NTNU

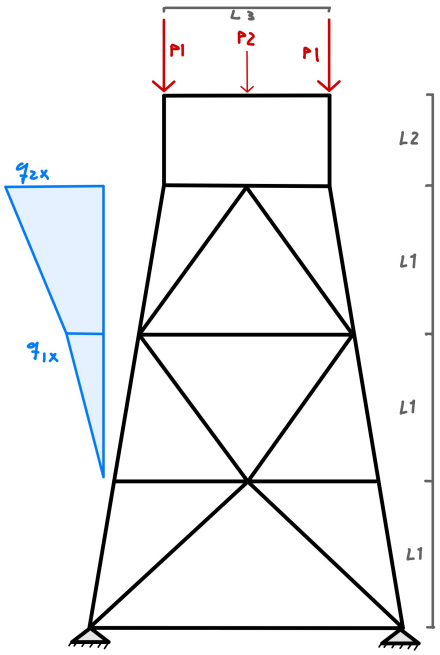
Kunnskap for en bedre verden

TMR4167 - Marin Teknikk - Konstruksjoner

Analysis of 2D frame in Python using the matrix method

Written by:

10013, 10058, 10067



October, 2021

Abstract

A general 2D frame simulation has been created in Python by using the matrix method. The simulation allows both concentrated and distributed loads on the frame. A jacket construction was chosen to run in the simulation, which has been compared to hand calculations and DNVGL's Nautic 3DBeam. The result is that the simulation has a high precision, with $[0-13]\%$ deviation for bending moment values along all elements and $[0-3]\%$ and $[0-8]\%$ deviation in axial and shear force respectively for almost all elements. By altering the dimensions of the elements in the jacket construction, all elements ended up with a maximum total stress within $[30-70]\%$ of yield stress. This report gives an insight into the theory behind and how the matrix method was implemented in a python simulation.

Preface

This report is the result of a project in the course "TMR4167 Marin Teknikk - Konstruksjoner" in the Marine Technology Deparntment of the Norwegian University of Science and Technology. The main essence of the task is to use the matrix method on a complex frame, and calculate the necessary cross-sectional dimensions reach desired values of element stress with respect to yield stress.

The task has been solved by three students during the time frame of about four weeks. Within the group, there has been good cooperation, division of responsibility and top class environment for school-work. Furthermore, the project has been a bit time-consuming, but at the same time motivating as the theory we have learned has been implemented in a realistic simulation.

Contents

List of Tables

List of Figures

1 Introduction

This report shows the theory and methods used when developing a simulation of a 2D frame exposed to concentrated and distributed loads. It also present the results from running the simulation on a jacket construction. The methods that have been used are mainly acquired from the course "TMR 4167 Marin tekninkk - Konstruksjoner" from the Norwegian University of Science and Technology. At the same time, these methods are built on theory learned from "TKT4118 Mekanikk 1" and "TKT4123 Mekanikk 2". Here, we learned basic relation between reaction forces, deformations and bending moments. In this project, we have used the matrix method and the calculation have taken place in Python. We have also completed simple hand calculations and DNVGL's Nautic 3DBeam to evaluate our results from python.

The Python program is based by the information it gets from a text-file. Further a lot of functions, explained more detailed in chapter 4, are used to calculate values for bending moments, shear forces, normal forces and stress. In addition to calculate these values at each node, we have also created an algorithm, which draws the whole moment diagram to find the maximum value along the beam. The final results are presented through tables and diagrams, and the deviations are also discussed.

2 Task

In this project, we have been asked to analyse a given structure by using the matrix method in Python. The structure we have been given is a plane displaceable frame, where a deck construction is placed upon a jacket. The deck construction consists of two I-profiles in aluminium, while the jacket consist of steel pipes with different diameters. Furthermore, the structure is also exposed by some external forces, due to loads and waves. Where, and how the forces acts on the structure is shown in figure 1.

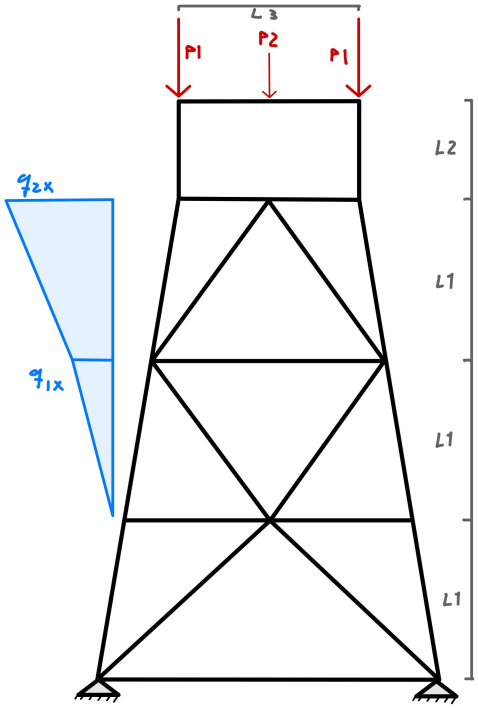


Figure 1: Illustration of the given frame

2.1 Assumptions

Before starting with the methods we have learned, for example the superposition principle, some assumptions had to be done. We have also been told some assumptions to be taken about the frame, to easier the calculation a little bit. All these assumptions are listed below:

-Hooke's law: Linear elasticity, relatively small deformation

-Two-dimensional frame (x,z)

-Beams with negligible mass

-End of beams pointing against each other have the same rotation

2.2 Construction- and load data

The data of the construction and loads are placed in Table 1 and Table 2. Note that the forces q1 and q2 have to be transformed to loads normal to the beams local orientation, and furthermore scaled to match the pipe diameter.

L_1[m]	22
L_2[m]	16
L_3[m]	15
E-module (steel) [GPa]	210
E-module (aluminium) [GPa]	70
Yield stress [MPa]	300

Table 1: Data of construction

P_1[MN]	P_2[MN]	q_1[N/m]	q_2[N/m]
3	1	$5.4 \cdot 10^5$	$1.8 \cdot 10^5$

Table 2: Load data

3 Theory and method

3.1 Discretization

First step in an analysis with the matrix method, is to execute a discretization of the jacket. Here, elements and nodes are connected with numbers, and systematized through a matrix of nodal point correspondence. Our discretization are shown in figure 2. In total, the number of elements and nodes counted 13 and 22, respectively.

Beam number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Node 1	1	3	5	4	6	8	7	10	9	11	1	3	12	5	7	13	1	12	5	12	5	13
Node 2	3	5	7	2	4	6	9	8	11	10	2	12	4	6	13	8	12	2	12	6	13	6

Figure 2: Matrix of nodal point correspondence.

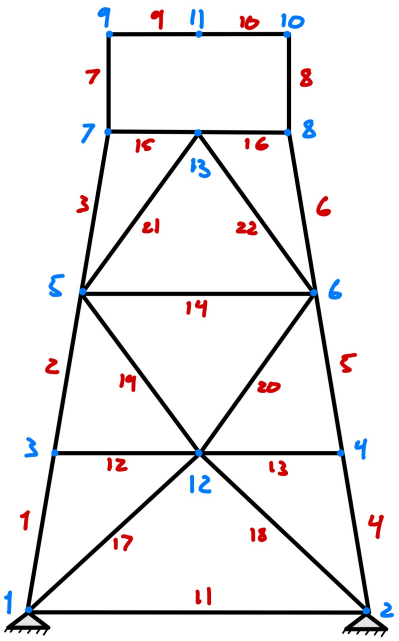


Figure 3: Jacket with elements and nodes enumerated.

3.2 Finding nodes

To find the node coordinates we used the given lengths and the steepness of the vertical pipe beams, r. Simple geometry gives that the length of the horizontal floor

beam is

$$L_{floorbeam} = L_3 + 6rL_1 = 28,2m \tag{1}$$

By choosing a coordinate system where origo is located at the center of the floor beam, this gives that node 1 and 2 have the coordinates $(-14.1, 0)$ and $(14.1, 0)$ respectively. Folowing the same simple geometry we find that the nodes 3 and 4 have the coordinates $(-11.9, 22)$ and $(11.9, 22)$.

3.3 Analysis of elements

Each element consist of two nodes and a beam connecting these. Each node has an axial-force, shear-force, and a moment as shown in figure 4.

Figure 4: Element with axial- and shear forces, and moments illustrated

3.3.1 Local stiffness matrix

Next step is to create the element stiffness matrix k. This matrix express the stiffness relation between loads and deformations, and solves the equation

$$\vec{s} = k \cdot \vec{v} \tag{2}$$

were $\vec{s}$ is the forces and moments acting on the nodes, $\vec{v}$ is the node deformations and rotation. This equation can in our 2D case be written as in equation 3 for each

element.

$$\begin{bmatrix} N_1 \\ V_1 \\ M_1 \\ N_2 \\ V_2 \\ M_2 \end{bmatrix} = \frac{E}{L} \begin{bmatrix} A & 0 & 0 & -A & 0 & 0 \\ 0 & \frac{12I}{l^2} & \frac{-6I}{l} & 0 & \frac{-12I}{l^2} & \frac{-6I}{l} \\ 0 & \frac{-6I}{l} & 4I & 0 & \frac{6I}{l} & 2I \\ -A & 0 & 0 & A & 0 & 0 \\ 0 & \frac{-12I}{l^2} & \frac{6I}{l} & 0 & \frac{12I}{l^2} & \frac{6I}{l} \\ 0 & \frac{-6I}{l} & 2I & 0 & \frac{6I}{l} & 4I \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ w_1 \\ \theta_1 \\ u_2 \\ w_2 \\ \theta_2 \end{bmatrix} \tag{3}$$

The selected stiffness matrix depends on the construction. Our 2D case requires a stiffness matrix with 6 degrees of freedom.

3.3.2 Transform element stiffness matrix to global coordinates

Before we create the global stiffness matrix we first have to express each local stiffness matrix in the global coordinates X and Z. Let T from equation 4 denote the transformation matrix that takes the local system to the global system, where θ is the elements orientation.

$$T = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 & 0 & 0 & 0 \\ -sin(\theta) & cos(\theta) & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & cos(\theta) & sin(\theta) & 0 \\ 0 & 0 & 0 & -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Furthermore we can express the effect of element stiffness matrix in global coordinates as $k_{X,Z} = T \cdot k_{x,z} \cdot T^T$, were T^T is the transponent of the transformation matrix. Note that the signs indicate that the T implies a clockwise rotation, and thus T^T a counter-clockwise rotation, opposite of whats common in mathematics.

3.3.3 Global stiffness matrix

Now that the local stiffness matrix has been established in global coordinates, the global stiffness matrix can be made. The dimension of this matrix depends on

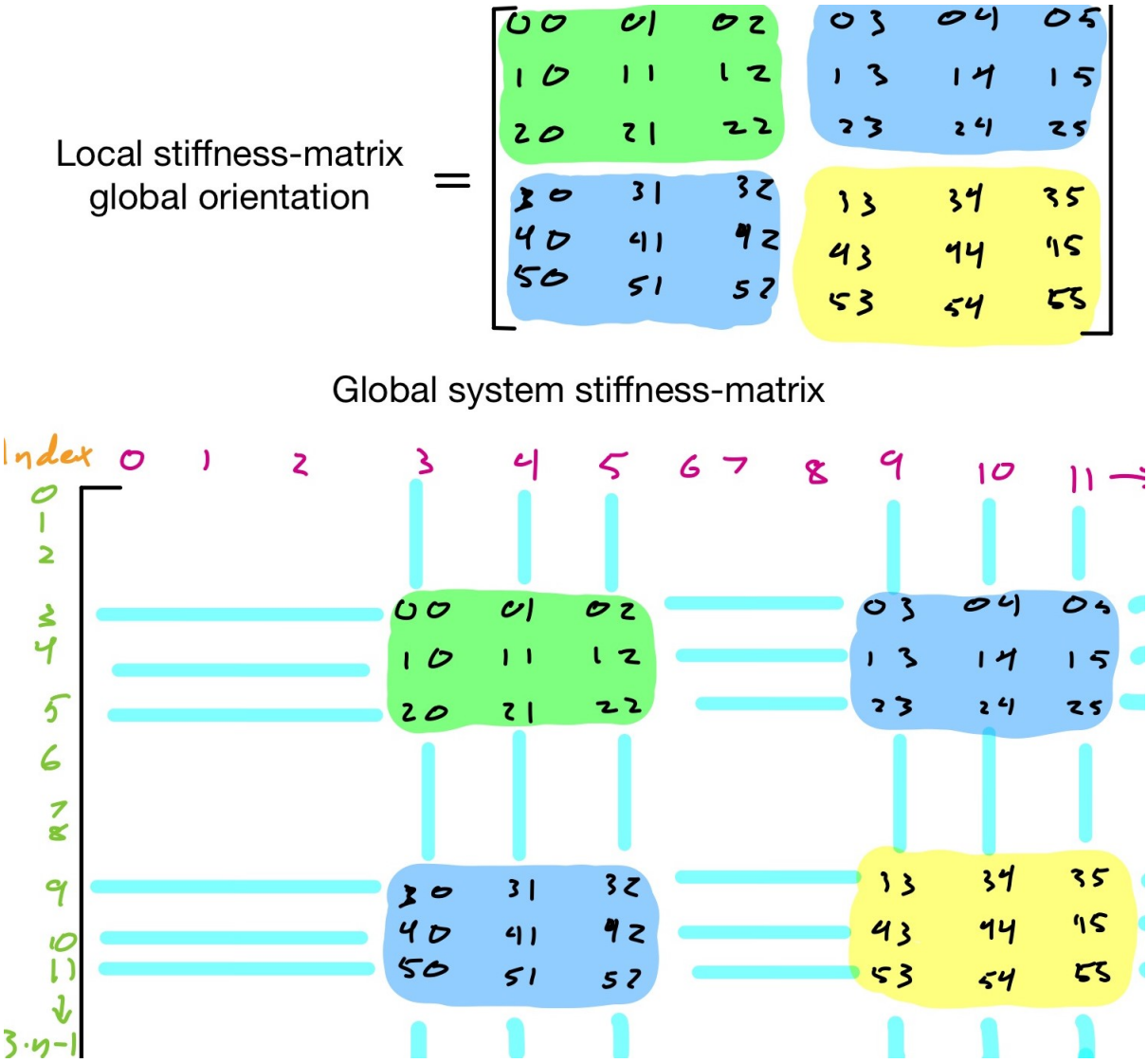


Figure 5: A representation of how the global oriented element stiffness matrix from node 2 to node 4 would be added into the global stiffness matrix.

the total number of nodes in the system. Each node has 3 degrees of freedom. Therefore, the size of a global stiffness matrix is $3n \times 3n$, which in our case gives a 39×39 matrix. To create this matrix, we combine the local stiffness matrices so that each force component in the load vector $\vec{R}$ is connected to the correct deformation component in $\vec{v}$ as shown in figure 5.

The next step is to account for what way the beams are connected in their ends. Here we allow three ways to attach nodes. A free support, fixed support and an unknown support. These are accounted for in the global stiffness matrix by 0 spring resistance, infinite spring resistance and a spring resistance as derived so far in

the theory section. To account for a free node, the row and column in the global stiffness matrix equal 0 and the diagonal element equal 1. No change has to be made to account for connections to other beams.

In our program a large value of 10^6 represent infinite. Also, no beam was free to move, while 11 nodes were unknown in all three degrees of freedom. The last two nodes, that is 1 and 6 were fixed in displacements but unknown in the rotational degree of freedom.

3.4 Load vector

To construct the load-vector, every load, both concentrated and distributed, need to be added together in relation to which nodes is affected by the load. The the concentrated loads can be added directly into the vector. The distributed loads however, needs some calculations before this can be done. Beams that are not exposed to distributed loads does not contribute to the load vector.

For beams exposed to a triangle load, we use the table shown in Appendix B, to calculate the fixed support moments. Here we see that the fixed support moment for a triangular load with peak q_2 at end 2 and 0 load in end 1 is as in equation 5 and 6.

$$(m_1)_2 = \frac{q_2 l^2}{30} \tag{5}$$

$$(m_2)_1 = -\frac{q_2 l^2}{20} \tag{6}$$

Given a situation as in figure 6, this leads to a fixed support moment as follows

$$(m_1)_2 = \frac{q_1 l^2}{20} + \frac{q_2 l^2}{30} \tag{7}$$

$$(m_2)_1 = -\frac{q_2 l^2}{20} - \frac{q_1 l^2}{30} \tag{8}$$

were the distributed load as two triangles facing each other has been considered.

