Extending Continuum mechanics module: Introducing classes for Cables and Improving the Truss class

Ishan Pandhare

Google Summer Of Code 2023 Proposal

Sympy

Abstract

As a part of the project for this summer, I would like to complete the following tasks:

- Improving methods for **Truss** class
- Introducing **Cables** class
    1. Defining the cables class
    2. Create a solver for Cable subjected to concentrated load
    3. Create a solver for Cable subjected to uniformly distributed load
    4. Introducing a draw method for the class
    5. Plotting tension developed in each segment due to the load(s)
- Stretch goal: Adding functions to plot **Influence Line diagrams** for **Truss**

Table of contents:

About me

Basic Information

Name: **ISHAN PANDHARE**
Email: ishan9096137017@gmail.com
GitHub: Ishanned
University: INDIAN INSTITUTE OF TECHNOLOGY (BHU), VARANASI
Degree: Integrated dual degree (B.tech + M.tech)
Time zone: IST (UTC +5:30)

Personal Background

I am a third-year undergraduate student pursuing a dual degree (B.Tech + M.Tech) in **Mathematics and Computing** at the Indian Institute of Technology (BHU), Varanasi. I started programming three years ago and I am proficient in several languages: Python, C, C++, Java, JavaScript, and R.

As a student of mathematics, I have studied the following subjects:

- Differential equations
- Algebra
- Graph Theory
- Probability, statistics, and Mathematical modeling
- Discrete mathematics
- Numerical techniques (involving topics such as the regula-falsi method, Newton-Rhapson method etc.)
- Mathematical methods (involving various transformation techniques including Laplace, Fourier, Hankel etc.)
- Complex analysis
- Number Theory

In addition to these, I have also studied Engineering Mechanics, Material Science, and Thermodynamics.

Programming Details

1. I work on Ubuntu Linux operating system, with VS code as my primary text editor
2. I have been programming in C, C++, and Python for over 3 years now.
3. I am well versed with using Git as version control.
4. Some of my previous projects are available on my GitHub account. These include a website for finding the weather details of a given location (using JavaScript and EJS) and using Machine learning algorithms to detect diabetes (using binary classification).
5. I have been a user of Sympy for over a year now, and have been contributing to it for nearly 8 months now. I have always admired the functionalities that are served by

Sympy, with my favorite being integration of various functions using Sympy. This feature has helped me calculate Picard's successive approximations, an otherwise tiring iterative method, in a matter of seconds. Following is the code I wrote:

In [17]:
```
import sympy as smp
from sympy import Dummy
```

In [18]:
```
def picard_solver(x_0,y_0, exp, it_count):
    phi= smp.symbols("phi")
    x=Dummy("x")
    phi=x_0

    for i in range(it_count+1):
        phi=y_0+smp.integrate(exp(x,phi),(x,x_0,x))
    return phi
```

In [19]:
```
x,y= smp.symbols("x y")
```

In [20]:
```
exp=lambda x,y:x+y
```

In [21]:
```
z=picard_solver(0,0,exp,5)
```

In [22]:
```
z
```

Out[22]:
$$\frac{x^7}{5040} + \frac{x^6}{720} + \frac{x^5}{120} + \frac{x^4}{24} + \frac{x^3}{6} + \frac{x^2}{2}$$

I was able to optimize the above piece of code thanks to the help of @smichr and the Sympy community.

- ## Motivation

I started using Sympy to model some questions in my academic work that would require large iterations to solve. Then I was introduced to the continuum mechanics module, which helped me solve questions for my Engineering Mechanics course.

This submodule can be extended to include structures such as cables and arches used extensively in engineering domains.

I have read the following books to study continuum mechanics:

- **Engineering Mechanics** by Irving H. Shames
- **Mechanics of Materials** by Dr. B.C. Punmia
- **Structural Analysis** by R.C. Hibbeler

Contributions to Sympy

The following are my contributions to Sympy:

- ## PRs
  1. #23900: Added sphinx endpoint for hadamard_product function (merged)
  2. #24030: Fix str of sum involving negative quantities (merged)
  3. #24194: replaced some broken reference links in doc with working ones (merged)
  4. #24219: updated trignometric.py to fix incorrect result for cos function (merged)
  5. #24371: rewrites for trigonometric functions as Besselj and gamma functions (closed)
  6. #24387: added rewrite for trigonometric functions as Besselj
  7. #24665: factor_list() collects factors from partially factored input

The Project

- ## Brief Overview

  The project aims at Extending the continuum mechanics module. In the past few years, this module has had commendable development and improvement, including adding various structures and related functionalities. The current state of the module proves to be helpful to solve many engineering problems, with the help of well-implemented classes for beams and trusses.

  There are still improvements to be made in the continuum mechanics module. After a discussion with past GSoC mentees and mentors, I have decided to base my project to implement the following ideas:

  - Improving methods for **Truss** class
  - Introducing **Cables** class
    a. Defining the cables class
    b. Create a solver for Cable subjected to concentrated load
    c. Create a solver for Cable subjected to uniformly distributed load
    d. Introducing a draw method for the class
    e. Plotting tension developed in each segment due to the load(s)
  - **Stretch goal**: Adding functions to plot **Influence Line diagrams** for **Truss**

  Considering the scope of the project, I would like to propose this as a **large** project (350 hours).

