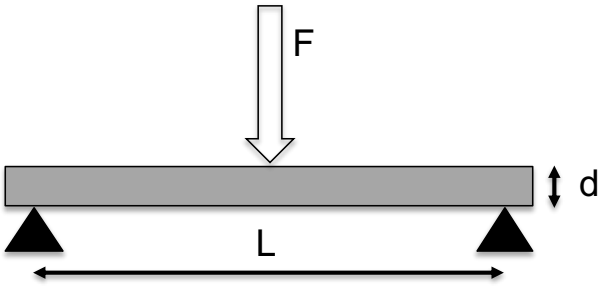

Figure 8: Set-up of mechanical experiment, three-point bending test

For the experiment the following parameters have been used:

F = 1000N ±0.5N
d = 10.4mm ±0.1mm
L = 150mm ±0.5mm

Where the second number is the standard deviation. Material number 1 and section 1 as hardcoded in MechRes2D have been used for the experiment. The experiment has been repeated 5 times. The results are given in Table 2.

Table 2: Experimental results of three-point bending test

Experiment number	Downward displacement center [mm]
1	0.100487
2	0.101454
3	0.100757
4	0.100316
5	0.100698

Truss Structure

The results of a test on a very simple truss structure are available (fig. 9).

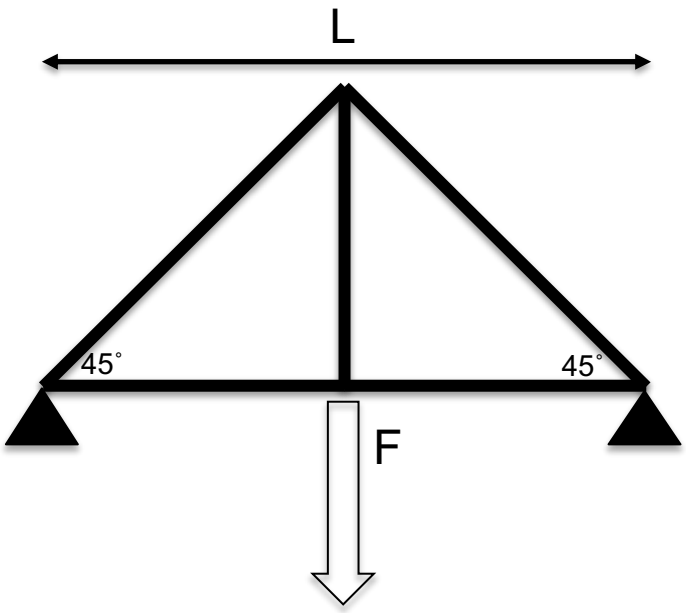

Figure 9: Set-up of mechanical experiment, truss structure

For the experiment the following parameters have been used:

F = 1000N ±0.5N
L = 200mm ±0.5mm

Where the second number is the standard deviation. Material number 2 and section 2 as hardcoded in MechRes2D have been used for the experiment. The experiment has been repeated 5 times. The results are given in Table 3.

Table 3: Experimental results of truss structure experiment

Experiment number	Downward displacement center [mm]
1	0.0034991
2	0.0034714
3	0.0034921
4	0.0034520
5	0.0034632

5. Structures Assignment

SpacePro asks your team to do the following:

1. Complete block 6 of the code, such that it calculates stresses and strains, equations 15 and 16. Additionally, complete the saving of the output and the plots in block 8 (fig. 5) of the code.
2. Perform unit tests and subsystem tests on all parts of MechRes2D specified in the flow diagram (fig. 5).
   Tip: start from the assumptions made and the theory behind the part that you are testing.
3. Verify the correct functioning of MechRes2D as a whole. Make sure to focus on the two types of structure for which MechRes2D is mainly used (fig. 3 & 4, and Table 1).
4. Discuss the effect of the symmetry boundary condition.
5. Validate the correct functioning of MechRes2D with the available experimental data.
6. Reflect on the accuracy of the output of MechRes2D for beam structures and for truss structures. Is there a difference? What does this difference mean for the use of the code as a design tool?
7. Reflect on the use of MechRes2D as a design tool for antenna dish structures. Is it possible to use the code to make a fair comparison between a beam structure and a truss structure of equal weight and dimension?

SpacePro suggests splitting your team in two parts, where one half of the team focusses on the beam elements, and one focusses on the rod elements.

6. Reporting requirements

Verification and Validation Plan

In the verification and validation (V&V) plan you describe how your team will approach the structures assignment. Describe in the V&V plan each verification and validation test, that you would like to perform. Describe which plots you plan to provide. Be specific of what will be on the axes of these plots. Describe which tables you plan to provide. In short, the V&V plan should describe in what way you plan to perform the verification and validation of MechRes2D on different levels of the code. It should also describe what analysis you will perform to answer point 6. of the assignment. The V&V plan also must include a detailed task distribution, indicating which team member will work on which part of the planned simulation, verification and validation.