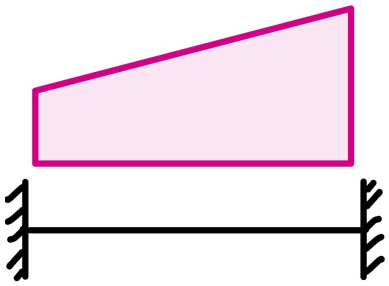


Figure 6: 2 times statically indeterminate beam exposed to a triangle load.

External loads generally also contribute with shear and axial forces in the endpoints. But in our case, all distributed loads act perpendicular to the element, and thus we only get fixed support sheer forces from these. Since we already have moments at each end, we can set up the following equations:

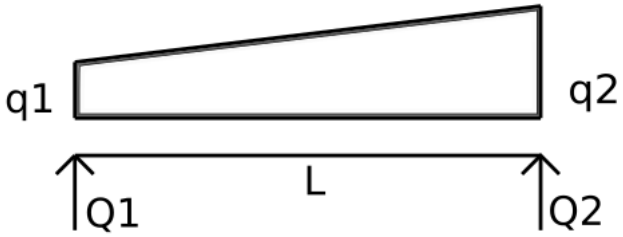


Figure 7: System for calculating shear forces

$$\sum M_1 = m_1 + m_2 - Q_2 L - \frac{q_1 L^2}{6} - \frac{q_2 L^2}{3} = 0 \tag{9}$$

$$\implies Q_2 = \frac{m_1 + m_2 - \frac{q_1 L^2}{6} L - \frac{q_2 L^2}{3}}{L} \tag{10}$$

$$\sum F_z = 0 \implies Q_1 = -\frac{(q_1 + q_2)L}{2} - Q_2 \tag{11}$$

The fixed support moments and forces can then be added to the load vector. The moments can be added directly, but the shear forces need to be decomposed to global coordinates. If this were a 3D program, moments would also be decomposed before added to the resulting load vector.

3.5 Solving the system relation

When the global system matrix K and load vector $\vec{R}$ has been constructed, we use the formula:

$$\vec{R} = K \cdot \vec{r} \tag{12}$$

To solve for $\vec{r}$, which is the displacement vector. Here we denote that $\vec{R} = K \cdot \vec{r}$ is called **the system relation**. Solving this equation by hand is a tedious process, but with a computer in hand, this becomes no problem at all.

3.6 Calculating reaction forces

With the results from the system relation, it is possible to calculate the reaction forces for each beam. Later, when analysing stress and dimensions, it is important to know where the maximum moment occur. To find the maximum moment and its location, bending moment at each node need to be calculated, and also moment along the beam for the beams exposed to distributed loads.

Reaction forces can be calculated using the formula:

$$S_i = k_i \cdot v_i + \bar{S}_i \tag{13}$$

Where S_i represent the end moment and $\bar{S}_i$ represents the external load contributions, and is calculated the same way as fixed support. v_i is the local displacement vector, and k_i the local stiffness matrix

3.6.1 Calculate moment diagram

To control if we have any maximum values of bending moment along the beam we calculated the beam moment diagram. To derive the moment diagram we consider a section cut at x along a beam as shown in figure 8 and use the moment equilibrium equation:

$$\sum M_x = 0 \tag{14}$$

$$M(x) + M_1 + xV_1 + q_2\frac{x}{L}\frac{x}{2}\frac{x}{3} + q_1 \cdot (1 - \frac{x}{L})\frac{x^2}{2} + q_1 \cdot (1 - \frac{x}{L})\frac{2x}{3}\frac{x}{2} = 0 \tag{15}$$

which implies equation 16.

$$M(x) = -M_1 - xV_1 - q_2 \cdot \frac{x^3}{6L} - q_1 \cdot (\frac{x^2}{2} - \frac{x^3}{6L}) \tag{16}$$

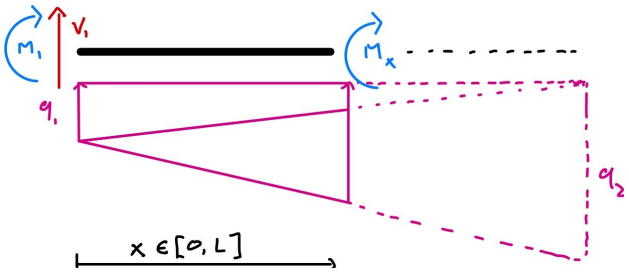


Figure 8: Illustration of how to calculate moment along the beam

This equation calculate the moment value as a function of x, for a beam exposed by a linear distributed load.

3.7 Analysis of stress

With known values for maximum moment and axial force, together with known cross-section parameters the stress can be calculated using the following formula:

$$\sigma = \frac{M_{max}}{I} \cdot z_{max} + \frac{N}{A} \tag{17}$$

where "I" is the moment of inertia, "z_{max}" the maximum distance from the neutral axis and A the cross-section area.

By comparing the bending stress and yield stress, it is possible to evaluate the beams margin to yield. This can be expressed through a safety factor.

$$C_{safety} = \frac{\sigma_B}{\sigma_Y} \tag{18}$$

3.8 Choosing dimensions

An important part before the building process of a construction, is to give the material the right dimensions with respect to yield. Based on this, we determine the material dimension to be, such that the stress level of the beam with the biggest strain in a group of elements, is in order of magnitude 30-70% by the yield stress. The group of elements are divided by their behavior. Our elements are divided into legs, diagonal braces, horizontal braces, vertical deck beams, and horizontal deck beams. Electing suitable cross-sections itself is an iteration process, where dimensions are tested until the most ideal is found.

3.9 3DBeam

DNVGL's Nautic 3DBeam is a software tool used for modelling and analysis of beam structures [2]. To get indications if our python-code was correct at an early stage, we actively used 3DBeam. The benefits by using 3DBeam are first of all that different shaped constructions exposed to external forces can easily be made. In addition the results are presented in tables easy to evaluate. By using 3DBeam we can quickly see whats wrong and right with our python code. The comparison of the displaceable frame can be seen in chapter 5.

3.10 Object oriented approach

We have chosen an object oriented approach. This makes our code much more readable than using matrices, where each individual index easily can get mixed up and create confusion. Especially for a program with a list as extensive at the beamsObjectList, which has over 30 member variables. Of which some are 6×6 matrices and some are numpy arrays of 101 elements.

For example

$$\text{beams[j][k]} = \text{beam.node2.supportTheta}$$

which is easier to read.

The object oriented approach also allows us to create member functions. These have been used extensively throughout the project. One possible downside to writing

object oriented code, is that it might consume less computational power to use lists of lists for everything when writing in python.

4 Python functions

All the functions in use have been commented in the script itself, as well as having quite descriptive names, to make our code as readable as possible. Still, this chapter provide a description of each function.

4.1 Input file

The input file should be the only input the user has to the program, and it is therefore important that the data in this file is correct. The lines are sorted using a keyword in the start of each line. This makes it possible to comment out lines, such as loads, only using a "'#" or some other random character next to the keyword. The program will print an error-message for each line it fails to read.

As long as the material and geometry libraries come in the right order, the other lines can in theory be written in random order. However, it is recommended to gather the data types together for an easier overview.

4.2 class Beam

Our code relies heavily on the beam class. It holds a magnitude of data for each beam, including the coordinates, length, orientation, stiffness-matrices, distributed load information and more. It also comes with over 30 member-functions. The beam class gives us the flexibility to handle almost every operation without keeping track of indexes. Each beam element also holds two node objects to make the class even more versatile, and really shows the strength of object oriented programming.

4.3 class Node

The node class is the fundamental building block of our code. It holds data for both coordinates and fixing point conditions. The node class is also essential in

construction of the resulting load vector, as the resulting loads is summed up and added in the node objects.

4.4 main()

Main is, as the name suggest, the main function in our code. From here we decided which input file the python program read from, and all the necessary functions are run in the correct order to avoid hiccups in our program.

4.5 readInputFile()

This is the function that converts the given text file into matrices containing appropriate data for the different categories. It works by iterating trough the file line by line. For each line it retrieves a code-word, for example "NODE", and then decides where to put that line accordingly.

4.6 initializeNodesAndBeamsList()

As the name suggests this function initializes all the objects, and puts them in lists, both for nodes and beams. In the initialisation of the beams, both the length and orientation is calculated by using the member-functions getLength(), and getGlobalOrientation() respectively. The length and orientation are then added to the beams as member-variables.

4.6.1 getLength()

Member function of the Beam-class. Calculates the length of the beam using pythagoras

$$L = \sqrt{(x_2 - x_1)^2 + (z_2 - z_1)^2} \tag{19}$$

where (x_1, z_2) and (x_2, z_2) are the beams node 1's and 2's coordinates, respectively.

4.6.2 getGlobalOrientation()

Member function of the Beam-class. Calculates the orientation θ of the beam by using

$$tan(\theta) = -\frac{\Delta z}{\Delta x} = -\frac{z_2 - z_1}{x_2 - x_1} \tag{20}$$

4.7 makeBeamsGeometry()

The geometry function first decides which type of geometry each beam has, and then calculates accordingly, either with the member-function makeIPE(), or makePipe(). Should it be necessary, we could also extend this function to handle other geometries.

4.7.1 makeIPE()

Member function of the Beam-class. Takes in height, width top, with bottom, thickness top and thickness bottom. Turns the beam into an IPE-profile, then calculates the cross section area and moment of inertia about strong and weak axis.

4.7.2 makePipe()

Member function of the Beam-class. Takes in radius r and fraction of radius that is hollow, turns the beam into an hollow pipe and then calculates area and moment of inertia for the beams cross-section.

4.8 giveEmodulToBeams()

Attaches the correct value of Youngs-modulus to the beam objects, according to which material the beam is made of. Utilizes the member function makeStiffness().

4.8.1 makeStiffness()

Member function of the Beam-class. Implements E modulus and appends stiffness to beam-object.

4.9 giveNumberToObjects()

Uses the member-function giveNumber() to enumerate the beams and nodes.

4.9.1 giveNumber()

Member function of both Node and Beam classes. Enumerates the element.

4.10 giveLocalStiffnessMatrixToBeamsLocalOrientation()

Uses the member-function makeLocalStiffnessMatrix() to attach the local stiffness-matrix to the beam objects.

4.10.1 makeLocalStiffnessMatrix()

Member function of the Beam-class. Makes a 6×6 local stiffness-matrix in local orientation and appends it to the beam element.

4.11 giveLocalStiffnessMatrixToBeamsGlobalOrientation()

Uses the member-function makeTransformedStiffnessMatrix() to attach the local stiffness-matrix transformed to global orientation to the beam objects.

4.11.1 makeTransformedStiffnessMatrix()

Member function of the Beam-class. Transforms the local stiffness-matrix in local orientation to global orientation. This is done using the transformation matrix, and the transposed of the transformation matrix. Since we use clockwise direction as positive, we switch up the two matrices from what is common in maths otherwise.

4.12 connectAndScaleDistributedLoadsAndCalculate FixedSupport()

Connects the distributed loads to the beam objects, and calculates fixed support for each beam affected by the distributed loads. To do this the function utilizes the member-functions addDistributedNormalLoad() and calculateFixedSupport()

for each beam in the beam-object-list affected. The function also scales the distributed loads to beam-dimensions if wanted using the funtion scaleDistributed-Load().

4.12.1 addDistributedNormalLoad()

Member function of the Beam-class. Transforms components of the distributed load to a distributed normal load, and adds it to the beam-element.

4.12.2 scaleDistributedLoad()

Member function of the Beam-class. Scales the ditributed loads to the dimensions used for the beam. For a pipe the formula used is:

$$q_{sc} = q\frac{d}{d_{sc}} \tag{21}$$

Where d_{sc} is the diameter given for the loads stated in the input file. d_{sc} is also given as a parameter in the input file, under the label "LOADSCALE". This is useful when dealing with loads dependent on the cross-section of the beam, such as fluid-loads.

4.12.3 calculateFixedSupport()

Member function of the Beam-class. Calculates fixed support, including moments and sheer, for the beam-element. The moments is calculated using appendix B.

4.13 connectNodeLoadsToNodes()

Connects the point-loads and moments to nodes with the member-function addNode-Load(). Everything regarding the nodes is done in global coordinates.

4.13.1 addNodeLoad()

Member function of the Node-class. Adds point loads directly to the node-element.

4.14 makeResultingLoadVector()

Runs trough the list of node-objects, and appends the loads to the resulting load vector.

4.15 makeGlobalStiffnessMatrix()

First makes a [3*nodes, 3*nodes] matrix of zeros. The function then goes trough each beam and adds the values from the local transformed stiffnessmatrix according to which nodes the beam is connected to.

The function then runs trough each node and accounts for fixing point conditions. If the node number "n" is held in place in z-direction the diagonal element at [3*n + 1][3*n + 1] will be multiplied by a factor of 10^6. If the node is free to move in z-direction however, the belonging row and column will be made 0, except the element on the diagonal which will be given the value 1.

4.16 np.linalg.solve()

This is a standard function from numpy's linear algebra library, which solves matrix equations on the form:

$$A \times x = B \tag{22}$$

We use it to solve for displacement x, where A is the global stiffness-matrix, and B is the resulting load vector.

4.17 calculateBeamReactionForces()

Calculates the reaction-forces for each beam and adds them, in local orientation, to each beam-object.

4.18 calculateMaxMomentAndBendingTension()

Runs trough each beam-element, but only calculates maximum middle-moment for the beams with distributed loads. The maximum bending tension is then calculated as well as the security factor. This is done using the member-functions calculateMaxMoment() and calculateMaxBendingTension().

4.18.1 calculateMaxMoment()

Member function of the Beam-class. Calculates the maximun moment and its location in distance from local node 1. We assume that maximum moment, and shear

force equals zero, occurs where the resultant of the distributed load attacks.

4.18.2 calculateMaxBendingTension()

Member function of the Beam-class. Calculates the maximum bending tension using Equation 17, and the safety-factor using Equation 18.

4.19 printBeamSpecsToTerminal()

Prints beam data for all beams in a readable way to terminal, using the member-function printBeam(). Mostly used for debugging.

4.19.1 printBeam()

Member function of the Beam-class. Prints data for the beam-element to the terminal.

4.19.2 printBeamMoments()

Member function of the Beam-class. Prints moment data to the terminal.

4.19.3 printSecurityFactor()

Member function of the Beam-class. Prints tension data to the terminal.

4.20 plotMomentDiagram()

This function plots the moment diagram for the elements that are exposed to a distributed load. All other elements will have a linear moment diagram. This function uses getMomentDiagram() which retrieves the moment diagram.

4.20.1 getMomentDiagram()

This function uses the derivation from chapter 3.6.1.

4.20.2 plotterForMomentDiagram()

This function makes a plot for each

4.21 plot()

This function utilizes the function nodeArrayDisplacedPosition() and the handed out funtions [3] setup_plots(), plot_structure(), and plot_structure_def()

Since the handed out functions [3] only handle rotations as deformation, we used needed to write our own function to handle displacements in x- and z-directions. We also had to modify the two plot-structure functions to handle our data-formats.

It's sensible to scale the displacement vector by quite a lot to make the deformations visible.

4.21.1 setup_plots()

Handed out code [3]. This sets up and initializes the plots.

4.21.2 nodeArrayDisplacedPosition()

Transforms the coordinates of the nodes to their displaced position. This is then sent into the plot_structure_def() function.

4.21.3 plot_structure()

Handed out code[3]. This plot visualizes how the construction looks like in its initial form, without deformations.

4.21.4 plot_structure_def()

Handed out code[3]. This plot visualizes the construction with rotational deformations. Combined with the function nodeArrayDisplacedPosition() the plot shows displacements in a much more real matter.

4.22 outputResultsToFile()

Exports output-data to txt-file in a readable format.

5 Results

5.1 Control of program

To control the Python program, we have compared the output with some hand calculations made on easier constructions. The constructions that have been compared are a fixed beam, and a portal frame. The comparisons of moment values are presented below, and more details about the hand calculations are shown in appendix A.

