Google
Summer of Code

Project on Creating a Rich Beam Solving System and Extending Continuum mechanics module

Organisation:

SymPy

Report by: Ishan Anirudh Joshi

Mentors: Jashanpreet Singh Sraw

Yathartha Joshi

Acknowledgement

I would like to say that it was a great pleasure & privilege to have got the opportunity of undertaking this project. I would also like to thank the organisation SymPy and Google Summer of Code to provide me with such an opportunity.

I would like to express my deep and sincere gratitude to my mentors: **Jashanpreet Singh Sraw, Jason K. Moore and Yathartha Joshi** for providing invaluable guidance throughout the project period.

These 12 weeks of the project gave me a good insight of continuum mechanics. I got to learn a lot about **beam bending, column buckling, the importance of symbolics in physics, and also about the development workflow of a software.**

I am sure that the knowledge and experience gained during this period will be of immense value for my growth in the field of Manufacturing and Automation Engineering.

Regards

Ishan Anirudh Joshi

Table of contents

Overview

SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python.

This project aims to improve and expand the existing continuum mechanics submodule within the physics module of SymPy.

It is divided into 3 phases with 4 weeks of time intended for each phase.

Phase I: Integrating geometry module with the Existing beam module

Phase II: Implementing a sub-module Column in the existing continuum mechanics module.

Phase III: Plotting beam diagrams.

Major additions

- #17001- Added a new method cut_section() which would return a tuple of new polygons which lie above and below the given line that intersects it.
- #17055- Made Beam class accept cross-section geometries.
- #17153- Added functionality to determine polar second moment of area, first moment of area and section modulus of a polygon.
- #17122- Introduced a class Column which provides functionality for column buckling calculations.
- #17345- A method to plot beam diagram using sympy's own plot().

Closed Pull Requests

- #16964- A new class CrossSection has been introduced in the continuum_mechanics module.
- #17240- Added functionality in the beam module to draw beam diagrams via a draw() function.

Discussions and issues

- #17072- [Discussion]: Column buckling calculations in continuum mechanics module
- #17162- Problem with solve in handling some trigonometric equations.

Phase I: Integrating geometry module with the Existing beam module

Defining a new attribute cross_section in the beam class.

The beam class takes in the input of the beam from the user in a **piecewise** form. It defines a new beam with the help of three basic **inputs**:

- **Length** of the beam (l)
- **Second moment** of area of the cross-section (I)
- **Elastic modulus** of the material of the beam (E)

The loads and the supports can be added to the beam using the class methods **apply_load(), apply_support(),** one by one.

This is how a basic definition of a beam object looks like:

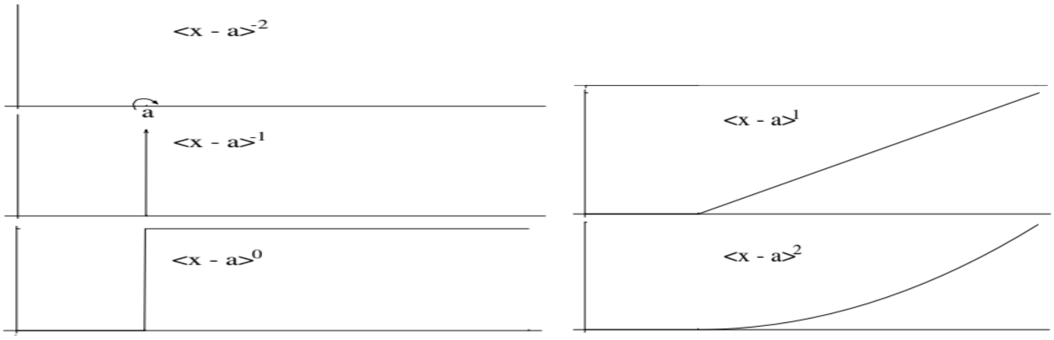
```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(5, 2, -1)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 9*SingularityFunction(x, 2, -1)
>>> b.applied_loads
[(-3, 0, -2, None), (4, 2, -1, None), (5, 2, -1, None)]
```

The beam module uses **singularity function** to solve for different equations (bending moment equation, slope equation, deflection equation) of the beam.

About singularity function: A singularity function is defined as:

$$f(x) \equiv < x - a >^n = \begin{cases} (x-a)^n & x \geq a, \\ 0 & x < a. \end{cases} \qquad \int_{-\infty}^{x} < x - a >^n dx = \begin{cases} < x - a >^{-1} & n = -2, \\ < x - a >^{0} & n = -1, \\ \frac{< x - a >^{n+1}}{n+1} & n \geq 0. \end{cases}$$

The **moment loads (order = -2), point loads (order = -1), uniformly distributed loads (order = 0), uniformly variable load (order = 1), parabolic loads (order = 2)** are depicted below:



The second moment previously was being explicitly entered by the user. The main aim here was to make the beam class **capable calculating the second moment** on its own. The **geometry** module of SymPy is quite efficient in dealing with geometries and their properties. Therefore, the idea was to give the user the option to make a geometry of their own and pass it as an argument to the beam object during its declaration.

A simple solution was to replace the existing **second_moment** attribute with a new **cross_section** attribute and give the user the option to either pass an object of the geometry module or directly enter the second moment. But this change might lead to **backwards incompatibility**.

Therefore, the **second_moment** attribute was kept in its place and now the user had the option of entering a geometry object as the cross-section of the beam.

