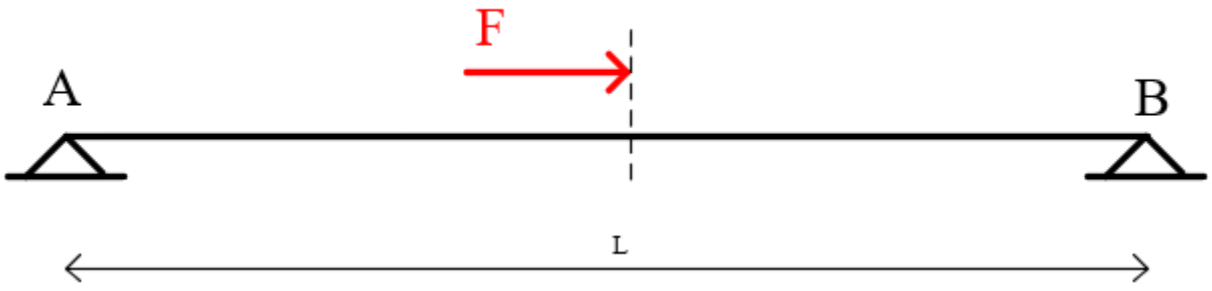
Implementation of Axial Force Analysis via Macaulay's Method Using Python

Bachelor's Thesis

Lucas Verlaan



Implementation of Axial Force Analysis via Macaulay's Method Using Python

Bachelor's Thesis

by

Lucas Verlaan

Supervisor:	T. van Woudenberg
Assistant Supervisor:	F. Messali
Project Duration:	April, 2025 - June, 2025
Faculty:	Faculty of Civil Engineering and Geosciences, Delft

Cover: Example of element with axial force

TU Delft

Preface

This report has been written as part of the final assignment for the Bachelor's program in Civil Engineering. I chose this project to combine and build upon my two favorite subjects from the program: structural mechanics and object-oriented programming in Python, which I explored further through a minor in computer science. I was also excited to gain experience with contributing to open-source Python projects via GitHub.

First and foremost, I would like to thank Tom van Woudenberg for supervising my project, particularly on the GitHub side. His expertise in GitHub workflows gave me the structure and confidence needed to contribute efficiently to SymPy. Furthermore, I am also grateful to Fransesco Messali for his thourough feedback on the report itself; his guidance on improving clarity for readers less familiar with the subject matter was especially valuable. Lastly, I would like to thank Sai Udayagiri from Google Summer of Code for his feedback on both the code and GitHub practices.

Lucas Verlaan
Delft, June 2025

Summary

In this report, the development and implementation of the new column module in SymPy is described. This module models and solves for axial force problems in one-dimensional structural elements. This implementation is based on Macaulay's method, which uses singularity functions to express the Euler-Bernoulli differential equations as a single piecewise expression. These functions are well-suited to symbolic manipulation and integration within SymPy, a Python library for symbolic mathematics and physics.

The column module addresses a limitation in the existing Structure2D module, which is also part of SymPy's continuum mechanics framework. Structure2D enables modeling of two-dimensional structures with members in both the x and y directions. Internally, these structures are ”unwrapped” into a one-dimensional system. While a 'beam' module is already used to solve for shear forces and bending moments, axial force support in Structure2D is currently limited. Only one horizontal reaction force can be solved, making use of a simple sum of horizontal loads. The column module provides the foundation for extending this functionality to include symbolic treatment of multiple axial loads and reactions, similar to how the beam module supports transversal loads.

The column module can be used by initializing a column object with its length, elastic modulus and cross-sectional areas. Users can apply supports and loads, including distributed loads, as well as define telescope hinges which allow discontinuities in displacement. The singularity functions of the supports, loads and hinges are added to a continuous load equation. The unknowns in the load equation are solved with force and deflection boundary conditions. At the supports, the deflection is zero and at the hinges the axial force is zero. Due to the load expression needing to be integrated twice to solve for displacements, two integration constants also have to be solved. The remaining equations are provided by boundary conditions for absence of axial forces just outside of the element.

The system of equations is solved symbolically using a linear solving function. The unknown forces and displacements are automatically replaced into the load equation. From this solved load equation, the singularity expressions for the axial force and deflection along the column can be set up and plotted. The column module has been tested for multiple different use cases with different load types and telescope hinges. The results have been compared with those from Matrixframe, showing that the they are correct.

While the current implementation offers core functionality for analyzing axially loaded elements, further improvements can be made by developers in the future. These include enhancements such as improved visualization, more flexible boundary conditions and inclusion of variable cross-sectional properties. While the column module is a standalone feature, it should also be adjusted to comply with future integration into the Structure2D module.

Contents

1 Introduction

Structural analysis plays a crucial role in civil engineering, as understanding the behaviour of structural elements under varying loading conditions is essential for safe design. Columns, being important load-bearing members, must be accurately modeled to predict responses due to axial stresses. While advanced computational solvers are widely used for analyzing complex structures, they offer little insight into the underlying mathematical formulations. SymPy, a Python library for symbolic computation, enables users to define, manipulate, and solve symbolic expressions, offering a transparent approach to problem-solving. Expanding the structural mechanics module within SymPy can enhance both educational and research applications by allowing users to explore parameter dependencies and build customizable models for structural elements.

1.1. Background

In this chapter the background of this research will be explained. First the underlying theory will be described and afterwards the previous research around this subject will be summarized.

1.1.1. Theory

The calculations in SymPy will be made using Macaulay's method. Macaulay's method makes use of the Euler-Bernoulli differential equations for structural analysis. The differential equation that is used to calculate extension due to axial forces is as follows:

$$EA\frac{d^2u(x)}{dx^2} + q_x(x) = 0 \tag{1.1}$$

where:

- $u(x)$: Axial displacement as a function of position x.
- E: Young's modulus of the material
- A: Cross-sectional area of the element.
- $q_x(x)$: Distributed axial load per unit length (load equation).

These differential equations would normally have to be applied separately to different parts of a discontinuous beam, using boundary conditions between these parts. Macaulay formulated a manner in which a continuous formula could express the element's behaviour by employing singularity functions, also known as Macaulay brackets. Singularity functions allow discontinued loads to be represented in a unified mathematical form. A typical singularity function is written as follows:

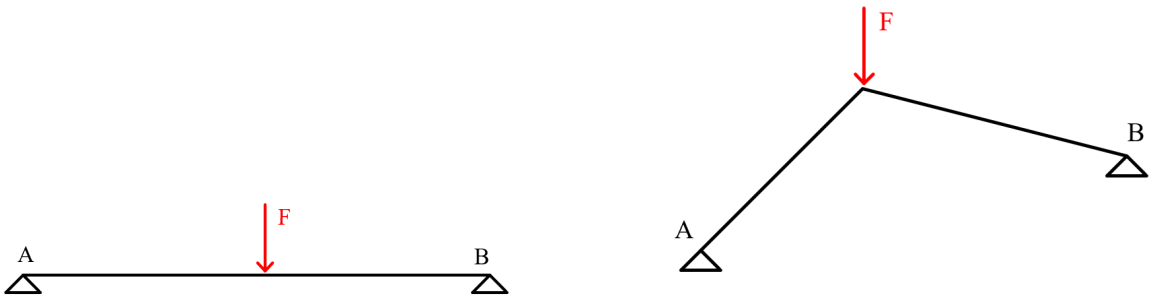
$$\langle x-a\rangle^n = \begin{cases} 0, & x < a \\ (x-a)^n, & x \geq a \end{cases} \tag{1.2}$$

where a is the location of the discontinuity and n is the order of the function. This formulation allows for a single expression for the entire length of the element, simplifying integration and enabling efficient computation through symbolic tools like SymPy.

Extensive documentation on this topic has been formulated using the advancements made in the bachelor theses of previous students (van Woudenberg, 2024).

1.1.2. Previous Research

Not only have many advancements already been made on the formulation of singularity functions using Macaulay's method, but this method has also been extensively applied in SymPy. The SymPy package contains a beam module which forms the singularity functions for a given one-dimensional beam and solve the reactions. In addition, implementations of hinges and influence lines have been added to this module (van Gelder, 2024). A proof of concept has also been formed for a two-dimensional module called Structure2D (Saheli, 2024). This two-dimensional model can solve reactions by projecting the structure as a one-dimensional beam, and using methods of the beam module. In figure 1.1 the difference between the 1D-beam and 2D structures is visualized.