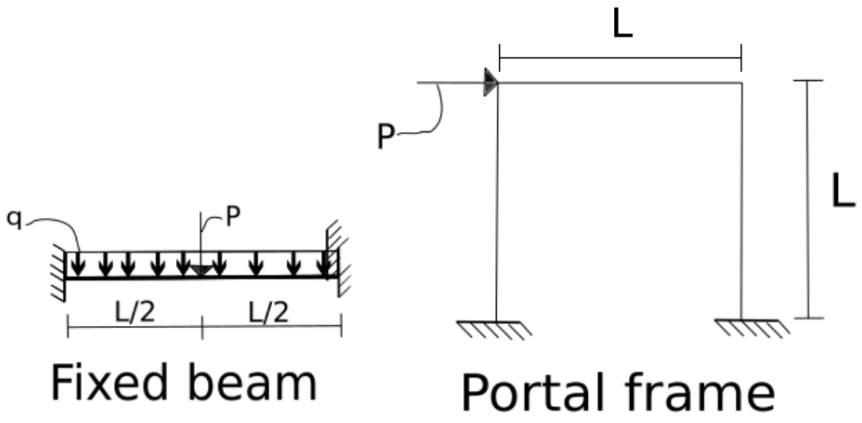


Figure 9: Fixed beam and portal frame we used to compare python code with

E-module[GPa]	70
Moment of inertia[m^4]	0.27
q [$\frac{MN}{m}$]	1.8
P [MN]	1
L [m]	20

Table 3: Selected values to get numbers to compare with the python-code.

5.2 Comparison of moment diagrams

Fixed beam

The fixed beam is divided into two beams and three nodes, where the indexing starts at the left side.

Hand calculation

By using super position and formulas from elementary beam deflection handed out in "TKT4123 Mekanikk 2", the bending moment for the fixed beam can be determined. This, combined with the selected values from Table 3, gave following result:

$$M_1 = M_2 = \frac{qL^2}{12} + \frac{PL}{8} = 62.5MNm \tag{23}$$

$$M_{mid} = \frac{qL^2}{24} + \frac{PL}{8} = 32.5MNm \tag{24}$$

$$Q_1 = Q_2 = \frac{ql}{2} + \frac{P}{2} = 18.5MN \tag{25}$$

Python-program

The python program gave approximately the same values for shear forces and bending moments as the hand calculations.

```
Beam 1 from node 1 to 2, Ø: 0.0, L: 10.0
u1: 0.0          N1: 0.0
w1: -0.0         V1: 18500000.0
ø1: 0.0          M1: -62499984.37
u2: 0.0          N2: 0.0
w2: -0.0419      V2: -500000.0
ø2: -0.0         M2: -32500015.63
```

```
Beam 2 from node 2 to 3, Ø: 0.0, L: 10.0
u1: 0.0          N1: 0.0
w1: -0.0419      V1: -500000.0
ø1: -0.0         M1: 32500015.63
u2: 0.0          N2: 0.0
w2: -0.0         V2: 18500000.0
ø2: -0.0         M2: 62499984.37
```

Figure 10: Output from the Python program.

3DBeam

The result from 3DBeam assures that the Python-program and hand calculation gives the right bending moment values. The deviation between the values are approximate 0%.

	Hand calculation	Python	3DBEAM
Beam 1 (Node 1 and node 2)	-62.5MNm and -32.5MNm	-62.5MNm and -32.5MNm	-62.5MNm and -32.5MNm
Beam 2 (Node 2 and node 3)	32.5MNm and 62.5MNm	32.5MNm and 62.5MNm	32.5MNm and 62.5MNm

Table 4: Results of moment calculations from all the platforms for the fixed beam

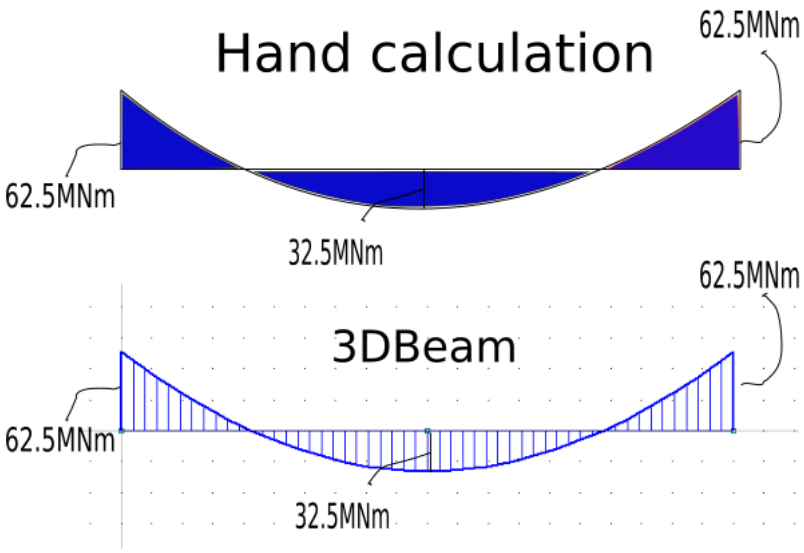


Figure 11: Moment diagrams of the fixed beam

Portal frame

The portal frame is divided into three beams and four nodes. The indexing starts at the left end, and follows the frame.

Hand calculation:

To calculate the moment values in the portal frame we used the deformation method explained in chapter 8.6 [1]. Positive orientation is defined clockwise.

$$M_{21} = -M_{23} = \frac{3PL}{14} = -4.286MNm \tag{26}$$

$$M_{32} = -M_{34} = \frac{3PL}{14} = 4.286MNm \tag{27}$$

$$M_{12} = -M_{43} = \frac{2PL}{7} = -5,714MNm \tag{28}$$

$$Q_{21} = Q_{34} = -\frac{P}{2} = -\frac{1}{2}MN \tag{29}$$

Python-program:

Both the bending moment and shear forces matches the result from the hand calculation.

```
Beam 1 from node 1 to 2, Ø: -90.0, L: 20.0
u1: 0.0          N1: -426760.77
w1: -0.0         V1: 501849.48
ø1: 0.0          M1: -5757051.93
u2: 0.002        N2: 426760.77
w2: -0.2226      V2: -501849.48
ø2: 0.0068       M2: -4279937.57

Beam 2 from node 2 to 3, Ø: 0.0, L: 20.0
u1: 0.2226       N1: 498150.52
w1: 0.002        V1: -426760.77
ø1: 0.0068       M1: 4279937.57
u2: 0.2203       N2: -498150.52
w2: -0.002       V2: 426760.77
ø2: 0.0067       M2: 4255277.89

Beam 3 from node 3 to 4, Ø: 90.0, L: 20.0
u1: 0.002        N1: 426760.77
w1: 0.2203       V1: 498150.52
ø1: 0.0067       M1: -4255277.89
u2: 0.0          N2: -426760.77
w2: 0.0          V2: -498150.52
ø2: 0.0          M2: -5707732.6
```

Figure 12: Output from the Python program.

3DBeam:

The result from 3DBeam assures that the Python-program gives approximately the same values for bending moment.

	Hand calculation	Python	3DBEAM
Node number	0 and 1	0 and 1	0 and 1
Beam 1	-5.714MNm and -4.286MNm	-5.757MNm and -4.280MNm	-5.802MNm and -4.233MNm
Beam 2	4.286MNm and 4.286MNm	4.280MNm and 4.255MNm	4.233MNm and 4.233MNm
Beam 3	-4.286MNm and -5.714MNm	-4.255MNm and -5.708MNm	-4.233MNm and -5.755MNm

Table 5: Results of moment calculations for the portal frame

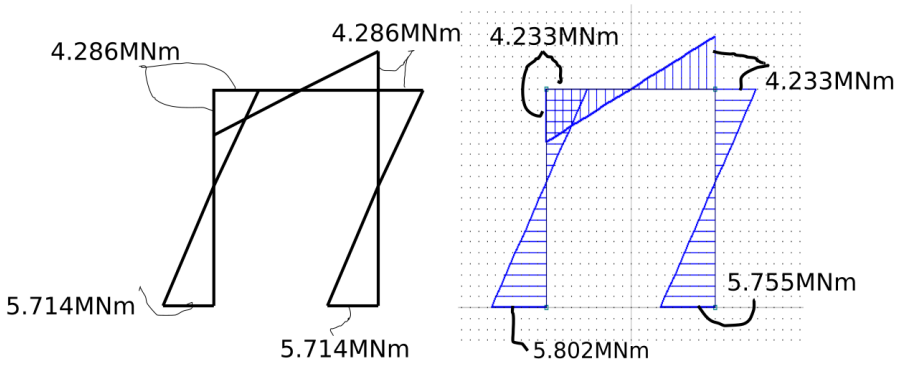


Figure 13: Moment diagrams of the portal frame. The left moment diagram is hand calculated and right moment diagram is from 3DBeam

5.3 Selecting dimensions

The cross section dimensions are selected by doing a lot of testing. We started by dividing the beams into groups, illustrated in figure 14. Then we calculated the bending stress, to the beam with the highest bending moment in each group. This value was compared to the yield stress value to get a safety factor against yield. Further, we checked if this factor was inside a magnitude of order between 30-70% as explained in chapter 3.7. In addition to create the dimensions with respect to yield stress, we have also taken moment capacity into account. Therefore the relation between the diameter and the thickness are calculated to be around 30.

After some iteration, we ended up with the dimensions in table 6. The "legs" were exposed to biggest stress in the jacket construction, and needed the biggest pipe-profile. At the deck construction, the horizontal elements needed the biggest IPE-profile. Table 7 views that all the stress values are within the percentage to yield that is required. Note that the IPE-profiles selected are not actually IPE, but rather HE-1000 B and HL-1100 B profiles. The looks of the profiles however is quite similar.

The results from the bending stress was used to select the right dimensions for cross sections. Based on the requirement from the task, different IPE-profiles and pipe-profiles were chosen for the beams. We grouped the beams by function in the frame. The legs were given Pipe-1000, the horizontal braces Pipe-850, the diagonals braces Pipe-950, the vertical components of the deck IPE-1000 and the horizontal components IPE-1100.

Throughout the project we used 3DBeam actively to get a indicate if our python code was correct.

5.4 Results from the plane displaceable frame

After the selection of the right dimensions, the python program is able to calculate the resulting values for bending moments, shear forces and normal forces. In this subchapter those values are presented through diagrams and compared with 3DBeam. The deviations are also commented. One factor to some deviation are

IPE-profile	Total height[mm]	Flange width[mm]	Web thickness [mm]	Flange thickness[mm]
IPE-1000	1000	300	36	19
IPE-1100	1100	400	36	20
Pipe-profile	Outer radius[mm]	Inner radius[mm]		
Pipe-850	850	790		
Pipe-950	950	887		
Pipe-1000	1000	930		

Table 6: Dimensions we ended up with after the iteration

Group	Maximum stress [MPa]	Element index	Percentage to yield
Pipe-1000	211	5	70%
Pipe-950	205	22	68%
Pipe-850	168	16	56%
IPE-1000	171	7	57%
IPE-1100	153	9 and 10	51%

Table 7: Percentage to yield

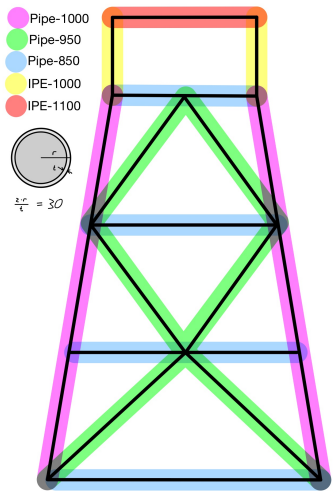


Figure 14: Illustration showing beams and their dimensions

that we in the python program have neglected the beams self weight, which 3DBeam doesn't. Another factor are that 3DBeam are based on a beam formulation that includes the contribution from shear deformations.

5.4.1 Bending moment

As table 8 illustrate, the deviation between the python and 3DBeam output is stably under 5%, except some places where the deviation are passing 10%. As there are none direct connection between the current beams, there are difficult to explain a specific reason to the deviation. However, the places where there are biggest deviations will not have an impact in the calculation of stress, as the node not represent the maximum value of bending moment. The node with the highest value of bending moment are marked by red. Table 9 contains all the extreme values of bending moment along the moment curve to the beams exposed to external forces. Beam 3 and 6 are the only beams were this result exceeds the end moment values. This is taken into account in the calculation of stress.

		Node 1			Node 2	
Beam	Python	3DBeam	Deviation	Python	3DBeam	Deviation
Unit	[MNm]	[MNm]	[%]	[MNm]	[MNm]	[%]
1	0.97	0.98	1.25	5.07	5.14	1.50
2	11.35	10.99	3.23	10.23	10.70	4.39
3	13.59	13.66	0.49	11.23	10.23	9.83
4	-5.09	-5.16	1.44	-1.03	-1.04	1.02
5	-9.96	-10.46	4.86	-11.48	-11.10	3.39
6	-13.00	-11.80	10.18	-12.79	-12.97	1.33
7	0.49	0.46	6.41	-1.02	-1.01	0.99
8	-1.00	-0.98	2.10	0.51	0.49	3.83
9	-1.02	-1.01	1.00	-2.74	-2.75	0.56
10	2.74	2.75	0.56	-1.00	-0.98	2.10
11	-1.32	-1.30	1.77	1.29	1.27	1.75
12	-6.28	-5.85	7.39	4.38	3.95	10.82
13	-4.47	-4.03	10.98	6.30	5.94	6.10
14	-19.06	-18.43	3.44	18.99	18.36	3.40
15	11.72	10.68	9.68	-15.93	-15.11	5.37
16	18.00	16.93	6.33	-12.49	-11.31	10.45
17	0.36	0.32	10.97	2.33	2.42	3.99
18	-2.46	-2.56	3.65	-0.26	-0.23	13.19
19	-3.10	-3.50	11.57	-6.67	-6.35	5.15
20	6.96	6.61	5.33	2.72	3.16	13.97
21	18.79	18.96	0.92	16.96	16.01	5.91
22	-16.97	-16.03	5.85	-18.87	-19.03	0.80

Table 8: Results of bending moment values

	python	3D Beam	Deviation
Unit	[MNm]	[MNm]	[%]
2	3.13	3.26	3.99
3	-17.37	-17.89	2.91
5	-3.02	-3.10	2.58
6	16.78	17.34	3.23
21	-18.46	-18.80	1.81
22	18.42	18.80	2.02

Table 9: Results of max field moment with deviation at beam 2,3,5,6,21 and 22

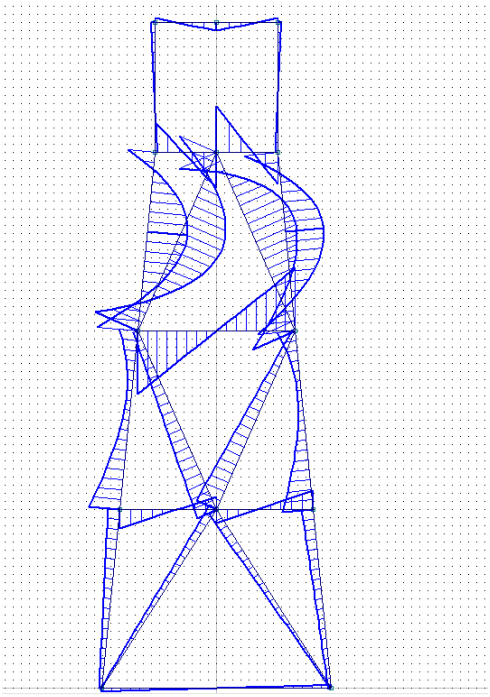


Figure 15: Moment diagram of the jacket construction from 3DBeam

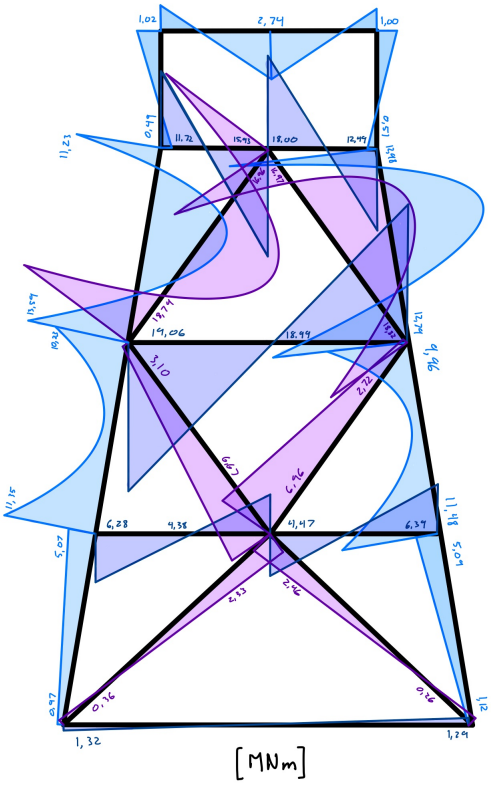


Figure 16: Moment diagram of the jacket construction from our program

5.4.2 Shear forces

Table 10 includes all the results of shear forces. The deviations are in general low excepts at beam 19 and 20. Both are diagonal braces at floor 2. At the beams with deviation, the python value are in general higher than in 3DBeam.