The V&V plan must fulfill all basic reporting requirements and should contain:

- Title page
- Table of contents
- Page numbers
- Introduction
- Numbered chapters
- Conclusions
- Task distribution

Please see Brightspace for the specifics regarding page limit, font-size and minimum clearings.

N.B.: The title page counts towards the specified page limit.

Structures Report

The structures report should describe the verification and validation of MechRes2D as has been carried out by the team. It should also contain a detailed task distribution indicating per team member how many hours were spent per task.

The structures report must fulfill all basic reporting requirements, i.e.:

- Title page
- Table of contents
- Page numbers
- Introduction
- Numbered chapters
- Summary and Conclusions
- Task distribution
- Appendices
- Appendices with code

Please see Brightspace for the specifics regarding page limit, font-size and minimum clearings.

N.B. 1: The title page counts towards the specified page limit.
N.B. 2: The regular appendices count towards the specified page limit.
N.B. 3: The appendices with code do not count towards the specified page limit.

References

[1] Lightsey, P.A., Atkinson, C.B., Clampin, M.C. and Feinberg, L.D., 2012. James Webb Space Telescope: large deployable cryogenic telescope in space. Optical Engineering, 51(1), p.011003.

[2] Megson, T.H.G., Aircraft Structures for Engineering Students, Fifth Edition, 2012.

[3] Gan, B.S., An Isogeometric Approach to Beam Structures, 1st Edition, 2018.