(a) 1D Beam with vertical load

(b) 2D structure with vertical load, leading to axial and shear forces

Figure 1.1: Comparison of 1D and 2D structural representations

1.2. Problem Definition

The one-dimensional beam module in SymPy is already very detailed, aside from springs and varying cross-sectional properties not yet having been implemented. The greatest opportunity for improvement lies in the two-dimensional implementation. While the structure2D module successfully calculates reaction forces, shear force diagrams, and moment diagrams, axial forces have not yet been fully implemented (Saheli, 2024). This is because the beam module can currently only apply vertical forces and calculate vertical reactions. An overview of the current beam module can be seen in figure 1.2

This figure shows the different methods of the beam module and how they relate to each other. The core functions of the module are applying different types of supports and hinges, after which the users can apply and remove loads. When these steps are completed a model for the beam is created, along with a load equation expressed as singularity functions. The user can solve the reactions of the load equations and the influence lines. Lastly, the shear force and bending moment lines can be constructed and plotted.

The two-dimensional module splits up the forces into axial and shear forces. For the shear forces, the module uses the beam class to calculate the vertical reactions and deformations. Due to the absence of functionality for axial loads, the 2D module calculates the horizontal reactions on its own. It does this with Newton's third law, equating the reaction force to the applied load. This means that only one horizontal reaction can be solved, as there is only one formula that can be used. The extra formulas can be gained by looking at the axial deformation and boundary conditions in the axial direction. This feature is too advanced to apply to the reaction solver in the 2D-module, so a new module will have to be established to solve this problem.

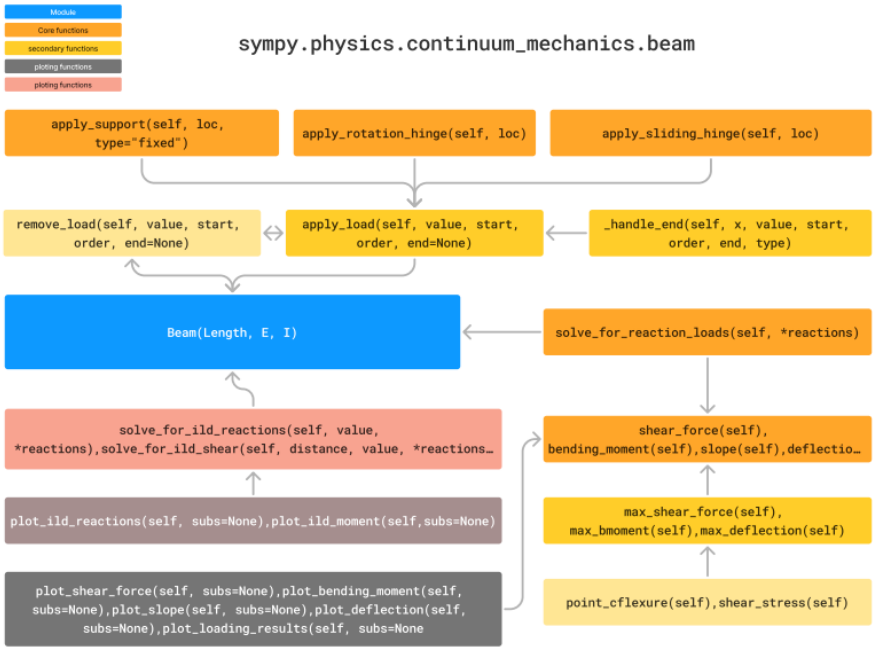


Figure 1.2: Overview of beam module (Saheli, 2024)

1.3. Objective and Scope

The main objective of this research is to implement axial force analysis in SymPy by creating a new one-dimensional column module. This module will make sure that one-dimensional axial force problems with more than one support are solvable. The reaction forces, axial force diagram, and extension/-compression due to horizontal forces should be calculable and plottable with this new implementation. Another main objective is the implementation of telescope hinges; hinges that provide a degree of freedom in the axial direction.

Creating a separate module for axial force analysis is a clean and effective approach, particularly because the Structure2D framework inherently separates shear and axial components. From a mechanics perspective, this separation is justified because axial and shear forces produce different stress and deformation behaviour. As a result, the two types of forces can be analyzed separately more effectively using different modules.

The research question is as follows:

How can axial force analysis be effectively integrated into a Python implementation of Macaulay's method?

The column module must be able to handle structures with the same level of complexity as supported by the Beam module. This includes support for various types of forces and hinges. One feature that will be excluded from the Column module is the implementation of influence lines, as they offer limited functionality for normal forces.

1.4. Methodology

As stated earlier, the effects of axial forces on a one-dimensional structure will be implemented through a new module called Column. A proposed structure for this module was outlined in a previous Bachelor's thesis and can be seen in figure 1.3 (Saheli, 2024).

As shown, the proposed model is very similar to the beam module, with the main difference being the exclusion of influence lines. The workflow similarly starts off by defining the supports and hinges. Rotation and sliding hinges will be left out, however. They are only important if there is bending or shear forces involved, but since the column module will be loaded purely with axial loads, these hinges will not have an influence. Comparable to the beam module, the user will then solve the reaction loads,

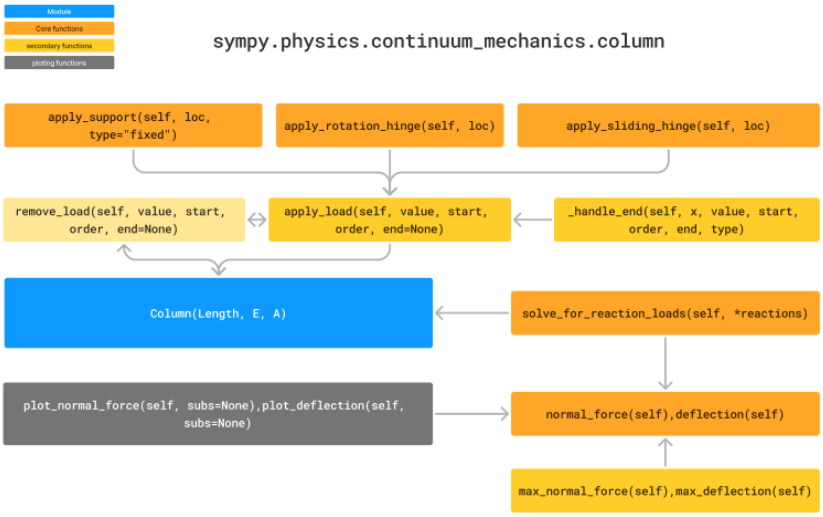


Figure 1.3: Proposed overview of the column module (Saheli, 2024)

and should be able to define and plot both the normal force and deflection equations. Due to this strong similarity, large parts of the existing code can and will be reused.

The module will be developed using a test-driven workflow through GitHub. The process begins by setting up an issue outlining the specific functionality to be implemented. Rather than building the entire module at once, development will be broken into manageable components, each tackled separately, allowing for feedback and iterative improvement. Each development cycle is small in scope, typically focusing on a single unit of functionality, such as defining an equation, implementing a method, or handling a specific structural behavior.

For each component, tests will first be created based on the mathematical background of the problem. These tests define the expected output for given inputs. They are designed to fail initially, since the corresponding code has not yet been implemented. Subsequently, the structure of the actual code will be written. The relevant methods will be developed and refined until all tests pass, indicating that the implementation behaves as expected. If certain tests are found to be incorrect or irrelevant, they can be adjusted accordingly. When satisfied with the code, the documentation will be added. The documentation is a pivotal part of the framework, as users need to understand how the code functions and how it can be used.

Finally, the code will undergo verification and review by moderators to ensure it adheres to established style guidelines and maintains code quality. Once approved, it will be merged into the main branch. This concludes one development cycle, after which a new issue will be created and the process is repeated for the next feature.

1.5. Report Structure

In this report, the design of the newly established column module will first be explained, accompanied by an exploration of the theoretical background. The methods and their functionality will be described, with reasoning provided for the decisions made, including what was left out or added with respect to the beam module. Examples will be presented to demonstrate potential use cases for the column module. The various methods will be tested to ensure that they produce correct results. Finally, the report will conclude with a summary of the research, a discussion of key topics, and recommendations for future improvements in the continuum mechanics section of SymPy.

2

Module Design and Functionality

This chapter presents the design and implementation of the newly developed column module. The framework's various functionalities are broken down and examined individually. For each functionality, the underlying mathematical and mechanical insight is described. This is followed by an explanation of how these theoretical concepts have been implemented in code. Alternative approaches are discussed, along with justification for the chosen design. The reasoning behind certain decisions is important to report, as it may not be directly apparent from the code itself, but remains crucial for future contributors to understand and maintain the module effectively.

The column object is modeled as as a horizontal, one-dimensional element that extends along the x-direction. This is counter intuitive, as columns in practice are vertical structures that support vertical loads due to gravity. This decision is motivated by the design of the Structure2D module. In that module, structures are divided into discrete elements that are represented as horizontal components, converting the problem into a one-dimensional system. Correspondingly, loads are decomposed into axial and shear components.

As a result, a vertical column in the two-dimensional module is projected as a horizontal element. The column module will thus also be constructed with horizontal orientation to provide seamless integration in the two-dimensional module. An example of this transition is given in figure 2.1. The one-dimensional object shown in the bottom graph can be modeled using the column module.