	Node 1			Node 2		
Beam	Python	3DBeam	Deviation	Python	3DBeam	Deviation
Unit	[kN]	[kN]	[%]	[kN]	[kN]	[%]
1	-185.46	-188.39	1.55	-185.46	-188.39	1.55
2	939.49	902.20	4.13	-1726.91	-1764.23	2.12
3	4550.82	4599.37	1.06	-6114.78	-6066.37	0.80
4	183.60	186.47	1.54	183.60	186.47	1.54
5	1708.84	1748.83	2.29	-957.56	-917.60	4.36
6	6230.85	6168.80	1.01	-4434.75	-4496.94	1.38
7	94.27	91.81	2.68	94.27	91.81	2.68
8	94.27	91.81	2.68	94.27	91.81	2.68
9	-501.38	-502.08	0.14	-501.38	-502.08	0.14
10	498.62	497.92	0.14	498.62	497.92	0.14
11	-92.65	-91.05	1.76	-92.65	-91.05	1.76
12	-895.84	-823.60	8.77	-895.84	-823.60	8.77
13	-912.17	-837.27	8.94	-912.17	-837.27	8.94
14	-1961.38	-1896.54	3.42	-1961.38	-1896.54	3.42
15	3685.98	3439.85	7.16	3685.98	3439.85	7.16
16	4065.67	3765.15	7.98	4065.67	3765.15	7.98
17	-75.40	-80.45	6.27	-75.40	-80.45	6.27
18	84.26	89.10	5.43	84.26	89.10	5.43
19	-148.85	-118.39	25.73	-148.85	-118.39	25.73
20	176.38	143.35	23.04	176.38	143.35	23.04
21	5068.62	5115.28	0.91	-6913.61	-6866.80	0.68
22	6910.33	6864.80	0.66	-5071.90	-5117.28	0.89

Table 10: Results of shear forces from 3DBeam and Python

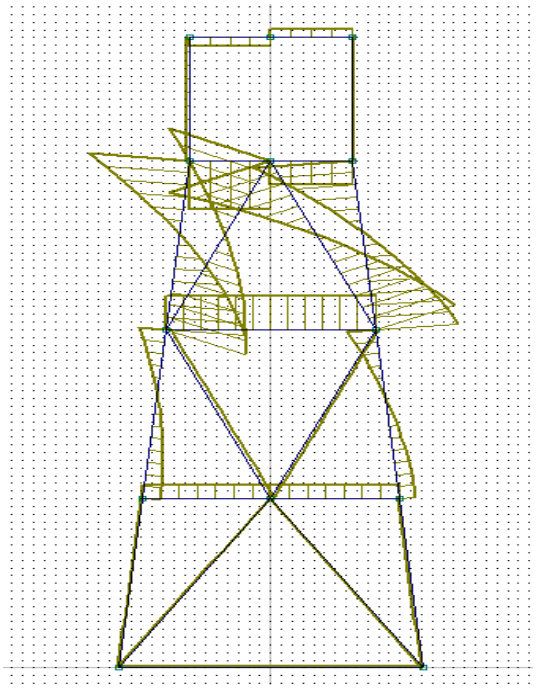


Figure 17: Shear force diagram to the frame construction from 3DBeam

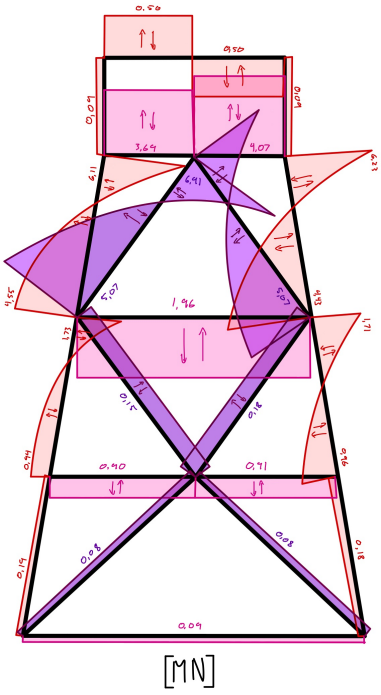


Figure 18: Shear force diagram to the frame construction from Python

5.4.3 Normal forces

The deviations by the the normal forces are pretty low in general, except beam 6 who is far of. Due to the low value of the normal force compared with the bending moment at beam 6, the deviation will not have any big impact to the stress value.

	Normal forces		
Beam	Python	3DBeam	Deviation
Unit	[MN]	[MN]	[%]
1	-61.21	-61.19	0.03
2	-60.42	-60.48	-0.09
3	7.83	7.58	3.32
4	65.73	65.72	0.02
5	64.93	64.99	-0.09
6	-1.19	-0.89	34.74
7	3.50	3.50	-0.02
8	3.50	3.50	0.02
9	0.09	0.09	2.69
10	0.09	0.09	2.69
11	0.00	0.00	0.00
12	1.04	1.01	2.69
13	-1.06	-1.03	2.91
14	-0.66	-0.66	0.12
15	6.77	6.70	1.06
16	-6.41	-6.32	1.50
17	-32.18	-32.20	-0.05
18	35.14	35.16	-0.04
19	43.26	43.41	-0.35
20	-40.52	-40.68	-0.38
21	-31.81	-31.53	0.89
22	32.22	31.88	1.07

Table 11: Normal forces results from Python and 3DBeam

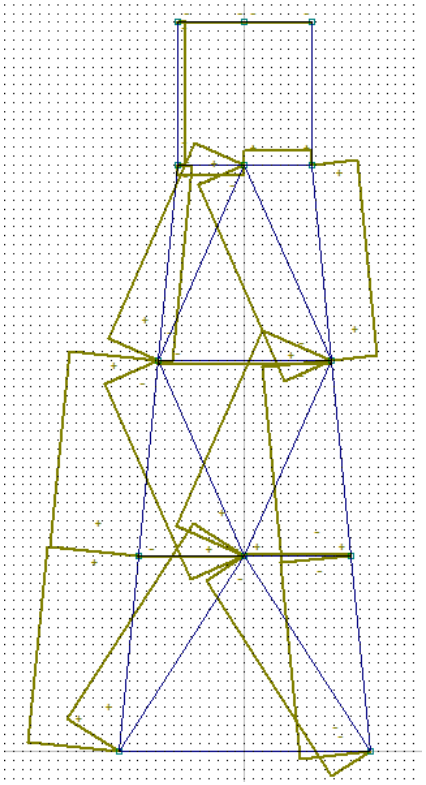


Figure 19: Normal force diagram to the frame construction from 3DBeam

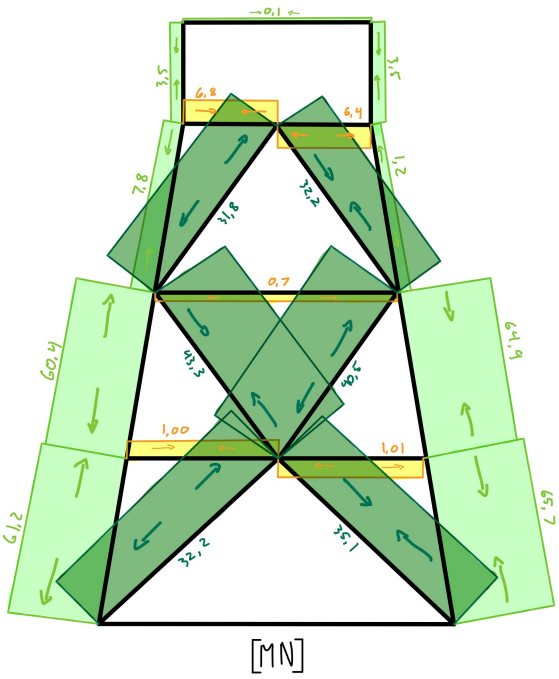


Figure 20: Normal force diagram to the frame construction from Python

5.5 Stress

The deviation in the stress values are more systematized. All the elements have pretty low deviation, excepts the horizontal braces.

		Stress value	
Beam	Python[MPa]	3DBeam[MPa}	Deviation[%]
Unit	[Mpa]	[Mpa]	[%]
1	169.80	169.00	0.47
2	199.70	196.00	1.89
3	-105.57	-103.00	2.50
4	-180.59	-180.00	0.33
5	-210.98	-207.00	1.92
6	86.41	85.00	1.66
7	-170.59	-170.00	0.35
8	-168.88	-167.00	1.13
9	-153.01	-154.00	-0.64
10	-153.01	-154.00	-0.64
11	-10.79	-10.00	7.90
12	-54.67	-49.00	11.57
13	55.58	50.00	11.16
14	157.84	147.00	7.37
15	-151.98	-141.00	7.79
16	167.77	153.00	9.65
17	102.91	103.00	-0.09
18	-111.91	-112.00	-0.08
19	-160.28	-157.00	2.09
20	154.53	152.00	1.66
21	203.73	200.00	1.86
22	-205.39	-202.00	1.68

Table 12: Results of stress values

5.6 Visualization of deformation

By running the handed out code, this deformation plot was found.

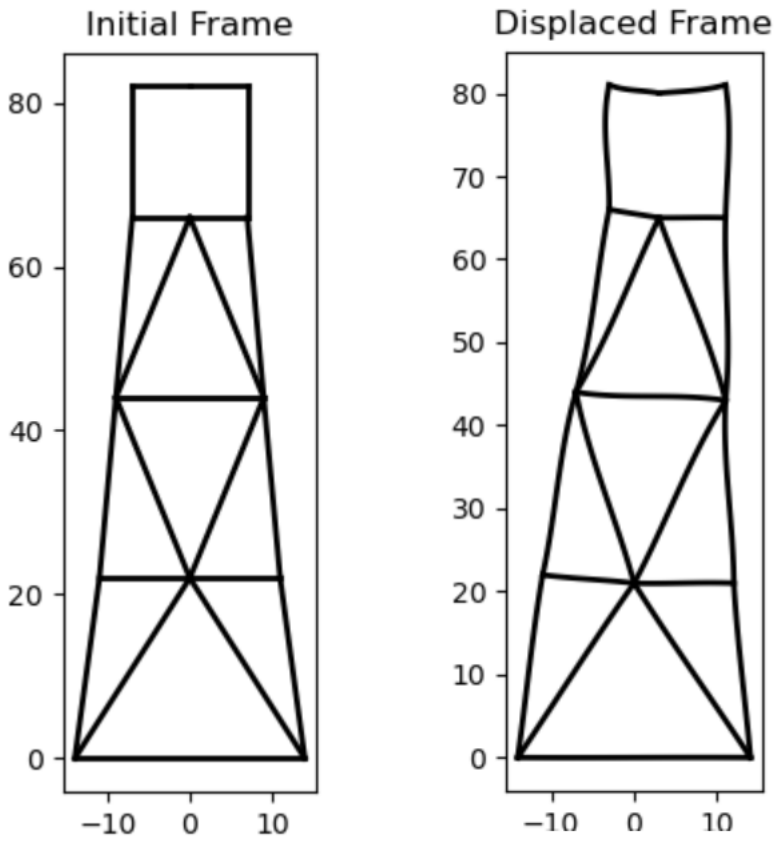


Figure 21: Initial and deformed frame.

5.7 Moment diagrams visualization

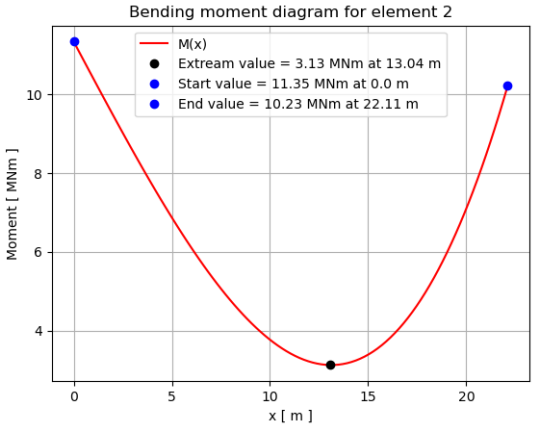


Figure 22: Python plot of moment diagram for element 2

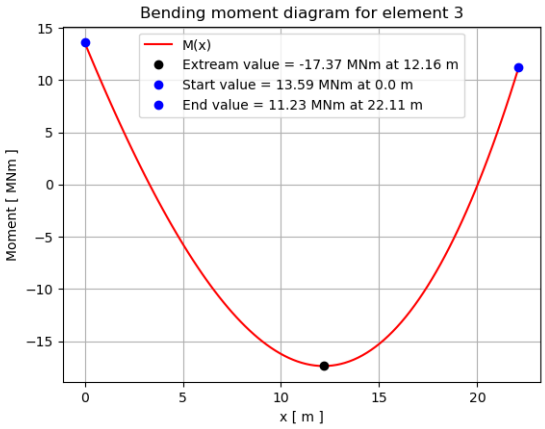


Figure 23: Python plot of moment diagram for element 3

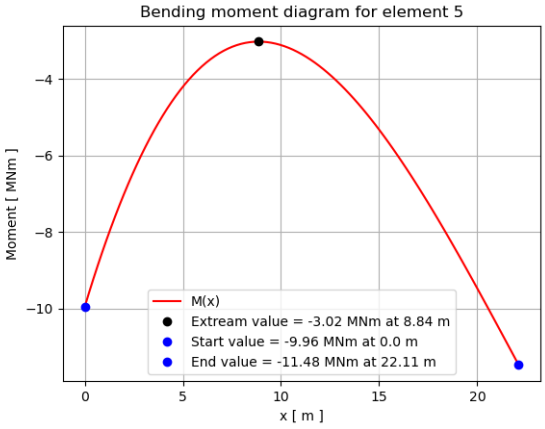


Figure 24: Python plot of moment diagram for element 5

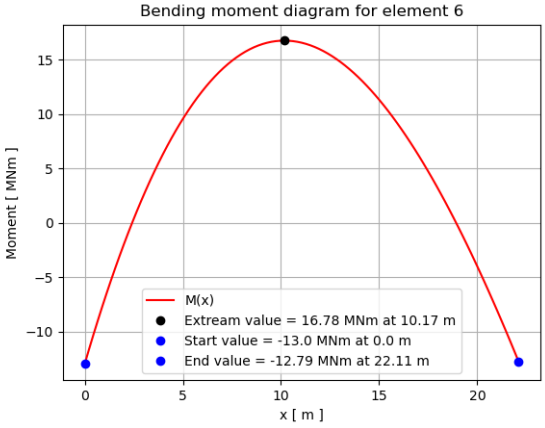


Figure 25: Python plot of moment diagram for element 6

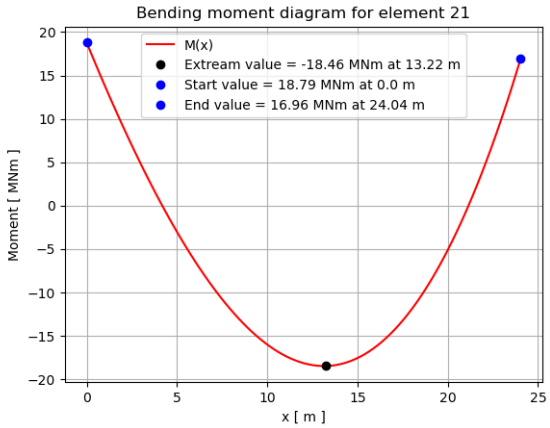


Figure 26: Python plot of moment diagram for element 21

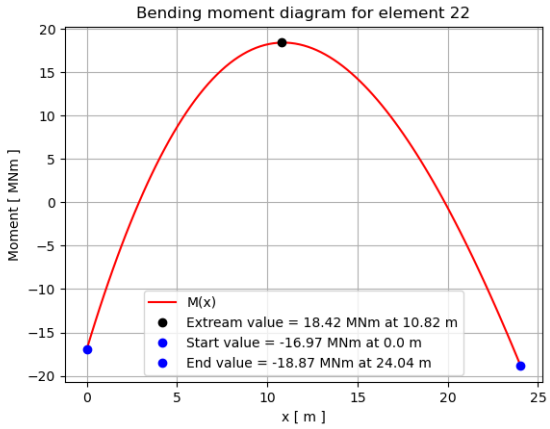


Figure 27: Python plot of moment diagram for element 22

6 Discussion

In this chapter the results from chapter 5 are discussed. In the first part of the chapter we compared the python program with the results from hand calculation and 3DBeam at the small frames. The deviation here was approximate 0%, and was good indication that our python program worked.