Appendix A: MechRes2D and ConstGeom Python Script

```
# -*- coding: utf-8 -*-


import numpy as np
import numpy.matlib as matlib
import matplotlib.pyplot as plt


def ConstGeom(inp):
    # This script constructs the geometry of the parts, as input for MechRes

    # angles dish
    phi1 = 0
    phi2 = inp['phi']/3/180*np.pi
    phi3 = 2*phi2
    phi4 = inp['phi']/180*np.pi
    # shared x-coordinates
    x1 = -inp['R']*np.sin(phi1)
    x2 = -inp['R']*np.sin(phi2)
    x3 = -inp['R']*np.sin(phi3)
    x4 = -inp['R']*np.sin(phi4)
    # shared y-coordinates
    y1 = inp['cl']+inp['R']*(1-np.cos(phi1))
    y2 = inp['cl']+inp['R']*(1-np.cos(phi2))
    y3 = inp['cl']+inp['R']*(1-np.cos(phi3))
    y4 = inp['cl']+inp['R']*(1-np.cos(phi4))
    # angle legs
    theta = np.arctan((inp['b']+x3)/y3)
    # create structure for selected type
    out = {}  # initialize output
    if inp['type']:
        # ---- beam structure ----
        # x-coordinates
        x5 = -inp['b']
        # y-coordinates
        y5 = 0
        # assemble points
        out['points'] = np.array([[x1, x2, x3, x4, x5], [y1, y2, y3, y4, y5]])
        out['parts'] = np.array([[1, 2, 3, 3], [2, 3, 4, 5]])
        # element type
        out['type'] = np.ones((max(out['parts'].shape)), dtype=int)
    else:
        # ---- rod structure ----
        # x-coordinates
        x5 = x3 + inp['cl']/2*np.cos(np.pi/3+theta)
        x6 = x3 - inp['cl']/2*np.cos(np.pi/3-theta)
        x7 = -inp['b']
        # y-coordinates
        y5 = y3 - inp['cl']/2*np.sin(np.pi/3+theta)
        y6 = y3 - inp['cl']/2*np.sin(np.pi/3-theta)
        y7 = 0
        # assemble points
        out['points'] = np.array(
            [[x1, x2, x3, x4, x5, x6, x7], [y1, y2, y3, y4, y5, y6, y7]])

        out['parts'] = np.array([[1, 2, 3, 1, 2, 3, 3, 4, 5, 5, 6], [
                                 2, 3, 4, 5, 5, 5, 6, 6, 6, 7, 7]])
        # element type
        out['type'] = np.zeros((max(out['parts'].shape)), dtype=int)

    # initialise seed for every part to 0
    out['seed'] = np.zeros((max(out['parts'].shape)))
    # initialise material for every part to 0
    out['material'] = np.zeros((max(out['parts'].shape)), dtype=int)
    # initialise section for every part to 0
    out['section'] = np.zeros((max(out['parts'].shape)), dtype=int)

    # create boundary conditions
    # 1 means d.o.f. is restrained, 0 means d.o.f. is free
    out['bc'] = np.zeros((3*max(out['points'].shape)))
    # final point is clamped
    out['bc'][-3:] = 1

    # create loading vector
    out['P'] = np.zeros((3*max(out['points'].shape), 1))

    # create thermal vector
    out['T'] = np.zeros((max(out['points'].shape), 1))
    if inp['type']:
        # first 4 nodes are exposed to deltaT
        out['T'][0:4] = inp['deltaT']*np.ones((4, 1))
    else:
        # all nodes are exposed to deltaT
        out['T'] = inp['deltaT']*np.ones((max(out['points'].shape)))

    # create mirror side of structure, if so selected
    if inp['sym']:
        # connectivity
        a = np.nonzero(out['parts'][0, :] == 1)[0]
        l = max(out['points'].shape)
        # points (first point should not be duplicated)
```

```
        out['points'] = np.vstack((np.concatenate((out['points'][0, :], -out['points'][0, 1:]), axis=0).reshape(
            1, 2*l-1), np.concatenate((out['points'][1, :], out['points'][1, 1:]), axis=0).reshape(1, 2*l-1)))
        # boundary conditions (first point should not be duplicated)
        out['bc'] = np.concatenate((out['bc'], out['bc'][3:]))
        # loading vector
        out['P'] = np.concatenate((out['P'], out['P'][3:]))
        # thermal vector
        out['T'] = np.concatenate((out['T'], out['T'][1:]))
        # advancing the node numbers to be connected apart for node 1
        c = (l-1)*np.ones((np.shape(out['parts'])), dtype=int)
        c[0, a] = np.zeros((1, len(a)), dtype=int)
        out['parts'] = np.concatenate((out['parts'], c+out['parts']), axis=1)
        # double the initialized vectors
        out['type'] = np.concatenate((out['type'], out['type']))
        out['seed'] = np.concatenate((out['seed'], out['seed']))
        out['material'] = np.concatenate((out['material'], out['material']))
        out['section'] = np.concatenate((out['section'], out['section']))
    else:
        # apply symmetry boundary conditions at point 1
        out['bc'][0] = 1   # 1 means d.o.f. is restrained, 0 means d.o.f. is free
        out['bc'][2] = 1

    # convert theta to degrees
    out['theta'] = theta*180/np.pi
    # store input data to output
    out['inp'] = inp
    out['mat'] = inp['mat']
    out['sec'] = inp['sec']
    out['usr'] = inp['usr']
    out['name'] = inp['name']
    out['plotScale'] = inp['plotScale']
    return out

    # TODO: write the input settings to a text file


def MechRes2D(inp):

    # ===============================================================================