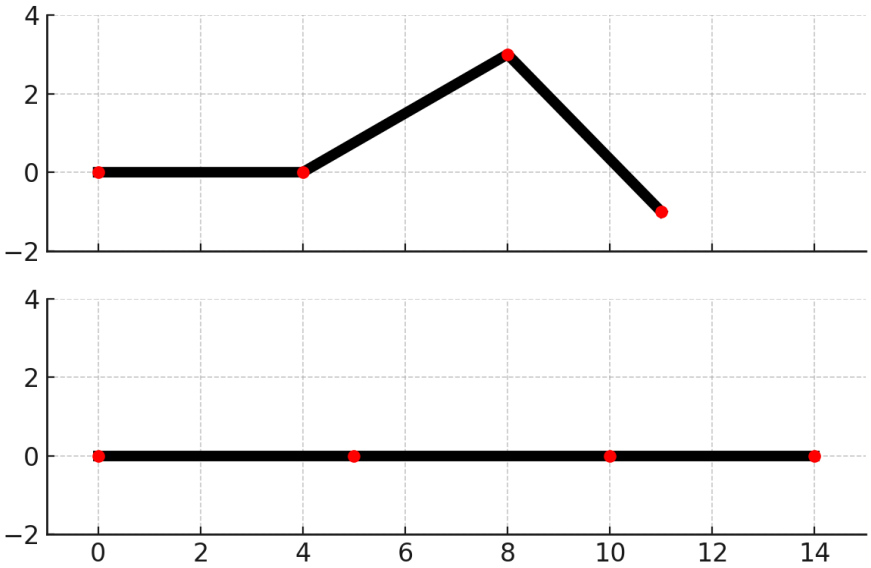


Figure 2.1: Visualization of unwrapping a two-dimensional structure (Saheli, 2024)

2.1. Initialization and Load Equation

The first steps to undertake when building the column object are the following: initializing the column, applying the reaction forces and applying the loads.

2.1.1. Initialization of the Column Object

To initialize the column, the user needs to input the parameters that define the shape and material of the column. The following parameters are required or optional when initializing a column:

- `length`: The total length of the column.
- `elastic_modulus` **(**E**)**: A measure of the stiffness of the column's material.
- `area` **(**A**)**: The cross-sectional area of the column.
- `variable` (optional): A symbolic variable used to express functions along the column. Defaults to x.
- `base_char` (optional): A base character for naming integration constants. Defaults to `'C'`.

The elastic modulus and the area are needed for axial equilibrium equation 1.1. The $variable$ parameter can be used in the future when looking at columns with varying cross-section, a column fixed at one end with a linearly increasing area for example. The base characteristics are used when integrating the load equation, resulting in integration constants. This initialization is very similar to the beam module, with the only key difference being the change in cross-sectional property. In the beam module, the second moment I is of importance as the measure of resistance against bending, but can be left out in the column module.

The module does not perform unit conversion or dimensional checking internally. Users are responsible for maintaining consistency in units. For instance, if the elastic modulus is provided in MPa, it should be converted appropriately. For example, it should be converted to kN/m^2 to match the units used for force and length in the rest of the model.

2.1.2. Macaulay's Method for Forces and Reactions

The goal of the column module is to set up a load equation that can be solved using boundary conditions. The load equation sums the applied loads and reaction forces using singularity functions. The influence of different forces on the load equation will first be researched.

Point Loads

Point loads can be represented as Dirac delta functions, which are singularity functions of order $n = -1$. The influence of point loads on the load equation is as follows:

$$q_x(x) = P \langle x - a \rangle^{-1} \tag{2.1}$$

Here, P represents the magnitude of the point load, and a is the location where the load is applied. (van der Wulp, 2023)

Reaction Loads

Reaction loads are also point loads and therefore their corresponding singularity function is identical. However, the reaction force is typically unknown and is denoted as R_a, where a is the location of the reaction force. The corresponding load equation is as follows:

$$q_x(x) = R_a \langle x - a \rangle^{-1} \tag{2.2}$$

(van der Wulp, 2023)

Uniformly Distributed Loads

Uniformly distributed loads are represented as step functions, which are singularity functions of order $n = 0$. In the load equation, a step function is added at the point of application and one is removed at the end point of the distributed load.

$$q_x(x) = w\langle x-a\rangle^0 - w\langle x-b\rangle^0 \tag{2.3}$$

Here, w is the constant magnitude of the uniform distributed load. The load is applied over the segment from a to b, where a and b represent the start and end locations of the load distribution, respectively. (van der Wulp, 2023)

Varying Distributed Loads

Although less common, varying distributed loads are also important to consider. These cases are more complex, as they don't allow for the simple subtraction of the same singularity function at a later point. While theory on this has been applied to the beam module, it has not been adequately explained or documented. Consider a linearly increasing axial load on a column as displayed in figure 2.2.

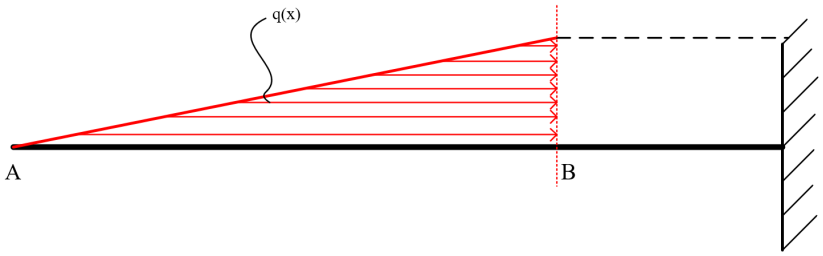


Figure 2.2: Linearly increasing distributed load

The load increases from A to B and afterwards the load disappears. At point A you would apply a singularity function of order $n = 1$ to the load equation to represent the linear increase. If you remove the same singularity function from the load equation at point b, you cancel out the slope that you applied at point A. This is not enough, however, as you are still left with a non-trivial distributed load (represented by the dotted line in the figure). A singularity function of order $n = 1$ thus has an influence on the slope and value of the distributed load. When the load stops, the value would also have to be brought to zero. A singularity function of order $n = 2$ has an influence on the angle of the slope, the slope and the value, and so on for higher orders.

To calculate the impact of the end of a distributed load on the load equation, the Taylor Series can be used. This is due to the fact that singularity functions from Macaulay's method and the terms in the Taylor Series are both localized power functions around a point. When a distributed load ends, the singularity functions at the endpoint are subtracted to reverse the influence of the previous term. The Taylor Series does the exact same, using successive terms to modify function behaviour. The load equation for distributed loads is given in equation 2.4.

$$q_x(x) = w\langle x-a\rangle^n - \sum_{i=0}^{n} \frac{f^{(i)}(b)}{n!}\langle x-b\rangle^i \tag{2.4}$$

Here a is again the left bound, b the right bound and n the order of the distributed load. The function $f^{(i)}(b)$ is the i^{th} derivative of the symbolic expression of the force, around point b. This expression can be formulated as the following:

$$f^{(i)}(b) = \frac{d^i}{dx^i}\left[w(x-a)^n\right]\Big|_{x=b} \tag{2.5}$$

This load equation can only be applied to orders $n = 0$ and greater. Formula 2.3 for order $n = 0$ can also be derived using these expressions. For example. $w(x-a)^0$ is equal to w, giving load expression:

$$q_x(x) = w\langle x-a\rangle^0 - \sum_{i=0}^{0} \frac{f^{(i)}(b)}{i!}\langle x-b\rangle^i = w\langle x-a\rangle^0 - w\langle x-b\rangle^0$$

2.1.3. Load Equation in the Column Module

After initializing the column, users can apply multiple supports and loads to the element. With the property `applied_loads` users can see which loads are currently applied to the object. Furthermore, with the property `load` users can get the total load equation for the structure.

Applying Supports

Supports can be added to the object using the `apply_support()` function, similar to the beam class. The method is very different, however, as it now only requires the location of the support as input. With the beam module the user also has to input what type of support it is; fixed, pinned or a roller. Due to applied forces only being axial, these types of supports do not have to be distinguished. Supports are either fixed in the horizontal direction (fixed and pinned supports) or are free in the horizontal direction (rolling supports). Supports that are free in the axial direction can be left out, as they have no influence on the column, and the other supports can simply be grouped as 'fixed' supports.

Figure 2.3 illustrates the differences in how supports are modeled in the beam and column modules. As shown, fixed support A can be modeled as a pinned support and rolling support B can be left out in the column object.

