Building a Symbolic 2D Structural Analysis Tool Using SymPy

A Proof of Concept for Symbolic Computation in Civil Engineering

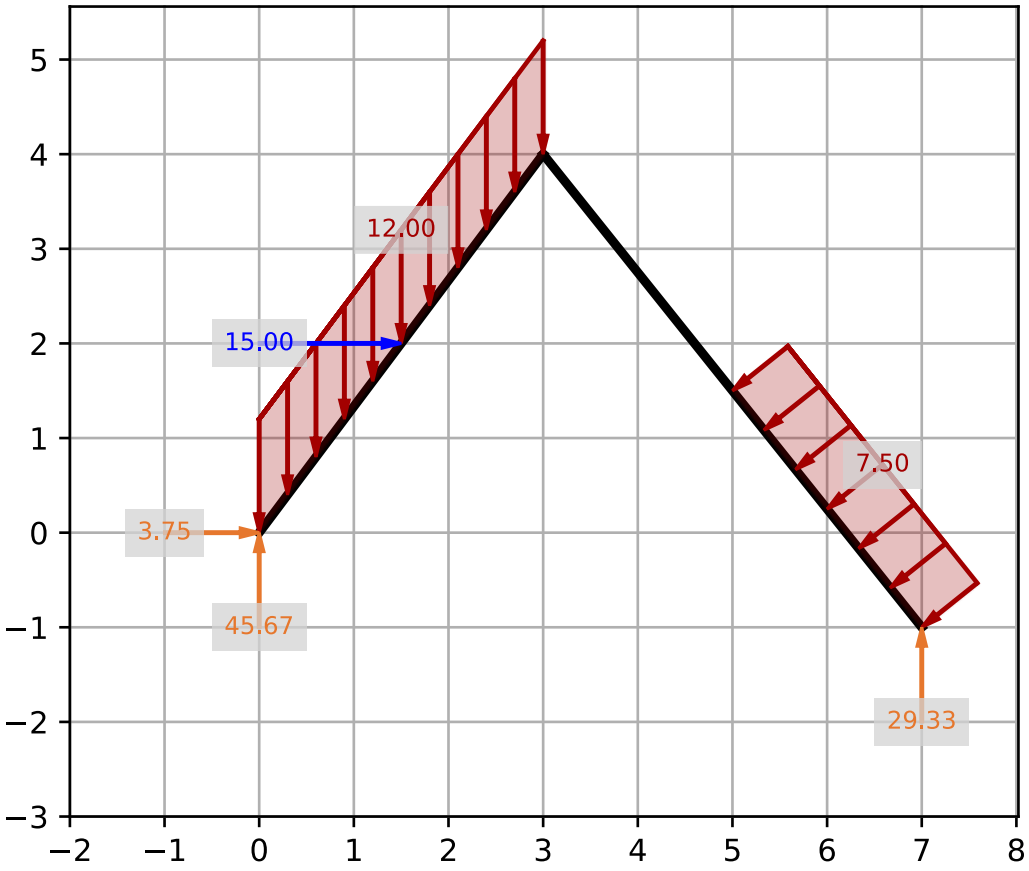

Figure 1 Title page example of solved reaction loads generated by the structure2d module.

B. Saheli
Technische Universiteit Delft
Faculty of Civil Engineering and Geosciences
[27-10-2024]

Title	:	Building a Symbolic 2D Structural Analysis Tool Using SymPy
Author	:	B. Saheli
Student number	:	4581741
First supervisor	:	T.R. van Woudenberg
Second Supervisor	:	J.K. Moore

I. Preface

I have written this report as an assignment for the bachelor end project of the faculty Civil engineering and geosciences. My personal goal was to expand my knowledge of python and structural mechanics, by contributing to open source. This project felt to me like a basic developer bootcamp which pushed me to learn many things that are outside of coding in Python but are essential for large project development. Alongside this report, I have written over 2200 lines of code, which include: a working proof of concept module, unit tests for the module and documentation. The code is available on my GitHub page (Saheli, 2024), and I encourage readers to try out the tool by cloning and installing it.

I would like to thank T.R. van Woudenberg for setting up the overarching Macaulay's method project, of which this project is a continuation. His insights at the start helped guide me in the right direction, and I'm grateful for his support. I would also like to thank J.K. Moore for steering me toward becoming a better programmer and helping me get familiar with GitHub. Finally, I would like to thank all the students that worked on the Macaulay's project before me, especially A.D.J. Baudoin as his examples and method were essential for debugging and testing the tool.

II. Summary

This report describes the development of a Python based tool for 2D structural analysis in civil engineering, using the SymPy library (Meurer A, 2017) and applying the Macaulay method (Macaulay, 1919) to enable symbolic computation for structural analysis. The tool serves as a proof of concept demonstrating the feasibility of symbolic computation in analyzing internal forces.

Central to the project is the extension of existing SymPy modules, specifically the Beam module, to support 2D analysis by introducing new modules: "Column" and "Structure2D." The existing "Beam" module in SymPy, which computes vertical load effects, was used to handle shear forces and bending moments by creating a load equation for the vertical direction. To handle normal forces a "Column" module was proposed (although not implemented in this iteration), designed to compute horizontal load effects based on the material's elasticity modulus, length, and cross-sectional area. The "Structure2D" module integrates both vertical and horizontal load computations by transforming a 2D structure into two corresponding 1D representations. This method enables separate computations for each direction, converting 2D problems into simpler 1D computations while preserving load and support behavior.

The tool's workflow involves users defining structure components (members, loads, supports) on a 2D grid. Once defined, the structure is "unwrapped" to map each member along a continuous line. Each load is split into its horizontal and vertical components and applied to the corresponding axis. Through examples, the report showcases the tool's ability to calculate shear forces, bending moments, and reaction loads.

Although the project successfully demonstrates the potential of symbolic computation for 2D structural analysis, several limitations exist. The tool currently lacks a functional Column module, restricting it to single horizontal reaction loads and limiting its ability to solve for horizontal deflections and axial forces fully. Additionally, the structure must be non-branching and linear, excluding complex configurations like truss networks. Material properties must be uniform across members, a constraint that simplifies the model but limits its real-world applicability. Addressing these limitations would involve developing a more advanced unwrapping algorithm for intersecting members and introducing a complete Column module.

In summary, this prototype validates the possibility of a symbolic 2D structural analysis tool, providing civil engineering students and educators with an analytical framework that combines Macaulay's method and modern symbolic computation. This tool lays the groundwork for future developments that could expand its functionality, to move away from proof of concept to a fully featured open-source product.

Contents

1. Introduction

This project focuses on developing a 2D structural analysis tool using Python's SymPy library, with the Macaulay method as the main calculation technique. By using the symbolic computation power of SymPy alongside Macaulay's method, the tool provides engineers and students with an symbolic way to model, analyze and understand the behavior of structures. This proof of concept demonstrates the potential of using symbolic computation for structural analysis, offering clear insights into how loads and supports affect internal forces and deflections. It also lays the foundation for future development, where additional features and functionalities could be added to enhance its capabilities.

In structural engineering, calculating internal forces and deformations in 2D structures is essential for understanding the principles of Euler-Bernoulli beam theory. There are several methods available to perform these calculations, and one of the lesser-known approaches is the Macaulay method. This method uses singularity functions, which allow complex loading scenarios to be modeled as a single continuous equation. As the load position changes along the structure, the relevant singularity functions are activated, and as this is a single continuous equation only boundary and deformation conditions are needed. This concept is further expanded to 2D by the Macaulay method project (Woudenberg T. R., 2024) which is an overarching research project into the expansion and advancement of the Macaulay method. This 2D method is the foundation of this project.

This leads to the central question of this report:

"Is it possible to develop a symbolic 2D structural analysis tool in Python, using Macaulay's method as the underlying mathematical framework?"

To answer the main question of this report, the following structure will be used. In Chapter 2, the design and architecture of the tool are discussed, including how existing SymPy modules were leveraged to build the 2D structural analysis functionality. Key concepts such as the integration of vertical and horizontal load equations using new modules are introduced here. Chapter 3 presents a detailed use case example illustrating how to input structure data, add loads and calculate reactions, shear force and bending moment diagrams. Chapter 4 evaluates the performance and limitations of the proof of concept, focusing on the results of the Macaulay method in 2D analysis. Finally, Chapter 5 offers a discussion on results and recommendations for further development.

2. Design of the Tool

This Chapter outlines the design and architecture of the 2D analysis tool. It builds on the existing SymPy continuum mechanics modules, to implement 2D functionality efficiently. The Chapter introduces two new modules, "Column" and "Structure2D," which work with the existing Beam module to handle both vertical and horizontal load equations, enabling full 2D structural analysis.

2.1 Overview of Existing SymPy Codebase

Rather than developing new functionality from scratch, it is essential to leverage as much of the existing SymPy codebase as possible. This ensures efficiency and maintains backward compatibility, a core principle of open-source libraries like SymPy.

Currently the SymPy library has a continuum mechanics module containing Beam, Truss, Cable and Arch sub modules.