The changes look like this.

```
-        self.second_moment = second_moment
+        if isinstance(second_moment, GeometryEntity):
+            self.cross_section = second_moment
+        else:
+            self.cross_section = None
+            self.second_moment = second_moment
```

```
    @second_moment.setter
    def second_moment(self, i):
-       self._second_moment = sympify(i)
+       self._cross_section = None
+       if isinstance(i, GeometryEntity):
+           raise ValueError("To update cross-section geometry use `cross_section` attribute")
+       else:
+           self._second_moment = sympify(i)
+
```

```
+    @property
+    def cross_section(self):
+        """Cross-section of the beam"""
+        return self._cross_section
+
+    @cross_section.setter
+    def cross_section(self, s):
+        if s:
+            self._second_moment = s.second_moment_of_area()[0]
+        self._cross_section = s
```

All these changes were done in the Pull request [PR#17055](PR#17055).

Defining new functions in the geometry module for determining polar second moment of area and section modulus of a cross-section.

After enabling the beam class to accept cross-sectional geometries, the second step was to make some additional functions in the geometry module which would help in determining the polar second moment of area, section modulus of a cross-section.

Polar second moment of area: It is a constituent of the second moment of area, linked through the perpendicular axis theorem. While the planar second moment of area describes an object's resistance to deflection (bending) when subjected to a force applied to a plane parallel to the central axis, the polar second moment of area describes an object's resistance to deflection when subjected to a moment applied in a plane perpendicular to the object's central axis (i.e. parallel to the cross-section)

Here J is the polar second moment of area:

$$J = I_z = I_x + I_y$$

The geometry module is able to calculate Ix, Iy, therefore we need to just add them to get our result.

Section modulus: Section modulus is a geometric property of an ellipse defined as the ratio of second moment of area to the distance of the extreme end of the ellipse from the centroidal axis.

Here Z is the section modulus:

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

The code for polar second moment of area and section modulus has been written in **PR # 17153**.
Tests and the documentation have been added in the same.
This Pull request has been successfully completed and merged now.

Code implementation of polar second moment of area (PR #17153)

```
+    def polar_second_moment_of_area(self):
+        """Returns the polar second moment of area of an Ellipse
+
+        It is a constituent of the second moment of area, linked through
+        the perpendicular axis theorem. While the planar second moment of
+        area describes an object's resistance to deflection (bending) when
+        subjected to a force applied to a plane parallel to the central
+        axis, the polar second moment of area describes an object's
+        resistance to deflection when subjected to a moment applied in a
+        plane perpendicular to the object's central axis (i.e. parallel to
+        the cross-section)
+
+        References
+        ==========
+
+        https://en.wikipedia.org/wiki/Polar_moment_of_inertia
+
+        Examples
+        ========
+
+        >>> from sympy import symbols, Circle, Ellipse
+        >>> c = Circle((5, 5), 4)
+        >>> c.polar_second_moment_of_area()
+        128*pi
+        >>> a, b = symbols('a, b')
+        >>> e = Ellipse((0, 0), a, b)
+        >>> e.polar_second_moment_of_area()
+        pi*a**3*b/4 + pi*a*b**3/4
+        """
+        second_moment = self.second_moment_of_area()
+        return second_moment[0] + second_moment[1]
```

Code implementation of section modulus (PR #17153)

```diff
+    def section_modulus(self, point=None):
+        """Returns a tuple with the section modulus of an ellipse
+
+        Section modulus is a geometric property of an ellipse defined as the
+        ratio of second moment of area to the distance of the extreme end of
+        the ellipse from the centroidal axis.
+
+        References
+        ==========
+
+        https://en.wikipedia.org/wiki/Section_modulus
+
+        Parameters
+        ==========
+
+        point : Point, two-tuple of sympifyable objects, or None(default=None)
+            point is the point at which section modulus is to be found.
+            If "point=None" section modulus will be calculated for the
+            point farthest from the centroidal axis of the ellipse.
+
```

```diff
+        x_c, y_c = self.center
+        if point is None:
+            # taking x and y as maximum distances from centroid
+            x_min, y_min, x_max, y_max = self.bounds
+            y = max(y_c - y_min, y_max - y_c)
+            x = max(x_c - x_min, x_max - x_c)
+        else:
+            # taking x and y as distances of the given point from the center
+            y = point.y - y_c
+            x = point.x - x_c
+
+        second_moment = self.second_moment_of_area()
+        S_x = second_moment[0]/y
+        S_y = second_moment[1]/x
+
+        return S_x, S_y
```

Defining a function in the geometry module to determine the first moment of area

The implementation was of the first moment of area was a bit tricky as no proper formula was being found.

First moment of area: First moment of area is a measure of the distribution of the area of a polygon in relation to an axis. The first moment of area of the entire polygon about its own centroid is always zero.

Therefore, it is calculated for an area, above or below a certain point of interest that makes up a smaller portion of the polygon. This area is bounded by the point of interest and the extreme end (top or bottom) of the polygon. The first moment for this area is then determined about the centroidal axis of the initial polygon.

The major part was to determine the area of the half above or below the centroidal axis. There wasn't any way to do that. This lead to the birth of a **new function** named **'cut_section()'** in the geometry module which **returned two new polygons** separated by a line cutting a given polygon.

The basic idea of the function **cut_section()** a **polygon clipping algorithm**, but is unique in its own way.

Working of cut_section():

Considering a polygon intersected by a line and we need to determine the part or the segment of the polygon that lies above it.

The algorithm works in two steps:

- Determining the vertices (or points) that lie above the given line.
- Adding those point to the list of the vertices of the new polygon segment.

How to determine whether a point lies above or below a line?

If we assume the equation of the line to be **ax + by + c** and the point to be checked be **(x1, y1)**, then:

- if **(ax1 + by1 + c)/b = 0** implies that the point lies on the line.

- if **(ax1 + by1 + c)/b > 0** implies that the point lies above the line.
- if **(ax1 + by1 + c)/b < 0** implies that the point lies below the line.

Adding the points to the list of the vertices of the new polygon segment.