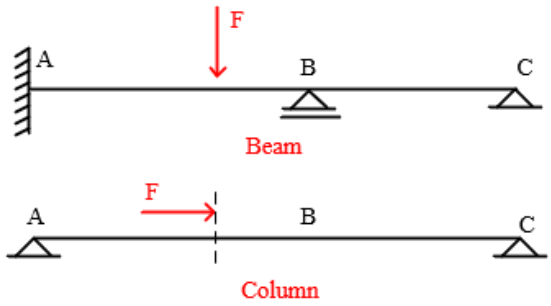


Figure 2.3: Difference between the modeling of beams and columns in SymPy

When a user applies a support, the method saves the location of the support in a list of boundary conditions. This list will be used in later methods to know where the boundary conditions of fixed displacements lie. The load equation for the reaction load is added up to the private variable `_load` which keeps track of the load equation. This is the variable called in the `load` property.

Applying and Removing Forces

Users can apply loads to the column object using the `apply_load()` method. This method is very similar to the method in the beam module. The user can input the value of the force, the start location (for point loads this is the point of application), the order of the load and the end value. These parameters are appended to the applied_loads list, and the corresponding singularity function(s) associated with the load are added to the load property. The `remove_load()` method does the opposite, it scans for the load in the applied_loads list and removes it. It also subtracts the corresponding singularity function(s). A remove load function could seem excessive, as users would only apply loads that they are going to use, and not remove them. However, if a user has a complicated column with many loads and supports, a remove load function can be useful. A user could want to see what would happen if a certain load weighed less and compare results. The remove load function is then a fast option to reach this goal.

For the subtraction term for distributed loads (see formula 2.4), a function called `_taylor_helper()` has been built. This method is called when an 'end' value is given. It subtracts or adds up the Taylor Series based term, depending on whether a load has been applied or removed. Changes have been made in respect to the helper function used in the beam module, which also uses the Taylor Series. In the beam module, many excessive lines of code were used by distinguishing between cases of removing and applying a load. In the column module it simply only returns the singularity functions, whether they are added or removed from the load property is up to whether the helper is called by the `apply_load()` or `remove_load()` method. This ensures that users and future developers can get a clearer understanding what the helper function is used for. In the future, this change could also be implemented in the beam module.

2.2. Solving for Reaction Loads

After all of the properties of the column and applied loads have been defined, the next step is to solve for the reaction loads and integration constants. First, the boundary conditions and equations are defined, after which the solution method in Python is given.

2.2.1. Boundary Conditions and Equations

To solve horizontal reactions loads, typically Newton's third law is used:

$$\sum F_x = 0 \tag{2.6}$$

This formula can be used for cases with one support but if there are multiple supports more equations are needed. These extra equations can be derived from the Euler-Bernoulli differential equation for axial extension given in formula 1.1. Here $q_x(x)$ is the load equation represented as singularity functions. At the supports, the column is fixed and thus can't be displaced. At each support the following boundary condition can be applied:

$$u(a) = 0 \tag{2.7}$$

where u is the displacement and a is the location of the support. With each unknown reaction force, an extra equation can be derived.

To solve differential equation 1.1, it has to be integrated twice. With this come two integration constants, which means that two extra equations have to be formulated to solve for these unknowns. Besides the essential boundary conditions, due to extension, there are also natural boundary conditions for the axial forces. Specifically, there are two traction-free boundary conditions representing the absence of axial forces just outside the ends of the column (van der Wulp, 2023).

$$\lim_{x \to 0^-} N(x) = 0 \qquad \lim_{x \to L^+} N(x) = 0 \tag{2.8}$$

The axial forces $N(x)$ can be derived by using static equation 2.9.

$$\frac{dN(x)}{dx} = -q_x \tag{2.9}$$

The singularity function expression for the load equation thus has to be integrated once to find a similar expression for the axial force within the column.

For example, take a column with two supports at the ends and a point load in the middle as displayed in figure 2.4.

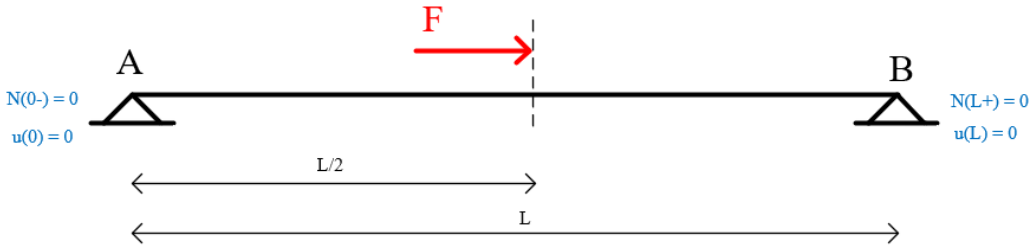


Figure 2.4: Column supported at both ends with boundary conditions

The load equation for this example is as follows:

$$q_x(x) = A_h\langle x\rangle^{-1} + F\langle x - L/2\rangle^{-1} + B_h\langle x - L\rangle^{-1}$$

Singularity functions can simply be integrated by going up one order. Point loads thus get turned into singularity functions of power 0, which is logical as point loads lead to a jump in axial force. The axial force equation for the column is formulated below, keeping the sign change in mind.

$$N(x) = -A_h\langle x\rangle^0 - F\langle x - L/2\rangle^0 - B_h\langle x - L\rangle^0 + C_N$$

C_N is the integration constant. To formulate the equation for the displacement, the equation has to once again be integrated and divided by EA.

$$EA * u(x) = -A_h\langle x\rangle^1 - F\langle x - L/2\rangle^1 - B_h\langle x - L\rangle^1 + xC_N + C_u$$

2.2.2. Equation Solver in Column Module

The reaction forces are calculated using the `solve_for_reaction_forces()` function. This function first sets up the two equations provided by the natural boundary conditions. Afterwards the equations are set up for the boundary conditions due to the supports. These equations are put in a list and the unknowns are solved with a linear solving function of SymPy. The values for the reaction loads are stored in the `reaction_forces` property which the user can call.

In the beam module, the user has to input the unknowns that they want solved. This function, however, seems redundant as there is no reason to only solve for a specific list of reaction forces and moments. It only makes using this method more complicated, as the user has to specify all unknowns themselves and define them all with variables. An example of how the equation solver function in the beam module is stated in listing 2.1.

Listing 2.1: Equation solver use case for beam module

```
1  from sympy.physics.continuum_mechanics.beam import Beam
2  from sympy import symbols
3  E, I = symbols('E, I')
4  b = Beam(20, E, I)
5  p0, m0 = b.apply_support(0, 'fixed')
6  p1 = b.apply_support(20, 'roller')
7  b.apply_load(-8, 10, -1)
8  b.apply_load(-4, 15, -1)
9  b.apply_load(100, 20, -2)
10 b.apply_load(10, 0, 0, end=10)
11 b.solve_for_reaction_loads(p0, m0, p1)
12 b.reaction_loads
```

In line 9, the user has to input all unknowns themselves, while the module itself can track which unknowns are present. In the column module, the user does not have to track the unknowns and input them in the function, which saves time and can minimize user errors and mistakes.

However, providing certain reactions symbolically does provide the possibility of relating the different reactions to each other. For example, the user wants to know how the p0 reaction relates to the symbolic p1 reaction. Reaction p1 can then be left out of the solver and will stay an unknown. If, in the future, users or developers find a good reason for leaving certain reactions as symbols in the column module, this can always be implemented.

Another large difference with the beam module is the simplified notation of boundary conditions. In the column module, boundary conditions can only be applied when adding a support. Alternatively in the beam module, the user can personalize boundary conditions by adjusting their location and the value of the condition. This too is excessive for the column module, as in practice boundary conditions are almost always zero. There could be complicated structures for there to be customizable boundary conditions, such as springed supports, but in the scope of this project only fairly simple column objects are analyzed. Therefore, only a list of boundary condition locations are saved as the value of the condition is always zero.

2.3. Telescope hinges

Along with supports, telescope hinges can also be applied to the column module. Telescope hinges are hinges that allow for for axial displacement at the applied location. Again, first the singularity functions and boundary conditions will be described, after which the implementation in the column module will be presented.

2.3.1. Singularity Equation and Boundary Condition

To clearly highlight the impact of a telescope hinge on a column object, one is applied to the example from chapter 2.2 (see figure 2.5). At location S of the telescope hinge, there is a new unknown: displacement u_S. To solve for this unknown, an extra equation is needed. This is provided by the boundary condition that there is no axial force at the location of the hinge.

$$N(S) = 0 \tag{2.10}$$

The impact of the sliding hinge on the load equation can be derived from formula 1.1. It now also depends on the cross-sectional properties.

$$q_x(x) = EA * u_S \langle x - S \rangle^{-2} \tag{2.11}$$

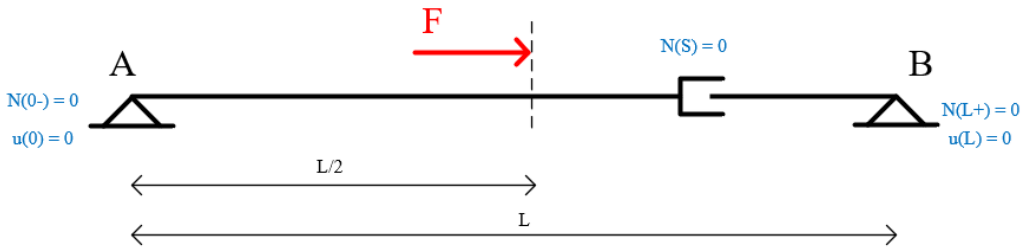


Figure 2.5: Column with two supports and telescope hinge

Telescope hinges practically split columns into multiple separate elements. This is due to them not being able to transmit axial forces. In the given example, the reaction force at B is now equal to zero, as force F cannot be transmitted past the sliding hinge, the column simply displaces. In this example, the left part of the column extends while the right part stays the same as there are no external forces. The left part slides over the right part due to this extension.