A brief overview of the existing continuum mechanics modules in SymPy:

- **Beam Module**: This module uses singularity functions to solve a horizontal beam with applied loads. It is well-written, well-documented, and has a good input style.
    - **Beam3D**: This is copy of the original beam module with built in functionality to support forces acting from 3 dimensions, a lot of code is repeated here. This should have been merged into the original beam module.
- **Truss Module**: Used to solve 2D statically determinate truss structures, this module does not use singularity functions for solving and uses a completely different input style that is not compatible with the beam module.
- **Cable Module**: This module solves cables and is outside the scope of this project.
- **Arch Module**: It solves problems related to a three-hinged arch (determinate) structure. This module is poorly documented and falls outside the scope of the project.

The Beam module, with its well documented and flexible input style, serves as a good foundation for developing a 2D analysis tool. It already includes many of the required features, and any additional functionality can be integrated into a new module rather than creating duplicate code. This is especially important for ensuring consistency and avoiding issues, like those seen with the Beam3D module, which introduces code duplication by not being properly merged into the original module.

2.2 Overview of Design

To implement 2D functionality while reusing as much existing code as possible, it is essential to analyze the current beam module and identify which parts can be leveraged (a detailed analysis of the beam module can be found in Appendix B - Overview of the Existing Beam Module). The beam module currently generates load equations for vertical forces acting on a straight horizontal beam. These vertical loads in the z-axis (aligned with the grid y-axis) create shear forces, which in turn cause bending moments, an important insight for understanding load behavior.

> *Note: the terms "x and y" and "x and z" are used interchangeably in the code and in this report. Z and Y both represent the vertical axis and should be understood as referring to the same vertical direction.*

Initially, it appears that the module's focus on strictly vertical loads might limit its functionality. However, a vertical load can also represent the vertical component of a load acting in any direction. With this perspective, it becomes clear that when a straight beam is subject to loads at various angles, these angled loads can be decomposed into their x and y components. According to the Euler-Bernoulli beam theory, the vertical component acts as a shear force, while the horizontal component represents a normal force. These forces are independent because they act perpendicularly.

This concept is demonstrated in Figure 2. In the top plot, a 20 *kN* load is applied at a 225° angle relative to the positive x-axis. To find the corresponding shear force equation, this load is decomposed into its vertical component by multiplying by sin(225°), yielding 14.14 *kN*. This value is then applied directly to the beam in the vertical direction, as shown in the bottom plot. The shear forces in these two plots are identical.

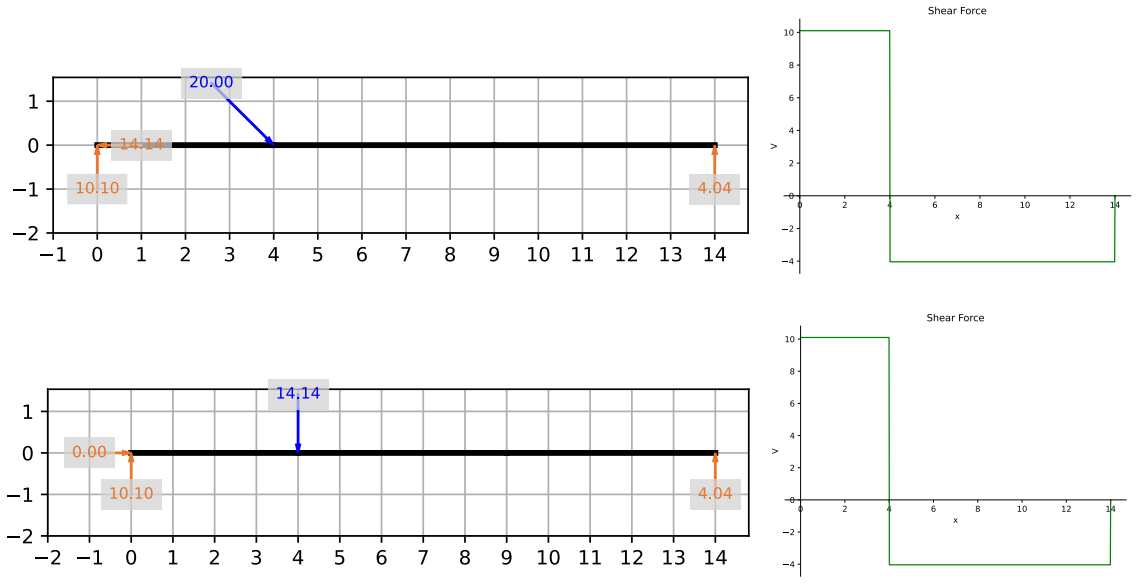

Figure 2 Equivalence of vertical decomposition, with shear force graphs on the right.

From these findings, it is clear that shear force and normal force can be calculated independently, provided all loads are decomposed into their vertical and horizontal components. The beam module can handle the vertical components, but currently, there is no module to compute horizontal load effects. To address this, a new module called "Column," will need to be developed to handle horizontal

loads. Each structure will then have a load equation for both axes: one for vertical deflection and one for horizontal deflection.

By combining the beam module with the new column module, it becomes possible to generate load equations for both x and y directions. However, both modules are designed to work with one-dimensional straight structures, with loads acting perpendicularly on the beam and normally on the column. See Figure 3 for a clarification of 1D vs 2D.

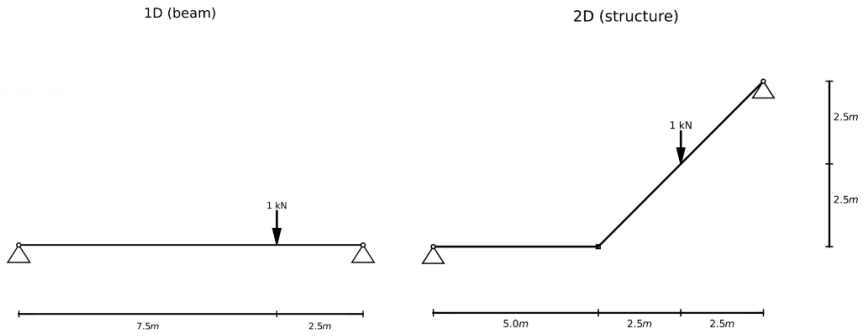

Figure 3 Left: 1D beam/column (a straight line along the x axis), Right: a 2D structure consisting of two members connecting at an angle

This setup presents a challenge for 2D problems, as neither module can directly process 2D coordinates. To solve this, the two modules must be integrated. This integration will be managed by a new master module, "Structure2D." The primary function of Structure2D will be to take 2D user inputs, split them into separate components for each axis, convert them into a format compatible with the beam and column modules, pass the data to these modules for computation, and then combine the results into a final 2D solution. A simplified overview is provided in Figure 4.

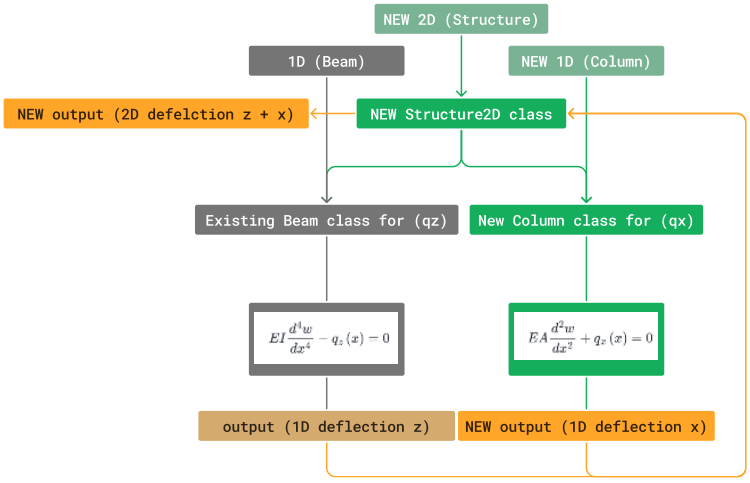

Figure 4 High level overview of structure2d module

The compatible format refers to a 1D beam or column with only its respective load components, achieved by laying the structure out as a continuous line. 2D structures can be "unfolded" to form a straight line, meeting the 1D input requirements of the beam and column modules. Let's illustrate this for the vertical direction starting from Figure 5. By unfolding the structure, we isolate and apply the vertical components of all loads to their respective positions along this line. However, the shear force distribution is incorrect due to the limitations of standard Macaulay's method. In a 2D structure, members can have angles so a load applied globally in the vertical direction might not act entirely

vertically on an angled member. This issue is illustrated in Figure 6, where a distributed load passed the first bend point acts on an angled member. Because the member is angled, the load isn't exclusively vertically relative to the member, it also has a horizontal (normal) component. This results in the load's effective contribution to the global vertical direction being lower than the distributed load's full value of 6 *kN/m*.

The solution to this problem lies in a 2D adaptation of Macaulay's method, which compensates by introducing additional loads to achieve a statically equivalent situation for each direction. In (Baudoin, 2024), these equations have been converted into an algorithm that adds the necessary compensating forces to maintain equivalence. This adjustment is demonstrated in Figure 7, where after adding the additional loads the shear force diagram aligns perfectly with the original 2D problem.