    #  AE3212-II Structures assignment 2021-2022
    # ===============================================================================
    # ============================Start of preamble================================
    # This Python-function belongs to the AE3212-II Sumulation, Verfication and
    # Validation Structures assignment. The file contains a function that
    # calculates the mechanical response of a 2D structure that has been
    # specified by the user.
    #
    # The function uses a structure as input and returns a structure as output.
    #
    # Proper functioning of the code can be checked with the following input:
    #
    # [input generated by ConstGeom; Constgeom will not be made available to
    # the student groups. They just get a printed set of input instructions]
    #
    # Written by Julien van Campen
    # Aerospace Structures and Computational Mechanics
    # TU Delft
    # October 2021 â€“ January 2022
    # ============================End of Preamble=================================

    # name of the structure

    inp = {}  # initializing input
    inp['name'] = 'SpaceTelescope'

    # Unit set used: kg - mm - GPa, N.B.: any consistent set will do
    inp['R'] = 2400  # [mm] Radius of the telescope dish
    inp['cl'] = 600  # [mm] Clearance of telescope with base plate
    inp['b'] = 700  # [mm] Base of standing legs telescope
    inp['phi'] = 30  # [deg] opening angle of dish
    inp['type'] = 1  # [-] 1 if beam, 0 if rod structure is selected
    # [-] 1 if full structure is portrayed, 0 if only half of structure is used
    inp['sym'] = 0

    # delta T (will disappear in real input)
    inp['deltaT'] = 5e2

    # user defined seed
    inp['usr'] = {}
    inp['usr']['seed'] = 8  # [-] results in n + 2 nodes per part
    # [-] 1 if lumped mass matrix is to be used, 0 if full mass matrix is to be used (only for rod structure)
    inp['usr']['lumped'] = 1

    inp['mat'] = [{}, {}, {}]  # initialize list of dict
    # material 1: aluminium
    inp['mat'][0]['name'] = 'aluminium'
    inp['mat'][0]['E'] = 70  # [GPa]
    inp['mat'][0]['rho'] = 2.700e-06  # [kg/mm^3]
    inp['mat'][0]['alpha'] = 24e-06  # [mm/(mm deg)]
    # material 2: steel
    inp['mat'][1]['name'] = 'steel'
    inp['mat'][1]['E'] = 210  # [GPa]
    inp['mat'][1]['rho'] = 7.850e-06  # [kg/mm^3]
    inp['mat'][1]['alpha'] = 12.5e-06  # [mm/(mm deg)]
```

```
# material 3: cfrp
inp['mat'][2]['name'] = 'cfrp'
inp['mat'][2]['E'] = 200  # [GPa]
inp['mat'][2]['rho'] = 1.800e-06  # [kg/mm^3]
inp['mat'][2]['alpha'] = 6e-06  # [mm/(mm deg)]

inp['sec'] = [{}, {}, {}]  # initialize list of dict
# section 1:
inp['sec'][0]['A'] = 100  # [mm^2]
inp['sec'][0]['I'] = 10000  # [mm^4]
# section 2:
inp['sec'][1]['A'] = 400  # [mm^2]
inp['sec'][1]['I'] = 8000000  # [mm^4]
# section 3:
inp['sec'][2]['A'] = 500  # [mm^2]
inp['sec'][2]['I'] = 250000  # [mm^4]

# factor to scale plot with
inp['plotScale'] = 1  # [-]

# call the function that creates the geometry
inp = ConstGeom(inp)

# force on point 3 in negative y-direction
inp['P'][3*(3-1)+1] = 0  # [N]
# number of eigenfrequencies returned
inp['nFreq'] = 5  # [-]

# !!! the input above is just for testing, will be removed for students !!!

# initialize output
out = {}
out['name'] = inp['name']

# %% 1. Reading the input structure and txt file
# Check the provided input
# ------------------------------------------------------------------------
if 'parts' in inp.keys() == False:
    print('Error: inp.parts has not been specified by user')
   # return

# Display the structure for the user to check it visually
# ------------------------------------------------------------------------
plt.figure(1)
# plot the parts
for ix in range(len(inp['parts'][0, :])):

    plt.plot(inp['points'][0, inp['parts'][:, ix]-1],
             inp['points'][1, inp['parts'][:, ix]-1], 'k-', marker='o', linewidth=2, markersize=12)

# plot the points
plt.plot(inp['points'][0, :], inp['points']
         [1, :], 'or', linewidth=4, markersize=10)
# format the axis of the plot
plt.xlim([min(inp['points'][0, :])-0.5, max(inp['points'][0, :])+0.5])
plt.ylim([min(inp['points'][1, :])-0.5, max(inp['points'][1, :])+0.5])
plt.rcParams.update({'font.size': 20})
plt.savefig('mesh_1.png')

# %% 2. Creating a mesh per part of the structure
# adjust the seed according to user specification
# ------------------------------------------------------------------------
if inp['usr']['seed'] != 0:
    if np.all(inp['type'] == 1):
        inp['seed'] = inp['usr']['seed']*np.ones(np.size(inp['seed']))

# write input to output
out['inp'] = inp

# Loop over the specified parts and create nodes and elements per part
# ------------------------------------------------------------------------
# initialize counter for nodes
iN = max(inp['points'].shape)

mesh = {}  # initialize mesh
mesh['part'] = [dict() for i_dic in range(len(inp['parts'][0, :]))]