2.3.2. Telescope Hinges in Column Module

Telescope hinges can be applied to the column object using the `apply_telescope_hinge()` method. The user has to input the location along the column where the hinge should be added. This function adds the location to a list of locations where these hinges are applied, and adds the singularity function (formula 2.11) to the load equation.

Additionally, the extra equations and unknowns are added to the reaction equations in the previous `solve_for_reaction_loads()` method. After the unknown hinge deflection u_S is solved, it is saved in the `hinge_deflections` property that the user can call.

2.4. Force and Deflection Diagrams

After all of the unknown reaction forces and deflections are solved, they can be filled back into the load equation. This is done with the helper function `_solved_load()`. It replaces the unknowns in the current load equation with the solutions from the `reaction_loads` and `hinge_deflections` properties. Using the integration methods from subsection 2.2.1, the axial force and deflection expressions are calculated by the `axial_force()` and `deflection()` methods, respectively.

The user can furthermore plot these equations to visualize the the forces and deflections. When calling the `plot_axial_force()` and `plot_deflection()` functions these plots can be generated.

2.5. Column Module Overview

In figure 1.3, a proposed overview of the column module was given. Due to different modules being added or left out, the current module design is for many parts very different to the proposed design. Thus, an updated module overview is provided in figure 2.6. This overview displays the various functions and their roles. The module workflow is given by the arrows, showing how the user should approach using the module and which functions relate to each other.

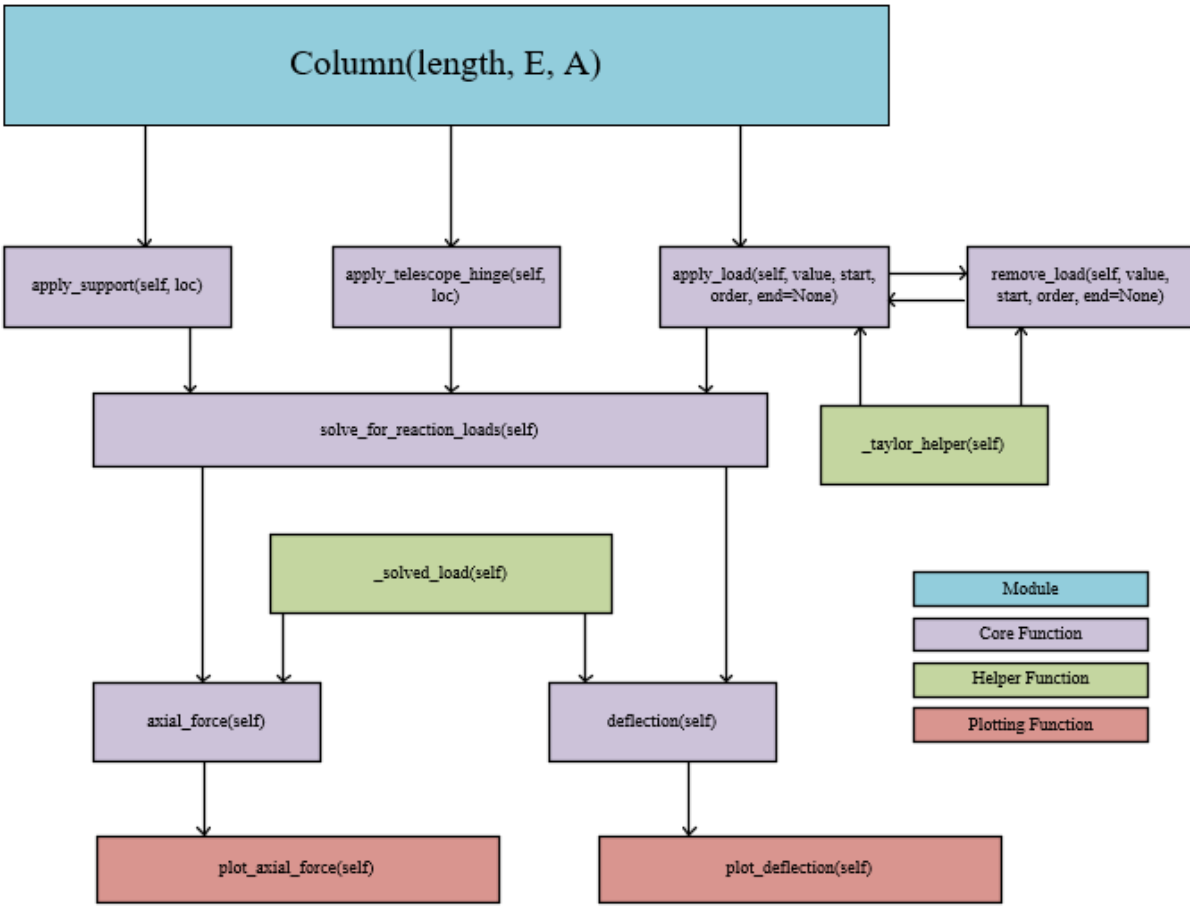


Figure 2.6: Column module overview

3

Module Use Cases

In this chapter, different column problems will be solved using the new column module in SymPy. First, a simple example will be worked out step by step to highlight how to use the different functions. Afterwards, more complicated cases with distributed loads and telescope hinges will be worked out. For each use case, the column module results are checked with results from equivalent models in MatrixFrame. In appendix A, these MatrixFrame models are displayed. Along with this, the axial force diagrams and nodal displacements are provided.

3.1. Simple Column with Point Loads

The situation that will be analyzed in this subsection is given in figure 3.1.

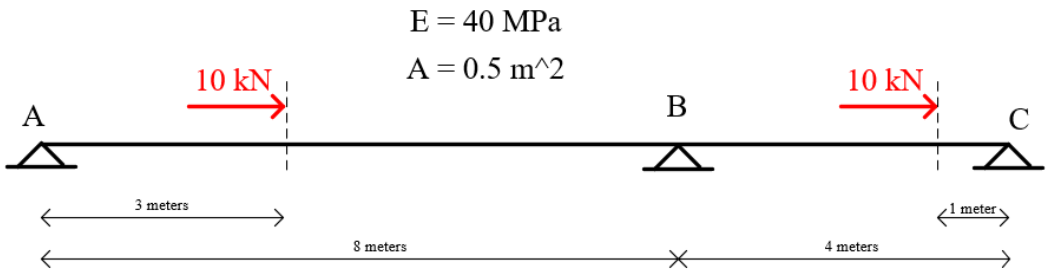


Figure 3.1: Column with three supports and two point loads

A column with a length of 12 meters is supported at both ends and at x = 8 meters (given x = 0 at A). Two point loads of 10 kN load the column in the positive x-direction, at x = 3 and x = 11 meters. The column has an elastic modulus of 40 MPa and a cross-sectional area of 0.5 m^2. When using this module, it is important to make sure that all units are consistent. 40 MPa is equal to $40 * 10^3 kN/m^2$. The second value should be used, as all other units are in kilonewtons and meters.

To make use of the Column module in SymPy, it first has to be imported. After, the class can be called and a column object can be constructed. The user has to provide values for the length, elastic modulus and area of the column to establish it. These values can also be provided symbolically, by making use of the `symbols` package from SymPy. In this case, all values are numerical so this package is not of importance. The process of initializing the column object for this example is given in listing 3.1.

Listing 3.1: Importing package and initializing column object

```
1 from sympy.physics.continuum_mechanics.column import Column
2 length, E, A = 12, 40000, 0.5
3 c = Column(length, E, A)
```

Next, the user has to define the location of the supports and the properties of the various loads. For each load, the magnitude, location, and order must be specified when calling the function. In this case, there are only point loads which have an order of $n = -1$. In listing 3.2, the process of adding supports and loads is given. In listing 3.3 the results of calling the properties `applied_loads` and `load` is displayed.