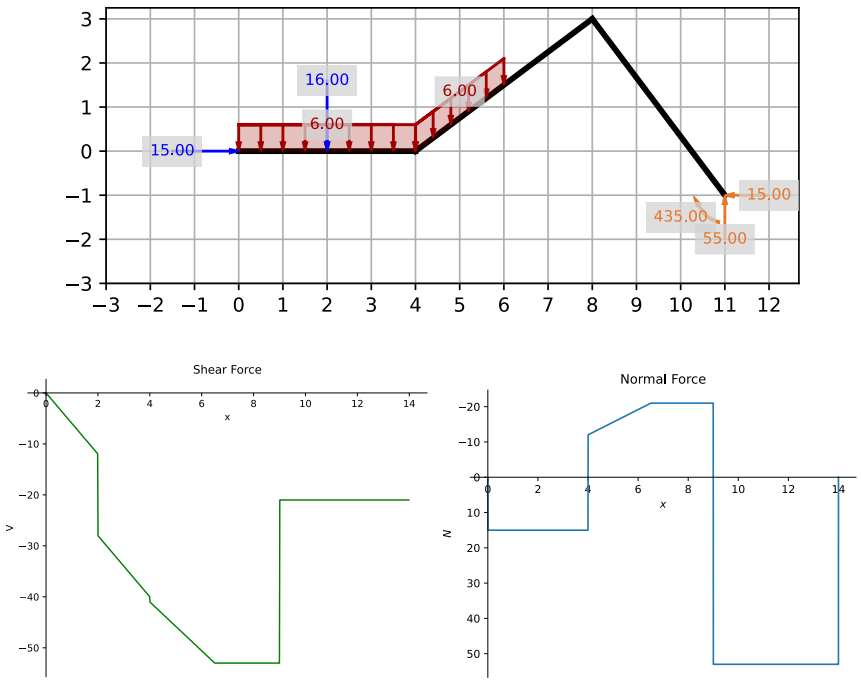

Figure 5 2D structure with bend points and loads with shear force and normal force diagrams.

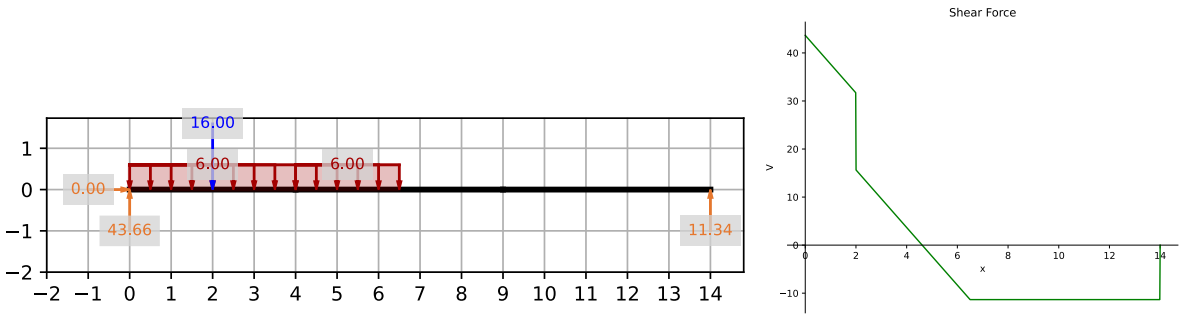

Figure 6 Unfolded structure with identical loads from Figure 5 (Notice how the shear force starts off correctly but as it reaches the first bend it fails.)

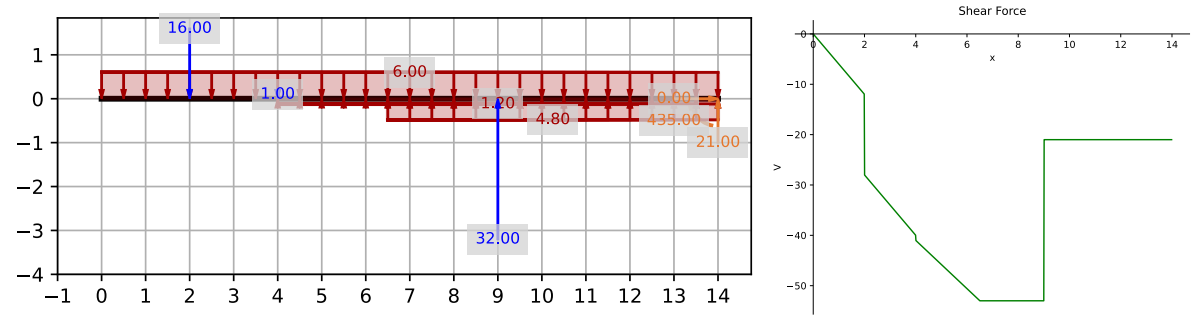

Figure 7 Unfolded structure with added compensating loads on top of the original loads from Figure 5. Shear force diagrams are now identical.

The same principle works for the horizontal normal loads. In Figure 8 additional loads are added based on the 2D Macaulay principle but this time in the horizontal direction. This again results in an identical normal force diagram.

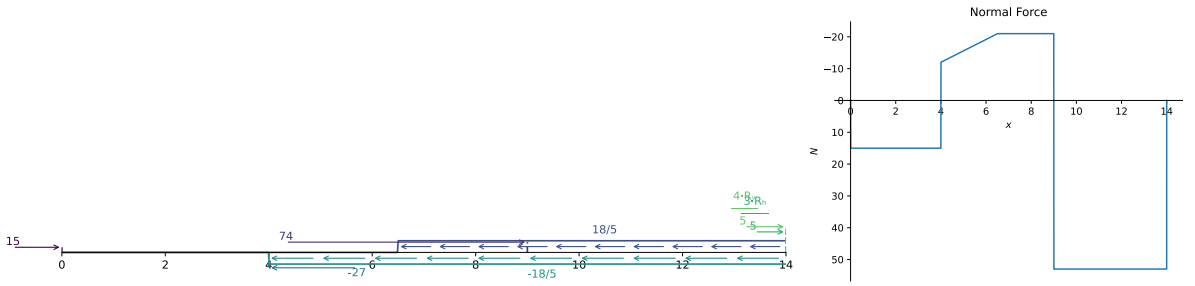

Figure 8 Source: (Woudenberg T. R., 2024) Unfolded structure with added compensation loads on top of original loads, for the horizontal direction.

After determining both shear and normal forces using the 2D Macaulay method, these forces are expressed as load equations. These load equations are then integrated according to the Euler-Bernoulli beam theory, resulting in two equations that represent deflection in the x and y directions. At this stage, however, it's no longer possible to solve these equations independently, as each one relies on the other. Solving this requires treating them as a system of equations.

2.3 New Modules

As discussed in Section 2.2 two new modules need to be created to handle 2D functionality. This Section will discuss the functionality of these new modules in detail.

2.3.1 Column

> *Note: This module is only proposed but due to time constraints not implemented in the proof of concept.*

The Column module is designed to work similarly to the Beam module but focuses on the horizontal direction, addressing loads applied along the length of the column. It uses properties of the column, such as its length, elastic modulus (*E*), and cross-sectional area (*A*). These properties are critical for determining how the column deflects horizontally under various normal forces.

For each type of load applied to the column, an appropriate singularity function is used to represent the effect of that load. The order of the singularity function is selected based on the type of the load, as explained in mathematical theory. These singularity functions are then combined into a total load equation that represents the forces acting in the horizontal direction. This total load equation sums the effects of all applied forces along the column and follows the same principles used for beams in the vertical direction.

An overview of the architecture of the column module can be seen in Figure 9 it based on the existing beam module design. The core functionalities are colored orange, and the secondary functionalities are colored yellow.