Four of the beams in the construction, beam 2, 5, 21, and 22 respectively, were on the brink of the range given in the task $(30 - 70\%)$. From the task we were in doubt whether to include both bending stress and axial stress in the process of deciding dimensions. However, it does not make sense to consider only bending stress, as the total stress in beam 2 and 5 will exceed yield stress if axial stress is counted. Therefore, we have considered stress from both bending moment and axial force as we didn't want to design a construction that will fall apart.

The biggest deviations in bending moment between our program and 3DBeam were around 10%. There are not any direct connection between these beams. However, since the biggest deviation mainly exists at the end with the least bending moment, the deviation will not have a big impact in deciding dimensions. There are no direct systems to the beams with higher moment-deviations, which makes it difficult to pinpoint the reasons for our programs relatively small deviations. Overall though, the program does not have any alarming moment outputs, compared to 3DBeam. The calculations of maximum field moment for the beams affected by distributed loads are also as precise as it can be compared to, given slightly off start values.

There are also difficult to explain the reason of deviation at shear forces and normal forces. Since the deviation in general are low, the deviation can partly be explained by the reason that the matrix method not consider shear deformations, just deformations from bending moment. In addition 3DBeam rounds numbers which can lead to deviations.

We have also noticed that the stress in the jackets horizontal beams, beam $12 - 16$, in our script is about $10\% \pm 2\%$ greater than the values from 3D Beam. This is shown in figure 12. However, we see no apparent systematic reason for this and believe that this is a coincidence. Since there are only 5 such beams, it is hard to

conclude that this is a systematic error.

7 Conclusion

In this project Python has been used to successfully analyze a frame with the matrix method. The simulation returns results for bending moments, shear forces and bending stress for the jacket. By comparing to 3DBeam, the result is that the simulation has a high precision, with $[0-13]\%$ deviation for bending moment values along all elements and $[0-3]\%$ and $[0-8]\%$ deviation in axial and shear force respectively for almost all elements. These values were used to create diagrams for bending moment, shear force, and normal force, in addition to choose dimensions for the construction. In the end, all elements has a maximum total stress within $[30-70]\%$ of yield stress.

The small deviations can be explained by some differences between 3DBeam and Python in the methods used to analyse the frame. However, these deviations did not have any in the process of choosing dimensions, and in general there were not any alarming output from the Python program. The results are trustworthy and simulates a 2D frames with a high precision.

Bibliography

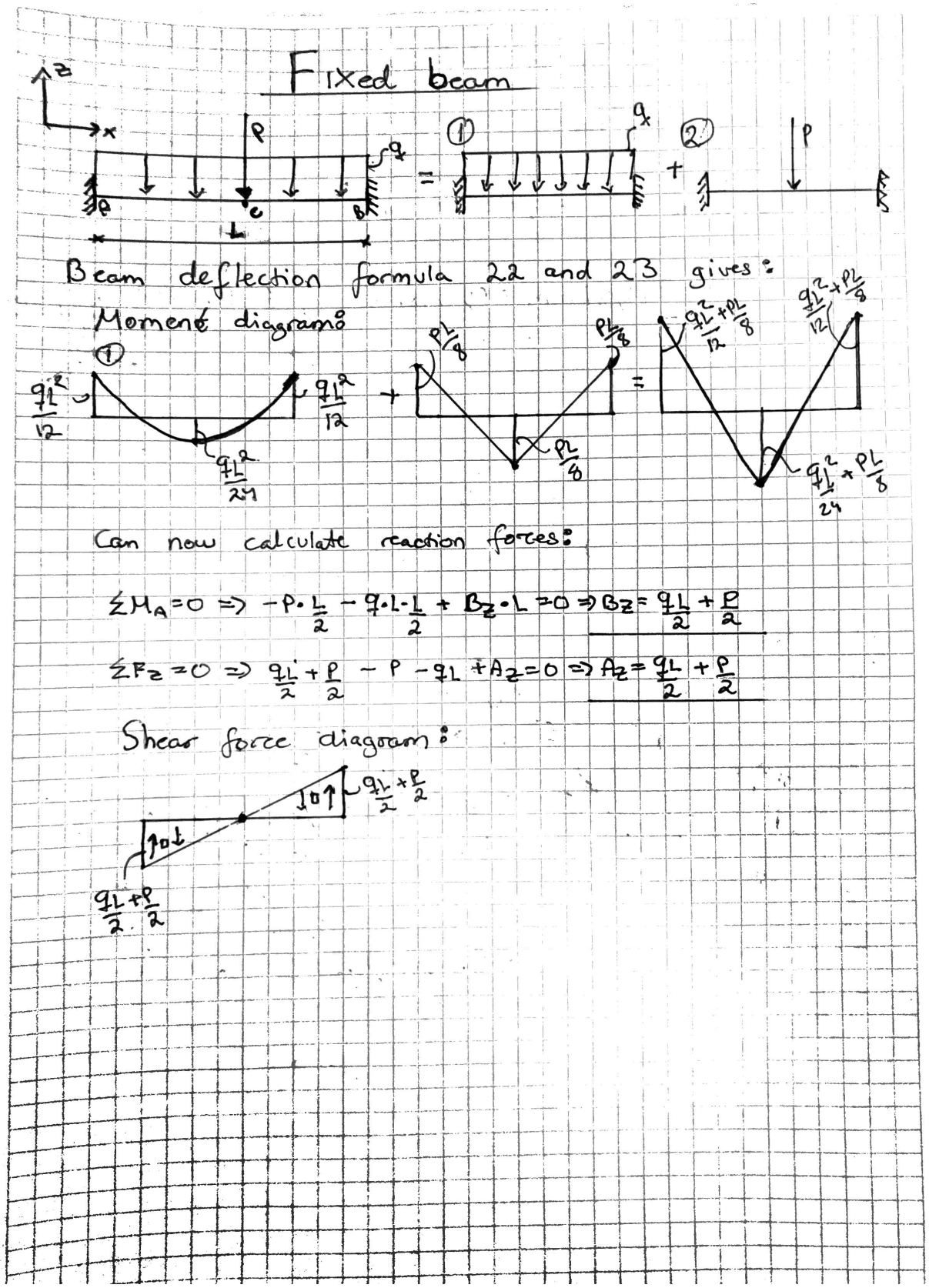
[1] Jørgen Amdahl. *Kompendium for bruk i faget TMR4167 Marin teknikk 2*. Fakultet for ingeniørvitenskap og teknologi, 2010.

[2] DNV GL. *Beam structural analysis - Nauticus Hull - 3D Beam*. URL: https://www.dnv.com/services/beam-structural-analysis-nauticus-hull-3d-beam-4541 (visited on 9th Nov. 2021).

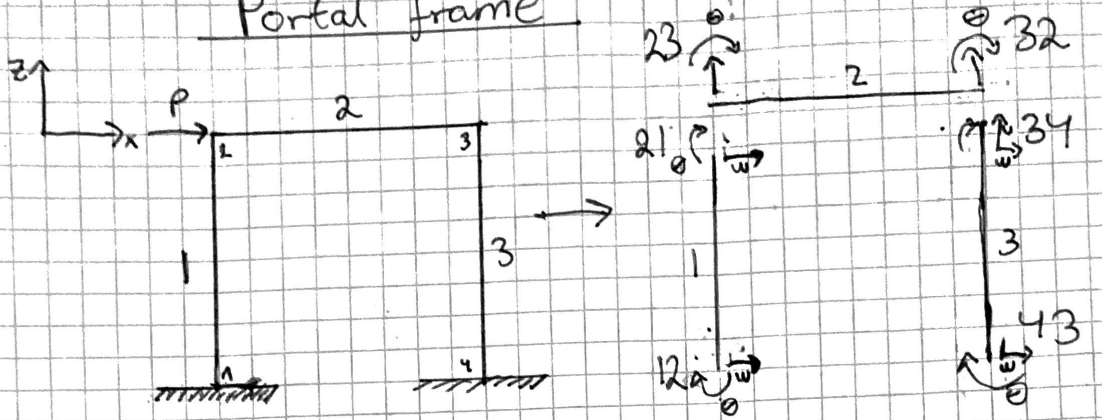
[3] Department of Marine Technology NTNU. *IMT Software Wiki - Prosjektoppgave i TMR4167 Marin teknikk - Konstruksjoner*. URL: https://www.ntnu.no/wiki/display/imtsoftware/Prosjektoppgave+i+TMR4167+Marin+teknikk+-+Konstruksjoner (visited on 9th Nov. 2021).

Appendix

A Calculations by hand



Portal frame



Due to boundary conditions and restrained degrees of freedom, we can create following local stiffness matrices to each element:

Element 1: $\begin{bmatrix} Q_{21} \\ M_{21} \end{bmatrix} = \frac{2EI}{L} \begin{bmatrix} 6/L^2 & 3/L \\ 3/L & 2 \end{bmatrix} \begin{bmatrix} w_{21} \\ \theta_{21} \end{bmatrix}$

Element 2: $\begin{bmatrix} M_{23} \\ M_{32} \end{bmatrix} = \frac{2EI}{L} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \theta_{23} \\ \theta_{32} \end{bmatrix}$

Element 3: $\begin{bmatrix} Q_{34} \\ M_{34} \end{bmatrix} = \frac{2EI}{L} \begin{bmatrix} 6/L^2 & 3/L \\ 3/L & 2 \end{bmatrix} \begin{bmatrix} w_{34} \\ \theta_{34} \end{bmatrix}$

At node 2 and 3 we need moment equilibrium, and the sum of the shear forces Q_{21} and Q_{34} has to be equal to the size of the external force P.

$w_{21} = w_{34}$

$$\sum M_2 = 0 \Rightarrow \frac{2EI}{L}\left(\frac{3}{L}w + 4\theta_2 + \theta_3\right) = 0$$

$$\sum M_3 = 0 \Rightarrow \frac{2EI}{L}\left(\theta_2 + 4\theta_3 + \frac{3}{L}w\right) = 0$$

$$\sum Q = 0 \Rightarrow \frac{2EI}{L}\left(\frac{12}{L^2}w + \frac{3\theta_2}{L} + \frac{3\theta_3}{L}\right) + P = 0$$

We transform the equations to matrices.

$$\frac{2EI}{L}\begin{bmatrix} 3/L & 4 & 1 \\ 3/L & 1 & 4 \\ 12/L^2 & 3/L & 3/L \end{bmatrix}\begin{bmatrix} w \\ \theta_2 \\ \theta_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -P \end{bmatrix}$$

By using inverse calculator:

$$\left(\frac{2EI}{L}\begin{bmatrix} 3/L & 4 & 1 \\ 3/L & 1 & 4 \\ 12/L^2 & 3/L & 3/L \end{bmatrix}\right)^{-1} = \frac{L}{28EI}\begin{bmatrix} -L & -L & 5/3L^2 \\ 13/3 & -1/3 & -L \\ -1/3 & 13/3 & -L \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} w \\ \theta_2 \\ \theta_3 \end{bmatrix} = \frac{L}{28EI}\begin{bmatrix} -L & -L & 5/3L^2 \\ 13/3 & -1/3 & -L \\ -1/3 & 13/3 & -L \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ -P \end{bmatrix}$$

$$w = -\frac{5L^3 \cdot P}{84EI} \quad , \quad \theta_2 = \frac{L^2 \cdot P}{28EI} = \theta_3$$

$$M_{21} = \frac{2EI}{L}\left(\frac{3}{L}w + 2\theta_2\right) = \underline{\underline{-\frac{3P}{14}}}$$

$$M_{23} = \frac{2EI}{L}\left(2\theta_2 + \theta_3\right) = \underline{\underline{\frac{3P}{14}}}$$

$$M_{32} = \frac{2EI}{L}\left(\theta_2 + 2\theta_3\right) = \underline{\underline{\frac{3P}{14}}}$$

$$M_{34} = \frac{2EI}{L}\left(\frac{3}{L}w + 2\theta_3\right) = \underline{\underline{-\frac{3P}{14}}}$$

$$Q_{21} = \frac{2EI}{L}\left(\frac{6}{L^2}w + \frac{3}{L}\theta_2\right) = \underline{\underline{-\frac{1}{2}P}}$$

$$Q_{34} = \frac{2EI}{L}\left(\frac{6}{L^2}w + \frac{3}{L}\theta_3\right) = \underline{\underline{-\frac{1}{2}P}}$$

By using the derived stiffness matrix in chapter 8.5.3:

$$M_{12} = \frac{2EI}{L}\left(\frac{3}{L}w + \theta_2\right) = -\frac{PL}{7}$$

$$M_{43} = \frac{2EI}{L}\left(\frac{3}{L}w + \theta_3\right) = -\frac{PL}{7}$$

B Fixed support moment for different load cases

Figure 8.3 in "kompendium for bruk i Marin Teknikk 2" (Amdahl *et al.*, 2010) shows the moment in a and b for a beam with a fixed support in both ends when exposed to different load situations.

Lasttilfelle	a, l, b	a, l, b
	m_{ab}	m_{ba}	m_{ba}
P, a, b	$-\frac{P \cdot ab^2}{l^2}$	$\frac{P \cdot a^2 b}{l^2}$	$\frac{P \cdot a}{2l^2}(l^2 - a^2)$
P, P, l/3, l/3, l/3	$-\frac{2}{9}$ Pl	$\frac{2}{9}$ Pl	$\frac{1}{3}$ Pl
P, P, P, l/4, l/4, l/4, l/4	$-\frac{5}{16}$ Pl	$\frac{5}{16}$ Pl	$\frac{15}{32}$ Pl
p	$-\frac{1}{12} pl^2$	$\frac{1}{12} pl^2$	$\frac{1}{8} pl^2$
p, s, b	$-\frac{ps^2}{12l^2}(6b^2+4bs+s^2)$	$\frac{ps^3}{12l^2}(4b + s)$	$\frac{ps^2}{8}(2 - \frac{s^2}{l^2})$
p, a, s	$-\frac{ps^3}{12l^2}(4a + s)$	$\frac{ps^2}{12l^2}(6a^2+4as+s^2)$	$\frac{ps^2}{8}(1 + \frac{a}{l})^2$
p, a, s, a	$-\frac{ps}{24l}(3l^2 - s^2)$	$-\frac{ps}{24l}(3l^2 - s^2)$	$\frac{ps}{16l}(3l^2 - s^2)$
p	$-\frac{1}{30} pl^2$	$\frac{1}{20} pl^2$	$\frac{1}{15} pl^2$
p	$-\frac{1}{20} pl^2$	$\frac{1}{30} pl^2$	$\frac{7}{120} pl^2$
p, l, s	$-\frac{ps^3}{60l}(5 - \frac{3s}{l})$	$\frac{ps^2}{60}(10-10\frac{s}{l}+3\frac{s^2}{l^2})$	$\frac{ps^2}{120}(20-15\frac{s}{l}+3\frac{s^2}{l^2})$
p, s, l	$-\frac{ps^2}{60}(10-10\frac{s}{l}+3\frac{s^2}{l^2})$	$\frac{ps^3}{60l}(5 - \frac{3s}{l})$	$\frac{ps^2}{120}(10 - 3\frac{s^2}{l^2})$
p, s, b	$-\frac{ps^2}{30}(10-15\frac{s}{l}+6\frac{s^2}{l^2})$	$\frac{ps^3}{20l}(1 + 4\frac{b}{l})$	$\frac{ps^2}{30}(5-5\frac{s}{l}+\frac{s^2}{l^2})$