Listing 3.2: Adding the supports, loads and calling the load equation

```
from sympy.physics.continuum_mechanics.column import Column
length, E, A = 12, 40000, 0.5
c = Column(length, E, A)
c.apply_support(0)
c.apply_support(8)
c.apply_support(12)
c.apply_load(10, 3, -1)
c.apply_load(10, 11, -1)
c.applied_loads
c.load
```

Listing 3.3: Output of applied_loads and load properties

```
[(10, 3, -1, None), (10, 11, -1, None)]
R_0*SingularityFunction(x, 0, -1) + R_12*SingularityFunction(x, 12, -1) + R_8*
    SingularityFunction(x, 8, -1) + 10*SingularityFunction(x, 3, -1) + 10*SingularityFunction
    (x, 11, -1)
```

Now that the load equation is set up, it can be solved. The solve function is added to the code in listing 3.4 and the `reaction_loads` property is called. In subsequent listing 3.5, the calculated reaction forces are given.

Listing 3.4: Solving the load equation and calling the reaction loads

```
from sympy.physics.continuum_mechanics.column import Column
length, E, A = 12, 40000, 0.5
c = Column(length, E, A)
c.apply_support(0)
c.apply_support(8)
c.apply_support(12)
c.apply_load(10, 3, -1)
c.apply_load(10, 11, -1)
c.solve_for_reaction_loads()
c.reaction_loads
```

Listing 3.5: Output reaction loads

```
{R_0: -25/4, R_8: -25/4, R_12: -15/2}
```

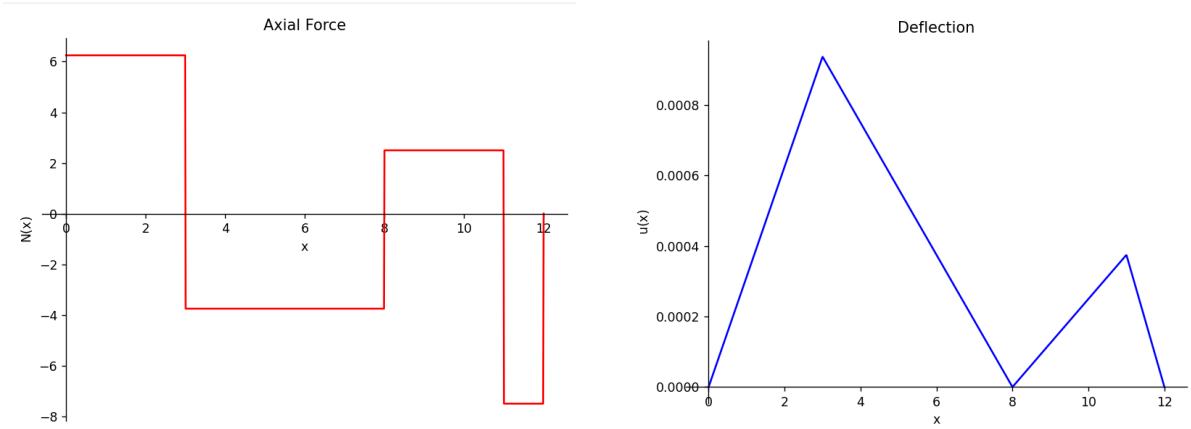
Lastly, the axial force and deflection formulas can be called and plotted. The functions are called in listing 3.6 and the singularity function expressions for axial force and deflection are provided in listing 3.7. In figure 3.2 the graphs for the axial force distribution and deflection along the column are provided.

Listing 3.6: Calling and plotting the axial force and deflection expressions

```
from sympy.physics.continuum_mechanics.column import Column
length, E, A = 12, 40000, 0.5
c = Column(length, E, A)
c.apply_support(0)
c.apply_support(8)
c.apply_support(12)
c.apply_load(10, 3, -1)
c.apply_load(10, 11, -1)
c.solve_for_reaction_loads()
c.axial_force()
c.deflection()
c.plot_axial_force()
c.plot_deflection()
```

Listing 3.7: Output axial force and deflection expressions

```
25*SingularityFunction(x, 0, 0)/4 - 10*SingularityFunction(x, 3, 0) + 25*SingularityFunction(
    x, 8, 0)/4 - 10*SingularityFunction(x, 11, 0) + 15*SingularityFunction(x, 12, 0)/2
0.0003125*SingularityFunction(x, 0, 1) - 0.0005*SingularityFunction(x, 3, 1) + 0.0003125*
    SingularityFunction(x, 8, 1) - 0.0005*SingularityFunction(x, 11, 1) + 0.000375*
    SingularityFunction(x, 12, 1)
```



(a) Axial force diagram

(b) Deflection diagram

Figure 3.2: Axial force and deflection graphs for the example column from figure 3.1

Positive axial forces lead to extension and negative forces lead to compression. Positive deflection leads to translation in the positive x-direction, usually to the right unless defined otherwise. The deflection diagram provides how much each part of the column is displaced after the loading. For example, a part of the column at x=2 displaced 0.0004 meters to the right after loading. As mentioned earlier, units are not tracked by the module and not given in the plots, users have to keep track of the units themselves.

3.2. Column with Distributed Loads

Now that the solution steps have been worked out for a simple example with point loads, a more complicated situation with distributed loads will be worked out. Distributed loads are realistic to model for columns, as columns in practice are always loaded by gravity, which is uniformly distributed along the column. The column object that will analyzed in this use case is modeled in figure 3.3.

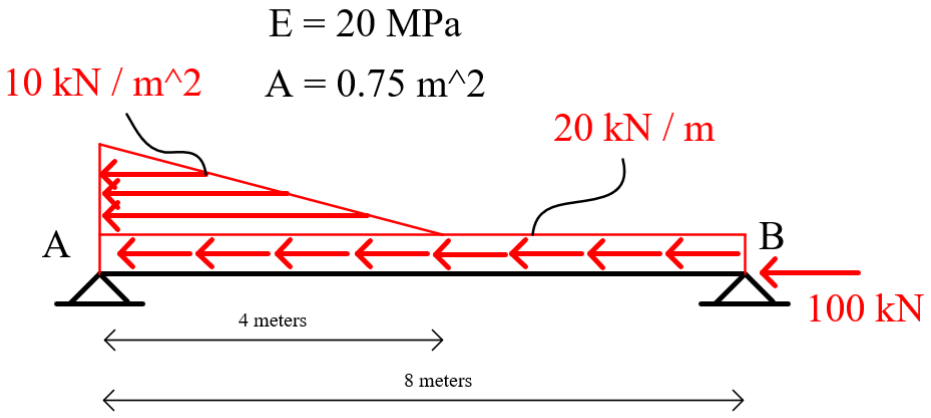


Figure 3.3: Column supported at both ends with different orders of loads

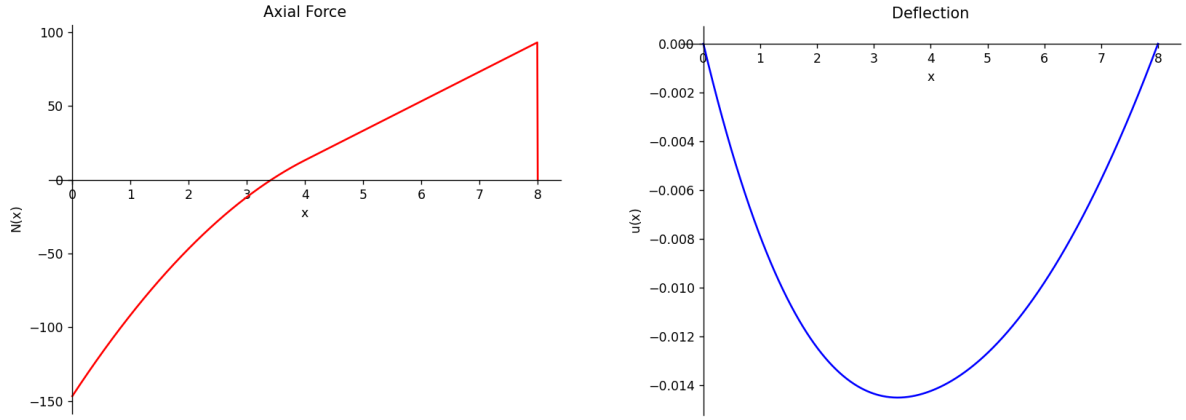
This column is supported at both ends, and is loaded with a point load of 100 kN in the negative direction. There is a uniformly distributed load along the whole length of the column, with an additional ramp load increasing from x = 4 towards x = 0. In listing 3.8 the code for modeling and solving this example is given. The output of the called properties are displayed in listing 3.9. In figure 3.4 the results for the axial force distribution and deflection are again provided.