# loop over parts
for ix in range(len(inp['parts'][0, :])):
    # length of the part
    mesh['part'][ix]['length'] = np.sqrt(sum(
        (inp['points'][:, inp['parts'][1, ix]-1]-inp['points'][:, inp['parts'][0, ix]-1])**2, 1))
    # unit vector of the part (local x-axis)
    mesh['part'][ix]['direction'] = (inp['points'][:, inp['parts'][1, ix]-1] -
                                     inp['points'][:, inp['parts'][0, ix]-1])/mesh['part'][ix]['length']
    # rotation of element w.r.t. global coord. system [radians]
    mesh['part'][ix]['rotation'] = np.arctan(
        mesh['part'][ix]['direction'][1]/mesh['part'][ix]['direction'][0])
    # element length is obtained by dividing the length of the elemnt by
    # the number of seeded nodes
    mesh['part'][ix]['elementLength'] = mesh['part'][ix]['length'] / \
        (inp['seed'][ix]+1)
    # create an array with the coordinates of the nodes
    multiplierVector = np.arange(
        0, 1+(1/(inp['seed'][ix]+1))/2, (1/(inp['seed'][ix]+1)))*mesh['part'][ix]['length']
```

```
    mesh['part'][ix]['nodes'] = inp['points'][:, inp['parts'][0, ix]-1].reshape((2, 1)) @\
        np.ones((len(multiplierVector)))[
        np.newaxis] + mesh['part'][ix]['direction'].reshape((2, 1))@multiplierVector[np.newaxis]
    # number nodes in the part
    # first node
    firstNode = inp['parts'][0, ix]
    # last node
    lastNode = inp['parts'][1, ix]
    # intermediate nodes
    nInterNodes = len(multiplierVector)-2
    if nInterNodes < 1:
        interNodes = []
    else:
        interNodes = [k for k in range(iN+1, iN+nInterNodes+1)]

    # advance counter node number
    iN = iN+nInterNodes
    # assign node numbers to part
    mesh['part'][ix]['nodeNumbers'] = [firstNode] + interNodes + [lastNode]
    mesh['part'][ix]['nNodes'] = 2+nInterNodes
    # number of elements in the part
    mesh['part'][ix]['nElements'] = len(multiplierVector)-1
    # temperature of the part (only if first node and last node have equal
    # non-zero temperature
    if (inp['T'][firstNode-1] > 0) and (inp['T'][firstNode-1] == inp['T'][lastNode-1]):
        mesh['part'][ix]['DeltaT'] = inp['T'][firstNode-1]
    else:
        mesh['part'][ix]['DeltaT'] = 0

    # plot the nodes in figure 1 for visual inspection

    plt.plot(mesh['part'][ix]['nodes'][0, :],
             mesh['part'][ix]['nodes'][1, :], 'ok', linewidth=4, markerfacecolor='None', markersize=5)

# store total amount of nodes
mesh['nNodes'] = iN

# transfer part nodes to global nodes
# ------------------------------------------------------------------------
# Initialise array for nodal coordinates
# (this is done here, because before the total amount of nodes was unknown)
mesh['nodes'] = np.zeros((2, iN))
# Each of the specified points is a node. These nodes have the lowest node
# numbers
mesh['nodes'][:, :max(inp['points'].shape)] = inp['points']
# loop over parts to collect nodal coordinates
for ix in range(len(inp['parts'][0, :])):
    # only the intermediate nodes need to be added to mesh.nodes
    if mesh['part'][ix]['nNodes'] > 2:
        for jx in range(2, mesh['part'][ix]['nNodes']):
            mesh['nodes'][:, mesh['part'][ix]['nodeNumbers']
                          [jx-1]-1] = mesh['part'][ix]['nodes'][:, jx-1]
# plot the nodes in a separate figure
plt.figure(2)
for ix in range(len(inp['parts'][0, :])):
    plt.plot(inp['points'][0, inp['parts'][:, ix]-1], inp['points']
             [1, inp['parts'][:, ix]-1], '0.8', linewidth=2)

plt.plot(mesh['nodes'][0, :], mesh['nodes'][1, :], 'ok',
         linewidth=4, markerfacecolor='None', markersize=5)
# format the axis of the plot

plt.xlim([min(inp['points'][0, :])-0.5, max(inp['points'][0, :])+0.5])
plt.ylim([min(inp['points'][1, :])-0.5, max(inp['points'][1, :])+0.5])
plt.savefig('mesh_2.png')
plt.show()
# plt.rcParams.update({'font.size': 20})


# pause for user to do visual inspection of structure and mesh
# ------------------------------------------------------------------------
input('Please inspect the mesh shown in figures 1 and 2. \
      They have also been saved in the working path as "mesh_1.png" and "mesh_2.png". \
      Press enter to continue.')

# %% 3. Assigning element properties
# Loop over the specified parts and assign properties to the elements
# ------------------------------------------------------------------------
# inintialize counter for elements
iE = 0
# initialize list of dict with large number of empty dict (here 1000)
mesh['element'] = [dict() for i_dic in range(1000)]
for ix in range(len(inp['parts'][0, :])):
    # register first element number of part
    firstElementNumber = iE+1
    # loop over elements in part
    for jx in range(mesh['part'][ix]['nElements']):
        # advance element count
        iE = iE+1
        # number of the part that the element belongs to
        mesh['element'][iE]['partNumber1'] = ix