For a point which lies above the line has two cases:

- **Whether the previous point also lies above the line:** If this is the case then it means we have to directly add that point to the new list.
- **Whether the previous point lies below the line:** Here it means that the polygon edge (b/w the current and the previous point) is moving upwards and intersecting the line, hence in such a case we have to add the intersection point first and then add the current point to the list.

For a point which lies below the line has again two cases:

- **Whether the previous point also lies below the line:** In such a case we will be ignoring the point and no action will be taken
- **Whether the previous point lies above the line:** It means the polygon edge (b/w the current and the previous point) is moving downwards and intersecting the line, hence in such a case only the intersection point is included.

The implementation of this algorithm has been proposed in PR #17001

Example use of cut_section():

```
>>> from sympy import Point, Symbol, Polygon, Line
>>> a, b = 20, 10
>>> p1, p2, p3, p4 = [(0, b), (0, 0), (a, 0), (a, b)]
>>> rectangle = Polygon(p1, p2, p3, p4)
>>> t = rectangle.cut_section(Line((0, 5), slope=0))
>>> t
(Polygon(Point2D(0, 10), Point2D(0, 5), Point2D(20, 5), Point2D(20, 10)),
Polygon(Point2D(0, 5), Point2D(0, 0), Point2D(20, 0), Point2D(20, 5)))
>>> upper_segment, lower_segment = t
>>> upper_segment.area
100
>>> upper_segment.centroid
Point2D(10, 15/2)
>>> lower_segment.centroid
Point2D(10, 5/2)
"""
```

Code Implementation of cut_section() is done in PR #17001

```
+        intersection_points = self.intersection(line)
+        if not intersection_points:
+            raise ValueError("This line does not intersect the polygon")
+
+        points = self.vertices
+        points.append(points[0])
+
+        x, y = symbols('x, y', real=True, cls=Dummy)
+        eq = line.equation(x, y)
+
+        # considering equation of line to be `ax +by + c`
+        a = eq.coeff(x)
+        b = eq.coeff(y)
+
+        upper_vertices = []
+        lower_vertices = []
+        # prev is true when previous point is above the line
+        prev = True
+        prev_point = None
+        for point in points:
+            # when coefficient of y is 0, right side of the line is
+            # considered
+            compare = eq.subs({x: point.x, y: point.y})/b if b \
+                    else eq.subs(x, point.x)/a
+
+            # if point lies above line
+            if compare > 0:
+                if not prev:
+                    # if previous point lies below the line, the intersection
+                    # point of the polygon egde and the line has to be included
+                    edge = Line(point, prev_point)
+                    new_point = edge.intersection(line)
+                    upper_vertices.append(new_point[0])
+                    lower_vertices.append(new_point[0])
+
+                upper_vertices.append(point)
+                prev = True
+            else:
+                if prev and prev_point:
+                    edge = Line(point, prev_point)
+                    new_point = edge.intersection(line)
+                    upper_vertices.append(new_point[0])
+                    lower_vertices.append(new_point[0])
+                lower_vertices.append(point)
+                prev = False
+            prev_point = point
+
+        upper_polygon, lower_polygon = None, None
+        if upper_vertices and isinstance(Polygon(*upper_vertices), Polygon):
+            upper_polygon = Polygon(*upper_vertices)
+        if lower_vertices and isinstance(Polygon(*lower_vertices), Polygon):
+            lower_polygon = Polygon(*lower_vertices)
+
+        return upper_polygon, lower_polygon
```

Code implementation of first_moment of area is done in PR #17153

```
+        Examples
+        ========
+
+        >>> from sympy import Point, Polygon, symbol
+        >>> a, b = 50, 10
+        >>> p1, p2, p3, p4 = [(0, b), (0, 0), (a, 0), (a, b)]
+        >>> p = Polygon(p1, p2, p3, p4)
+        >>> p.first_moment_of_area()
+        (625, 3125)
+        >>> p.first_moment_of_area(point=Point(30, 7))
+        (525, 3000)
+        """
+        if point:
+            xc, yc = self.centroid
+        else:
+            point = self.centroid
+            xc, yc = point
+
+        h_line = Line(point, slope=0)
+        v_line = Line(point, slope=S.Infinity)
+
+        h_poly = self.cut_section(h_line)
+        v_poly = self.cut_section(v_line)
+
+        x_min, y_min, x_max, y_max = self.bounds
+
+        poly_1 = h_poly[0] if h_poly[0].area <= h_poly[1].area else h_poly[1]
+        poly_2 = v_poly[0] if v_poly[0].area <= v_poly[1].area else v_poly[1]
+
+        Q_x = (poly_1.centroid.y - yc)*poly_1.area
+        Q_y = (poly_2.centroid.x - xc)*poly_2.area
+
+        return Q_x, Q_y
```

Phase II: Implementing a sub-module 'Column' in the existing continuum mechanics module

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**`.

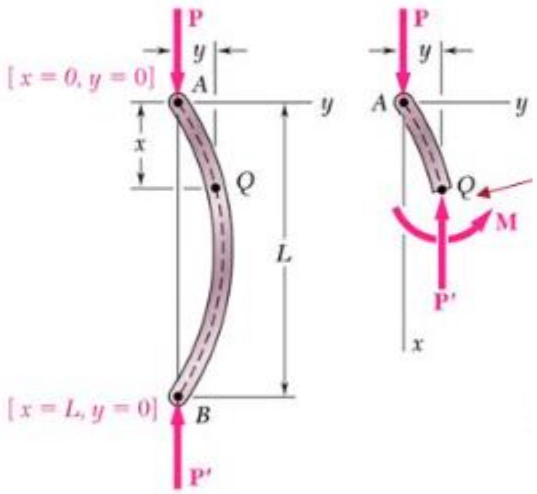
And for eccentric load another moment of magnitude `**P*e**` is added to the moment.