Listing 3.8: Code for modeling and solving use case for distributed loads

```
from sympy.physics.continuum_mechanics.column import Column
c = Column(8, 20000, 0.75)
c.apply_support(0)
c.apply_support(8)
c.apply_load(-100, 8, -1)
c.apply_load(-20, 0, 0, end = 8) # Uniform distributed load
c.apply_load(-10, 4, 1, end = 0) # Ramp load, starts at x = 4
c.solve_for_reaction_loads()
c.reaction_loads
c.plot_axial_force()
c.plot_deflection()
```

Listing 3.9: Output for use case distributed loads

```
{R_0: 440/3, R_8: 580/3}
```



(a) Axial force diagram

(b) Deflection diagram

Figure 3.4: Axial force and deflection graphs for the example column from figure 3.3

3.3. Column with Telescope Hinge

Lastly, the column module will be used to solve an axial force problem including a telescope hinge. In figure 2.5 an example was given for a column with a telescope hinge. An adjusted version with numeric values is given below in figure 3.5.

At x = 3 there is a telescope hinge, providing a discontinuity in the deflection. As stated before, the telescope hinge can't transfer forces. The results of solving this example should reflect those aspects. This example is modeled and solved in listing 3.10 and the results of the unknown reaction forces and hinge displacement are provided in listing 3.11. After adding the supports, the hinge is applied to the column using `apply_telescope_hinge()` and providing the location of the telescope hinge. The property `hinge_deflections` is called to display the solved displacement of the hinge.

Listing 3.10: Code for modeling and solving the example with telescope hinge

```
from sympy.physics.continuum_mechanics.column import Column
c = Column(4, 10000, 1)
c.apply_support(0)
c.apply_support(4)
c.apply_telescope_hinge(3)
c.apply_load(50, 2, -1)
```

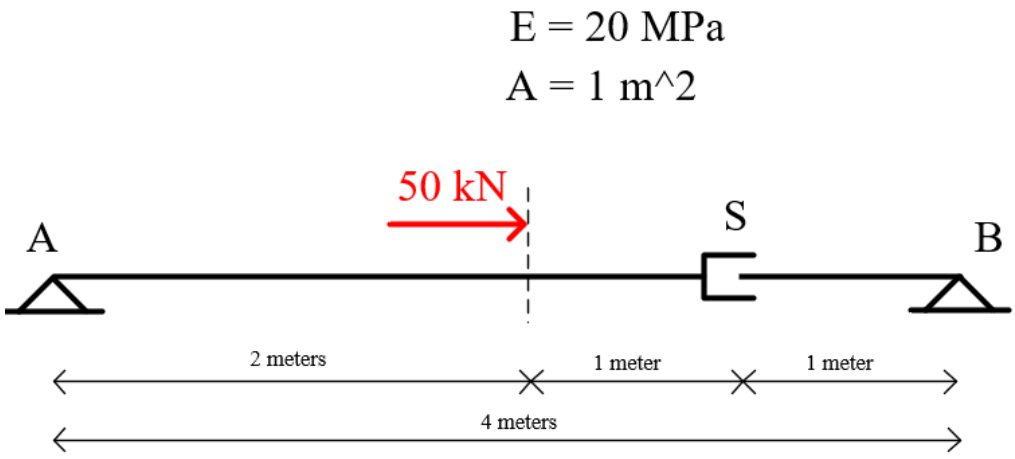


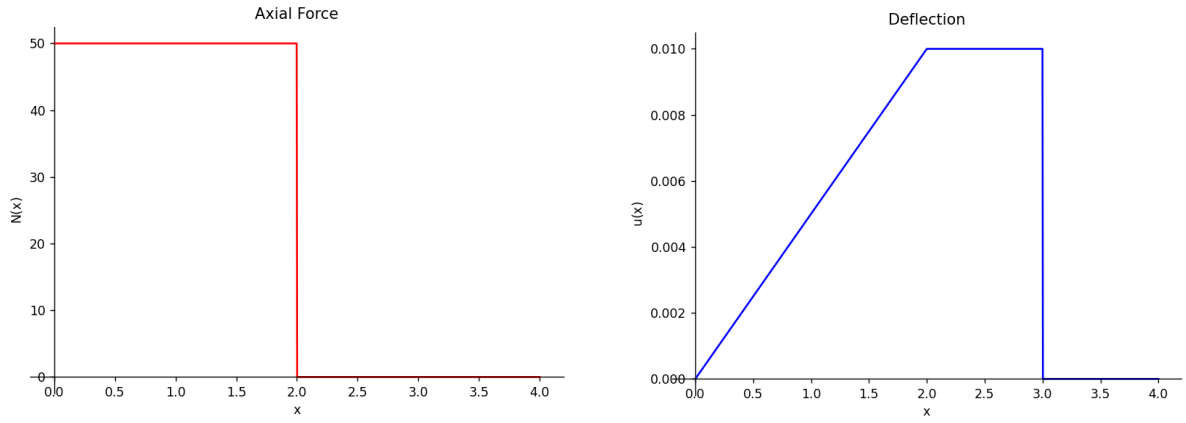
Figure 3.5: Column with two supports and telescope hinge

```
c.solve_for_reaction_loads()
c.reaction_loads
c.hinge_deflections
c.plot_axial_force()
c.plot_deflection()
```

Listing 3.11: Output for use case with telescope hinge

```
{R_0: -50, R_4: 0}
{u_3: 1/100}
```

These solutions reflect the theory, the right support has no reaction force. The deflection at the telescope hinge is 0.01 meters, element AS thus 'slides' past element SB. The plotted axial force and deflection diagrams are shown in figure 3.6.



(a) Axial force diagram

(b) Deflection diagram

Figure 3.6: Axial force and deflection graphs for the example column from figure 3.5

Figure 3.6a correctly shows that there is no axial force beyond the application point of the applied load. In the deflection diagram, the absence of displacement on of the element right of the hinge can also be seen.

4

Discussion and Recommendations

The implementation of the column module introduces support for axial forces in the SymPy structural mechanics tool. While the module achieves it's primary purpose, there remains many areas for further development and optimization. This chapter discusses these opportunities and provides recommendations for future improvements. Furthermore, the integration into the Structure2D module will be discussed.

4.1. Future Improvements to the Column Module

There are several potential enhancements that could improve the current version of the column module. During the module design, certain features—such as support for symbolic reaction expressions and fully customizable boundary conditions— were simplified with respect to the beam module. Although these features were not necessary within the scope of this project, they can be implemented in the future should specific use cases require them.

When implementing the column module into the Structure2D framework, maintaining close structural similarity to the beam module may become beneficial. Therefore, some of the simplifications made in this project, such as the handling of supports and boundary conditions, should be revisited to make sure it aligns well with the needs of the Structure2D module.

Another improvement could be making functions for calculating specific values for the maximum deflection and axial forces. The plots provide good visual indication, but precise values are also very useful. The maximum value is useful when choosing the material properties of the column, the column has to be strong enough to not yield at the maximum value. Another extension could be to generate a list of important locations where jumps or maximums occur, and provide the forces and deflections at these locations. A GitHub issue on this topic can be found *here*.

In addition to these refinements, several new features could further increase the module's applicability.

Springed Hinges and Supports

Introducing springed telescope hinges and supports would enhance both the flexibility and realism of the column module. These elements simulate partially restrained connections and support conditions, which can be related to practical engineering problems. Their implementation is expected to be relatively straight-forward, as their mathematical formulations are very similar to their non-springed counterparts. For springed supports, the boundary condition is expressed as:

$$u(a) = -\frac{R_s}{K_u} \tag{4.1}$$

where R_s denotes the reaction force of the spring and K_u spring's axial stiffness. This boundary condition demonstrates that the displacement is not necessarily zero, unlike in fixed supports. Therefore, the

handling of boundary conditions within the module must be adapted to accommodate for these cases. A GitHub issue for this topic can be found *here*.

Support for Variable Cross-Section and Elastic Modulus

Currently, the column module assumes a constant area and elastic modulus along the whole length. Supporting variable cross sectional properties or jumps in their values could greatly expand the range of realistic scenarios that the module can handle.

A scenario where this can be useful is when modeling tapered columns. Tapered columns are columns that increase in cross-sectional area closer to the ground, as lower sections need to carry more weight. Upper sections carry less weight, and thus material can be saved at those spots. An example of a tapered column is given in figure 4.1.

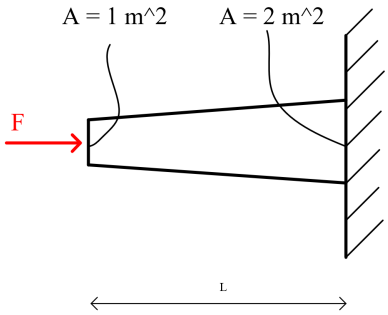


Figure 4.1: Column with increasing cross section

Another issue for this topic can be found *here*.

Drawing function

Another useful feature to implement in the future is a drawing function. This function is already implemented in the beam and Structure2D modules and allows for the users to easily visualize what exactly they are constructing and where the forces are being applied. However, a discussion point arises regarding the realism in the visual representation of the column objects.