- ## Execution Plan

  I will be implementing the proposed project in four phases, excluding the community bonding period. Each phase will have a set of goals to be achieved, listed as follows:

1. **Phase 1:**
   In this phase, I will be working on improving the methods of Truss class. This would allow the user to relatively larger data with a single call of the method, rather than repeatedly calling methods such as `add_node`.

2. **Phase 2:**
   In this phase, I will be introducing the **Cable** class to the module. This will also include a solver for a cable subjected to concentrated load as well as a cable subjected to uniformly distributed load.

3. **Phase 3:**
   This phase will focus on the 'representation' part. I would be working on the 'draw' method to represent the cable and its state, while also working on a function to plot the tension developed in each segment of the cable.

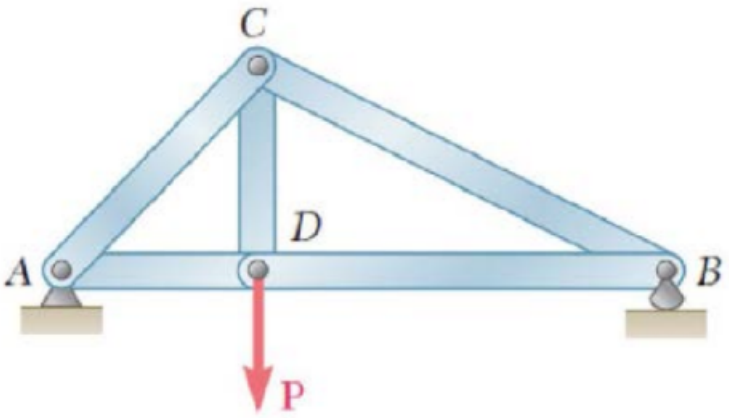
4. **Phase 4 (stretched phase):**
   This phase would be to achieve the stretch goals mentioned. This time would be utilised (if available) to develop a function to plot an influence line diagram for Truss class.

- ## Community bonding period

  I will start discussing the project details with the mentor assigned during this period. Since I have been contributing to Sympy for some time now, I have had conversations with many developers. If the discussions with the mentor are completed early, I would like to start to code (implementing phase 1) during the community bonding period itself.

- ## Phase 1

  I would like to start this project by working on improving the existing class first. Truss is an important structure from an engineering perspective. Sympy's Truss is a well implemented class, although some changes to it can improve user experience.
  In this phase, I would be working on improving the methods of the truss class. Currently, if a user wants to add more than one node at the same time, then the user has no other option but to call the `add_node` method multiple times. This is the case with some other methods as well.
  Consider the following truss:



To add the four nodes, the user will have to do the following:

```
>>> t = Truss()

>>> t.add_node("A", 0, 0)
>>> t.add_node("B", 3, 0)
>>> t.add_node("C", 1, 1)
>>> t.add_node("D", 1, 0)
```

This is a rather tiring job for the user, especially when the number of nodes is large.

To reduce these efforts, the method(s) can be modified in the following way:

```
def add_node(self, *args):

        for i in args:
            label = i[0]
            x = i[1]
            x = sympify(x)
            y = i[2]
            y = sympify(y)

            if label in self._node_labels:
                raise ValueError("Node needs to have a unique label")

            elif x in self._node_position_x and y in
self._node_position_y and
self._node_position_x.index(x)==self._node_position_y.index(y):
                raise ValueError("A node already exists at the given
position")

            else :
                self._nodes.append((label, x, y))
                self._node_labels.append(label)
                self._node_positions.append((x, y))
                self._node_position_x.append(x)
                self._node_position_y.append(y)
```

The above code is a modification of Advait Pote's work. Such tweaks could also be made to other methods as well, such as `remove_nodes`, `add_member`, `remove_member`, etc.

The above code can also be modified to add labels automatically. Say, the user only passes label 'A' for the first node, then the method should automatically assign 'B', 'C', and 'D' nodes to the subsequent nodes. This would save the user some time to assign each node a label, since generally the labels are assigned in lexicographic fashion. Of course, if the user wishes to add the labels manually, then the user would be welcome to do so.

This idea could be implemented by using the keyword argument (**kwargs).

Now the user would be able to add these nodes with a single call:

```
>>> t = Truss()
>>> t.add_node(("A", 0, 0), ("B", 3, 0), ("C", 1, 1), ("D", 1, 0))

  # or if the user wants to assign the labels automatically,
>>> t = Truss()
>>> t.add_node(label='A', (0,0), (3,0), (1,1), (1,0))
```

Along with these changes, I would also like to add some more functionalities to the Truss class. Since in a truss, it is seen that a closed polygon is formed. Consider the above-given truss, where a closed polygon is formed by A, D, B, and C.

Even after modifying the `add_member` method, the user would have to provide four 2-tuple as the input. For a relatively larger truss, it would be tiring to add members this way, when clearly it can be seen that some kind of polygon will be formed by the edge members.

To tackle this, I would like to introduce a new method `add_member_cyclic`, which would take labels as arguments, and add members between the nodes in a cyclic manner. This function would tackle the need for tuples as arguments. The following script is a rough implementation of the idea:

```
    def add_members_cyclic(self, *args):

      if(len(args)<2):
           return error

      for i in range(len(args)-1):
          add_member(args[i], args[(i+1)%len(args)])
```

The above code can be modified to **not** give an error if there already exists a member between two nodes and go about making the polygon. The above code is just one part of the method that I would like to implement. I would also be automating the labels, just like the idea proposed for the 'add_node' method.

