Continuum mechanics: Creating a Rich beam Solver and Extending Continuum Mechanics module

Table of Contents:

About Me

Basic Information

Name: Ishan Anirudh Joshi
Email: ishanaj98@gmail.com
University: Netaji Subhas Institute of Technology, New Delhi
Github: ishanaj
Blog: https://ishanaj.wordpress.com
Time Zone: IST (UTC + 5:30)

Personal Background

I am 3rd year undergraduate pursuing Bachelors in Engineering in Manufacturing Processes and Automation(MPAE) at N.S.I.T, New Delhi.

Programming Details

I work on Windows operating system with *Sublime Text 3* as my primary editor. It's been about more than two years since I started competitive coding. I began coding with C++ and C. I like these languages because of their simplicity and also because I had C++ in my intermediate. Implementation of what we think becomes an easy task in C/C++. I am comfortable with STL and algorithms. The reason I liked and chose Python was because of its flexibility and simplicity. It has been more than a year after I started programming in python and also more than a year contributing in sympy.

Contributions to sympy

PR's(merged/unmerged)

- **(merged)** PR #16276 Added functionality to obtain subplots from already created plot objects. A new class `**PlotGrid**` was introduced in the plotting module which would take some already created plot objects and adjust them in the required grid size of subplot
- **(open)** PR #15951 Added a method to remove support
- **(open)**PR #15775. Added functionality to calculate shear stress.
- **(open)** PR #14583 solving equations if floating point numbers are used
- **(open)**PR #14284 Ability to solve modular equations in solveset(with integer solutions)
- **(open)**PR #14053 solve_trig1 now able to convert hyperbolic-trig to exp.
- **(merged)**PR #14012 solveset is now able to solve XFAIL
- **(open)**PR #13975 Made solveset return CRootof() for unsolvable inequalities
- **(closed)**PR #13913 made solveset to work with eq having Max(),Min()
- **(open)**PR #13669 rs_exp(): prec converted to int before entering for loop
- **(closed)**PR #13391 Making `Eq(True, 1)` type commands evaluate to False
- **(merged)**PR #13035 link added for as_content_primitive() in basic.py and typo-fix

Issues/Bugs Reported

- **(open)**Issue #16572 Problem with plotting discontinuities in the current master.
- **(open)**Issue #16537 Plotting geometric identities using plotting module
- **(open)**Issue #14565 Redundancy in the output of nonlinsolve

The Project

Brief Overview

Major areas of concern:

Stage-1:

- Integrating the geometry module with beam module
  - Defining a cross section in the beam module
  - Calculations of first moment
  - Calculations of second moment
  - Section Modulus
  - I-section and T-section geometry definition and its calculations

Stage-2:
- Implementing Column Buckling and its calculations.
- Plotting graphs related to column buckling

Stage-3:
- `Draw` function: A function to draw the diagram of the beam with its loads using matplotlib. For this, we can take some ideas from here.

Stage-4
- Truss structure implementation and its calculations using method of joints. It won't be a part of beam module, but can be taken under continuum mechanics.

The Implementation Plan

Stage-1: Integrating geometry module with beam module

The basic idea would be to implement a **class `CrossSection`** which would use the geometry module to define a cross section of a beam.
Need for class: Since this cross-section would also be used by the `Column` class and other classes which are to be implemented in the future, therefore a separate class is required outside the beam module.
- The following basic cross-sections need to be defined:
  i) Circular/hollow circular
  ii) Triangular
  iii) Rectangular/ Hollow rectangular
  iv) I-shape
  v) T-shape

- After defining these cross-sections, it needs to return the following information related to the cross-section:
  i) Area
  ii) Second moment
  iii) Centroid
  iv) First moment of area
  v) Section Modulus
  vi) Polar moment of inertia

Out of these the first three are already implemented in the geometry module, while the last three have to be implemented

Proposed API of the class:

```
# API
>>> shape = CrossSection(circle, 10)
>>> shape = CrossSection(rectangle, (width=4, height=6))
>>> shape = CrossSection(I-shape, (5, 2), (2, 5), (6, 2))
# dimensions of upper flange, web and lower flange respectively

>>> shape.area
>>> shape.second_moment
>>> shape.centroid
>>> shape.first_moment
>>> shape.section_modulus
>>> shape.polar_modulus

# class design
Class CrossSection:
    def __init__(self, shape, *args):
        # would inspect the shape and dimensions(args)and give a call to
        # the respective class method.
    # The class methods being:
    def _circle(self)
    def _triangle(self)
    def _rectangle(self)
    ...
    ...
    # and so on for every cross-section.
    # these methods would take the dimensions of the cross section and
    # then use the geometry module to define the shape.
```