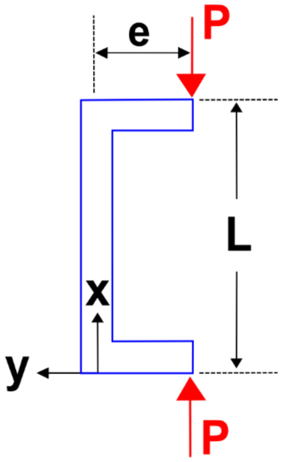
Simple load:

$$M = Py$$

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:**__

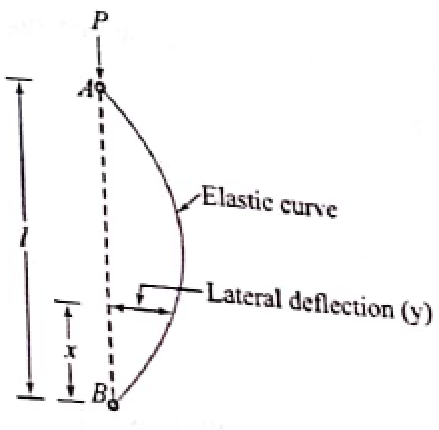
Four basic supports are to be implemented:

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

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

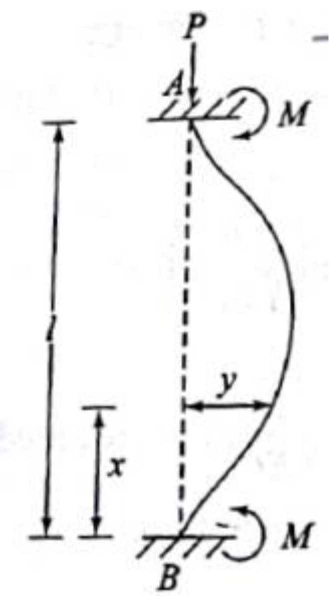
- **Pinned-Pinned:** no change in moment

$$moment = Py$$



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

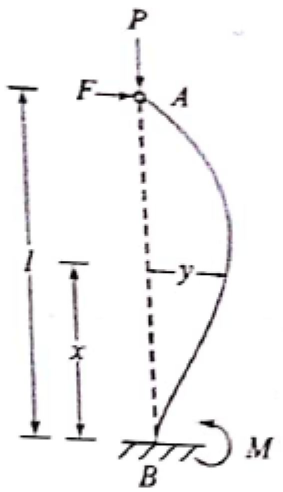
$$moment = Py - M$$



- **Fixed-pinned:**

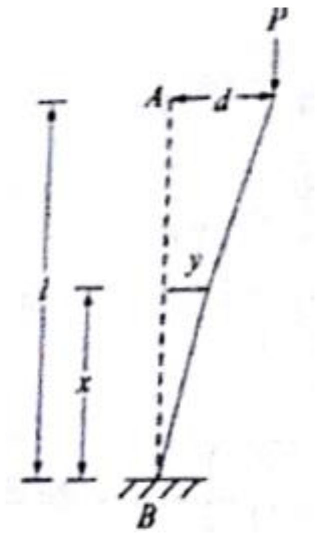
$$moment = Py - F(l - x))$$

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:**

$$moment = Py - Pd$$



<u>Reason for creating a non-mutable class</u>: Most things are immutable in SymPy which is useful for caching etc. Matrix is an example where allowing mutability has lead to many problems that are now impossible to fix without breaking backwards compatibility.

From a backwards compatibility perspective it is always possible to change your mind and add mutability later but not the other way around.

Code implementation of Column class PR #17122

Necessary imports for the Column class:

```
+ from sympy.solvers import linsolve, solve
+ from sympy.core import Symbol, diff, symbols
+ from sympy import dsolve, Function, Derivative, Eq, cos, sin, sqrt, tan
+ from sympy.core.symbol import Dummy
+ from sympy.printing import sstr
+
```

Beginning of the Column class along with documentation and examples:

```
+ class Column(object):
+     """
+     A column is a structural member designed to undertake axial
+     compressive loads. A column is characterized by its
+     cross-sectional profile(second moment of area), its length and
+     its material.
+
+     Examples
+     ========
+
+     There is a solid round bar 3 m long with second-moment I is used as a
+     column with both the ends pinned. Young's modulus of the Column is E.
+     The buckling load applied is 78KN
+
+     >>> from sympy.physics.continuum_mechanics.column import Column
+     >>> from sympy import Symbol, symbols
+     >>> E, I, P = symbols('E, I, P', positive=True)
+     >>> c = Column(3, E, I, 78000, top="pinned", bottom="pinned")
+     >>> c.end_conditions
+     {'bottom': 'pinned', 'top': 'pinned'}
+     >>> c.boundary_conditions
+     {'deflection': [(0, 0), (3, 0)], 'slope': [(0, 0)]}
+     >>> c.moment()
+     78000*y(x)
+     >>> c.solve_slope_deflection()
+     >>> c.deflection()
+     C1*sin(20*sqrt(195)*x/(sqrt(E)*sqrt(I)))
+     >>> c.slope()
+     20*sqrt(195)*C1*cos(20*sqrt(195)*x/(sqrt(E)*sqrt(I)))/(sqrt(E)*sqrt(I))
+     >>> c.critical_load()
+     pi**2*E*I/9
+     """
```