        # node numbers belonging to the element
        mesh['element'][iE]['nodeNumber1'] = mesh['part'][ix]['nodeNumbers'][jx]
```

```
            mesh['element'][iE]['nodeNumber2'] = mesh['part'][ix]['nodeNumbers'][jx+1]
            # retrieve properties
            mesh['element'][iE]['E'] = inp['mat'][inp['material'][ix]]['E']
            mesh['element'][iE]['rho'] = inp['mat'][inp['material'][ix]]['rho']
            mesh['element'][iE]['alpha'] = inp['mat'][inp['material'][ix]]['alpha']
            mesh['element'][iE]['A'] = inp['sec'][inp['section'][ix]]['A']
            mesh['element'][iE]['I'] = inp['sec'][inp['section'][ix]]['I']
            mesh['element'][iE]['type'] = inp['type'][ix]
            mesh['element'][iE]['rotation'] = mesh['part'][ix]['rotation']
            mesh['element'][iE]['length'] = mesh['part'][ix]['elementLength']
            # divide the mass of the element over its two nodes
            mesh['element'][iE]['mass'] = mesh['element'][iE]['length'] * \
                mesh['element'][iE]['A']*mesh['element'][iE]['rho']
            mesh['element'][iE]['lumpedMassNodeNumber1'] = mesh['element'][iE]['mass']/2
            mesh['element'][iE]['lumpedMassNodeNumber2'] = mesh['element'][iE]['mass']/2
            # element temperature
            mesh['element'][iE]['DeltaT'] = mesh['part'][ix]['DeltaT']

        # register last element number of part
        lastElementNumber = iE
        mesh['part'][ix]['elementNumbers'] = range(
            firstElementNumber, lastElementNumber+1)
    # store total amount of elements
    mesh['nElements'] = iE
    # keep only filled dict
    mesh['element'] = mesh['element'][1:iE+1]

    # Create a stiffness matrix, thermal loadvector and mass matrix per element (in global coordinate system)
    # ---------------------------------------------------------------------------
    for ix in range(mesh['nElements']):
        # Local stiffness matrix
        # ---------------------------------------------------------------------
        mesh['element'][ix]['Kbar'] = ((mesh['element'][ix]['E']*mesh['element'][ix]['A']) /
                                       mesh['element'][ix]['length']) * \
            np.concatenate((np.array([1, 0, 0, -1, 0, 0]).reshape(1, 6), np.zeros(
                (2, 6)), np.array([-1, 0, 0, 1, 0, 0]).reshape(1, 6), np.zeros((2, 6))), axis=0)
        # Add terms for bending stiffness if the element is a beam element
        if mesh['element'][ix]['type']:
            mesh['element'][ix]['Kbar'] = mesh['element'][ix]['Kbar'] +
((mesh['element'][ix]['E']*mesh['element'][ix]['I'])/(mesh['element'][ix]['length']**3)) * \
                np.concatenate((np.zeros((1, 6)),
                                np.array([0, 12, 6*mesh['element'][ix]['length'], 0, -12,
                                          6*mesh['element'][ix]['length']]).reshape(1, 6),
                                np.array([0, 6*mesh['element'][ix]['length'], 4*mesh['element'][ix]['length']**2, 0, -
                                          6*mesh['element'][ix]['length'],
2*mesh['element'][ix]['length']**2]).reshape(1, 6),
                                np.zeros((1, 6)),
                                np.array([0, -12, -6*mesh['element'][ix]['length'], 0,
                                          12, -6*mesh['element'][ix]['length']]).reshape(1, 6),
                                np.array([0, 6*mesh['element'][ix]['length'], 2*mesh['element'][ix]['length']**2, 0, -
6*mesh['element'][ix]['length'], 4*mesh['element'][ix]['length']**2]).reshape(1, 6)))

        # Thermal load vector
        # ---------------------------------------------------------------------
        mesh['element'][ix]['Qbar'] = mesh['element'][ix]['E']*mesh['element'][ix]['A']*mesh['element'][ix]['alpha'] *
\
            mesh['element'][ix]['DeltaT'] * \
            np.array([[1], [0], [0], [-1], [0], [0]], dtype=float)
        # Rotation matrix
        # ---------------------------------------------------------------------
        T = np.diag([np.cos(mesh['element'][ix]['rotation']), np.cos(mesh['element'][ix]['rotation']), 1, np.cos(
            mesh['element'][ix]['rotation']), np.cos(mesh['element'][ix]['rotation']), 1])
        T[0, 1] = np.sin(mesh['element'][ix]['rotation'])
        T[1, 0] = -np.sin(mesh['element'][ix]['rotation'])
        T[3, 4] = np.sin(mesh['element'][ix]['rotation'])
        T[4, 3] = -np.sin(mesh['element'][ix]['rotation'])
        # Rotate the local stiffness matrix to the global coordinate system
        mesh['element'][ix]['K'] = np.transpose(
            T)@mesh['element'][ix]['Kbar']@T
        # Rotate the local thermal load vector to the global coordinate system
        mesh['element'][ix]['Q'] = np.transpose(T)@mesh['element'][ix]['Qbar']
        # Mass matrix
        # ---------------------------------------------------------------------
        rhoAL = mesh['element'][ix]['rho'] * \