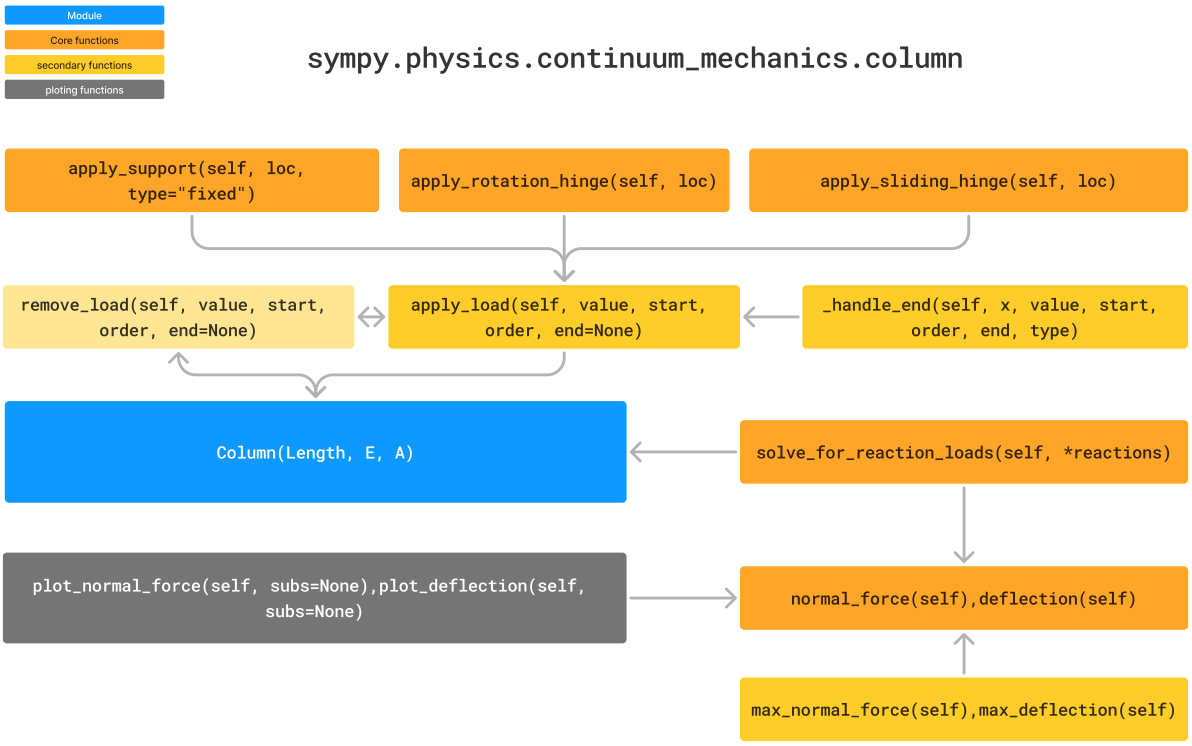

Figure 9 Overview of proposed Column module architecture

2.3.2 Structure2d

The Structure2D module serves as the central component of the tool, extending the capabilities of the existing Beam module to analyze two-dimensional structures. The module is designed to compute deflections, shear forces, and bending moments in both the vertical (z) and horizontal (x) directions using the singularity function approach (2D Macaulay method) as previously described. By decoupling the computations into two directions, the module simplifies the complexity of 2D structural analysis into two 1D problems. Which are then solved by the beam and column module independently. An overview of the architecture can be seen in Figure 10.

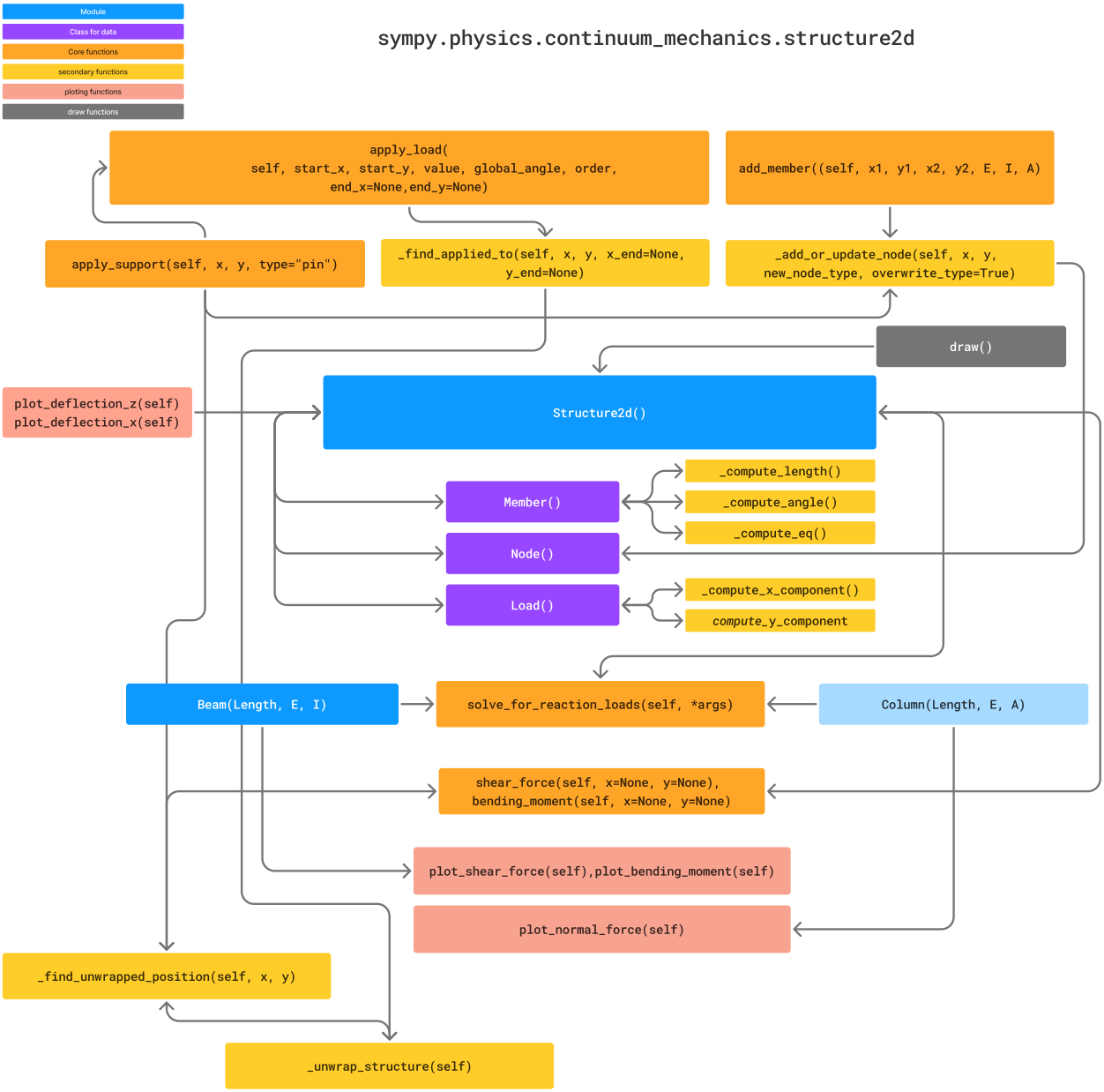

Figure 10 Overview of structure2d module architecture

Module Functionality and Key Functions

The Structure2D module serves as the core component of the tool, managing data and facilitating interactions between user inputs and computational functions. Building upon the existing Beam module, it introduces new capabilities to handle two-dimensional structures. This section details the functionality of the Structure2D module and its key functions and user interaction.

In the architecture of the code Figure 10, different components are color-coded for clarity: blue represents continuum mechanics modules (with the light blue the column module is included for completeness but not implemented in this proof of concept), purple indicates classes that primarily serve as data objects, orange are core functions that users interact with directly, and yellow are helper functions.

The Structure2D module acts as a data management system, storing all information about the structure being analyzed. To make data handling easier, three additional classes are implemented to store data: Member, Node, and Load. These classes contain related data and compute necessary attributes, such as angles and equations for members, and the x and y components of loads.

Users interact with the module primarily through core functions like **add_member()**, **apply_load(),** and **apply_support()**. When a user calls **add_member()**, the module creates a new Member object with the provided input and adds it to the list of members within the Structure2D instance. Similarly, nodes are managed by checking whether a node already exists at a given coordinate to avoid duplication, using the **_add_or_update_node()** function. This function checks if a node is present at the specified coordinates and either adds a new node or updates an existing one, which is essential when support reactions are added later.

When using **apply_load()**, the function creates Load objects and adds them to the structure's list of loads. The Load class automatically splits any applied loads into their x and y components. It then calls the **_find_applied_to()** function to determine whether the load is applied to a node or a member, this function checks the coordinates of the load against existing nodes and members. For nodes, it compares the load's coordinates to each node's coordinates and checks if the difference is zero. For members, it uses the member's equation, computed by the Member class, and substitutes the load's x-coordinate into the equation, checking if the difference between the computed y-value and the load's y-coordinate is zero. There is a special case when a member is vertical, this case is handled separately. Once a corresponding member is found, a local x-coordinate is computed.

If the applied load is a distributed load, the code uses the midpoint of where the distributed load is acting for the check. This saves one check but introduces a small limitation: distributed loads must act on one member only. The function returns a member ID, the local x-values (the coordinate along the member where the load is acting), and optionally the local end values if it’s a distributed load, or only the node ID if the load is applied to a node.

> *Note: Why not compare directly, and why always check if the difference equals zero? In mathematics, there is no difference between 0.5, ½, and 4/8, as humans can immediately tell these all equal one-half. However, for computers, all of these are different, some even have different data types. Therefore, it is easier to take the difference and check if it equals zero to avoid confusion.*

Next, the **apply_load()** function calls **the _unwrap_structure()** function, which "unwraps" the structure. This process involves mapping the members of the structure onto a continuous, linear axis, effectively transforming the 2D problem into a 1D one that can be analyzed using the Beam module's capabilities. The module then lays out each member sequentially along the unwrapped x-axis, using a cumulative sum of lengths. This method positions each member correctly relative to the others, maintaining integrity in the unwrapped form. During this process, each member and node are checked for loads that are acting on them, and then the local x-coordinates get updated with unwrapped ones. An example of how a structure is unfolded can be seen in Figure 11.

Figure 11 Unwrapping visualization. Top: Original 2D structure, Bottom: Unwrapped structure.

These unwrapped coordinates are also put in a list and added to the Structure2D attributes for unwrapped loads points and for unwrapped bend points. Finally, these two attribute lists are than converted into a format that can easily be passed into an algorithm described in (Baudoin, 2024). This algorithm takes coordinates of all loads and bend points and builds the actual equation for loads in the vertical direction.

The **apply_load()** function then updates the beam's load equation using this algorithm. It translates the 2D loads into equivalent 1D loads along the unwrapped beam, considering the angles of the members and the positions of the loads.