Constructor of the column class which is automatically called when a new object is created:

```
def __init__(self, height, elastic_modulus, second_moment, load, eccentricity=None, top="pinned", bottom="pinned", bc_slope=None, bc_deflection=None):
    """
```

```
+    def __init__(self, height, elastic_modulus, second_moment, load, eccentricity
+        """
+        Parameters
+        ==========
+
+        height: Sympifyable
+            A symbol or a value representing column's height
+
+        elastic_modulus: Sympifyable
+            A symbol or a value representing the Column's modulus of
+            elasticity. It is a measure of the stiffness of the Column
+            material.
+
+        second_moment: Sympifyable
+            A symbol or a value representing Column's second-moment of area
+            It is a geometrical property of an area which reflects how its
+            points are distributed with respect to its neutral axis.
+
+        load: Sympifyable
+            A symbol or a value representing the load applied on the Column.
+
+        eccentricity: Sympifyable (default=None)
+            A symbol or a value representing the eccentricity of the load
+            applied. Eccentricity is the distance of the point of application
+            of load from the neutral axis.
+
```

```
+        top: string (default="pinned")
+            A string representing the top-end condition of the column.
+            It can be: pinned
+                       fixed
+                       free
+
+        bottom: string (default="pinned")
+            A string representing the bottom-end condition of the column.
+            It can be: pinned
+                       fixed
+
+        bc_slope: list of tuples
+            A list of tuples representing the boundary conditions of slope.
+            The tuple takes two elements `location` and `value`.
+
+        bc_deflection: list of tuples
+            A list of tuples representing the boundary conditions of deflection
+            The tuple consists of two elements `location` and `value`.
+        """
```

Documentation ends and the main code inside the constructor begins:

```
+        self._height = height
+        self._elastic_modulus = elastic_modulus
+        self._second_moment = second_moment
+        self._load = load
+        self._eccentricity = eccentricity
+        self._moment = 0
+        self._end_conditions = {'top':top, 'bottom': bottom}
+        self._boundary_conditions = {'deflection': [], 'slope': []}
+        if bc_deflection:
+            self._boundary_conditions['deflection'] = bc_deflection
+        if bc_slope:
+            self._boundary_conditions['slope'] = bc_slope
+        self._variable = Symbol('x')
+        self._deflection = None
+        self._slope = None
+        self._critical_load = None
+        self._apply_load_conditions()
```

The ‘str’ function:

```
+     def __str__(self):
+         str_sol = 'Column({}, {}, {})'.format(sstr(self._height), sstr(self._elastic_modulus), sstr(self._second_moment))
+         return str_sol
```

Defining different attribute properties:

```
+     @property
+     def height(self):
+         """Height of the column"""
+         return self._height
+
+     @property
+     def elastic_modulus(self):
+         """Elastic modulus of the column"""
+         return self._elastic_modulus
+
+     @property
+     def second_moment(self):
+         """Second moment of the column"""
+         return self._second_moment
+
+     @property
+     def load(self):
+         """Load applied on the column"""
+         return self._load
+
+     @property
+     def eccentricity(self):
+         """Eccentricity of the load applied on the column"""
+         return self._eccentricity
+
```

Defining a helper function '_apply_load_conditions()' which is automatically called when a new Column object is created. It calculates the moment acting on the beam.

```
+    def _apply_load_conditions(self):
+        y = Function('y')
+        x = self._variable
+        P = Symbol('P', positive=True)
+
+        self._moment += P*y(x)
+        if self.eccentricity:
+            self._moment += P*eccentricity
+
+        # Initial boundary conditions, considering slope and deflection
+        # the bottom always zero
+        self._boundary_conditions['deflection'].append((0, 0))
+        self._boundary_conditions['slope'].append((0, 0))
+
+        if self._end_conditions['top'] == "pinned" and self._end_conditions['bottom'] == "pinned":
+            self._boundary_conditions['deflection'].append((self._height, 0))
+
+        elif self._end_conditions['top'] == "fixed" and self._end_conditions['bottom'] == "fixed":
+            # `M` is the reaction moment
+            M = Symbol('M')
+            self._boundary_conditions['deflection'].append((self._height, 0))
+            # moment  = P*y - M
+            self._moment -= M
+
```

```
elif self._end_conditions['top'] == "pinned" and self._end_conditions['bottom'] == "fixed":
    # `F` is the horizontal force at the pinned end to counter the rection moment at fixed end
    F = Symbol('F')
    self._boundary_conditions['deflection'].append((self._height, 0))
    # moment = P*y - F(l - x)
    self._moment -= F*(self.height - x)

elif self._end_conditions['top'] == "free" and self._end_conditions['bottom'] == "fixed":
    # `d` is the deflection at the free end
    d = Symbol('d')
    self._boundary_conditions['deflection'].append((self._height, d))
    # moment = P*y - P*d
    self._moment -= P*d

else:
    raise ValueError("{} {} end-condition is not supported".format(sstr(self._end_conditions['top']), sstr(self._end_conditions['bottom'])))
```

Solve_slope_and_deflection() which solves for the slope and deflection on the column:

```
+        y = Function('y')
+        x = self._variable
+        P = Symbol('P', positive=True)
+
+        C1, C2 = symbols('C1, C2')
+        E = self._elastic_modulus
+        I = self._second_moment
+
+        # differnetial equation of Column buckling
+        eq = E*I*y(x).diff(x, 2) + self._moment
+
+        self._deflection = dsolve(Eq(eq, 0), y(x), ics={y(0): 0, y(x).diff(x).subs(x, 0): 0}).args[1]
```