Now, instead of the following procedure

```
>>> t.add_member("member_1", "A", "C")
>>> t.add_member("member_2", "A", "D")
```

```
>>> t.add_member("member_3", "B", "C")
>>> t.add_member("member_4", "B", "D")
```

The user can call a single method

```
>>> add_members_cyclic("A", "C", "B", "D")
```

The labels ‘member_1’, ‘member_2’, etc. would be assigned automatically here (after a little modification to the above code). If the user wants to manually assign the labels, again, we would welcome the user to do so.

- # Phase 2
  - **a) Introducing the Cable class**

    The cable class will be introduced here. This will enable users to initialize a cable object, and define its properties such as its length, the points of the hinge, loads attached, etc.

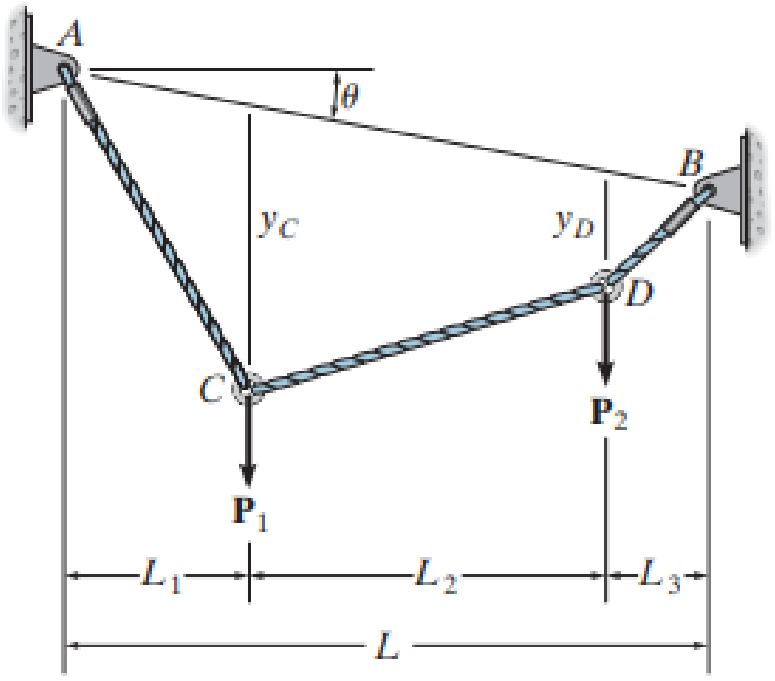
    A sample code for the idea would be on the given lines:

```
class Cable:
    def __init__(self, **kwargs):
        self._length = kwargs["length"]
        self._left_support = kwargs["left_support"]
        self._right_support = kwargs["right_support"]
        self._loads_magnitude = []
        self._loads_position_x = []
        self._loads_position_y = []
        self._loads_dir = []
        self._dist_load = []
        self._dist_load_dir = []


    @property
    def length(self): #an example of class method
        return self._length

    # On similar grounds, we define methods for other properties
    # as well

    # function to add load
    def add_conc_load(self, x, y, magnitude, dir):

        self._loads_magnitude.append(magnitude)
        self._loads_dir.append(dir)
        self._loads_position_x.append(x)
        self._loads_position_y.append(y)

    def add_dist_load(self, magnitude, dir):
```

```
        self._dist_load.append(magnitude)
        self._dist_load_dir.append(dir)

    # In the same fashion other, other methods such as
    # remove load can also be added
```

The length property would depict the length of the cable, the left and right support would be the positions of the support.
The loads list would define the loads applied to the cable, the direction of the load and the position of it.
This class would be taking inspiration from classes that are already implemented, i.e., the truss class and the beam class.

After the implementation of the Cable class, we would be able to achieve something whose physical significance would correspond to the following image:



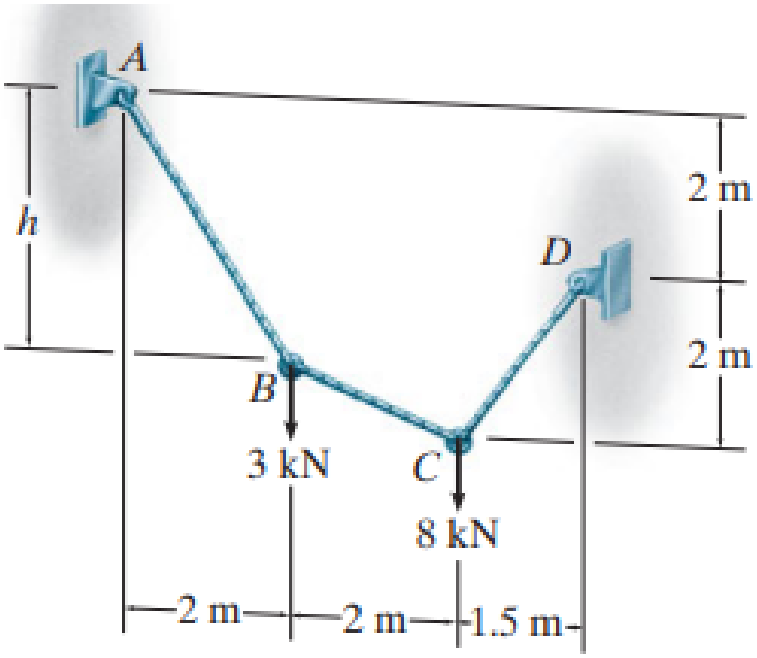
b) Developing a solver for a cable subjected to concentrated load

Taking reference of the image given above, If the distances L_1, L_2 and L_3 and the loads P_1 and P_2 and are known, then the problem is to determine the nine unknowns consisting of the tension in each of the three segments, the four components of reaction at A and B, and the sags y_c and y_d and at the two points C and D.