This approach has limitations. It assumes that the structure does not contain any branching or intersecting members and that members are added in order. There is no internal tracking of where each member is mapped on the unwrapped line beyond this cumulative process. In structures with branches or intersections, this method would fail because it lacks the capability to "cut" the structure into sections and reassemble them while keeping track of each member's position and orientation.

To address these limitations, future versions of the Structure2D module would need a more advanced unwrapping algorithm. Such an algorithm would handle branching and intersecting structures by systematically decomposing the structure into sections, logically reassembling them in onto the unwrapped axis, and tracking the relationships between members. This improvement would also remove the requirement for members to be added in a specific order, increasing the module's flexibility.

When the user uses the **apply_support()** function, it calls the **_find_unwrapped_position()** function. This function finds the unwrapped x-location based on x and y coordinates to find the unwrapped position of the support. It calls the **_unwrap_structure()** function to obtain this data. Once this is complete, the **apply_support()** function checks what type of support is input and adds corresponding loads and boundary conditions to the unwrapped location. These loads are yet to be computed and are therefore added symbolically, representing reaction loads that can vary based on the type of support. **apply_support()** finally returns the symbols that represent the reaction loads.

When the user wants to solve the structure, the **solve_for_reaction_loads()** function is used. This function takes all the reaction load symbols and passes them into the beam's **solve_for_reaction_loads()** function. This function returns the solved vertical and moment reaction loads only. This leaves the horizontal reaction loads still unsolved. To solve the horizontal reaction loads, a horizontal solver should be used, this solver should be part of the Column module. As previously mentioned, this module is not implemented. To get around this problem without developing the Column module, a simple solver is built. This solver only uses Newton's third law ($\Sigma F = -\Sigma F$) to solve for simple equilibrium. This introduces a limitation where only one horizontal unknown support reaction can currently be solved. Once all the support reactions are solved, they are combined into a dictionary and saved as attributes of Structure2D.

The plotting functions for shear force and bending moment are directly connected to the plotting functions of the Beam module. The **shear_force()** and **bending_moment()** functions are also connected to the Beam module directly, however, the Beam module only takes in the 1D unwrapped position. This needs to be converted to the unwrapped position by using the previously mentioned **_find_unwrapped_position()** function. When these functions are called without arguments, they return the full equation. If only an x-coordinate is provided, they compute the value at that unwrapped position. If both x and y coordinates are provided, they compute the value at the corresponding position in the 2D structure.

The **summary()** function works by calling two helper functions: **_print_reaction_loads()** and **_print_points_of_interest().** The first function prints the reaction loads. The second function prints points of interest for shear force and bending moment. The points of interest for bending moment are the bend points and are directly taken from the Structure2D attributes. The shear force points of interest are the bend points and the points where a point load is acting. These points often have jumps in the graph and are therefore evaluated just before the point of interest and just after the point of interest. The values are computed using the previously mentioned **shear_force()** and **bending_moment()** functions.

The **draw()** function handles drawing plots of the situation using matplotlib. It uses data from the attributes of Structure2D. It loops over all the member coordinates and adds them to the plot. After this, it loops over the loads. When looping over the loads, they are checked based on their order and whether they are a support reaction. Support reactions are marked with a different color. It is important to check the order of the loads, based on this, it's possible to determine the type of load whether it is a point load, distributed load, or a bending moment. Icons for the respective load are then selected and drawn. There is also a check to see if the load is positive or negative. If the value is negative, the load gets flipped. This is to ensure that the arrows representing the loads are pointing in the right direction. The loads also get checked for whether they are of symbolic value, if this is the case, a grey color is selected. Finally, the **draw()** function sets the grid to have an equal ratio and recomputes the vertical ticks on the y-axis to ensure a square grid.

3. Use case example

In this Chapter, we demonstrate the implementation of the new modules described in Chapter 2 through a step-by-step example. The example is solved using the newly developed Structure2D module. Readers can follow along with the example in Python using the structure2d fork from (Saheli, 2024). The goal of this example is to solve the reaction loads and to determine the shear force and bending moment influence lines. The full code can be found in Appendix A - Example from Chapter 3

Example

Consider a two-dimensional structure shown in Figure 12, consisting of three connected members. The first member is 4 meters long and horizontal. The second member is connected to the end of the first member and is 5 meters long, forming an angle of arctangent (3/4) relative to the x-axis. The third member is also 5 meters long and forms an angle of arctangent (4/3) relative to the x-axis. There is a fixed support at the end of the third member. The sign convention used in this example is, forces acting downward and to the right are considered positive. Angles for loads are measured relative to the positive x-axis in the anticlockwise direction, and angles for members are measured relative to the positive x-axis in the clockwise direction.

The material properties are defined as follows: Elasticity Modulus (*E*): 30000 kN/m^2

- Second Moment of Area (*I*): 1 m^4
- Cross-sectional Area (*A*): 10000 m^2

The structure has two-point loads:

- Fh: A horizontal load of 15 *kN* applied at the start of the first member, acting to the right (positive x).
- Fv: A vertical load of 16 *kN* applied at the midpoint of the first member, acting downward. (negative y)

Additionally, there are two distributed loads, noted as qv:

- The first distributed load is 6 *kN/m*, applied along the entire length of the first member, acting downward (negative y).
- The second distributed load is 6 *kN/m*, applied over the first half of the second member, also acting downward (negative y).

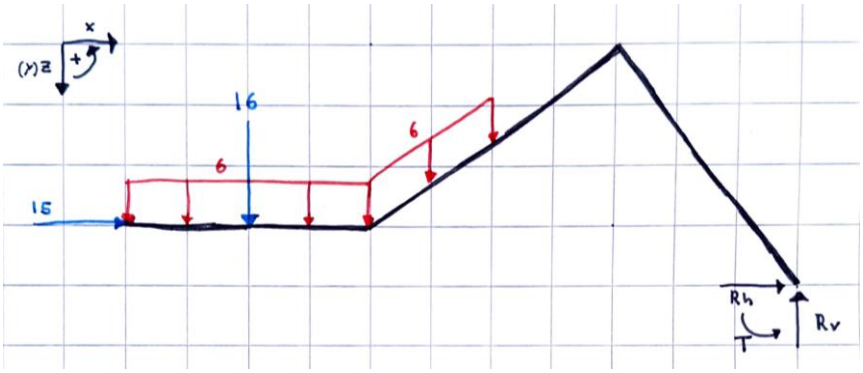

Figure 12 Hand sketch of the example

Let's solve the example using the structure2d module. First, we import the Structure2d module from the SymPy library with sympy.physics.continuum_mechanics.structure2d. We then define the material properties: the elasticity modulus (*E*), the second moment of area (*I*), and the cross-sectional area (*A*). Next, we create an instance of the Structure2d class, assigning it to the variable s. This instance sets up the code to run method functions on the Structure2d object, as shown in Listing 1.

Listing 1 Initializing Structure2D module

```
from sympy.physics.continuum_mechanics.structure2d import Structure2d
%config InlineBackend.figure_format = 'svg'
E = 3e4
I = 1
A = 1e4
s = Structure2d()
```

Note that the line "%config InlineBackend.figure_format = 'svg'" is used in Jupyter notebooks to configure the output format of plots for better image quality.

3.1 Input for structure2d

Once the structure object is initialized, we need to input the example problem into the Structure2D object. This input consists of two main steps: adding members and adding loads. Both steps involve defining x and y coordinates on a two-dimensional grid, where the structure can be drawn. Additionally, the user can generate a plot of the structure at any step, allowing for easy visualization throughout the process. Values for loads can be either SymPy symbols or numerical values.

3.1.1 Add members

Let's build the structure from the example. Start by selecting an arbitrary reference point on the grid in this case, the reference point is chosen to be (0, 0). The first member is horizontal and 4 meters long, starting at the reference point and extending from left to right. Therefore, the coordinates for the start and end of this member are (0, 0) and (4, 0), respectively.

For the two remaining members, the coordinates are less straightforward. Since Structure2D currently doesn't support adding members using angles and lengths directly, we must first convert these into sets of coordinates based on the geometry of the structure. This design choice was made to save time, there is a possibility to make a new function that can add members based on their angle and length. Suppose the second member is angled and extends from (4, 0) to (8, 3), and the third member extends from (8, 3) to (11, -1). These coordinates are determined by calculating the end points based on lengths and angles or from known coordinates in the problem.

Once the coordinates are determined, we assign the material properties defined earlier to all members. The code to add the members is seen in Listing 2

Listing 2 Adding members

```
s.add_member(x1=0, y1=0, x2=4 , y2= 0, E=E, I=I, A=A)
s.add_member(x1=4, y1=0, x2=8 , y2= 3, E=E, I=I, A=A)
s.add_member(x1=8, y1=3, x2=11, y2=-1, E=E, I=I, A=A)
```

3.1.2 Plots (draw)

At any point during the process of solving, it is possible to use the draw method to obtain a plot that shows the current state of the structure. This not only helps to visualize the result but also assists during the setup process by verifying the correct placement of members. The draw method offers several parameters that allow you to customize the visualization:

- forced_load_size: A numerical value that forces all loads to be drawn with the specified magnitude, based on the grid units. This can help standardize the appearance of loads for better visual comparison, when load values are very high.
- show_load_values: A Boolean value (True or False) that toggles whether the numerical values of the loads are displayed on the plot. Setting it to True will annotate the loads with their magnitudes.
- draw_support_icons: A Boolean value (True or False) that determines whether icons representing the types of supports (fixed, pinned, roller) are drawn at the support locations.