A condition usually occurs where the deflection calculated by the above method comes out to be zero, which implies that no buckling occurs but that is not the case, so we try to solve it using a different method

```
+        # if deflection is zero, no buckling occurs, which is not the case,
+        # so trying to solve for the constants differently
+        if self._deflection == 0:
+            self._deflection = dsolve(Eq(eq, 0), y(x)).args[1]
+
+            defl_eqs = []
+            # taking last two bounndary conditions which are actually
+            # the initial boundary conditions.
+            for point, value in self._boundary_conditions['deflection'][-2:]:
+                defl_eqs.append(self._deflection.subs(x, point) - value)
+
+            # solve for C1, C2 along with P
+            solns = solve(defl_eqs, (P, C1, C2), dict=True)
+            for sol in solns:
+                if self._deflection.subs(sol) == 0:
+                    # removing trivial solutions
+                    solns.remove(sol)
+
+            # checking if the constants are solved, and subtituting them in
+            # the deflection and slope equation
+            if C1 in solns[0].keys():
+                self._deflection = self._deflection.subs(C1, solns[0][C1])
+            if C2 in solns[0].keys():
+                self._deflection = self._deflection.subs(C2, solns[0][C2])
+            if P in solns[0].keys():
+                self._critical_load = solns[0][P]
+
+        self._slope = self._deflection.diff(x)
```

Solving for the critical load:

```
+        y = Function('y')
+        x = self._variable
+        P = Symbol('P', positive=True)
+
+        if self._critical_load is None:
+            defl_eqs = []
+            # taking last two bounndary conditions which are actually
+            # the initial boundary conditions.
+            for point, value in self._boundary_conditions['deflection'][-2:]:
+                defl_eqs.append(self._deflection.subs(x, point) - value)
+
+            # C1, C2 already solved, solve for P
+            self._critical_load = solve(defl_eqs, P, dict=True)[0][P]
+
+        return self._critical_load
```

Writing tests for the Column Class

- **Test for pinned-pinned end condition:**

```
+    # test for pinned-pinned end-condition
+    c1 = Column(l, E, I, P, top="pinned", bottom="pinned")
+    c1.solve_slope_deflection()
+    assert c1.moment() == P*y(x)
+    assert c1.deflection() == C1*sin(sqrt(P)*x/(sqrt(E)*sqrt(I)))
+    assert c1.slope() == C1*sqrt(P)*cos(sqrt(P)*x/(sqrt(E)*sqrt(I)))/(sqrt(E)*sqrt(I))
+    assert c1.critical_load() == pi**2*E*I/l**2
```

- **Test for fixed-fixed end-condition:**

```
+    # test for fixed-fixed end-condition
+    c2 = Column(l, E, I, P, top="fixed", bottom="fixed")
+    c2.solve_slope_deflection()
+    assert c2.moment() == -M + P*y(x)
+    assert c2.deflection() == -M*cos(sqrt(P)*x/(sqrt(E)*sqrt(I)))/P + M/P
+    assert c2.slope() == M*sin(sqrt(P)*x/(sqrt(E)*sqrt(I)))/(sqrt(E)*sqrt(I)*sqrt(P))
+    assert c2.critical_load() == 4*pi**2*E*I/l**2
+
```

- **Test for free-fixed end condition:**

```
+     # test for free-fixed end-condition
+     c3 = Column(l, E, I, P, top="free", bottom="fixed")
+     c3.solve_slope_deflection()
+     assert c3.moment() == -P*d + P*y(x)
+     assert c3.deflection() == -d*cos(sqrt(P)*x/(sqrt(E)*sqrt(I))) + d
+     assert c3.slope() == sqrt(P)*d*sin(sqrt(P)*x/(sqrt(E)*sqrt(I)))/(sqrt(E)*sqrt(I))
+     assert c3.critical_load() == pi**2*E*I/(4*l**2)
+
```

- **Test for pinned-fixed end condition:**

```
# test for pinned-fixed end-condition
c4 = Column(l, E, I, P, top="pinned", bottom="fixed")
c4.solve_slope_deflection()
assert c4.moment() == -F*(l - x) + P*y(x)
assert c4.deflection() == sqrt(E)*F*sqrt(I)*sin(sqrt(P)*x/(sqrt(E)*sqrt(I)))/P**(3/S(2)) - F*l*cos(sqrt(P)*x/(sqrt(E)*sqrt(I)))/P + F*l/P - F*x/P
assert c4.slope() == F*cos(sqrt(P)*x/(sqrt(E)*sqrt(I)))/P - F/P + F*l*sin(sqrt(P)*x/(sqrt(E)*sqrt(I)))/(sqrt(E)*sqrt(I)*sqrt(P))


raises(ValueError, lambda: Column(l, E, I, P, top="free", bottom="free"))
```

- **<u>Writing an XFAIL</u>:** The deflection equation of pinned-fixed end-condition comes out to be of the form **`a*cos(x) - b*sin(x)`** for which **solve()** should return in the form **`x = atan(a/b)`**. **solve()** currently gives the answer in other form. Either we can get a way to convert it into desired form or make **solve()** return it in the required form.

```
+ @XFAIL
+ def test_critical_load_pinned_fixed():
+     # the deflction equation of pinned-fixed end-condition
+     # comes out to be of the form `a*cos(x) - b*sin(x)` for which
+     # solve should return in the form `x = atan(a/b)`. solve() currently
+     # gives the answer in other form. Either we can get a way to convert it
+     # into desired form or make solve() return it in the required form.
+     c = Column(l, E, I, P, top="pinned", bottom="fixed")
+     c.solve_slope_deflection()
+     c.critical_load() == 2*pi**2*E*I/l**2
```

Phase III: Plotting beam diagrams.

A new function **draw()** was to be implemented inside the beam class, which would make the beam class capable of plotting the beam diagram corresponding to the beam object.

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

- Draw a rectangle (which is a beam):
- Draw a curve for the load equation above the beam and fill it with colour to depict different loads applied on the beam
- Draw supports
- At the end put the axis 'OFF'

The draw function was initially intended to use **matplotlib** as an external module to plot the beam diagram, as SymPy's own plotting module wasn't capable to support geometry shapes, annotations and ability to fill colour inside a curve.