For example:
An **I-section** of:
upper/lower flange width = 5
upper/lower flange height = 2
web width = 3
web height = 6, can be defined using geometry module as:

```
# I-section definition using geometry module
>>> from sympy import Polygon
>>>  p = Polygon((0, 0), (5, 0), (5, 2), (4, 2), (4, 8), (5, 8), (5, 10)
     , (0, 10), (0, 8), (1, 8), (1, 2), (0, 2))
>>> p.area
38
>>>  p.second_moment_of_area()
(1142/3, 331/6, 0)
```

Also, a function **`add_cross_section()`** will have to be defined in the beam module which would take the input of the cross-section from the user and then use class `CrossSection` to define a cross-section and get the results

Proposed API of `add_cross_section():

```
>>> b = Beam(l, E, I)
>>> ..
>>> ...
>>> b.add_cross_section(circle, 10)
# `add_cross_section()` would use class CrossSection to calculate
# second_moment, first_moment, polar_modulus and other things and
# and substitute those values in the attributes of the beam.
>>> b.first_moment  # returns first moment
>>> b.polar_moment  # returns polar moment
```

- <u>Calculating **first moment** of area:</u>

$$S_x = A\bar{y}$$

$$S_y = A\bar{x}$$

Where **A** is the area above the point of interest(i.e neutral axis for general cases) and, **y/x** is the distance between the neutral axis of the cross section and the centroid of the area **A**.

- Calculating **section modulus**:

$$Z = \frac{I}{y}$$

I : second moment of the cross section

y : distance of the top most layer from the neutral axis

- Calculating **polar moment**:

$$J = Ixy = Ix + Iy$$

The geometry module is able to calculate `Ixy`. We need to just take it as polar moment.

Later, the way of beam module taking the input could also be changed, where it would be directly taking the cross-section and its dimensions

Kind of like:

```
>>> b = Beam(l, E, rectangle, (5, 7))
# (5, 7) being the width and height resp of the rectangle
```

Stress Calculations using cross section analysis:

With the implementation of the cross-section, it would now be easier to include calculation related to **stress**. PR #15775 has partially implemented the functionality to calculate stress.

The main aim would be to make it use the cross-section to get certain values.

Stage-2: Implementing Column buckling and its calculations

For Column Buckling it would be better to define a separate **class `Column`** as the calculations are very different from that of beam.

The **governing equation** for column buckling is:

$$EI\frac{d^2y}{dx^2} = -M$$

Step-1: To determine the internal moment. This is simply done by assuming deflection at any arbitrary cross section at a distance **`x`** from the bottom as **`y`** and then multiplying this by the load **`P`**.

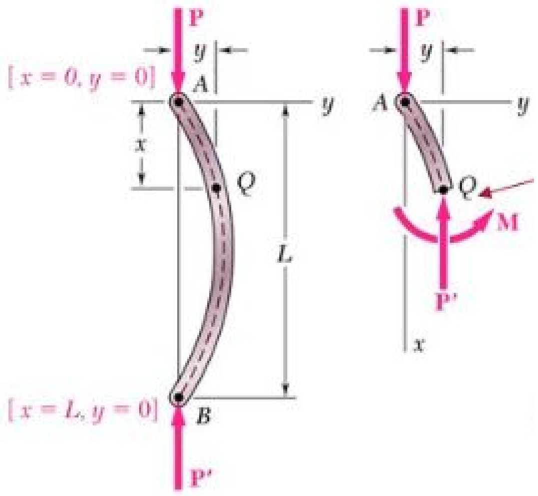
And for eccentric load another moment of magnitude **`P*e`** is added to the moment.

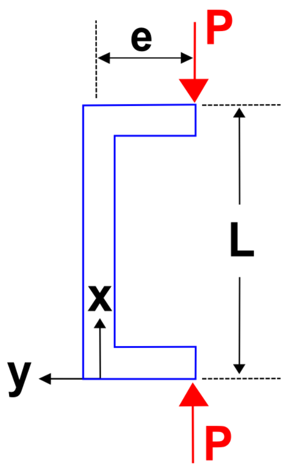
Simple load:

$$M = Py$$

Eccentric load:

$$M = Py + Pe$$





Step-2: This moment can then be substituted in the governing equation and this differential equation can be solved using **dsolve()** for the deflection \`y\`.