Let's use the draw method with these parameters to visualize the current state of the structure. Run Listing 3 to see the plot.

Listing 3 Draw function

```
s.draw()
```

The output should look like Figure 13 the structure now appears identical to the example.

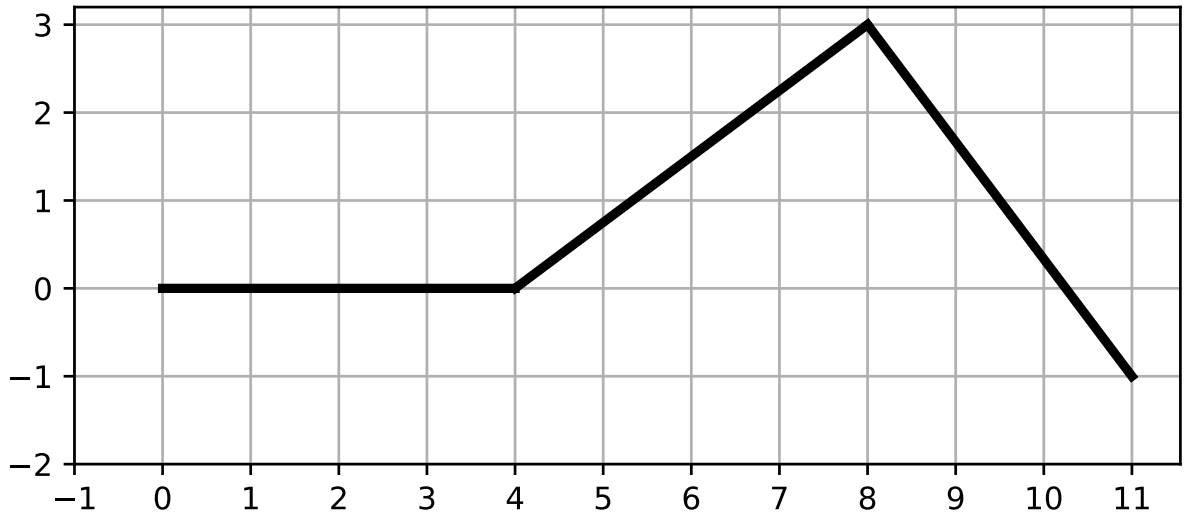

Figure 13 Plot generated by the draw function. (structure only)

3.1.3 Add loads

Now that the structure is set up, we can add loads to it. The Structure2D object provides two methods to apply loads: **apply_load()** for external loads and **apply_support()** for support reactions. In the example, there are four external loads: two-point loads and two distributed loads.

The **apply_load()** method takes several parameters:

- start_x and start_y: the coordinates where the load starts.
- value: the magnitude of the load.
- global_angle: the angle at which the load acts, measured in degrees from the positive x-axis.
- order: the order of the load, based on the mathematical theory of singularity functions -1 for point loads and 0 for constant distributed loads.
- end_x and end_y (optional): the coordinates where the load ends (required for distributed loads).

Let's input the first two-point loads, Fh and Fv as seen in Listing 4 and Listing 5. Fh is a point load of 15 *kN* acting at (0, 0) along the positive x-axis (0 degrees).

Listing 4 Apply Fh point load

```
s.apply_load(start_x=0, start_y=0, value=15, global_angle=0  , order=-1)
```

Fv is a point load of 16 *kN* acting at (2, 0) downward (270 degrees)

Listing 5 Apply Fv point load

```
s.apply_load(start_x=2, start_y=0, value=16, global_angle=270, order=-1)
```

For the distributed loads, we need to specify the start and end coordinates. For example, a distributed load of *6 kN/m* acting downward between (0, 0) and (4, 0) is applied as:

Listing 6 Apply first distributed load

```
s.apply_load(
    start_x=0,
    start_y=0,
    value=6,
    global_angle=270,
    order=0,
    end_x=4,
    end_y=0)
```

Another distributed load from (4, 0) to (6, 1.5) is applied as:

Listing 7 Apply second distributed load

```
s.apply_load(
    start_x=4,
    start_y=0,
    value=6,
    global_angle=270,
    order=0,
    end_x=6,
    end_y=1.5)
```

Next, we add the support reactions using the **apply_support()** method. This method takes the following parameters:

- x and y: the coordinates of the support.
- type: the type of support ('fixed', 'pin', 'roller').

In the example, we have a fixed support at (11, -1). The **apply_support()** method returns the symbolic reaction forces, these need to be assigned to variables (we will use these in the next step). As our example has a fixed support, 3 reaction loads are returned.

Listing 8 Apply fixed support

```
Rv, Rh, T = s.apply_support(x=11, y=-1, type="fixed")
```

In this code, Rv and Rh represent the vertical and horizontal reaction forces, and T represents the moment reaction at the fixed support. By using the **apply_support()** method, we have added support reactions to our structure.

Let's use the draw method in Listing 9 to visualize what we just added, this time we can set the show_load_values input to True to view the load values.

Listing 9 Draw structure with load values visible

```
s.draw(show_load_values=True)
```

The output should look like Figure 14. We are done with the setup and have input all the information from the example into the structure2d class and are ready to solve.

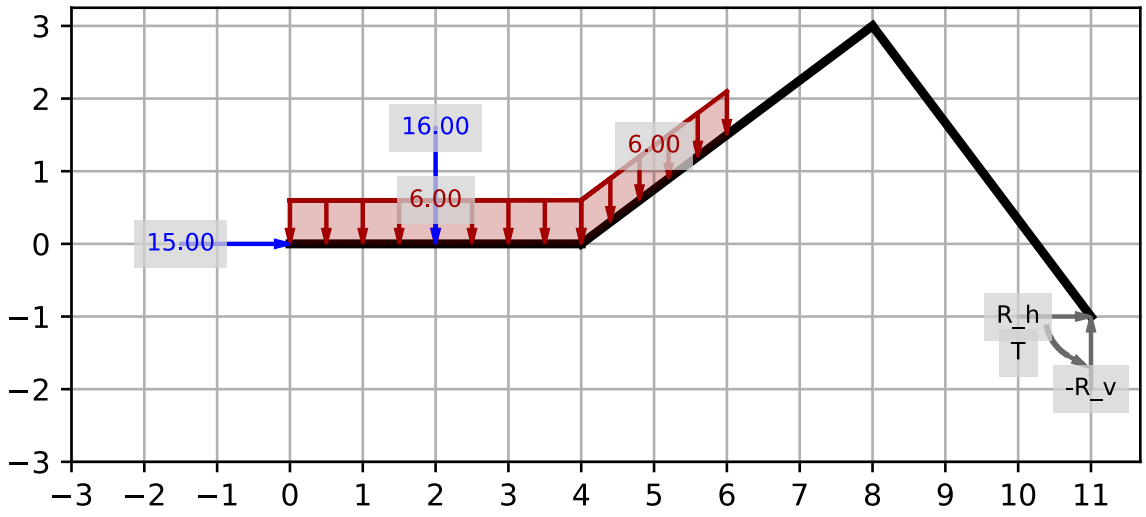

Figure 14 Plot generated by the draw function. (structure with loads and unknown reaction loads)

3.2 Solving reaction loads

To solve the structure, we use the **solve_for_reaction_loads()** method of the Structure2d object. This method calculates the reaction forces at the supports and determines the shear force and bending moment equations along the structure. The method takes the following parameter:

- *reaction_force_variables: Variables representing all reaction loads acting on the structure.

In our example, we have reaction forces Rv, Rh, and T at the fixed support located at (11, -1). We pass these variables to the **solve_for_reaction_loads()** method to solve the structure, as seen in Listing 10.

Listing 10 Solving reaction loads

```
s.solve_for_reaction_loads(Rv, Rh, T)
```

After running this code, the structure is in a solved state and the method will return the reaction loads. This means that the reaction loads at the supports have been determined, and the shear force and bending moment equations for each member have been calculated.

3.3 Output for structure2d

With the structure solved, we can generate outputs to analyze the results. The Structure2D object provides several methods to display the results, including plotting the structure with loads and supports, printing a summary of the reaction forces and internal forces, and plotting the shear force and bending moment diagrams.

First, we can print a summary of the solved reaction loads and internal forces using the summary method, as seen in Listing 11. This method takes an optional parameter round_digits to specify the number of decimal places for rounding, if set to None, it will display exact analytical values. Currently the exact analytical values in some cases are not always computed for the reaction loads resulting in numerical values, this is a known bug.

Listing 11 Generate summary of solved state

```
s.summary()
```

In this summary, as seen in Listing 12:
Reaction Loads: Displays the calculated reaction forces at the supports. The coordinates of the support are given in square brackets, and the unwrapped coordinate along the structure is given in parentheses.

Points of Interest - Bending Moment: Lists the bending moment values at specific points along the structure.

Points of Interest - Shear Force: Lists the shear force values at specific points along the structure.