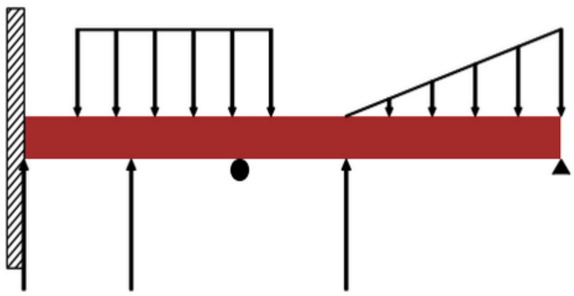
Initially matplotlib was directly used to draw the beam diagram. This implementation was done in **PR #17240.**

The implementation gave the following results:

```
>>> E, I = symbols('E, I')

>>> b1 = Beam(50, E, I)

>>> b1.apply_load(-10, 0, -1)
>>> b1.apply_load(R1, 10, -1)
>>> b1.apply_load(R2, 30, -1)
>>> b1.apply_load(9, 5, 0, 23)
>>> b1.apply_load(9, 30, 1, 50)
>>> b1.apply_support(50, "pin")
>>> b1.apply_support(0, "fixed")
>>> b1.apply_support(20, "roller")
>>> b1.draw()
```



This implementation had the following drawbacks:

- Drawing superimposed loads was a bit tricky to handle
- It made the beam class dependent on an external plotting module despite sympy having its own module for plotting.

After rigorous discussion with the mentors and fellow contributors, we came up with a solution to these problems

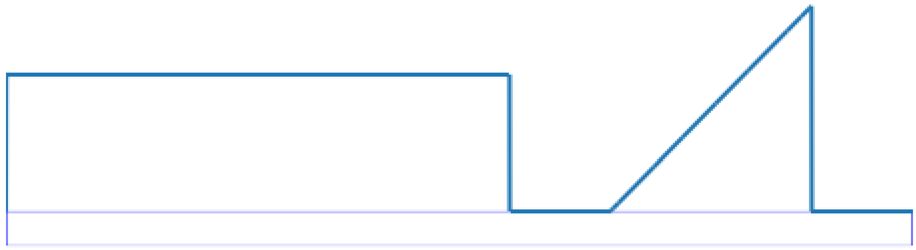
If we directly plot the loading equation (which is in terms of singularity function) using sympy's plot, we might be able to tackle the first problem. And to make the load look cleaner we could use colour filling instead of placing arrows below the curve.

I also came up with a solution for the second problem. I implemented the features inside the sympy's plotting module and then used the same inside the **draw()** function.

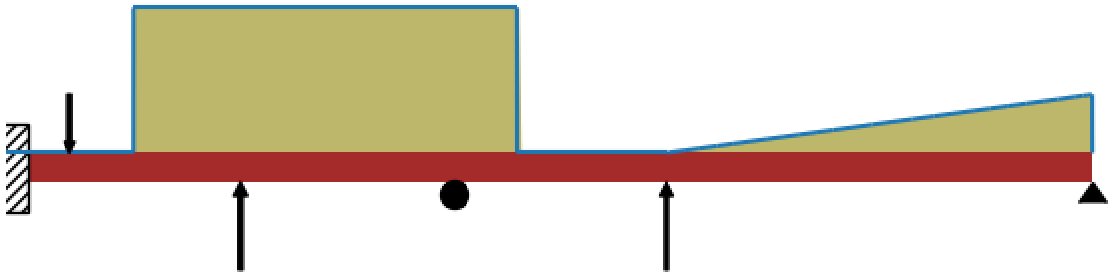
The following are the features implemented in sympy's plotting module.

```
+     ``annotations``: list. A list of dictionaries specifying the type of
+     annotation required. The keys in the dictionary should be equivalent
+     to the arguments of the matplotlib's annotate() function.
+
+     ``markers``: list. A list of dictionaries specifying the type the
+     markers required. The keys in the dictionary should be equivalent
+     to the arguments of the matplotlib's plot() function along with the
+     marker related keyworded arguments.
+
+     ``rectangles``: list. A list of dictionaries specifying the dimensions
+     of the rectangles to be plotted. The keys in the dictionary should be
+     equivalent to the arguments of the matplotlib's patches.Rectangle class.
+
+     ``fill``: dict. A dictionary specifying the type of color filling
+     required in the plot. The keys in the dictionary should be equivalent
+     to the arguments of the matplotlib's fill_between() function.
+
```

After implementing the features in the plotting module the **draw()** function was then rewritten in a separate pull request **PR #17345**. The first step was to use plot the singularity function itself. This is how the singularity function above the beam looks like:



Now the next task was to fill colour inside the beam rectangle and under the load plot, and add different supports (fixed, roller, pin). The beam diagram now looks like:



The **draw()** function was now almost complete. Only certain tests were to be performed and the results were to be verified. During this time Jason suggested that it would be good to give user the option whether to scale down the diagram as in cases where the magnitude of load was very high the plot went outside the plotting window.

The **draw()** function was then modified with an incoming argument **pictorial** which is a Boolean kept by default as True. The user would get a pictorial representation of loads when pictorial is set to TRUE and a representation of the exact dimensions when pictorial is set to FALSE.