For the scope of this project, I would be focusing on creating a solver in which the loads are applied to specified positions. The aim of this solver would be to compute the tension that is developed in each segment of the cable and to compute the reaction forces at the support. This solver would also be able to raise an error if the position of any load would stretch out the cable, based on the initial assumption that

the cable is inextensible (and the user provides the length of the cable beforehand). Since the sags would now be known, the unknowns would now reduce and hence less computation would be required to achieve our goal.
Based on the above implementation idea, it would not be necessary for the user to specify the length of the cable. The cable length property could then be made optional and can be calculated as another unknown.

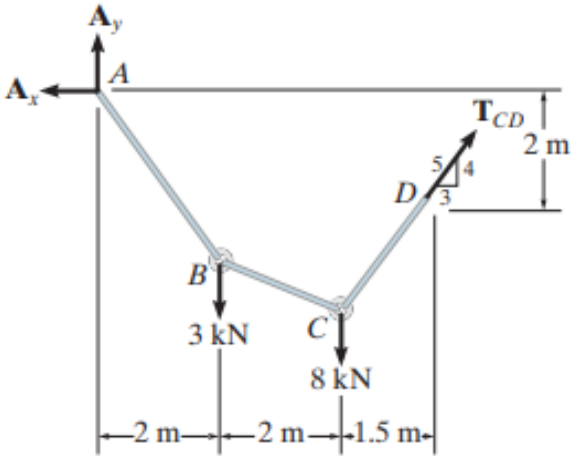
Consider the following example to get a better picture of the implementation:



Here, the value of h is specified to be 2.74m.

The unknowns here would be the Tensions in the 3 segments- AB, BC and CD, the reaction forces at supports A and B, and the length of the cable (if not specified beforehand).

The free-body diagram for the cable would be



We can use equilibrium equations

$$\Sigma F_x = 0, \Sigma F_y = 0$$

on points A, B, C, and D and deduce our eight unknowns.

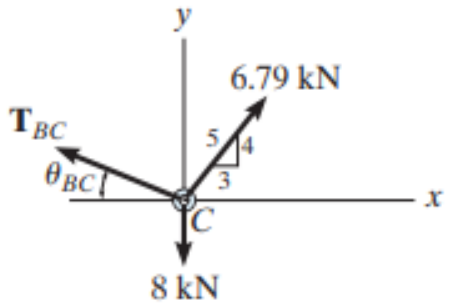
Tension in segment CD can be obtained as follows:

$$\downarrow + \Sigma M_A = 0;$$

$$T_{CD}(3/5)(2\text{ m}) + T_{CD}(4/5)(5.5\text{ m}) - 3\text{ kN}(2\text{ m}) - 8\text{ kN}(4\text{ m}) = 0$$

$$T_{CD} = 6.79\text{ kN}$$

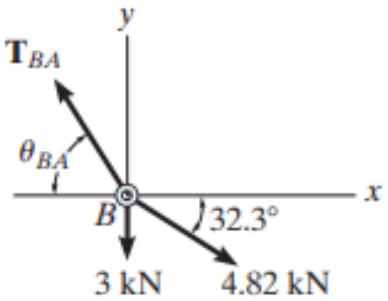
Applying equations of equilibrium to C and B,



$$\overset{+}{\rightarrow} \Sigma F_x = 0; \quad 6.79\text{ kN}(3/5) - T_{BC}\cos\theta_{BC} = 0$$

$$+\uparrow \Sigma F_y = 0; \quad 6.79\text{ kN}(4/5) - 8\text{ kN} + T_{BC}\sin\theta_{BC} = 0$$

$$\theta_{BC} = 32.3^\circ \qquad T_{BC} = 4.82\text{ kN}$$



$$\overset{+}{\rightarrow} \Sigma F_x = 0; \quad -T_{BA}\cos\theta_{BA} + 4.82\text{ kN}\cos 32.3^\circ = 0$$

$$+\uparrow \Sigma F_y = 0; \quad T_{BA}\sin\theta_{BA} - 4.82\text{ kN}\sin 32.3^\circ - 3\text{ kN} = 0$$

$$\theta_{BA} = 53.8^\circ \qquad T_{BA} = 6.90\text{ kN}$$

Using a similar way, we can compute the reaction forces at A and D as well.
Another approach of solving this question would be to

Make 8 equilibrium equations (2 for each A, B, C and D)

This would help to find out the 8 unknowns we would be solving for.
These equations could be solved using Sympy's solver.