The location computation for the coordinates on the structure is not yet implemented for bending moments and shar forces, only the unwrapped coordinates are shown. The square brackets currently have place holders.

Listing 12 Summary output

```
==================== Structure Summary ====================
Reaction Loads:
R_v   [11.00,-1.00]  (14.00)            = -55.0000000000000
T     [11.00,-1.00]  (14.00)            = -435
R_h   [11.00,-1.00]  (14.00)            = -15.0000000000000

Points of Interest - Bending Moment:
bending_moment at [x.xx,y.yy]  (0.00)    = 0
bending_moment at [x.xx,y.yy]  (4.00)    = -80
bending_moment at [x.xx,y.yy]  (9.00)    = -330
bending_moment at [x.xx,y.yy]  (14.00)-  = -434.999979000000

Points of Interest - Shear Force:
shear_force at [x.xx,y.yy]  (0.00+)      = 0
shear_force at [x.xx,y.yy]  (2.00-)      = -11.9999940000000
shear_force at [x.xx,y.yy]  (2.00+)      = -28
shear_force at [x.xx,y.yy]  (4.00-)      = -39.9999940000000
shear_force at [x.xx,y.yy]  (4.00+)      = -41
shear_force at [x.xx,y.yy]  (9.00-)      = -53.0000000000000
shear_force at [x.xx,y.yy]  (9.00+)      = -21
shear_force at [x.xx,y.yy]  (14.00-)     = -21.0000000000000
```

Next, we can redraw the structure as seen in Listing 13, now numerical values of the reaction loads are displayed. The result will look like Figure 15.

Listing 13 Draw result including solved support reactions

```
s.draw(show_load_values=True)
```

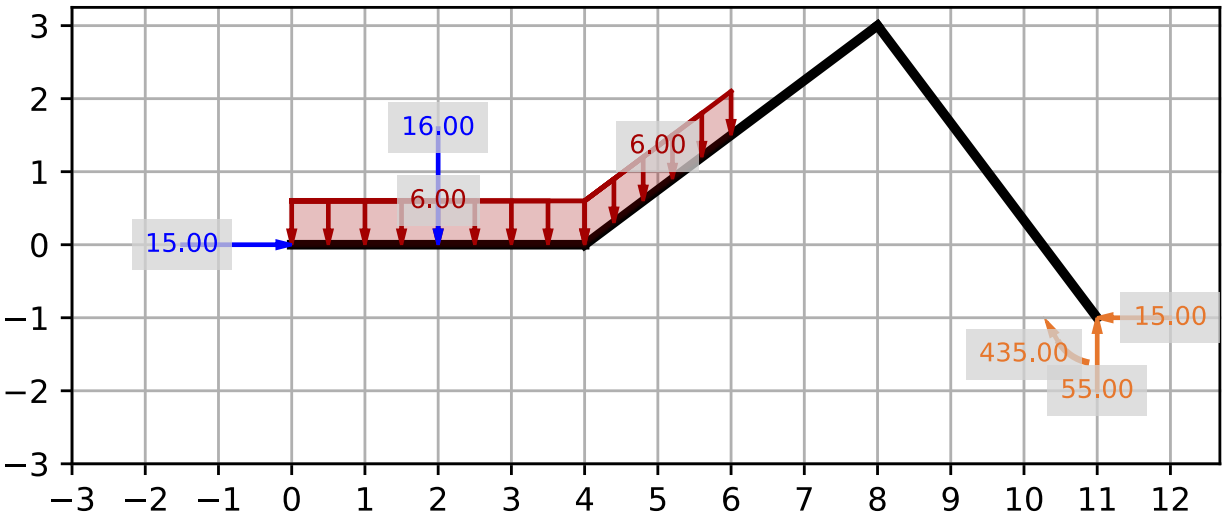

Figure 15 Plot generated by the draw function. (solved structure)

We can also plot the shear force and bending moment diagrams using the **plot_shear_force()** and **plot_bending_moment()** methods, as seen in Listing 14.

Listing 14 Plot shear force and bending moment diagrams

```
s.plot_shear_force()
s.plot_bending_moment()
```

These methods generate plots of the shear force and bending moment along the length of the unwrapped structure, this can be seen in Figure 15. Note that due to current limitations, these diagrams are plotted along the unwrapped coordinate of the structure and are not overlaid onto the original two-dimensional geometry, this could be done another possible future enhancement for the module.

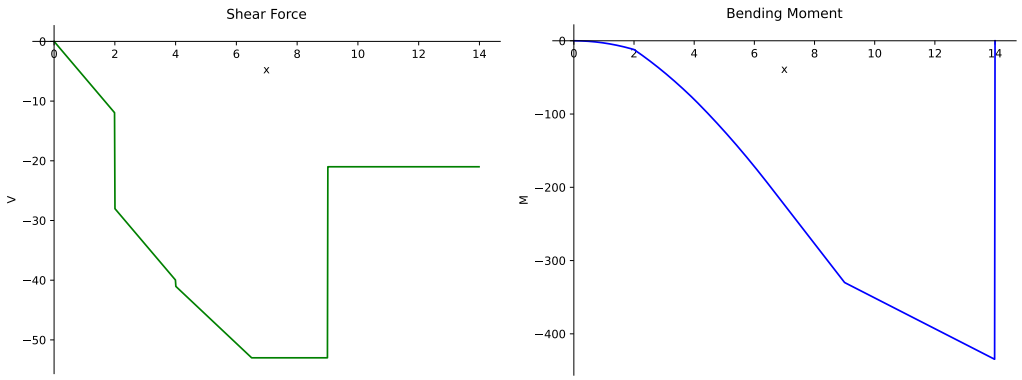

Figure 16 Solved plots for internal forces Left: shear force graph, Right: bending moment graph

Additionally, we can evaluate the shear force and bending moment at any point along the structure using the **shear_force()** and **bending_moment()** methods. These methods accept either an x coordinate or both x and y coordinates, as seen in Listing 15.

Listing 15 Shear force and bending moment values at (2,0)

```
print(s.shear_force(2,0))
print(s.bending_moment(2,0))
```

In this example, we evaluate the shear force and bending moment at the point (2, 0) on the structure. Due to current limitations, when using both x and y coordinates, the methods do not support evaluating values just before or just after a point where discontinuities may occur (e.g., at points of applied loads). This is important because shear forces can exhibit sudden jumps at these locations. As a workaround, if only the x coordinate is provided, the evaluation is performed along the unwrapped coordinate of the structure. This allows us to evaluate the values just before or just after a discontinuity by using a very small value dx as seen in Listing 16.

Listing 16 Display shear force values just before a jump in the graph

```
dx = 1e-6
print(s.shear_force(2 - dx))
print(s.shear_force(2 + dx))
```

4. Conclusion

This project successfully demonstrated the possibility of developing a symbolic 2D structural analysis tool in Python using Macaulay's method as the mathematical foundation. The Structure2D module extends the existing SymPy Beam module, allowing it to handle 2D problems. It manages the complexity of 2D structural analysis by unwrapping the structure into a 1D equivalent, allowing for the separation and independent analysis of vertical and horizontal load effects.

Through the development of the Structure2D module, the project leveraged the existing capabilities of the SymPy library, particularly the Beam module, to extend its functionality into two dimensions without duplicating code. By introducing new classes such as Member, Node, and Load, the tool effectively manages data and interactions between user inputs and computational functions. The unwrapping process transforms complex 2D structures into equivalent 1D representations, simplifying the analysis while preserving the integrity of the original structure. This tool offers significant educational value by providing a hands-on approach to learning structural analysis. It bridges the gap between theoretical concepts and practical application, allowing students and engineers to visualize and compute internal forces and deflections in 2D structures symbolically.

The project's success also highlights the potential of open-source collaboration in advancing engineering tools. By making the code available on GitHub, it invites contributions from the community, inspiring further development and refinement of the tool. This collaborative approach can lead to the creation of new features and enhancements.

In conclusion, the successful implementation of this proof of concept provides engineers and students with an symbolic approach to understanding structural behavior in two dimensions. It validates the central hypothesis that symbolic computation and Macaulay's method can be effectively combined to analyze 2D structures offering exact solutions and deeper insights into how loads and supports influence internal forces and deflections.

5. Discussion

While the project achieved its primary goal of developing a symbolic 2D structural analysis tool, several limitations were identified that present possibilities for future improvement. These limitations are detailed in Appendix C - Limitations of the Proof of Concept, but a few of the important ones are described here.

One significant limitation of the current implementation is the absence of a Column module, which would handle horizontal load effects and normal forces. Currently, the tool focuses only on vertical loads and bending moments, excluding horizontal components necessary for computing deflections. A new Column module could add a secondary solver to address these effects which would also resolve the current limitation of only one unknown horizontal support reaction. Addressing this limitation would be an ideal first focus for future development, as it is essential to expanding the tool's functionality. Another limitation lies in the handling of complex structural geometries. The tool currently supports only non-branching, non-intersecting structures with members added in order. This restriction limits the analysis to simpler structures and excludes more complex frameworks like truss structures. Developing a more advanced unwrapping algorithm capable of handling branching and intersecting members would significantly increase the number of problems that could be solved with the tool.