The deflection \`**y**\` would be obtained in terms of \`**x**\`

Applying supports:

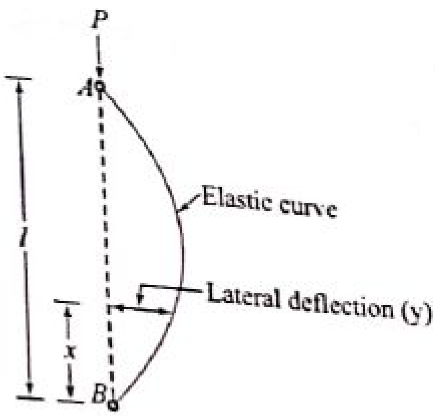
Four basic supports are to be implemented:

Pinned-pinned, fixed-fixed, fixed-pinned, one pinned-other free.

Depending on the supports the moment due to applied load would change as:

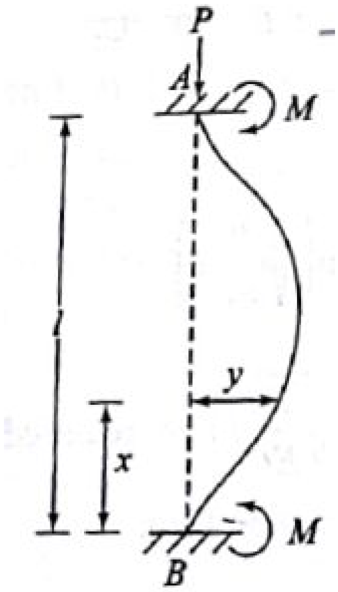
- **Pinned-Pinned:** no change in moment

$$moment = Py$$



- **Fixed-fixed**: reaction moment M is included

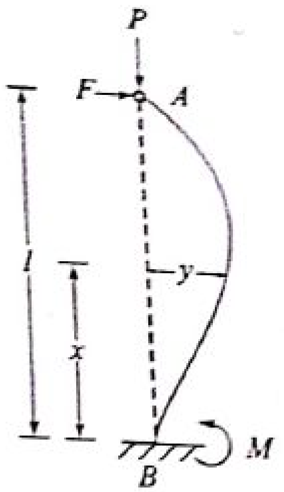
$$moment = Py - M$$



- **Fixed-pinned:**

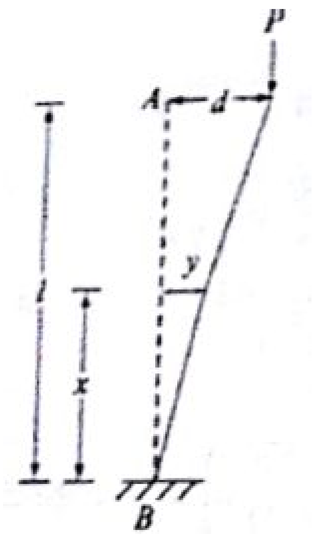
  $$moment = Py - F(l - x))$$

  Here **M** is the restraint moment at **B(**which is fixed**).** To counter this, another moment is considered by applying a horizontal force **F** at point A.



- **One pinned- other free:**

  $$moment = Py - Pd$$



Proposed API:

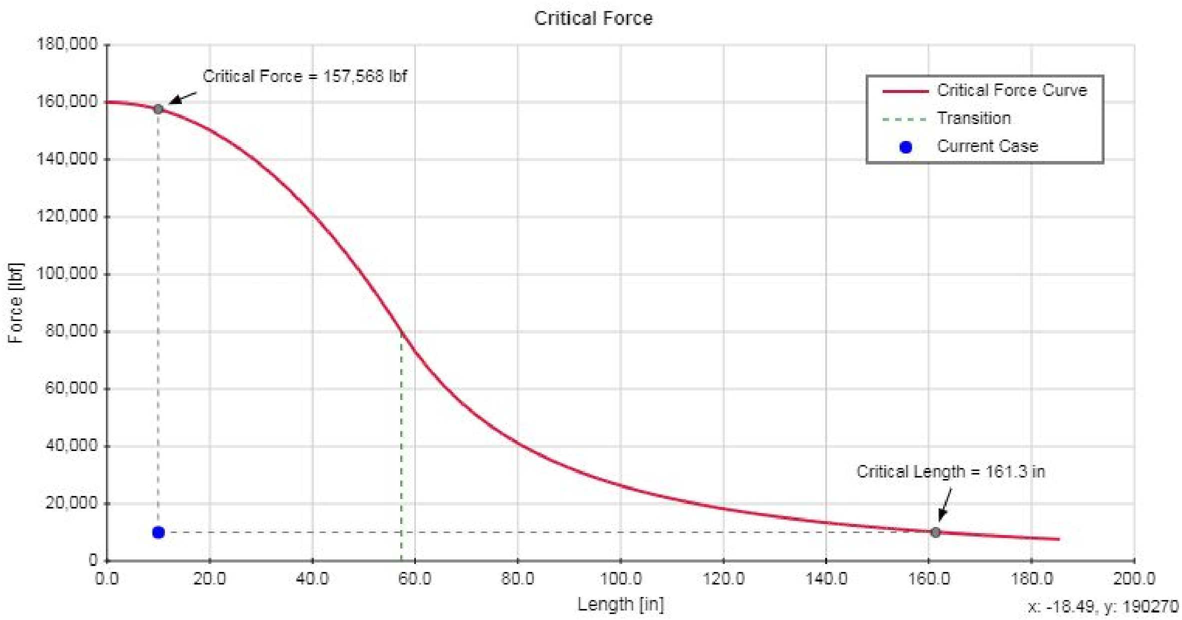
```
>>> c = Column(height = 4, E, I)
>>> c.apply_load(load_magnitude, eccentricity=3)
# would directly create a moment of magnitude `P*y` with P being the load and y
# being deflection at any arbitrary section
# Also would add `P*e` in the moment if the load is eccentric

>>> c.apply_support(top=fixed, bottom=pin)
# would add reaction moment or forces acc. to the type of support and add
boundary conditions.
>>> c.load
>>> c.moment
>>> c.deflection            # can be obtained solving the differential equation
                            # in terms of `x` which is the distance of any
                            # arbitrary cross-section from the bottom
>>> c.slope                 # can be obtained by differentiating the deflection
>>> c.critical_load(fos = 3)       # with the help of boundary conditions
>>> c.critical_length(fos=3)
>>> c.crippling stress()
# A variable 'x' will have to be defined in terms of which the deflection and
the slope are to be calculated
# Class attributes:
  height
  elastic_modulus
  second_moment
  load
  moment
  deflection
  slope
  boundary_conditions
```

Plotting graphs related to column:

- **Critical force vs length:** As the length/height of the column increases the critical force at which a column buckles decreases. This would vary according to the material and the end conditions of the beam. Using sympy's **plot** we can plot this.
- **Deflection vs height:** Since the deflection would be a function of the length, therefore a plot could be obtained.

  A general plot of critical force vs length looks like:



Stage 3: Draw function

The `draw()` function would mainly do the following tasks:

- Draw a rectangle (which is a beam):
- Draw arrows of the load
  - Moment arrow
  - Point load arrow
  - Uniformly distributed load arrow
  - Increasing load arrow
  - Parabolic load arrow
- Draw supports
- Draw x-axis, y-axis
- Draw deflection dotted line
- At the end put the axis 'OFF'

Matplotlib modules/classes used:

- [matplotlib.patches.Rectangle]() -to draw the beam
- [matplotlib.patches.Circle]() - to draw roller support
- [matplotlib.patches.Polygon]() - to draw pin support
- [Matplotlib.pyplot.annotate]() - to draw arrows of load and moment
- [Matplotlib.pyplot.text]()- to add text in the plot