            mesh['element'][ix]['A']*mesh['element'][ix]['length']
        rhoIL = mesh['element'][ix]['rho'] * \
            mesh['element'][ix]['I']/mesh['element'][ix]['length']
        if mesh['element'][ix]['type']:
            # mass matrix for beam element
            mRhoA = rhoAL/420*np.concatenate((np.array([140, 0, 0, 70, 0, 0]).reshape(1, 6),
                                              np.array([0, 156, 22*mesh['element'][ix]['length'], 0,
                                                        54, -13*mesh['element'][ix]['length']]).reshape(1, 6),
                                              np.array([0, 22*mesh['element'][ix]['length'],
4*mesh['element'][ix]['length']**2, 0,
                                                        13*mesh['element'][ix]['length'], -
3*mesh['element'][ix]['length']**2]).reshape(1, 6),
                                              np.array([70, 0, 0, 140, 0, 0]).reshape(
                                                  1, 6),
                                              np.array([0, 54, 13*mesh['element'][ix]['length'], 0,
                                                        156, -22*mesh['element'][ix]['length']]).reshape(1, 6),
                                              np.array([0, -13*mesh['element'][ix]['length'], -
3*mesh['element'][ix]['length']**2, 0, -22*mesh['element'][ix]['length'],
4*mesh['element'][ix]['length']**2]).reshape(1, 6)))
            mRhoI = rhoIL/30*np.concatenate((np.zeros((1, 6)),
```

```
                                            np.array([0, 36, 3*mesh['element'][ix]['length'], 0, -36*mesh['element']
                                                     [ix]['length'], 3*mesh['element'][ix]['length']]).reshape(1, 6),
                                            np.array([0, 3*mesh['element'][ix]['length'],
4*mesh['element'][ix]['length']**2, 0, -
                                                     3*mesh['element'][ix]['length'], -
mesh['element'][ix]['length']**2]).reshape(1, 6),
                                            np.zeros((1, 6)),
                                            np.array([0, -36*mesh['element'][ix]['length'], -3*mesh['element'][ix]
                                                      ['length']**2, 0, 36, -
3*mesh['element'][ix]['length']]).reshape(1, 6),
                                            np.array([0, 3*mesh['element'][ix]['length'], -
mesh['element'][ix]['length']**2, 0, -3*mesh['element'][ix]['length'], 4*mesh['element'][ix]['length']**2]).reshape(1,
6)))
            mesh['element'][ix]['m'] = mRhoA + mRhoI
        else:
            # mass matrix for rod element
            if inp['usr']['lumped']:
                # lumped mass matrix
                mesh['element'][ix]['m'] = rhoAL/2*np.eye(6)
            else:
                # regular mass matrix
                mesh['element'][ix]['m'] = rhoAL/6*np.concatenate((np.concatenate((2*np.eye(
                    3), np.eye(3)), axis=1), np.concatenate((np.eye(3), 2*np.eye(3)), axis=1)))

    # %% 4. Assembling the structure
    # Assemble the Stiffness matrix
    # ------------------------------------------------------------------------
    # initialise the global stiffness matrix
    # each node has 3 degrees of freedom: u,v, and theta
    mesh['K'] = np.zeros((3*mesh['nNodes'], 3*mesh['nNodes']))
    for ix in range(len(inp['parts'][0, :])):
        # assemlbe the global stiffness matrix
        for jx in mesh['part'][ix]['elementNumbers']:
            # bounds
            lb1 = 3*(mesh['element'][jx-1]['nodeNumber1']-1)+1
            lb2 = 3*mesh['element'][jx-1]['nodeNumber1']
            ub1 = 3*(mesh['element'][jx-1]['nodeNumber2']-1)+1
            ub2 = 3*mesh['element'][jx-1]['nodeNumber2']
            bounds = np.concatenate(
                (np.arange(lb1, lb2+1), np.arange(ub1, ub2+1)))-1
            # assemble
            mesh['K'][np.ix_(bounds, bounds)] = mesh['K'][np.ix_(
                bounds, bounds)]+mesh['element'][jx-1]['K']

    # Assemble the Thermal load vector
    # ------------------------------------------------------------------------
    out['Q'] = np.zeros((np.shape(mesh['K'])[0], 1))
    for ix in range(len(inp['parts'][0, :])):
        # assemlbe the thermal load vector
        for jx in mesh['part'][ix]['elementNumbers']:
            # bounds
            lb1 = 3*(mesh['element'][jx-1]['nodeNumber1']-1)+1
            lb2 = 3*mesh['element'][jx-1]['nodeNumber1']
            ub1 = 3*(mesh['element'][jx-1]['nodeNumber2']-1)+1
            ub2 = 3*mesh['element'][jx-1]['nodeNumber2']
            bounds = np.concatenate(
                (np.arange(lb1, lb2+1), np.arange(ub1, ub2+1)))-1
            # assemble
            out['Q'][bounds] = out['Q'][bounds]+mesh['element'][jx-1]['Q']

    # Assemble the Mass matrix
    # ------------------------------------------------------------------------
    # initialise the global mass matrix