The assumption of uniform material properties across all members is another area for improvement. In practical engineering scenarios, structures often consist of members with varying elastic moduli, cross-sectional areas, and moments of inertia. However currently the beam module does not support different moments of inertia for beam sections. This needs to be developed first before the Sturecture2D module is upgraded and is therefore a lower priority.

By addressing these areas, the tool can evolve beyond a proof of concept into a robust analytical resource for structural engineering students and educators. The synergy between classical engineering methods and modern computational techniques demonstrated in this project shows potential in educational environments.

References

Baudoin, A. (2024). *Continue Macaulay methode voor geknikte, vertakte en gesloten constructies*. Retrieved from repository.tudelft: https://repository.tudelft.nl/record/uuid:f921c7d0-dde9-4d3c-9572-dd8462e9df4a

Macaulay, W. (1919). A note on the deflection of beams. *Messenger of Mathematics*, 48:129–130.

Meurer A, S. C. (2017). *doi.org*. Retrieved from Sympy: https://doi.org/10.7717/peerj-cs.103

Saheli, B. (2024). *Sympy-structure2d-fork*. Retrieved from GitHub: https://github.com/BorekSaheli/sympy/tree/structure2d

Woudenberg, T. R. (2024). *Extension to Macaulay's method @ TU Delft*. Retrieved from Teachbooks.github.io: https://teachbooks.github.io/Macaulays_method/intro.html

Woudenberg, T. R. (2024). *Macaulay-2D*. Retrieved from GitHub: https://github.com/Tom-van-Woudenberg/Macaulay-2D/blob/plots/BEP%20v1.ipynb

Appendix A - Example from Chapter 3

This is the example as one complete snippet of code that is used in Chapter 3, feel free to paste this in your IDE to try it out.

```
from sympy.physics.continuum_mechanics.structure2d import Structure2d

%config InlineBackend.figure_format = 'svg'

E = 3e4
I = 1
A = 1e4

s = Structure2d()
s.add_member(x1=0, y1=0, x2=4 , y2=0 , E=E, I=I, A=A)
s.add_member(x1=4, y1=0, x2=8 , y2=3 , E=E, I=I, A=A)
s.add_member(x1=8, y1=3, x2=11, y2=-1, E=E, I=I, A=A)

Rv, Rh, T = s.apply_support(x=11, y=-1, type="fixed")

s.apply_load(start_x=0, start_y=0, value=15, global_angle=0   , order=-1)
s.apply_load(start_x=2, start_y=0, value=16, global_angle=270, order=-1)

s.apply_load(
    start_x=0,
    start_y=0,
    value=6,
    global_angle=270,
    order=0,
    end_x=4,
    end_y=0,
)

s.apply_load(
    start_x=4,
    start_y=0,
    value=6,
    global_angle=270,
    order=0,
    end_x=6,
    end_y=1.5,
)
s.solve_for_reaction_loads(Rv, Rh, T)

s.summary(round_digits=None)

s.draw(show_load_values=True)
```

Appendix B - Overview of the Existing Beam Module

In Figure 17 the functionality of the Beam module is detailed. The module creates a beam class where loads can be applied using prebuilt load configurations. For instance, when a distributed load (order 0 or higher) is added, the function automatically manages the inverse equation to turn the load off after a specified length. Another example is when a rotational hinge is applied the code automatically introduces the necessary boundary conditions and loads with the appropriate order. After all loads are added an equation for the entire beam is assembled after which the equation simply needs to be integrated to get the equations for shear force bending moment and deflection.

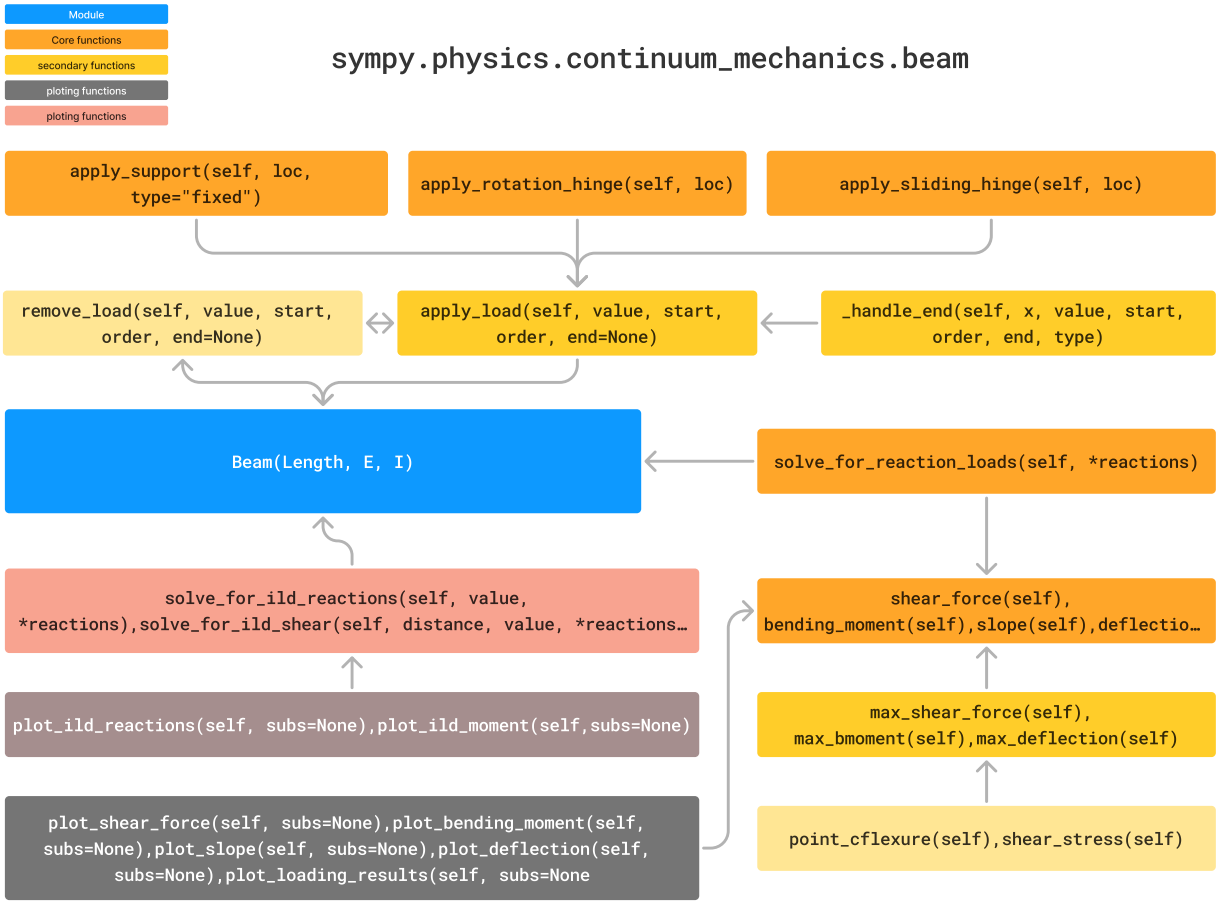

Figure 17 Overview of the Existing Beam module

Appendix C - Limitations of the Proof of Concept

Due to time constraints, several design choices have been made to limit the scope of the project and ensure the development of a functional proof of concept. These limitations include the following:

- **Sign Convention**: A consistent sign convention is applied to all forces and reactions. The positive direction for the x-axis is to the right, and for the y-axis, it is downward.

  Reason: This is the standard convention used at Technische Universiteit Delft. Implementing a more general sign convention would require multiplying equations by negative values in various places where angles and directions are computed. Additionally, the drawing function automatically flips signs, which would necessitate an in-depth analysis of each equation to determine how to adjust the sign.

- **Column Module**: The column module was not implemented.

  Reason: Developing this module would require building an entirely new system that functions similarly to the existing beam module. Since the principles and architecture used in the structure2D module could be applied to the beam module, building the column module is unnecessary for this stage as it would add little conceptual benefit.

- **Analysis Scope**: The analysis considers only bending moments and shear forces, excluding other factors.

  Reason: This decision is a direct result of choosing not to implement the column module, which would have enabled the consideration of other forces.

- **Non-Branching Structures**: The tool supports only non-branching or non-intersecting structures. Each member must be connected to only one other member at each end or remain unconnected. Members are added sequentially, following the unwrapping order of the structure from left to right or right to left.

  Reason: Branching requires "cutting" the structure into pieces before unwrapping. Implementing an algorithm to effectively do this is challenging, as it involves searching a tree structure and determining how to minimize the number of cuts. The current implementation is faster and simpler, assembling all members in order, end-to-end.

- **Uniform Material Properties**: All members must have the same values for elastic modulus (*E*), cross-sectional area (*A*), and moment of inertia (*I*).

  Reason: The current code tracks only the coordinates of members, not their individual properties. Adding support for varied material properties would require significant changes in how members are tracked.

- **Support Restrictions**: Supports can have only one unknown reaction force in the horizontal direction.

  Reason: The current solver is based on the beam module, which does not include a solver for horizontal forces. To work around this limitation, the tool allows only one free horizontal support reaction, making use of Newton's third law ($\Sigma F = -\Sigma F$) to solve for a simple equilibrium.