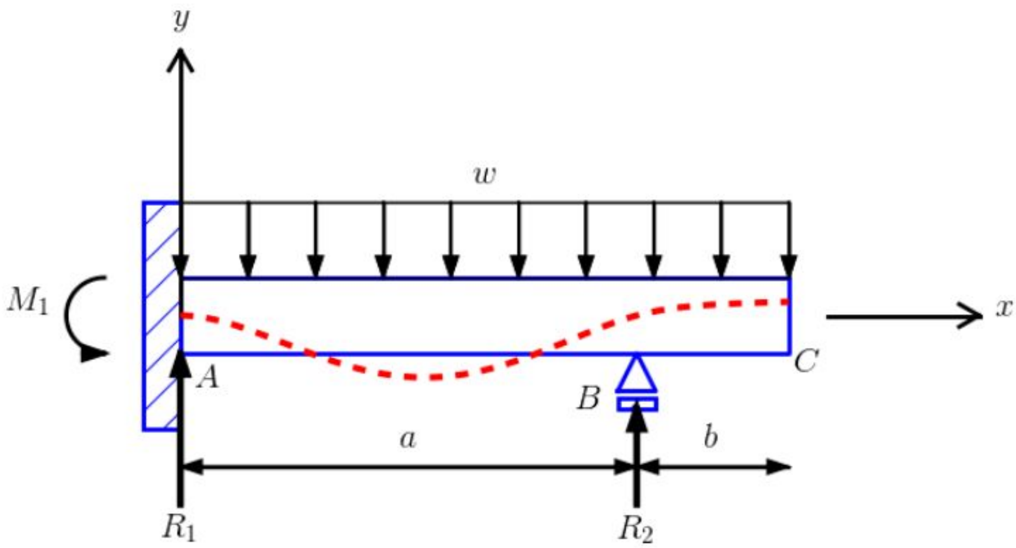
```
>>> b.draw()
# opens a plot with diag of the beam

# function overview
def draw(self):
    length = self.length
    height = self.height # height attribute would be defined by the
                         # `add_cross_section()` method after the
                         # completion of Phase-I.
    fig = plt.figure()
    ax = fig.subplot(111)
    rect = patches.Rectangle((0, 0), length, height)
    ax.add_patch(rect)

    self._draw_arrows()
    self._draw_support() # again would use patches to plot support
    self._draw_deflection_line()
    # and some other functions if required
    plt.axis(‘off’) # to turn off axis

# _draw_arrows() would use annotate() function of matplotlib to draw
# arrows
def _draw_arrows(self):
    loads = self._applied_loads
    for load in loads:
        if load[2] == -2     # checking the order of applied load
            # make moment arrow using annotate()
        if load[2] == -1
            # make point arrow using annotate()
        if load[2] == 0
            # make udl arrow using annotate()
        if load[2] == 1
            # make linearly var load arrow using annotate()
        if load[2] == 2
            # make a parabolic load arrow using annotate()
```

It would draw something like:



Some other hints can be taken from here.

Stage 4: Implementation of Truss Structure and its calculation

Truss: A Truss is a structure that consists of two force members only, where the members are organized so that the assemblage as a whole behaves as a single object

In other words, a truss is an assembly of beams or other elements that creates a rigid structure.

A Truss basically consists of : **a member, and a node**.

Each of the beam/member is under tensile or compressive load which is referred to as **internal force**.

There are basically two methods to analyse a truss:

- Method of joints
- Method of sections

Here, the plan is to analyse the truss using **method of joints**, which analyses the forces on every joint and then determines the internal forces on members and reactions.

Proposed API:

```
>>> t = Truss()
>>> t.add_node("node_1", x_position, y_position)
>>> t.add_member(start_node, end_node)
>>> t.apply_load(node_1, magnitude=10, dir='x') # only point loads
>>> t.apply_moment(node_3, magnitude, dir='z')
>>> t.apply_support(node_4, type="fixed")
>>> t.solve()      # solve for reaction and internal forces
>>> t.reactions
# returns a dict with the values of reactions
```

```
>>> t.internal_forces
# returns a dict of internal forces of each member
```

`**t.solve()**` would calculate the internal forces and reactions using the method of joints.
It would be kind of similar to the purpose current `**solve_for_reaction_loads()**` in the beam module, except for its way of working.
The method of joints basically does a force analysis on every joint.
For each joint, it determines the static equations of equilibrium for it,
i.e.,

$$\sum F_x = 0,$$
$$\sum F_y = 0.$$

Combining equations of all the nodes, we would get a bunch of equations, like:

$$1 : \sum F_x = R_{1x} + F_3 \cdot \frac{1}{\sqrt{2}} = 0$$
$$1 : \sum F_y = R_{1y} - F_1 - F_3 \cdot \frac{1}{\sqrt{2}} = 0$$
$$2 : \sum F_x = R_{2x} + F_2 = 0$$
$$2 : \sum F_y = F_1 = 0$$
$$3 : \sum F_x = -F_2 - F_3 \cdot \frac{1}{\sqrt{2}} = 0$$
$$3 : \sum F_y = F_3 \cdot \frac{1}{\sqrt{2}} - 100N = 0$$
$$\Rightarrow F_3 \cdot \frac{1}{\sqrt{2}} = 100N$$