    mesh['m'] = np.zeros((3*mesh['nNodes'], 3*mesh['nNodes']))
    for ix in range(len(inp['parts'][0, :])):
        # % assemlbe the global stiffness matrix
        for jx in mesh['part'][ix]['elementNumbers']:
            # bounds
            lb1 = 3*(mesh['element'][jx-1]['nodeNumber1']-1)+1
            lb2 = 3*mesh['element'][jx-1]['nodeNumber1']
            ub1 = 3*(mesh['element'][jx-1]['nodeNumber2']-1)+1
            ub2 = 3*mesh['element'][jx-1]['nodeNumber2']
            bounds = np.concatenate(
                (np.arange(lb1, lb2+1), np.arange(ub1, ub2+1)))-1
            # assemble
            mesh['m'][np.ix_(bounds, bounds)] = mesh['m'][np.ix_(
                bounds, bounds)]+mesh['element'][jx-1]['m']

    # %% 5. Applying Loads and Boundary Conditions
    # Assemble the loading vector
    # ------------------------------------------------------------------------
    # applied loads
    out['P'] = np.zeros((np.shape(mesh['K'])[0], 1))
    out['P'][:max((inp['points']).shape)*3] = inp['P']

    # displacements of entire system
    out['U'] = np.zeros((np.shape(mesh['K'])[0], 1))

    # reaction forces of entire system
    out['R'] = np.zeros((np.shape(mesh['K'])[0], 1))

    # reduce the amount of degrees of freedom if rod element is selected
    # ------------------------------------------------------------------------
    if np.all(inp['type'] == 1):
```

```
    # nothing happens
    pass
else:
    # find the indices of the remaining DOFs
    remainDF = np.nonzero(matlib.repmat(
        [1, 1, 0], 1, int(max(mesh['K'].shape)/3)))[1]
    remainBC = np.nonzero(matlib.repmat(
        [1, 1, 0], 1, int(max(inp['bc'].shape)/3)))[1]
    # reduce vector with boundary conditions
    inp['bc'] = inp['bc'][remainBC]
    #  reduce stiffness and mass matrices
    mesh['K'] = mesh['K'][np.ix_(remainDF, remainDF)]
    mesh['m'] = mesh['m'][np.ix_(remainDF, remainDF)]
    # reduce displacement, load, thermal load and reaction force vectors
    out['U'] = out['U'][remainDF]
    out['P'] = out['P'][remainDF]
    out['Q'] = out['Q'][remainDF]
    out['R'] = out['R'][remainDF]

# Remove blocked degrees of freedom
# ------------------------------------------------------------------------
# active degrees of freedom
# 1 means d.o.f. is restrained, 0 means d.o.f. is free
activeDF = np.ones((np.shape(mesh['K'])[0]))
# find the clamped nodes
inactiveDF = np.nonzero(inp['bc'])[0]
# inactive degrees of freedom
activeDF[inactiveDF] = 0
inactiveDF = np.ones((np.shape(mesh['K'])[0])) - activeDF
# convert to zeros and ones to indices
activeDF = np.nonzero(activeDF)[0]
inactiveDF = np.nonzero(inactiveDF)[0]

# reduced stiffness matrices
Kr = mesh['K'][np.ix_(activeDF, activeDF)]
Ksr = mesh['K'][np.ix_(inactiveDF, activeDF)]
# reduced load vectors
Pr = out['P'][activeDF]
Ps = out['P'][inactiveDF]
Qr = out['Q'][activeDF]
Qs = out['Q'][inactiveDF]
# reduced mass matrix
mr = mesh['m'][np.ix_(activeDF, activeDF)]

# return mesh as output
out['mesh'] = mesh

# %% 6. Performing the analysis
# displacements and reaction forces
# ------------------------------------------------------------------------
# displacements of reduced system
KrInv = np.linalg.inv(Kr)
Ur = KrInv@(Pr-Qr)  # ok<MINV>
# displacements of entire system
out['U'][activeDF] = Ur
# reaction forces
Rs = Ksr@Ur-(Ps-Qs)
# reaction forces of entire system
out['R'][inactiveDF] = Rs

# eigenfrequency analysis
# ------------------------------------------------------------------------
# compute eigenvalues
E = np.linalg.eig(-KrInv@mr)[0]
# proces first userdefined number of eigenvalues
E = np.sqrt(np.abs(E[:inp['nFreq']]**(-1)))
# return eigenfrequencies to output.
out['eigenfrequency'] = E

# %% 7. Plotting results
if np.all(inp['type'] == 1):
    # reshape 3 displacements per node
    locDisp3D = out['U'].reshape(3, mesh['nNodes'], order='F')
else:
    # reshape 2 displacements per node
    locDisp3D = out['U'].reshape(2, mesh['nNodes'], order='F')

# 2D locations of the displaced nodes
locDisp = mesh['nodes'] + inp['plotScale']*locDisp3D[0:2, :]

plt.figure(3)
for ix in range(max(inp['parts'][0, :].shape)):
    plt.plot(inp['points'][0, inp['parts'][:, ix]-1], inp['points'][1, inp['parts'][:, ix]-1],
             '0.8', linewidth=2)

plt.plot(mesh['nodes'][0, :], mesh['nodes'][1, :], 'o', '0.8',
         linewidth=2, markerfacecolor='None', markersize=5)

plt.plot(locDisp[0, :], locDisp[1, :], 'or', linewidth=2,
         markerfacecolor='None', markersize=5)

# format the axis of the plot
plt.xlim([min(min(inp['points'][0, :]), min(locDisp[1, :])) -
         50, max(inp['points'][0, :])+50])
```

```
    plt.ylim([min(inp['points'][1, :])-50,
              max(max(inp['points'][1, :]), max(locDisp[1, :]))+50])

    # %% 8. Storing results to txt file and output structure

    # TODO: writing to output file
    return out


inp = []
out = MechRes2D(inp)
```