The **implementation** of this procedure would include:

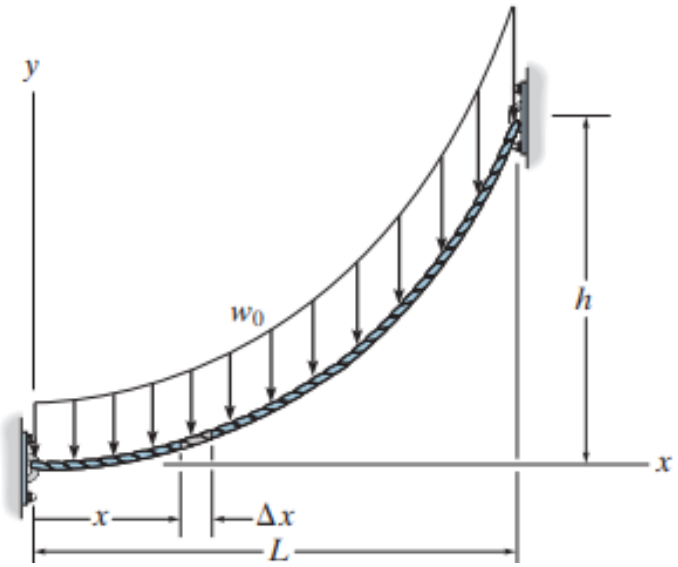
a. Calculating the angles
   For this, a helper function could be developed that would calculate the angles formed between the cable and support and between two cable segments.
b. For every load and the point of loading,
   Form two equilibrium equations (also applicable to the points of support). This would require the outputs from (a) part.
c. Solving these equations, it can be seen that we would have enough equations to find these variables even if we increase the number of loads.
d. The reaction forces can also be calculated using equilibrium equations at the supports.
e. If the length is not specified, it can be calculated using Pythagoras' theorem. If the specified length is not equal to the one that is computed, an error can be raised.

Following is a rough implementation of the "solving" part. This code will also be worked on to give out errors and better efficiency.

```
def solve_for_conc_load(self):

        tension = []
        # calculating first tension using equilibrium equation
        # moment = 0 for right support
        t=0
        for i in range(len(self._loads_magnitude)):
            t = t +
self._loads_magnitude[i]*sin(self._loads_dir[i])*(self._loads_pos
ition_x[i] - self._right_support)
            +
self._loads_magnitude[i]*cos(self._loads_dir[i])*(self._loads_pos
ition_y[i] - self._right_support)

        t =
t/(sin(a)*(-self._left_support[0]+self._right_support[0]) +
        cos(a)*(self._right_support[1]-self._left_support[1]))

        tension.append(t)

        # now that the tension in first segment is known,
        # we can use sum of Fx=0 and Fy=0 to find tension
        # in each segment

        for i in range(len(self._loads_magnitude)):
            # each successive tension can be found using the
previous
            # known tension (tension[i]) and applying the
equations of equilibrium
            #
```

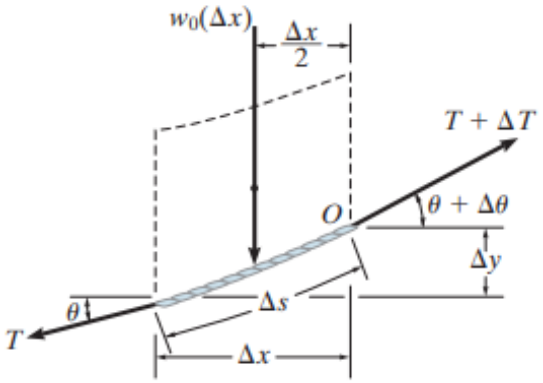
```
            tension.append(t)

        return tension
```

c) Developing a solver for a cable subjected to uniform distributed load



We will first determine the shape of a cable subjected to a uniform horizontally distributed vertical load

The free-body diagram of a small segment of the cable having a length Δs is shown:



Using the equations of equilibrium,

$$\stackrel{+}{\rightarrow} \Sigma F_x = 0; \quad -T\cos\theta + (T + \Delta T)\cos(\theta + \Delta\theta) = 0$$
$$+\uparrow \Sigma F_y = 0; \quad -T\sin\theta - w_0(\Delta x) + (T + \Delta T)\sin(\theta + \Delta\theta) = 0$$
$$\circlearrowleft + \Sigma M_O = 0; \quad w_0(\Delta x)(\Delta x/2) - T\cos\theta\,\Delta y + T\sin\theta\,\Delta x = 0$$

Dividing these equations by Δx and taking limit,

$$\frac{d(T\cos\theta)}{dx} = 0$$

$$\frac{d(T\sin\theta)}{dx} = w_0$$

$$\frac{dy}{dx} = \tan\theta$$

Integrating the first and second equation with the corresponding boundary condition,

$$T\cos\theta = F_H$$

$$T\sin\theta = w_0 x$$

Where F_H is a constant.

Which leads to

$$\tan\theta = \frac{dy}{dx} = \frac{w_0 x}{F_H}$$

Integrating the above equation,

$$y = \frac{w_0}{2F_H}x^2$$

The constant can be found using the boundary condition y=h and x=L,

$$y = \frac{h}{L^2}x^2$$

Using this procedure, we can find the shape of the cable and the tension as well.

Implementation:

a. Unlike the above-discussed example, the left support would not be at origin, rather it would be defined by the user. The first task would be to deduce the shape of the cable, which is seen to be a parabola.
   Since the length of the cable will be defined, we can use the arc-length formula

$$L = \int_a^b \sqrt{1 + f'(x)^2}\,\mathrm{d}x$$

   This along with the coordinates of the left and right support, will give us three equations to find the equation of the parabola,

$$y = ax^2 + bx + c$$

b. Alternatively, if the user specifies the coordinates of the lowest point, we can use the information to form the equation of the parabola. Similar to the previous solver, we can raise an error if the calculated length and the specified length do not match.
c. Subsequently, once the shape of the cable is known, we can find the other unknowns using the derived formulae. Since these are in reference to a parabola that passes through the origin, we can use shifting of origin to solve them.