C Input file for Jacket construction

```
NODE,   -14.1,  0.0,    2,  2,  1
NODE,   14.1,   0.0,    2,  2,  1
NODE,   -11.9,  22.0,   1,  1,  1
NODE,   11.9,   22.0,   1,  1,  1
NODE,   -9.7,   44.0,   1,  1,  1
NODE,   9.7,    44.0,   1,  1,  1
NODE,   -7.5,   66.0,   1,  1,  1
NODE,   7.5,    66.0,   1,  1,  1
NODE,   -7.5,   82.0,   1,  1,  1
NODE,   7.5,    82.0,   1,  1,  1
NODE,   0.0,    82.0,   1,  1,  1
NODE,   0.0,    22.0,   1,  1,  1
NODE,   0.0,    66.0,   1,  1,  1
---
BEAM,   1,  3,  1, 2,  5
BEAM,   3,  5,  1, 2,  5
BEAM,   5,  7,  1, 2,  5
BEAM,   4,  2,  1, 2,  5
BEAM,   6,  4,  1, 2,  5
BEAM,   8,  6,  1, 2,  5
BEAM,   7,  9,  2, 1,  4
BEAM,   10, 8,  2, 1,  4
BEAM,   9,  11, 2, 1,  5
BEAM,   11, 10, 2, 1,  5
BEAM,   1,  2,  1, 2,  3
BEAM,   3,  12, 1, 2,  3
BEAM,   12, 4,  1, 2,  3
BEAM,   5,  6,  1, 2,  3
BEAM,   7,  13, 1, 2,  3
BEAM,   13, 8,  1, 2,  3
BEAM,   1,  12, 1, 2,  4
BEAM,   12, 2,  1, 2,  4
BEAM,   5,  12, 1, 2,  4
BEAM,   12, 6,  1, 2,  4
BEAM,   5,  13, 1, 2,  4
BEAM,   13, 6,  1, 2,  4
---
```

```
MATERIAL,   210000000000,   300000000
MATERIAL,   70000000000,    300000000
---
NODELOAD,   9,  0,  -3000000, 0
NODELOAD,   10, 0,  -3000000, 0
NODELOAD,   11, 0,  -1000000, 0
---
BEAMLOAD,   2,  0,      0,  180000, 0
BEAMLOAD,   3,  180000, 0,  540000, 0
BEAMLOAD,   5,  180000, 0,  0,      0
BEAMLOAD,   6,  540000, 0,  180000, 0
BEAMLOAD,   21, 180000, 0,  540000, 0
BEAMLOAD,   22, 540000, 0,  180000, 0
---
REFERENCEDIAMETER,  1.5
---
PIPE,   0.75, 0.7
PIPE,   0.8,  0.74
PIPE,   0.85,  0.79
PIPE,   0.95,  0.887
PIPE,   1,    0.93
---
IPE,    0.4, .18, .0086, .0135
IPE,    .6, .22, .012, .019
IPE,    0.75, .264, .012, .0155
IPE,    1, .3, .019, .036
IPE,    1.1, .4, .02, .036
```

D Python-Output

```
Node displacements
Node  u [mm]  w [mm]  theta [rad]
 1    0.0   0.0   0.0003
 2    0.0   -0.0    0.0003
 3    22.0    13.0    0.0019
 4    22.0    -14.0   0.0019
 5    102.0   20.0    0.005
 6    102.0   -22.0   0.005
 7    176.0   11.0    0.0012
 8    176.0   -15.0   0.0014
 9    200.0   -10.0   0.0109
 10   199.0   -35.0   -0.0076
 11   200.0   -82.0   0.0017
 12   21.0    -1.0    0.0014
 13   175.0   -1.0    0.0004

Moments
Beam  M1 [MNm]  M2 [MNm]  M_field_max [ MNm ]
 1    -0.97     5.07
 2    -11.35    10.23   3.13
 3    -13.59    11.23   -17.37
 4    5.09    -1.03
 5    9.96    -11.48    -3.02
 6    13.0    -12.79    16.78
 7    0.49    1.02
 8    -1.0    -0.51
 9    -1.02     -2.74
 10   2.74    1.0
 11   1.32    1.29
 12   6.28    4.38
 13   4.47    6.39
 14   19.06     18.99
 15   -11.72    -15.93
 16   -18.0     -12.49
 17   -0.36     2.33
 18   2.46    -0.26
 19   3.1     -6.67
```

```
 20    -6.96      2.72
 21    -18.79     16.96    -18.46
 22    16.97      -18.87     18.42

Sheer
Beam   Q1 [MN]  Q2 [MN]
 1     -0.19      0.19
 2     0.94     1.73
 3     4.55     6.11
 4     -0.18      0.18
 5     -1.71      -0.96
 6     -6.23      -4.43
 7     -0.09      0.09
 8     0.09     -0.09
 9     0.5      -0.5
 10    -0.5     0.5
 11    -0.09      0.09
 12    -0.9     0.9
 13    -0.91      0.91
 14    -1.96      1.96
 15    3.69     -3.69
 16    4.07     -4.07
 17    -0.08      0.08
 18    -0.08      0.08
 19    0.15     -0.15
 20    0.18     -0.18
 21    5.07     6.91
 22    -6.91      -5.07

Axial  Force
Beam   N [MN]
 1     61.21
 2     60.42
 3     -7.83
 4     -65.73
 5     -64.93
 6     1.19
 7     -3.5
 8     -3.5
```

```
9     -0.09
10    -0.09
11    0.0
12    -1.04
13    1.06
14    0.66
15    -6.77
16    6.41
17    32.18
18    -35.14
19    -43.26
20    40.52
21    31.81
22    -32.22

Stress
Beam   Sigma_x [MPa] %fy
1        170.0      57.0%
2        200.0      67.0%
3        87.0       29.0%
4        181.0      60.0%
5        211.0      70.0%
6        68.0       23.0%
7        171.0      57.0%
8        169.0      56.0%
9        153.0      51.0%
10       153.0      51.0%
11       11.0       4.0%
12       55.0       18.0%
13       56.0       19.0%
14       158.0      53.0%
15       152.0      51.0%
16       168.0      56.0%
17       103.0      34.0%
18       112.0      37.0%
19       160.0      53.0%
20       155.0      52.0%
21       204.0      68.0%
22       205.0      68.0%
```

A Main

```
'''
Developed at the Norwegian University of Science and Technology, Department of Marine Technology
08.11.2021 as part of TMR4167 Marin teknikk - Konstruksjoner
'''

# Imports sub python files
from functions import *
from plotVisualization import *
from readTextToArray import *
from structure_visualization import *


# This is the command room, here we control the program.

def main():
    # Reads from inputfile and sorts the data in appropriate numpy arrays.
    # Make sure the file path is correct for your enviroment's working-directory.
    nodeArray, beamArray, materialArray, nodeloadArray, beamloadArray, pipeLibrary, IPELibrary, referenceDiameter = readInputFile(
        "InputDataJacket.txt")

    # Initialize beams and nodes objects and appends to list.
    nodesObjectList, beamsObjectList = initializeNodesAndBeamsList(nodeArray, beamArray)

    # Uses specifications from input files to create geometry for each beam.
    beamsObjectList = makeBeamsGeometry(beamArray, beamsObjectList, pipeLibrary, IPELibrary)

    # Gives correct Young's modulus to each beam.
    beamsObjectList = giveYoungsModulusToBeams(beamsObjectList, materialArray, beamArray)
```

```
29
30      # Enumerates each beam.
31      beamsObjectList = giveNumberToObjects(beamsObjectList)
32
33      # Enumerates each node.
34      nodesObjectList = giveNumberToObjects(nodesObjectList)
35
36      # Creates the local stiffness matrix for each beam.
37      beamsObjectList =
    giveLocalStiffnessMatrixToBeamsLocalOrientation(beamsObjectList)
38
39      # Orients the local stiffness matrix to global coordinates for
    each beam.
40      beamsObjectList =
    giveLocalStiffnessMatrixToBeamsGlobalOrientation(beamsObjectList
    )
41
42      # Connects distributed loads to corresponding beams.
43      beamsObjectList = addDistributedLoadsToBeam(beamsObjectList,
    beamloadArray)
44
45      # Scale distributed loads with respect to reference diameter.
46      beamsObjectList = scaleDistributedBeamLoads(beamsObjectList,
    referenceDiameter)
47
48      # Calculates fixed support moment and forces for each beam.
49      beamsObjectList = calculateFixedSupportMomentAndForces(
    beamsObjectList)
50
51      # Connects node loads to corresponding nodes.
52      nodesObjectList = connectNodeLoadsToNodes(nodesObjectList,
    nodeloadArray)
53
54      # Makes the resulting load vector.
55      resultingLoadVector = makeResultingLoadVector(nodesObjectList)
56
57      # Makes the global stiffness matrix.
58      globalStiffnessMatrix = makeGlobalStiffnessMatrix(
    beamsObjectList, nodesObjectList)
```

```
59
60      # Uses numpy.linalg.solve() to solve the system relation for
        the global displacement vector.
61      globalDisplacementVector = np.linalg.solve(
        globalStiffnessMatrix, resultingLoadVector)
62
63      # Calculates reaction forces for each beam.
64      beamsObjectList = calculateBeamReactionForces(beamsObjectList,
        globalDisplacementVector)
65
66      # Calculates max moment and bending tension for each beam.
67      beamsObjectList = calculateMaxMomentAndBendingTension(
        beamsObjectList)
68
69      # Makes and appends moment diagrams for beams with distributed
        loads to beams.
70      beamsObjectList = appendMomentDiagramToBeams(beamsObjectList)
71
72      # Prints specifications for each beam to terminal for debugging
        .
73      printBeamSpecsToTerminal(beamsObjectList)
74
75      # The handed out code structure_visualization.py is altered to
        match our code and used to plot our jacket
76      # construction. Displacements er scaled by 20.
77      plot(nodeArray, beamArray, globalDisplacementVector * 20)
78
79      # Plots moment diagrams for beams with distributed load.
80      plotMomentDiagram(beamsObjectList)
81
82      # Export data to txt-file
83      outputResultsToFile('Results', beamsObjectList, nodesObjectList
        , globalDisplacementVector)
84
85      return 0
86
87
88  main()
```

B Functions

```
'''
Developed at the Norwegian University of Science and Technology, Department of Marine Technology
08.11.2021 as part of TMR4167 Marin teknikk - Konstruksjoner
'''

from classes import *
from importedLibraries import *


# This file includes relevant functions.

def initializeNodesAndBeamsList(nodeArray, beamArray):
    '''
    Makes and initializes the beams and nodes object lists
    param nodeArray: matrix containing node-data from input file
    param beamArray: matrix containing beam-data from input file
    return: Object lists for nodes, beams
    '''
    nodesObjectList = []
    beamsObjectList = []

    for i in range(len(nodeArray)):
        nodesObjectList.append(
            Node(nodeArray[i][0], nodeArray[i][1], nodeArray[i][2], nodeArray[i][3], nodeArray[i][4]))

    for j in range(len(beamArray)):
        node1 = nodesObjectList[int(beamArray[j][0]) - 1]
        node2 = nodesObjectList[int(beamArray[j][1]) - 1]
        beamsObjectList.append(Beam(node1, node2))

    return nodesObjectList, beamsObjectList


def makeBeamsGeometry(beamArray, beamsObjectList, pipeLibrary, IPELibrary):
```

```
    '''
    Decides from the input file which geometry to add to each beam object
    param beamArray: matrix containing beam-data from input file
    param beamsObjectList: list containing all beam-objects
    param pipeLibrary: matrix containing pipe-data from input file
    param IPELibrary: matrix containing IPE-data from input file
    return: Beams object list
    '''
    i = 0
    for beam in beamsObjectList:
        IPEorPipe = beamArray[i][3]
        if IPEorPipe == 1:
            # If beam geometry is IPE
            geo = int(beamArray[i][4] - 1)
            # IPE library includes hight, w top, w bot, w mid, t top, t bot
            beam.makeIPE(IPELibrary[geo][0], IPELibrary[geo][1], IPELibrary[geo][2],
                         IPELibrary[geo][3])

        elif IPEorPipe == 2:
            # Else if beam geometry is pipe
            geo = int(beamArray[i][4] - 1)
            # Pipe library includes radius, ratio air
            beam.makePipe(pipeLibrary[geo][0], pipeLibrary[geo][1])

        i += 1

    return beamsObjectList


def giveYoungsModulusToBeams(beamsObjectList, materialArray, beamArray):
    '''
    This function appends Youngs modulus to elements
    :param beamsObjectList: List of beam objects
    :param materialArray: Library of materials
    :param beamArray: Array of beam data from input file
```

```
70      :return: Beams object list
71      '''
72      i = 0
73      for beam in beamsObjectList:
74          materialType = int(beamArray[i][2] - 1)
75          # Matrial array includes aluminium and steel
76          beam.makeStiffness(materialArray[materialType])
77          i += 1
78
79      return beamsObjectList
80
81
82  def giveNumberToObjects(objectList):
83      '''
84      Enumerates objects in a list
85      :param objectList: A list of objects, can be both nodes and
        beams
86      :return: A list of objects
87      '''
88      for i in range(len(objectList)):
89          objectList[i].giveNumber(i + 1)
90      return objectList
91
92
93  def giveLocalStiffnessMatrixToBeamsLocalOrientation(beamsObjectList
        ):
94      '''
95      Uses the Beam member function makeLocalStiffnessMatrix() to
        make a local stiffness
96      matrix to each beam in the list of beam objects
97      :param beamsObjectList: A list of beam objects
98      :return: A list of beam objects
99      '''
100     for beam in beamsObjectList:
101         beam.makeLocalStiffnessMatrix()
102     return beamsObjectList
103
104
105 def giveLocalStiffnessMatrixToBeamsGlobalOrientation(
```

```
    beamsObjectList):
106     '''
107     Transforms the element stiffness matrix.
108     :param beamsObjectList: A list of beam objects
109     :return: A list of beam objects
110     '''
111     for i in range(len(beamsObjectList)):
112         beamsObjectList[i].transformElementStiffnessMatrix()
113     return beamsObjectList
114 
115 
116 def addDistributedLoadsToBeam(beamsObjectList, beamloadArray):
117     '''
118     Adds distributed loads from input file to beam objects
119     :param beamsObjectList: A list of beam objects
120     :param beamloadArray: An array of beam loads
121     :return: A list of beam objects
122     '''
123     for i in range(len(beamloadArray)):
124         for beam in beamsObjectList:
125             if (beam.number == beamloadArray[i][0]):
126                 beam.addDistributedNormalLoad(beamloadArray[i])
127     return beamsObjectList
128 
129 
130 def scaleDistributedBeamLoads(beamsObjectList, referenceDiameter):
131     '''
132     Scales distributed loads on beam objects according to their
    diameter with respect
133     to the reference diameter.
134     :param beamsObjectList: A list of beam objects
135     :param referenceDiameter: A reference diameter
136     :return: A list of beam objects
137     '''
138     if referenceDiameter == -1:
139         pass
140     else:
141         for beam in beamsObjectList:
142             beam.scaleDistributedLoad(referenceDiameter)
```

```
143     return beamsObjectList
144
145
146 def calculateFixedSupportMomentAndForces(beamsObjectList):
147     '''
148     Calculates each beams fixed support moment and forces at both
        ends.
149     :param beamsObjectList: A list of beam objects
150     :return: A list of beam objects
151     '''
152     for beam in beamsObjectList:
153         beam.calculateFixedSupport()
154     return beamsObjectList
155
156
157 def connectNodeLoadsToNodes(nodesObjectList, nodeloadArray):
158     '''
159     Conncts the nodeloads to the node objects
160     :param nodesObjectList: a list of node objects
161     :param nodeloadArray: an array node data from input file
162     :return: a list of node objects
163     '''
164
165     for i in range(len(nodeloadArray)):
166         for j in range(len(nodesObjectList)):
167             if (nodesObjectList[j].number == nodeloadArray[i][0]):
168                 nodesObjectList[j].addNodeLoad(nodeloadArray[i])
169     return nodesObjectList
170
171
172 def makeResultingLoadVector(nodesObjectList):
173     '''
174     Makes a list of fixed support forces and moments that satisfy
175     the system relation, K*r = R.
176     :param nodesObjectList: A list of nodes
177     :return: An 1x3n array
178     '''
179     R = []
180     for i in range(len(nodesObjectList)):
```

```
        R.append(-nodesObjectList[i].Fx)  # u
        R.append(-nodesObjectList[i].Fz)  # w
        R.append(-nodesObjectList[i].M)  #
    return np.array(R)


def makeGlobalStiffnessMatrix(beamsObjectList, nodesObjectList):
    '''
    Makes the global stiffness matrix by adding each beams
    transformed stiffness matrix in position
    given by the nodes each beam is connected to.
    We then account for fixed support conditions.
    :param beamsObjectList: A list of beam objects
    :param nodesObjectList: A list of node objects
    :return: A 3n x 3n array
    '''


    M = np.zeros((3 * len(nodesObjectList), 3 * len(nodesObjectList
    )))

    # Make stiffness matrix
    for beam in beamsObjectList:
        n1 = beam.node1.number - 1
        n2 = beam.node2.number - 1
        K = beam.transformedStiffnessMatrix
        for i in range(3):
            for j in range(3):
                M[n1 * 3 + i][n1 * 3 + j] += K[i][j]
                M[n2 * 3 + i][n2 * 3 + j] += K[3 + i][3 + j]
                M[n1 * 3 + i][n2 * 3 + j] += K[i][3 + j]
                M[n2 * 3 + i][n1 * 3 + j] += K[3 + i][j]

    # Account for fixed support conditions
    for n, node in enumerate(nodesObjectList):
        for k in range(3):
            if k == 0:
                displacement = node.supportX
            elif k == 1:
```

```
218                 displacement = node.supportZ
219             elif k == 2:
220                 displacement = node.supportTheta
221 
222             if displacement == 1:  # Do nothing
223                 pass
224             elif displacement == 2:  # Multiply diagonal with 10^6
225                 M[n * 3 + k][n * 3 + k] = M[n * 3 + k][n * 3 + k] *
    10 ** 6
226             elif displacement == 0:  # Make diagonal 1, rest of row
   /collumn 0
227                 for l in range(len(nodesObjectList)):
228                     M[n * 3 + k][l] = 0
229                     M[l][n * 3 + k] = 0
230                 M[3 * n + k][3 * n + k] = 1
231 
232     return M
233 
234 
235 def calculateBeamReactionForces(beamsObjectList, r):
236     '''
237     Calculates beams reaction force and adds these as member-
   variables to each beam
238     param beamsObjectList: List of beam objects
239     :param beamsObjectList: Array containing resulting loads
240     :param r: A list of beam objects
241     :return: a list of beam objects
242     '''
243     for beam in beamsObjectList:
244         localDisplacements = np.zeros(6)
245         n1 = beam.node1.number - 1
246         n2 = beam.node2.number - 1
247         for i in range(3):
248             localDisplacements[i] = r[3 * n1 + i]
249             localDisplacements[3 + i] = r[3 * n2 + i]
250         beam.localDisplacements = np.matmul(beam.T_transponent,
   localDisplacements)
251         beam.reactionForces = np.matmul(beam.localStiffnessMatrix,
   beam.localDisplacements)
```

```
252         if (beam.hasDistributedLoad):
253             m1 = (1 / 20) * beam.q1 * (beam.length) ** 2 + (1 / 30)
    * beam.q2 * (beam.length) ** 2
254             m2 = -(1 / 30) * beam.q1 * (beam.length) ** 2 - (1 /
    20) * beam.q2 * (beam.length) ** 2
255
256             v2 = (m1 + m2 - (beam.q1 * beam.length ** 2) / 6 - (
    beam.q2 * beam.length ** 2) / 3) / beam.length
257             v1 = -(beam.length / 2) * (beam.q1 + beam.q2) - v2
258
259             beam.reactionForces += np.array([0, v1, m1, 0, v2, m2],
    dtype=float)
260         else:
261             pass
262     return beamsObjectList
263
264
265 def appendMomentDiagramToBeams(beamsObjectList):
266     '''
267     Appends moment diagram and related values as member variabels
    to beam.
268     :param beamsObjectList: A list of beam objects
269     :return: A list of beam objects
270     '''
271     for beam in beamsObjectList:
272         if beam.hasDistributedLoad:
273             beam.appendMomentDiagramToBeam()
274     return beamsObjectList
275
276
277 def printBeamSpecsToTerminal(beamsObjectList, s=0, n=999):
278     '''
279     Prints the reaction forces for beam s to s + n. This is mostly
    used for debugging.
280     :param beamsObjectList: A list of beam objects
281     :param s: First beam number
282     :param n: First beam + n is last beam to be printed
283     :return: prints to console
284     '''
```

```
285
286     if n > len(beamsObjectList) - s:
287         n = len(beamsObjectList) - s
288     for i in range(n):
289         beamsObjectList[i + s].printBeam()
290         beamsObjectList[i + s].printBeamMoments()
291         beamsObjectList[i + s].printSecurityFactor()
292
293
294 def calculateMaxMomentAndBendingTension(beamsObjectList):
295     '''
296     Calculates max moment and bending tension by using beam class
    member functions.
297     :param beamsObjectList: A list of beam objects
298     :return: A list of beam objects
299     '''
300     for beam in beamsObjectList:
301         beam.calculateMaxTension()
302     return beamsObjectList
303
304
305
306 def plotMomentDiagram(beamsObjectList):
307     '''
308     Takes in moment diagram and related member variables and plots
    the moment diagram
309     with extreme values maked in the diagrams for each beam with a
    distributed load.
310     :param beamsObjectList: a list of beam objects
311     :return: nothing
312     '''
313     Micro = 10 ** -6
314     i = 0
315     for beam in beamsObjectList:
316         if (beam.hasDistributedLoad):
317             plt.figure(i)
318             plt.plot(np.array(beam.equidistributedX), np.array(beam
    .momentDiagram) * Micro, c='r', label='M(x)')
319             plt.xlabel('x [ m ]')
```

```
320             plt.ylabel('Moment [ MNm ]')
321             plt.title('Bending moment diagram for element ' + str(
    beam.number))
322             plt.grid()
323             plt.plot()
324 
325             # Extract extreme values and makes the graphs more
    readable.
326             xMax = beam.localMaxMomentX
327             maxValueMNm = beam.localMaxMoment * Micro
328             startValueMNm = beam.momentDiagram[0] * Micro
329             endValueMNm = beam.momentDiagram[-1] * Micro
330             plt.plot(xMax, maxValueMNm, 'o', c='black',
331                      label='Extream value = ' + str(round(
    maxValueMNm, 2)) + ' MNm at ' + str(round(xMax, 2)) + ' m')
332             plt.plot(beam.equidistributedX[0], startValueMNm, 'o',
    c='blue',
333                      label='Start value = ' + str(round(
    startValueMNm, 2)) + ' MNm at ' + str(
334                          round(beam.equidistributedX[0], 2)) + ' m'
    )
335             plt.plot(beam.equidistributedX[-1], endValueMNm, 'o', c
    ='blue',
336                      label='End value = ' + str(round(endValueMNm,
    2)) + ' MNm at ' + str(
337                          round(beam.equidistributedX[-1], 2)) + ' m
    ')
338             plt.legend()
339 
340             i += 1
341 
342     plt.show()
343 
344 
345 def outputResultsToFile(filename, beamsObjectList, nodesObjectList,
     r):
346     '''
347     Makes a file and writes result data to it.
348     :param filename: file name
```

```
349     :param beamsObjectList: a list of beam objects
350     :param nodesObjectList: a list of node objects
351     :param r: the resulting load vector
352     :return: nothing
353     '''
354     f = open(filename + '.txt', 'w')
355     f.write('Node displacements\nNode\tu [mm]\tw [mm]\ttheta [rad]\
    n')
356     for i in range(len(nodesObjectList)):
357         f.write(f' {i+1}\t\t{round(r[i*3]*10**3)}\t\t{round(r[i
    *3+1]*10**3)}\t\t{round(r[i*3+2],4)}\n')
358     f.write('\nMoments\nBeam\tM1 [MNm]\tM2 [MNm]\tM_field_max [ MNm
     ]\n')
359     for beam in beamsObjectList:
360         if(beam.hasDistributedLoad):
361             f.write(f' {beam.number}\t\t{round(beam.reactionForces
    [2]/10**6,2)} \t\t{round(beam.reactionForces[5]/10**6,2)}\t\t{
    round(beam.localMaxMoment/10**6,2)}\n')
362         else:
363             f.write(f' {beam.number}\t\t{round(beam.reactionForces
    [2]/10**6,2)} \t\t{round(beam.reactionForces[5]/10**6,2)}\n')
364     f.write('\nSheer\nBeam\tQ1 [MN]\tQ2 [MN]\n')
365     for beam in beamsObjectList:
366         f.write(f' {beam.number}\t\t{round(beam.reactionForces
    [1]/10**6,2)} \t\t{round(beam.reactionForces[4]/10**6,2)}\n')
367     f.write('\nAxial Force\nBeam\tN [MN]\n')
368     for beam in beamsObjectList:
369         f.write(f' {beam.number}\t\t{round(beam.reactionForces
    [3]/10**6,2)}\n')
370     f.write('\nStress\nBeam\tSigma_x [MPa]\t%fy\n')
371     for beam in beamsObjectList:
372         f.write(f' {beam.number}\t\t\t{round(beam.sigmax / 10 ** 6)
    }\t\t\t{round(beam.securityFactor * 100)}%\n')
```