In figure 4.1 the column has a fixed support. The column module doesn't recognize this support as fixed, only that it is fixed in the axial direction. A drawing of this column object would result in an unrealistic element with one pinned support. Functionality could be added that the user as to specify what type of support they want to apply to the column, which can then be reflected in the drawing. Specifying this should be optional, and the basic support would always be the pinned support. An issue for this can be found *here*.

4.2. Integration of Column Module into Structure2D Module

Due to the similarity between the load formulation and solution methods of the column and beam modules, the integration of the column module into the Structure2D framework can follow a similar approach to that of the beam. Currently, when a 2D structure is defined, a beam object with the unwrapped length is initialized using the `_init_beam()` function. A corresponding `_init_column()` function can be implemented to initialize a column object in a similar manner. When a load is applied to the two-dimensional structure, it is decomposed into axial and shear components. The axial loads should be applied to the column object, similar as to which shear forces are currently being applied to the beam object.

Additionally, the 2D `apply_support()` function should be extended to apply appropriate boundary conditions to the column object in cases of fixed or pinned supports. Lastly, the axial reactions should not be determined by summing axial loads manually, as is currently done. Instead, the Structure2D's own `solve_for_reaction_loads()` function should be modified to call the solve method of the column object. This approach allows for accurate computation of axial displacements and internal forces, particularly in statically indeterminate structures with multiple horizontal reactions.

5

Conclusion

This project successfully introduces a new column module capable of modeling and solving axially loaded columns, including statically indeterminate cases. The implementation supports a range of loading scenarios, including point loads and distributed loads. Telescope hinges, that allow for a discontinuity in displacement, can also be modeled and solved for. Lastly, the expressions for the axial force and deflection can be set up and plotted.

Large parts of the code structure and mathematical framework from the beam module have been reused to develop the column module. The process of applying supports and loads, setting up the load equation, and solving for the unknown is parallel to that of the beam module. However, within the methods, simplifications or alternate processes have been used with the column implementation. For example, how the column module models boundary conditions and supports is different. Simplifications have also been made for the input of the solver functions, allowing for users to call the function without referencing the unknowns themselves.

The use of helper functions has also been changed. The helper function for the Taylor expression has been modified to remove excess lines and improve clarity of it's purpose. Furthermore, the column module also uses a helper function to provide a load expression with the solved unknowns, which can be used to derive the axial force and deflection expressions.

Given the structural similarity to the beam module, the column module integrates well into the Structure2D framework. While the current Structure2D implementation does not fully support scenarios with multiple horizontal reactions, the integration of the column module lays the groundwork for extending its capabilities to include accurate handling of axial responses and multiple horizontal supports.

Overall, the column module extends the functionality of the continuum mechanics tools in SymPy by enabling axial analysis of structural members and offering a basis for further integration into two-dimensional structural systems.

References

Saheli, B. (2024). *Building a symbolic 2d structural analysis tool using sympy* [Bachelor's Thesis]. TU Delft.

van Gelder, M. (2024). *Expanding the implementation of macaulay's method in python* [Bachelor's Thesis]. TU Delft.

van Woudenberg, T. R. (2024). *Extension to macaulay's method* [Accessed: 2024-06-14]. https://teachbooks.github.io/Macaulays_method/intro.html

van der Wulp, J. (2023). *De methode van macaulay voor tweedimensionale constructies* [Bachelor's Thesis]. TU Delft.

A

Appendix A: MatrixFrame Model Calculations

In this appendix, the MatrixFrame models and results used to check the SymPy column module is shown. For each case the model is provided, after which the axial force diagram and nodal displacement values are given.

A.1. Use Case 1

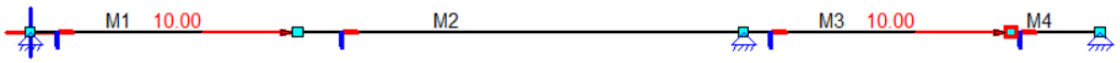


Figure A.1: MatrixFrame model for use case 1

Figure A.2: Axial force diagram for use case 1

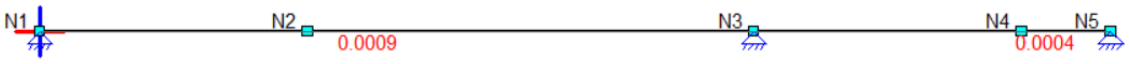


Figure A.3: Nodal displacements for use case 1

At the two point load location the deflections are 0.0009 and 0.0004 meters, respectively. These values reflect the peak values from the graph in figure 3.2b.

A.2. Use Case 2

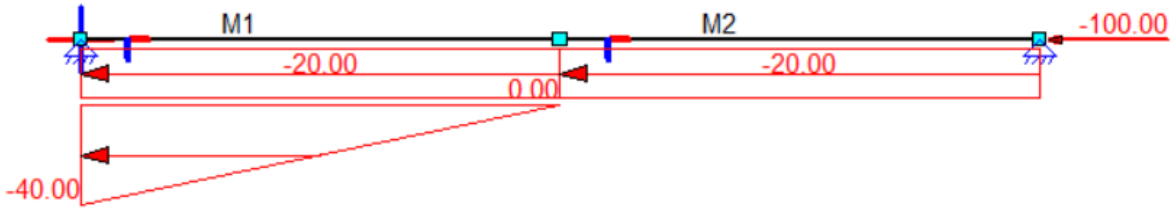


Figure A.4: MatrixFrame model for use case 2

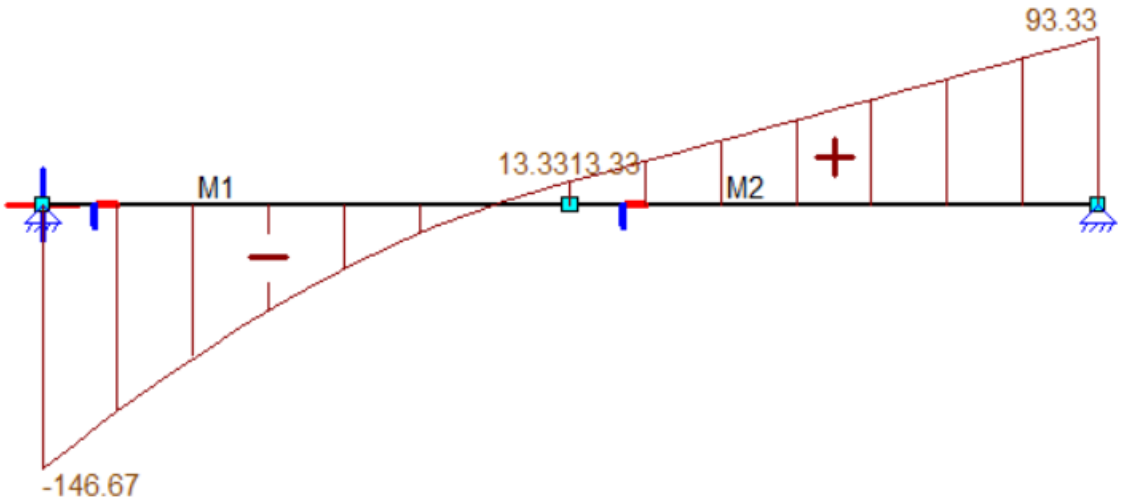


Figure A.5: Axial force diagram for use case 2

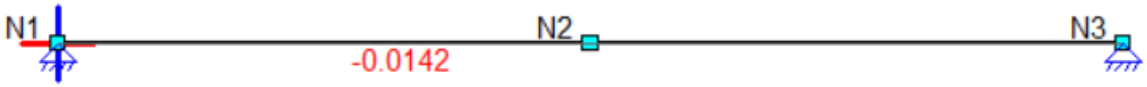


Figure A.6: Nodal displacements for use case 2

The displacement of the center of the column is -0.0142 meters. This reflects the value provided by the deflection graph from figure 3.4b.

A.3. Use Case 3

Figure A.7: MatrixFrame model for use case 3

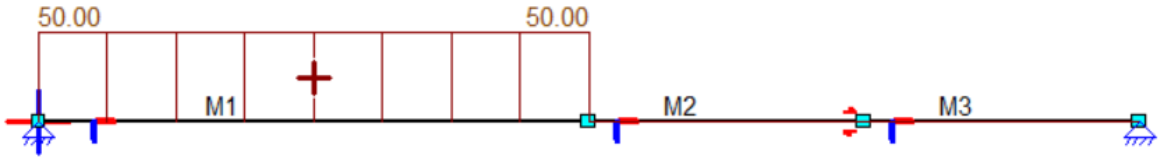


Figure A.8: Axial force diagram for use case 3

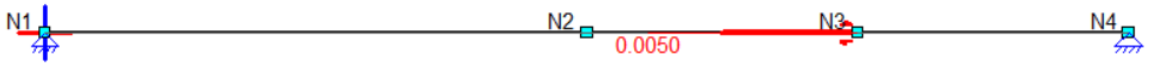


Figure A.9: Nodal displacements for use case 3