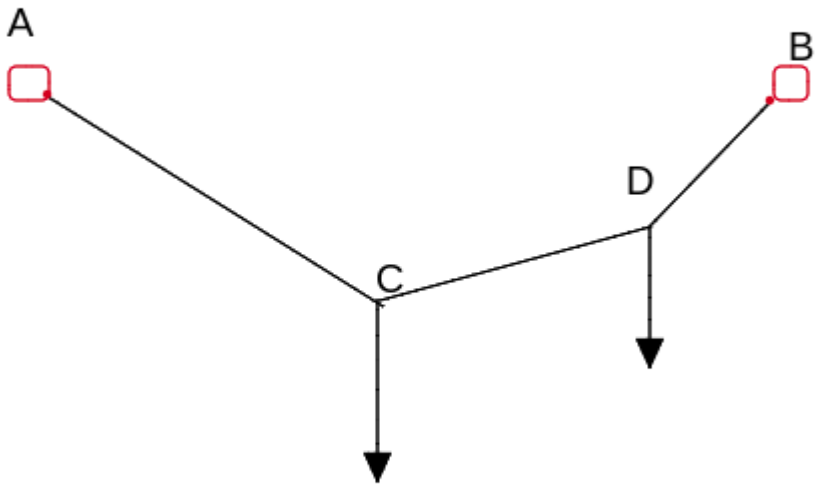
Here is a rough implementation of the above mentioned idea

```
def solve_for_dist_load(self):

        # first, we find the equation of the parabola
        system = Matrix(( (self._left_support[0]**2,
self._left_support[0], 1, self._left_support[1]),
        (self._right_support[0]**2, self._right_support[0], 1,
self._right_support[1]),
        self._lowest_point[0]**2, self._lowest_point[0], 1,
self._lowest_point[1]))

        solve_linear_system(system, a, b, c)

        # now we have the equation y = ax**2 + bx + c

        # to find the tension, we shift the origin to
self._lowest_point
        # hence the new equation would be y = ax**2, where a =
w/2Fh

        # tension at any given point = Fh/cos(theta)
        theta = atan(w*x/Fh)
        t = Fh/cos(theta)

        # the solver would return tension as a function of
position
        return t
```

- **Phase 3**

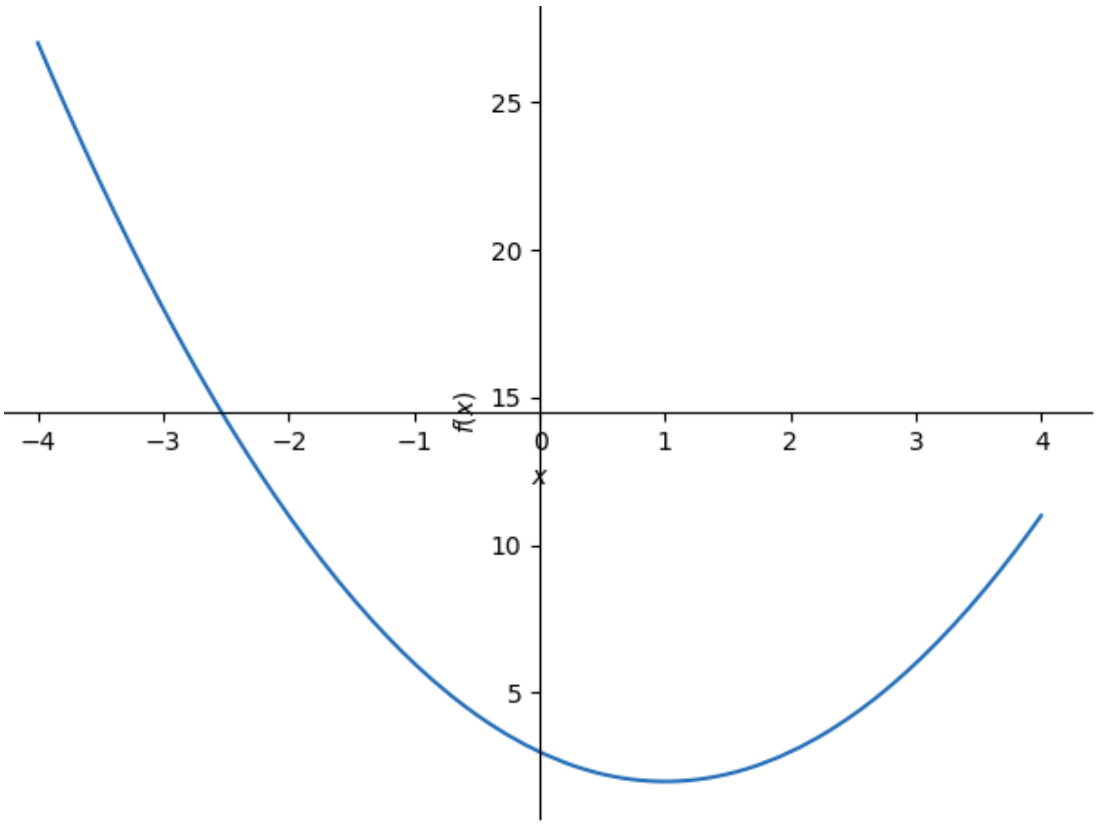
d) Introducing a draw method for the class

The draw method for cable class will be based on the previous work by Advait Pote during last year's GSoC. This method would return a plot object that would be depicting the state of the cable, the positions of the loads, position of supports, etc. This method would make use of various objects of sympy.plotting.