C Classes

```
'''
Developed at the Norwegian University of Science and Technology, Department of Marine Technology
08.11.2021 as part of TMR4167 Marin teknikk - Konstruksjoner
'''

from functions import *
from importedLibraries import *


# This file handles the class declaration for Nodes and Beams.

class Beam:
    '''
    This is a beam class. It includes multiple in house functions. Some is called when initializing and some
    can be called if you want to add info to the beam, such as geometry or number.
    '''

    def __init__(self, NODE1, NODE2):
        '''
        Initializes a Beam object. Calculates orientation and length in global coordinates.
        :param NODE1: start node
        :param NODE2: end node
        '''
        self.node1 = NODE1
        self.x1 = NODE1.x
        self.z1 = NODE1.z
        self.node2 = NODE2
        self.x2 = NODE2.x
        self.z2 = NODE2.z
        self.orientation = self.getGlobalOrientation()
        self.length = self.getLength()
        self.hasDistributedLoad = False

```

```
34      def getGlobalOrientation(self):
35          '''
36          Calculates the objects orientation and makes a variable
    called orientation.
37          :return: Returns an angle in float format
38          '''
39
40          dz = self.z2 - self.z1
41          dx = self.x2 - self.x1
42
43          if dz != 0:
44              if dx != 0:
45                  theta_ref_global = -np.arctan2(dz, dx)
46              elif self.z1 > self.z2:
47                  theta_ref_global = np.pi / 2
48              else:
49                  theta_ref_global = -np.pi / 2
50          else:
51              if self.x1 < self.x2:
52                  theta_ref_global = 0
53              else:
54                  theta_ref_global = np.pi
55          return theta_ref_global
56
57      def getLength(self):
58          '''
59          Calculates the objects length. This function is used when
    initializing the beam.
60          :return: length of beam.
61          '''
62          dx = self.x1 - self.x2
63          dz = self.z1 - self.z2
64          return np.sqrt(dx * dx + dz * dz)
65
66
67      def makePipe(self, r, ratioAir):
68          '''
69          Turns the beam object into a pipe, and calculates
    corresponding area and moment of inertia.
```

```
70          :param r: radius
71          :param ratioAir: fraction of radius that is air (hollow
    pipe)
72          :return: nothing
73          '''
74          I = np.pi / 4 * (r ** 4 - ratioAir ** 4)
75          A = np.pi * (r ** 2 - ratioAir ** 2)
76          self.area = A
77          self.momentOfInertiaStrong = I
78          self.momentOfInertiaWeak = I
79          self.diameter = 2 * r
80          self.Zc = r
81
82
83      def makeIPE(self,H, b,t_mid,t_f):
84          '''
85          Turns the beam object into a IPE profle,
86          and calculates corresponding area and moment of inertia.
87          :param H: height
88          :param w_top: width top
89          :param w_bot: width bottom
90          :param w_mid: width middle
91          :param t_top: thickness top
92          :param t_bot: thickness bottom
93          :return: nothing
94          '''
95          A_f = b * t_f
96          Amid = (H - 2 * t_f) * t_mid
97
98          I_f= b * t_f**3 / 12
99          Imid = (H- 2 * t_f)**3 * t_mid / 12
100
101         area = 2 * A_f + Amid
102         momInertiaStrong = 2 * I_f + Imid + 2 * A_f * (H/2 - t_f/2)
    **2
103         momInertiaWeak = 2 * b**3 * t_f + t_mid**3 * (H - 2 * t_f)
104
105         self.area = area
106         self.momentOfInertiaStrong = momInertiaStrong
```

```
107         self.momentOfInertiaWeak = momInertiaWeak
108         self.diameter = b
109         self.Zc = H / 2
110 
111     def giveNumber(self, beamNumber):
112         '''
113         Enumerates the beam.
114         :param beamNumber: beam number
115         :return: nothing
116         '''
117         self.number = beamNumber
118 
119     def makeStiffness(self, materialArray):
120         '''
121         Appends Youngs modulus, stiffness and yield stress to beam.
122         :param E: Youngs modulus
123         :return: nothing
124         '''
125         self.E = materialArray[0]
126         self.sigmafy = materialArray[1]
127         self.stiffnessStrongAxis = self.momentOfInertiaStrong *
    self.E
128         self.stiffnessWeakAxis = self.momentOfInertiaWeak * self.E
129 
130     def makeLocalStiffnessMatrix(self):
131         '''
132         Makes an element stiffness matrix.
133         :return: nothing
134         '''
135         localStiffnessMatrix = np.zeros((6, 6))
136 
137         # Uses local variables to make variables that simplify
    later calculations
138         EA_L = self.area * self.E / self.length
139         EI_L = self.momentOfInertiaStrong * self.E / self.length
140         EI_L2 = self.momentOfInertiaStrong * self.E / (self.length
    ** 2)
141         EI_L3 = self.momentOfInertiaStrong * self.E / (self.length
    ** 3)
```

```
142
143         # All E*A/L dependent matrix elements
144         localStiffnessMatrix[0][0] = EA_L
145         localStiffnessMatrix[3][3] = EA_L
146         localStiffnessMatrix[0][3] = -1 * EA_L
147         localStiffnessMatrix[3][0] = -1 * EA_L
148
149         # All E*I/L dependent matrix elements
150         localStiffnessMatrix[2][2] = 4 * EI_L
151         localStiffnessMatrix[5][5] = 4 * EI_L
152         localStiffnessMatrix[5][2] = 2 * EI_L
153         localStiffnessMatrix[2][5] = 2 * EI_L
154
155         # All E*A/L^2 dependent matrix elements
156         localStiffnessMatrix[4][5] = 6 * EI_L2
157         localStiffnessMatrix[5][4] = 6 * EI_L2
158         localStiffnessMatrix[1][5] = -6 * EI_L2
159         localStiffnessMatrix[5][1] = -6 * EI_L2
160         localStiffnessMatrix[4][2] = 6 * EI_L2
161         localStiffnessMatrix[2][4] = 6 * EI_L2
162         localStiffnessMatrix[1][2] = -6 * EI_L2
163         localStiffnessMatrix[2][1] = -6 * EI_L2
164
165         # All E*A/L^3 dependent matrix elements
166         localStiffnessMatrix[1][1] = 12 * EI_L3
167         localStiffnessMatrix[4][4] = 12 * EI_L3
168         localStiffnessMatrix[1][4] = -1 * 12 * EI_L3
169         localStiffnessMatrix[4][1] = -1 * 12 * EI_L3
170
171         self.localStiffnessMatrix = localStiffnessMatrix
172
173     def transformElementStiffnessMatrix(self):
174         '''
175         Transforms the beams local stiffnessmatrix to the global
    orientation,
176         making it ready to be put in the global stiffness matrix.
177         :return: applies the stiffness matrix in the global
    orientation to the beams
178         '''
```

```
179         a = self.orientation
180 
181         self.T = np.array([
182             [np.cos(a), np.sin(a), 0, 0, 0, 0],
183             [-np.sin(a), np.cos(a), 0, 0, 0, 0],
184             [0, 0, 1, 0, 0, 0],
185             [0, 0, 0, np.cos(a), np.sin(a), 0],
186             [0, 0, 0, -np.sin(a), np.cos(a), 0],
187             [0, 0, 0, 0, 0, 1]])
188 
189         self.T_transponent = np.array([
190             [np.cos(a), -np.sin(a), 0, 0, 0, 0],
191             [np.sin(a), np.cos(a), 0, 0, 0, 0],
192             [0, 0, 1, 0, 0, 0],
193             [0, 0, 0, np.cos(a), -np.sin(a), 0],
194             [0, 0, 0, np.sin(a), np.cos(a), 0],
195             [0, 0, 0, 0, 0, 1]])
196         self.transformedStiffnessMatrix = list(
197             np.matmul(self.T, np.matmul(self.localStiffnessMatrix,
    self.T_transponent)))
198 
199     def addDistributedNormalLoad(self, load):
200         '''
201         Adds the distributed load from imput file to the beam. Adds
     q1 and q2 to the beam object,
202         where q1 is the normal load working on NODE1, and q2 for
    NODE2.
203         :param load: list with load data, in global orientation
204         :return:
205         '''
206         self.hasDistributedLoad = True
207         if (np.sin(self.orientation) == 0):
208             self.q1 = load[2]
209             self.q2 = load[4]
210         elif (np.cos(self.orientation) == 0):
211             self.q1 = load[1]
212             self.q2 = load[3]
213         else:
214             self.q1 = load[1] / np.sin(self.orientation) + load[2]
```

```
/ np.cos(self.orientation)
215             self.q2 = load[3] / np.sin(self.orientation) + load[4]
/ np.cos(self.orientation)
216
217     def calculateFixedSupport(self):
218         '''
219         Calculates Fixed Support for beams affected by distributed
loads
220         Adds Fixed Support to the nodes affected
221         Based on table found in "TMR4167 Marin teknikk 2
Konstruksjoner - Del 1", page 281
222         :return: nothing
223         '''
224
225         if (self.hasDistributedLoad):
226             m1 = (1 / 20) * self.q1 * (self.length) ** 2 + (1 / 30)
 * self.q2 * (self.length) ** 2
227             m2 = -(1 / 30) * self.q1 * (self.length) ** 2 - (1 /
20) * self.q2 * (self.length) ** 2
228
229             self.node1.M += m1
230             self.node2.M += m2
231
232             v2 = (m1 + m2 - (self.q1 * self.length ** 2) / 6 - (
self.q2 * self.length ** 2) / 3) / self.length
233             v1 = -(self.length / 2) * (self.q1 + self.q2) - v2
234
235             self.node1.Fx += v1 * np.sin(self.orientation)
236             self.node1.Fz += v1 * np.cos(self.orientation)
237             self.node2.Fx += v2 * np.sin(self.orientation)
238             self.node2.Fz += v2 * np.cos(self.orientation)
239         else:
240             pass
241
242
243     def scaleDistributedLoad(self, referenceDiameter):
244         '''
245         Scales forces linearly with respect to reference diameter
246         :param referenceDiameter: a float thar reperesent the
```

```
    reference diameter
247         :return: nothing
248         '''
249         if(self.hasDistributedLoad):
250             self.q1 *= self.diameter / referenceDiameter
251             self.q2 *= self.diameter / referenceDiameter
252         else:
253             pass
254 
255     def appendMomentDiagramToBeam(self, N=100):
256         '''
257         Calculates the moment diagram for an element.
258         :param N: Number of equidistributed moment values
    calculated along the beam
259         :return: nothing
260         '''
261 
262         X = np.linspace(0, self.length, N + 1)
263         dx = self.length / N
264         momentList = []
265 
266         for k in range(N + 1):
267             x = k * dx
268             ms2 = -self.q2 * (x ** 3 / (6 * self.length))
269             ms1 = -self.q1 * (x ** 2 / 2 - x ** 3 / (6 * self.
    length))
270             ms0 = -self.reactionForces[2] + - x * self.
    reactionForces[1]
271             momentList.append(ms0 + ms1 + ms2)
272 
273         self.equidistributedX = np.array(X)
274         self.momentDiagram = momentList
275 
276         if (momentList[0] < 0):
277             self.localMaxMoment = np.max(momentList)
278             self.localMaxMomentIndex = np.argmax(momentList)
279         else:
280             self.localMaxMoment = np.min(momentList)
281             self.localMaxMomentIndex = np.argmin(momentList)
```

```
282
283         self.localMaxMomentX = self.localMaxMomentIndex * self.
    length / N
284
285
286     def calculateMaxTension(self):
287         '''
288         Calculates maximun bending tension and security factor.
289         :return: nothing
290         '''
291         try:
292             self.sigmax = max([abs(self.reactionForces[2]), abs(
    self.localMaxMoment),
293                                abs(self.reactionForces[5])]) * self
    .Zc / self.momentOfInertiaStrong
294         except AttributeError:
295             self.sigmax = max(
296                 [abs(self.reactionForces[2]), abs(self.
    reactionForces[5])]) * self.Zc / self.momentOfInertiaStrong
297         self.sigmax += abs(self.reactionForces[0])/self.area
298         self.securityFactor = self.sigmax / self.sigmafy
299
300
301     def printSecurityFactor(self):
302         '''
303         Prints the security factor to terminal.
304         :return: nothing
305         '''
306         print(f'Sigma_x: {round(self.sigmax / 10 ** 6)} [MPa], {
    round(self.securityFactor * 100)}% of fy\n\n')
307
308     def printBeam(self):
309         '''
310         Prints beam data in a more readable way
311         :return: nothing
312         '''
313
314         string = f'Beam {self.number} from node {self.node1.number}
     to {self.node2.number},    : {round(self.orientation * 180 / np.
```

```
    pi, 2)}, L: {round(self.length, 2)}\n'
315         for i in range(2):
316             string += f'u{i + 1}: {round(self.localDisplacements[i
    * 3], 4)} \tN{i + 1}: {round(self.reactionForces[i * 3], 2)}\n'
317             string += f'w{i + 1}: {round(self.localDisplacements[i
    * 3 + 1], 4)} \tV{i + 1}: {round(self.reactionForces[i * 3 + 1],
     2)}\n'
318             string += f'  {i + 1}: {round(self.localDisplacements[i
     * 3 + 2] * 180 / np.pi, 4)} \tM{i + 1}: {round(self.
    reactionForces[i * 3 + 2], 2)}\n'
319         print(string)
320
321     def printBeamMoments(self):
322         '''
323         Prints moment values to terminal.
324         :return: nothing
325         '''
326         try:
327             string = f'M1: {round(self.reactionForces[2])}, M_max:
    {round(self.localMaxMoment)} at {round(self.localMaxMomentX, 3)
    }, M2: {round(self.reactionForces[5])}'
328         except AttributeError:
329             string = f'M1: {round(self.reactionForces[2])}, No
    ditributed load, M2: {round(self.reactionForces[5])}'
330         print(string)
331
332
333 class Node:
334     '''
335     This class is used to create beams and store
336     data about displacements and fixed supported forces and moments
    .
337     We use these to create the R-vector.
338     '''
339
340     def __init__(self, x, z, supportX, supportZ, supportTheta):
341         '''
342         Initializes a Node object.
343         :param x: x coordinate
```

```
344         :param z: z coordinate
345         :param supportX: The nodes support condition in X
346         :param supportZ: The nodes support condition in Z
347         :param supportTheta: The nodes rotational support condition
348         '''
349         # Coordinates
350         self.x = x
351         self.z = z
352         # Displacement and rotation support conditions
353         self.supportX = supportX
354         self.supportZ = supportZ
355         self.supportTheta = supportTheta
356         # Fixed support forces and moment
357         self.Fx = 0
358         self.Fz = 0
359         self.M = 0
360 
361     def giveNumber(self, nodeNumber):
362         '''
363         Enumerates the node. Not sure if we will use this or not.
364         :param nodeNumber: number of node
365         :return: applies number to node.
366         '''
367         self.number = nodeNumber
368 
369     def addNodeLoad(self, nodeLoad):
370         '''
371         Adds the nodeloads directly to the nodes
372         param nodeLoad: vector containing the nodeloads
373         -= because we want the reactionforces from the loads
374         '''
375         self.Fx -= nodeLoad[1]
376         self.Fz -= nodeLoad[2]
377         self.M -= nodeLoad[3]
```