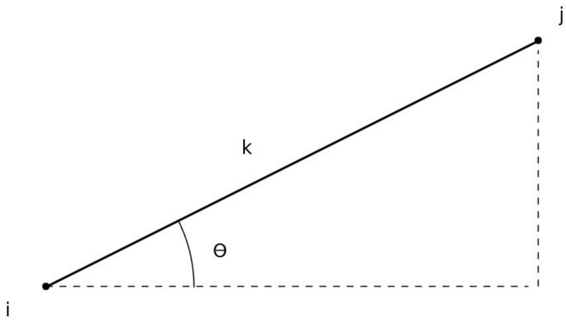
These equations could be collectively represented in matrix form, similar to: $C \cdot F = P$
The matrix would look something like:

$$\begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & 1 & 0 & 0 \\ -1 & 0 & -\frac{1}{\sqrt{2}} & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ R_{1x} \\ R_{1y} \\ R_{2x} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 100 \end{bmatrix}$$

Now the solution can be determined by taking the inverse of C, as: $F = C^{-1} \cdot P$

The main part here would be to form the coefficient matrix `C` which can be easily formed as described here:

Determining the coefficient matrix `C`:
Assuming an arbitrary member `**k**` in a truss connected between two nodes `**i**` and `**j**`.



The equation of the nodes `**i**` and `**j**` are:

$$i : \sum F_x, \quad i : \sum F_y, \quad j : \sum F_x, \quad j : \sum F_y$$

The member `**k**` only contributes to these four equations.

Numbering these equations as **2i-1, 2i, 2j-1, 2j**
So taking the angle:

$$l_{ij} = cos\theta = \frac{\Delta x}{L_k}$$

$$m_{ij} = sin\theta = \frac{\Delta y}{L_k}$$

In the matrix the entire column `**k**` will be zero except the following rows:

$$\begin{aligned} C_{2i-1,k} &= l_{ij} \\ C_{2i,k} &= m_{ij} \\ C_{2j-1,k} &= -l_{ij} \\ C_{2j,k} &= -m_{ij} \end{aligned}$$

This can be done for every member and thus coefficient matrix `**C**` and thus the coefficient matrix can be made.

Timeline

The following timeline aligns with what has been mentioned in the Implementation plan.

Community Bonding Period (May-6 - May 26)

- In this period I will try to plan out the things the way it needs to be implemented by discussing with the mentors.
- I will be studying Sympy's codebase thoroughly and try to use its advantage in the implementation to the fullest.
- Major part would be studying the geometry module in detail, which would help in the first stage of the project.

Week 1, 2, 3 (May 27 to June 16)

- Integrating geometry module as planned in stage-1
- Adding documentation and necessary examples
- Implementing stress calculations

Week (4, 5, 6) (June 17 - July 7)

- Implementing column buckling as discussed in stage-2
- Adding documentation, tests and necessary examples.
- Creating functions to plot necessary plots related to column buckling

Week (7, 8, 9) (July 8 - July 28)

- Implementing the draw function as discussed in stage-3
- Testing various outputs
- Adding documentation and necessary examples

Week (10, 11, 12) (July 29 - August 18)

- Implementing Truss structure analysis as discussed in stage-4
- Adding documentation and necessary examples
- Adding necessary tests

Week 13(August 19- August 26)

- Implementing what is left.
- Submitting final evaluations

Any Plans/Commitment (During GSoC)

- I have no major commitments for this summer and I am positive that I will be able to contribute for about 40-50 hours a week for the project. This project at will form the core of all my working and learning throughout the Summer.
- I will be maintaining a weekly Blog at https://ishanaj.wordpress.com which I have made especially for the purpose of GSoC.
- College will restart in 3rd week of August. In 1st month of the semester, we don’t have any exams so I can concentrate on the project during these days and also for this month I have put relatively less work, in the timeline.

Post GSoC

- If I stuck somewhere in implementation, I will try to complete them after the GSoC period. I will continue my contribution and will be active in SymPy community also will help new contributors.
- If possible, I want to contribute to SymEngine.

References:

- Sampad Saha GSoC Application
- Jashanpreet singh GSoC Application
- Column Buckling and calculations
- Sketching beam diagram
- Truss analysis using method of joints
- Cross Section analysis of Beam
- Calculation of stresses and first moment of area