The above image is the expected result when a cable is subjected to concentrated load(s).

For cable subjected to distributed load, the result is expected to be as follows:

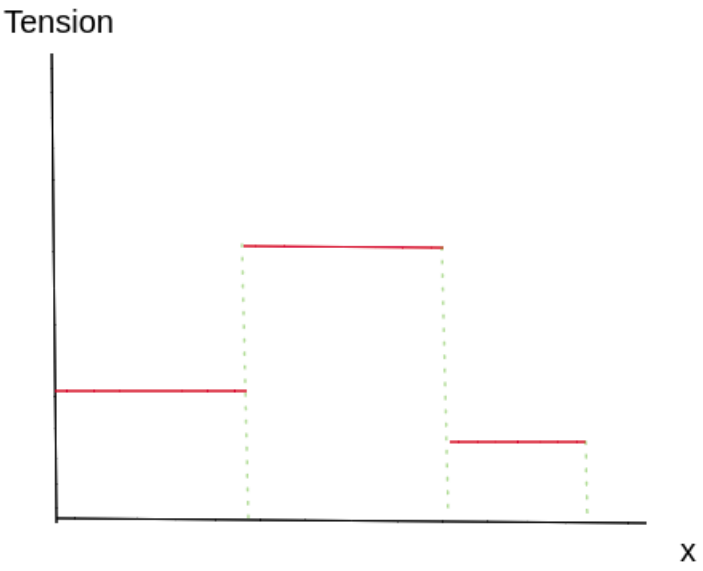


This plot was made using sympy.plotting. There are improvements to be made to this, such as adding geometry, proper legends for supports and depiction of load.

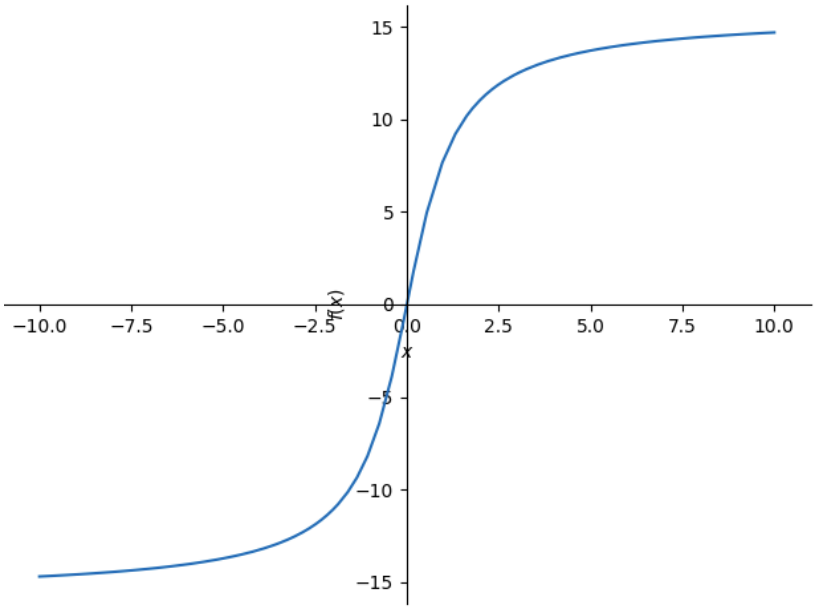
e) Plotting tension developed in each segment

In this part, I would be developing functions within the class to plot the tensions developed in the cable.

For concentrated load, the tension plot would look like following



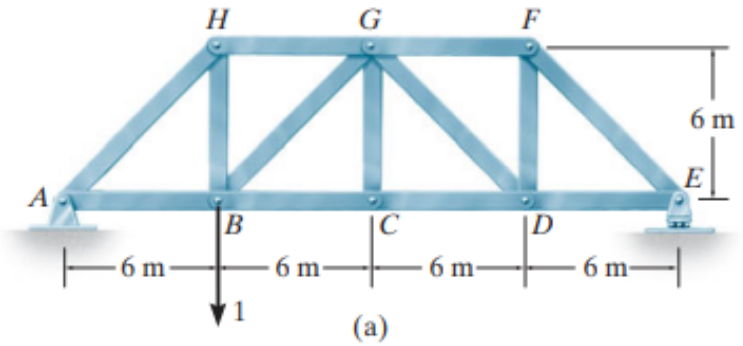
For uniform distributed load, the plot for tension would be the following:



- ## Stretch goal: Phase 4

  In this phase, I would be developing a function to plot the influence line diagram for a truss. Influence lines have important applications for the design of structures that resist large live loads. In the case of truss, these lines can tell what force is developed in a particular member. This can provide engineers and material scientists an insight as to how the truss could be constructed and what material could be used.