D Read from input file

```
'''
Developed at the Norwegian University of Science and Technology,
    Department of Marine Technology
08.11.2021 as part of TMR4167 Marin teknikk - Konstruksjoner
'''

from importedLibraries import *

def readInputFile(filepath):
    '''
    Reads input file and sorts info into appropriate array
    :param filepath: Lists containing appropriate data
    :return: Arrays of input file data
    '''
    f = open(filepath, 'r')
    lineList = f.readlines()
    NODE, BEAM, MATERIAL, NODELOAD, BEAMLOAD, PIPE, IPE = [], [],
    [], [], [], [], []
    # Set default reference diameter
    REFERENCEDIAMETER = -1
    for i, line in enumerate(lineList):
        line = line.split(',')
        if line[0] == "NODE":
            NODE.append(line[1:])
            # x, z, support x, support z, supprot theta
            # support x, support z, supprot theta is
            # 0 = "This is not fixed"
            # 1 = "This is unknown"
            # 2 = "This is fixed"
        elif line[0] == "BEAM":
            BEAM.append(line[1:])
            # N1, N2, M, G, n
            # N1-Node left most node (low X value)
            # N2-Node right most node (high X value)
            # M-Material, 1=steel, 2=aluminium
            # Geometry. 2 = pipe, 1 = IPE
            # The n. th geometry dimensions from library
```

```
        elif line[0] == "MATERIAL":
            MATERIAL.append(line[1:])
            # Youngs moulus, yield stress
            # 1 is steel and 2 is aluminium
        elif line[0] == "NODELOAD":
            NODELOAD.append(line[1:])
            # Node number, Fx, Fz, My
        elif line[0] == "BEAMLOAD":
            BEAMLOAD.append(line[1:])
            # Beam number, Fx1, Fz1, Fx2, Fz2
        elif line[0] == "PIPE":
            PIPE.append(line[1:])
            # outer radius, inner radius
        elif line[0] == "IPE":
            IPE.append(line[1:])
        elif line[0] == "REFERENCEDIAMETER":
            REFERENCEDIAMETER = float(line[1].strip())
        elif line[0] == "---\n":
            pass
        else:
            print("Error reading line {i + 1} in input file!")
    f.close()
    return np.array(NODE, dtype=float), np.array(BEAM, dtype=float)
, np.array(MATERIAL, dtype=float), \
           np.array(NODELOAD, dtype=int), np.array(BEAMLOAD, dtype=
int), np.array(PIPE, dtype=float), \
           np.array(IPE, dtype=float), REFERENCEDIAMETER
```

E Imported libraries

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
```

F Plot visualizations

This code is not ours. It is handed out.

```
from importedLibraries import *
from structure_visualization import *

'''
This subfile controlls plotting. It is dependent on the given code from data studass, "structure_visualization.py"
'''

def nodeArrayDisplacedPosition(nodeArray, r):
    for i,node in enumerate(nodeArray):
        node[0] += r[3*i]
        node[1] += r[3*i + 1]
    return nodeArray

def plot(NODE, BEAM, r):
    '''
    Makes a plot using given structure visualization plots.
    :param NODE: Array of nodes
    :param BEAM: Array of beams
    :param MATERIAL:
    :param NODELOAD:
    :param BEAMLOAD:
    :return: A plot
    '''
    # Konstant that says whats the nr of the first beam and node is.
    indexStart = 1

    fig_init, ax_init, fig_def, ax_def = setup_plots()

    plot_structure(ax_init, NODE, BEAM, 0, indexStart)

    plot_structure_def(ax_def, nodeArrayDisplacedPosition(NODE, r), BEAM, 0, indexStart, r)

    plt.show()
```

```
# This visualization is based on the original Matlab code by Josef Kiendl, and has been modified to fit TMR4176 by Jon Arnt K rstad
# NB! Denne filen krever at du har installert Python-pakkene: NumPy, SciPy og Matplotlib
# More detailed information regarding Python (matrices, visualizations etc.) may be found at 'https://www.ntnu.no/wiki/display/imtsoftware'

from importedLibraries import *

def setup_plots():
    '''
    Function that defines initial and deformed axis and figures.
    :return: Figures and axis
    '''
    fig_init, ax_init = plt.subplots()
    fig_def, ax_def = plt.subplots()
    ax_init.set_title('Initial Frame')
    ax_def.set_title('Displaced Frame')
    ax_init.axes.set_aspect('equal')
    ax_def.axes.set_aspect('equal')
    return fig_init, ax_init, fig_def, ax_def

def plot_structure(ax, punkt, elem, numbers, index_start):
    '''
    A function that plots a somewhat realistic image of the initial noedes and beams before..
    :param ax:
    :param punkt:
    :param elem:
    :param numbers:
    :param index_start:
    :return: Nothing.
    '''
    # This is a translation of the original function written by Josef Kiendl in Matlab
    # It has been slightly modified in order to be used in TMR4176

```

```
33      # This function plots the beam structure defined by nodes and
        elements
34      # The bool (0 or 1) 'numbers' decides if node and element
        numbers are plotted or not
35
36      # Change input to the correct format
37      nodes = np.array(punkt[:, 0:2], copy = 1, dtype = int)
38      el_nod = np.array(elem[:, 0:2], copy=1, dtype=int) + 1
39
40      # elNodToNodesKonstant is a subtracting factor to go from
        el_nod[i,0] to a nodes element.
41      # This constant is made by Hugo to make this script compatable
        with our nodes and beams arrays.
42      elNodToNodesKonstant = 2
43
44      # Start plotting part
45      for iel in range(0, el_nod.shape[0]):
46          # Plot element
47          ax.plot([nodes[el_nod[iel, 0] - elNodToNodesKonstant, 0],
        nodes[el_nod[iel, 1] - elNodToNodesKonstant, 0]],
48                  [nodes[el_nod[iel, 0] - elNodToNodesKonstant, 1],
        nodes[el_nod[iel, 1] - elNodToNodesKonstant, 1]], '-k',
        linewidth = 2)
49
50          if numbers == 1:
51              # Plot element numbers. These are not plotted in the
        midpoint to
52              # avoid number superposition when elements cross in the
         middle
53              ax.text(nodes[el_nod[iel, 0] - elNodToNodesKonstant, 0]
         +
54                      ( nodes[el_nod[iel, 1] - elNodToNodesKonstant,
        0] - nodes[el_nod[iel, 0] - elNodToNodesKonstant, 0] ) / 2.5,
55                      nodes[el_nod[iel, 0] - elNodToNodesKonstant, 1]
         +
56                      ( nodes[el_nod[iel, 1] - elNodToNodesKonstant,
        1] - nodes[el_nod[iel, 0] - elNodToNodesKonstant, 1] ) / 2.5,
57                      str(iel + index_start), color = 'blue',
        fontsize = 16)
```

```
58
59      if numbers == 1:
60          # Plot node number
61          for inod in range(0, nodes.shape[0]):
62              ax.text(nodes[inod, 0], nodes[inod, 1], str(inod +
    index_start), color = 'red', fontsize = 16)
63
64
65  def plot_structure_def(ax, punkt, elem, numbers, index_start, r):
66      '''
67      Makes a plot of the deformed beams. Assumes nodes to be fixed.
68      :param ax:
69      :param punkt:
70      :param elem:
71      :param numbers:
72      :param index_start:
73      :param r:
74      :return: A plot of the structures nodes and deformed beams.
75      '''
76      # This is a translation of the original function written by
    Josef Kiendl in Matlab
77      # This function plots the deformed beam structure defined by
    nodes and elements
78      # The bool (0 or 1) 'numbers' decides if node and element
    numbers are plotted or not
79
80      # Change input to the correct format
81      nodes = np.array(punkt[:, 0:2], copy = 1, dtype = int)
82      el_nod = np.array(elem[:, 0:2], copy=1, dtype=int)
83      nod_dof = np.arange(1, nodes.shape[0] + 1, 1, dtype=int)
84
85      if numbers == 1:
86          # Plot node number
87          for inod in range(0, nodes.shape[0]):
88              ax.text(nodes[inod, 0], nodes[inod, 1], str(inod +
    index_start), color = 'red', fontsize = 16)
89
90      elNodToNodesKonstant = 1
91
```

```
92      for iel in range(0, el_nod.shape[0]):
93          delta_x = nodes[el_nod[iel, 1] - elNodToNodesKonstant, 0] -
     nodes[el_nod[iel, 0] - elNodToNodesKonstant, 0]
94          delta_z = nodes[el_nod[iel, 1] - elNodToNodesKonstant, 1] -
     nodes[el_nod[iel, 0] - elNodToNodesKonstant, 1]
95          L = np.sqrt(delta_x ** 2 +  delta_z ** 2)
96          if delta_z >= 0:
97              psi = np.arccos(delta_x / L)
98          else:
99              psi = -np.arccos(delta_x / L)
100
101         phi = np.zeros((2, 1))
102         for inod in range(0, 2):
103             if nod_dof[el_nod[iel, inod] - elNodToNodesKonstant] >
    0:
104                 phi[inod] = r[(el_nod[iel, inod] -
    elNodToNodesKonstant)*3 + 2]
105         x = np.array([0, L])
106         z = np.array([0, 0])
107         xx = np.arange(0, 1.01, 0.01)*L
108         cs = CubicSpline(x, z, bc_type = ((1, -phi[0, 0]), (1, -phi
    [1, 0])))
109         zz = cs(xx)
110
111         # Rotate
112         xxzz = np.array([[np.cos(psi), -np.sin(psi)], [np.sin(psi),
     np.cos(psi)]]) @ np.vstack([xx, zz])
113
114         # Displace
115         xx2 = xxzz[0, :] + nodes[el_nod[iel, 0] -
    elNodToNodesKonstant, 0]
116         zz2 = xxzz[1, :] + nodes[el_nod[iel, 0] -
    elNodToNodesKonstant, 1]
117         ax.plot(xx2, zz2, '-k', linewidth = 2)
118
119         if numbers == 1:
120             # Plot element numbers. These are not plotted in the
    midpoint to
121             # avoid number superposition when elements cross in the
```

```
    middle
122             ax.text(xx2[round(xx2.size / 2.5)], zz2[round(xx2.size
    / 2.5)], str(iel + index_start), color = 'blue', fontsize = 16)
```