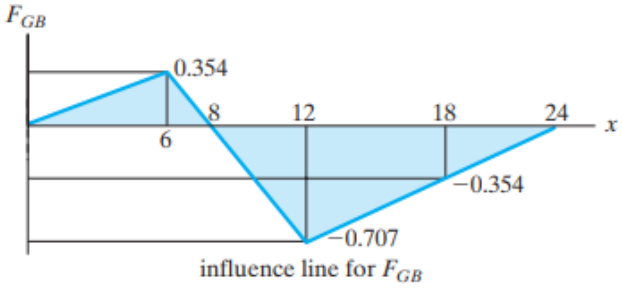
  Consider the given truss:



  The influence line for the force in member GB can be calculated in the following manner:
  Each successive joint at the bottom cord is loaded with a unit load and the force in member GB is calculated.
  The tabulated form of the computation would be:

x	F_{GB}
0	0
6	0.354
12	−0.707
18	−0.354
24	0

The corresponding plot would be:



influence line for F_{GB}

Implementation:

1) The user can be asked to create a truss of their choice.
2) Nodes at the bottom joint can be identified
3) Once identified, unit load can be applied to them to identify the force in the member in question. Once the forces are calculated, they can be plotted.

A rough implementation of this idea is presented below

```
def influence_line(self):

        bottom_nodes = []
        member_force = []

        for i in _nodes:
            if(i==node_with_min_x):
                bottom_nodes.append(i)

        for i in bottom_nodes:
            apply_load(i, 1, 270)
            t.solve()
            # we'll now store the values
            member_force.append(force_in_member)
            # the force_in_member can be calculated
            # from the truss solver
            # then remove the load again
            remove_load(i, 1, 270)

        # for plotting
        plot(member_force,
```

```
            (x, 0, max_value), title='Influence Line for force in
member',
            xlabel=x, ylabel=force, line_color='blue', show=False)
```

Alternatively, the data can also be presented in tabular form.

This implementation would take inspiration from Prakhar Saxena's work (1, 2, and 3) in GSoC'21, where influence line diagrams were introduced for beam.

Timeline

- ## Community Bonding Period (May 4-28)

  During this time, I will be discussing the details about the project with the mentors and taking their suggestions. If the discussions related to the project are completed before the assigned time, then I would like to start implementing the project on the lines of the discussion with the mentor

- ## Phase 1 (May 29 - June 11, 2 Weeks)

  The implementation of the first phase of the project would be done during these 2 weeks. The goal would be to implement the methods and improvements discussed in the project section. This should not take much time as the truss class is already implemented.

- ## Phase 2 (June 12 - July 23, 6 weeks)

  Part a) Implementation of Cable class (June 12 - June 25, 2 weeks)

  The class 'Cable' will be implemented during this period, taking inspiration from the Truss and the Beam class. The goal for this period would be the successful implementation of this class.

  Part b) Implementation of the solvers (June 26 - July 23, 4 weeks)

  In this period, I would be developing the solvers, for both- concentrated loads as well as distributed load. Since developing solvers would require adequate testing, this would take the most time of the project.

  By the time of midterm evaluation, which will be during this phase, I will have completed the Phase 1 of my project and implemented the Cable class. I would be working on the solvers during that duration.

- ## Phase 3 (July 24 - August 20, 4 weeks)

  Part a) Implementation of ‘draw’ method (July 24 - August 6, 2 weeks)

  During this time, I would be implementing the ‘draw’ function for the cable class, similar to the ones implemented for truss and beam classes. This would be assigned 2 weeks as it would be using various aspects of plotting and geometry.

  Part b) Implementing the plotting function for tension (August 7 - August 20, 2 weeks)

  This would be the last part of the project (barring stretch goals). Here, I would be implementing a method to plot the tension developed in the string due to the load(s) applied.

- ## Final week (August 21 - August 28)

  During this time I would make sure that all my work is finished and merged. If this is done well before the final week, I would like to start working on the stretch goal.

Other commitments during GSoC

I do not have any other commitments during the summer. I will be through with my end-semester exams in the first week of May, and will have the entire summer vacation dedicated to the project. Vacations would end in the month of August, but there is no major load for at least a month after the classes start. Hence, I would be able to devote 25-30 hours per week.

In case of any disruption, I would make sure to inform my mentor in advance and make up for the work in the time before the break. In case of any unforeseeable happening, I would dedicate extra time in the coming weeks to make up for the lost time.

Post GSoC

After GSoC, I would like to continue contributing to Sympy, fixing and addressing issues that may arise in the continuum_mechanics module, while also addressing PRs and doing code reviews. I would also like to implement my stretch goal in case I am not able to do so during my project duration. I would also try to implement the arch class in the continuum_mechanics module, which is another important structure in engineering.

I would like to thank Christopher Smith(@smichr), Jason Moore(@moorepants), Oscar Benjamin(@oscarbenjamin), Aaron Meurer(@asmeurer), Sam Brockie(@brocksam), Prakhar Saxena(@Psycho-Pirate), Advait Pote(@AdvaitPote), and all the members of this wonderful community for helping me contribute to Sympy and always helping me in required times.

References

1) **Engineering Mechanics** by Irving H. Shames
2) **Mechanics of Materials** by Dr. B.C. Punmia
3) **Structural Analysis** by R.C. Hibbeler
4) Prakhar Saxena's GSoC proposal
5) Advait Pote's GSoC proposal
6) Sympy Documentation for Continuum mechanics