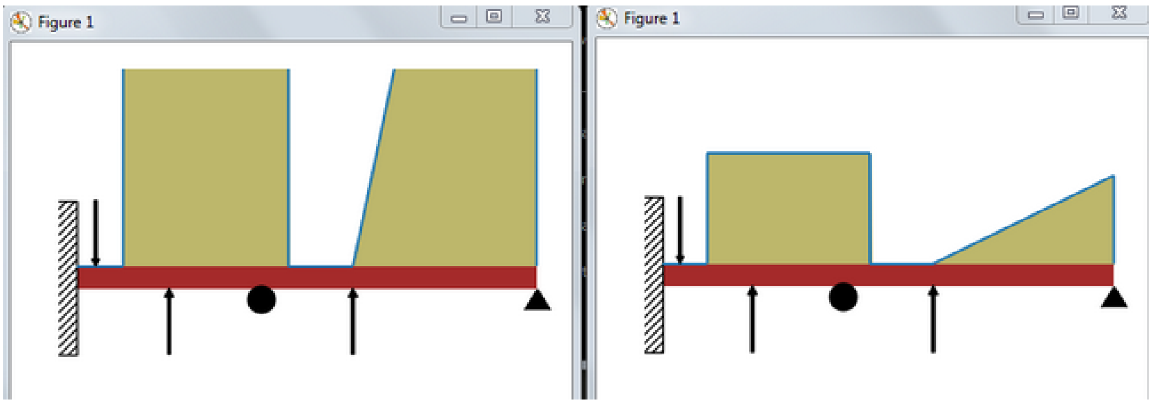
<u>Using pictorial argument with the draw() function:</u>

```
>>> R1, R2 = symbols('R1, R2')
>>> E, I = symbols('E, I')

>>> b1 = Beam(50, 20, 30)

>>> b1.apply_load(10, 2, -1)
>>> b1.apply_load(R1, 10, -1)
>>> b1.apply_load(R2, 30, -1)
>>> b1.apply_load(90, 5, 0, 23)
>>> b1.apply_load(10, 30, 1, 50)
>>> b1.apply_support(50, "pin")
>>> b1.apply_support(0, "fixed")
>>> b1.apply_support(20, "roller")
# case 1 on the left
>>> p = b1.draw()
>>> p.show()

# case 2 on the right
>>> p1 = b1.draw(pictorial=True)
>>> p1.show()
```



Code implementation of the draw function:

Basic API along with documentation:

```
+    @doctest_depends_on(modules=('numpy',))
+    def draw(self, pictorial=True):
+        """Returns a plot object representing the beam diagram of the beam.
+
+        Parameters
+        ==========
+
+        pictorial: Boolean (default=True)
+            Setting ``pictorial=True`` would simply create a pictorial (scaled) view
+            of the beam diagram not with the exact dimensions.
+            Although setting ``pictorial=False`` would create a beam diagram with
+            the exact dimensions on the plot
```

The main driving code:

```
+        if not numpy:
+            raise ImportError("To use this function numpy module is required")
+
+        x = self.variable
+
+        # checking whether length is an expression in terms of any Symbol.
+        from sympy import Expr
+        if isinstance(self.length, Expr):
+            l = list(self.length.atoms(Symbol))
+            # assigning every Symbol a default value of 10
+            l = {i:10 for i in l}
+            length = self.length.subs(l)
+        else:
+            l = {}
+            length = self.length
+        height = length/10
+
+        rectangles = []
+        rectangles.append({'xy':(0, 0), 'width':length, 'height': height, 'facecolor':"brown"})
+        annotations, markers, load_eq, fill = self._draw_load(pictorial, length, l)
+        support_markers, support_rectangles = self._draw_supports(length, l)
+
+        rectangles += support_rectangles
+        markers += support_markers
+
+        sing_plot = plot(height + load_eq, (x, 0, length),
+         xlim=(-height, length + height), ylim=(-length, 1.25*length), annotations=annotations,
+          markers=markers, rectangles=rectangles, fill=fill, axis=False, show=False)
+
+        return sing_plot
```

_draw_load() : A helper function to draw()

```
+    def _draw_load(self, pictorial, length, l):
+        loads = list(set(self.applied_loads) - set(self._support_as_loads))
+        height = length/10
+        x = self.variable
+
+        annotations = []
+        markers = []
+        load_args = []
+        scaled_load = 0
+        load_eq = 0
+        higher_order = False
+        fill = None
+
+        for load in loads:
+            # check if the position of load is in terms of the beam length.
+            if l:
+                pos =  load[1].subs(l)
+            else:
+                pos = load[1]
+
+            # point loads
+            if load[2] == -1:
+                if isinstance(load[0], Symbol) or load[0].is_negative:
+                    annotations.append({'s':'', 'xy':(pos, 0), 'xytext':(pos, height - 4*height)
+                else:
+                    annotations.append({'s':'', 'xy':(pos, height),  'xytext':(pos, height*4), '
```

```
+            # moment loads
+            elif load[2] == -2:
+                if load[0].is_negative:
+                    markers.append({'args':[[pos], [height/2]], 'marker': r'$\circlearrowleft$', 'markersize':15})
+                else:
+                    markers.append({'args':[[pos], [height/2]], 'marker': r'$\circlearrowright$', 'markersize':15})
+            # higher order loads
+            elif load[2] >= 0:
+                higher_order = True
+                # if pictorial is True we remake the load equation again with
+                # some constant magnitude values.
+                if pictorial:
+                    value, start, order, end = load
+                    value = 10**(1-order) if order > 0 else length/2
+
+                    scaled_load += value*SingularityFunction(x, start, order)
+                    if end:
+                        f2 = 10**(1-order)*x**order if order > 0 else length/2*x**order
+                        for i in range(0, order + 1):
+                            scaled_load -= (f2.diff(x, i).subs(x, end - start)*
+                                           SingularityFunction(x, end, i) / factorial(i))
```

```
        # `fill` will be assigned only when higher order loads are present
        if higher_order:
            if pictorial:
                if isinstance(scaled_load, Add):
                    load_args = scaled_load.args
                else:
                    # when the load equation consists of only a single term
                    load_args = (scaled_load,)
                load_eq = [i.subs(l) for i in load_args]
            else:
                if isinstance(self.load, Add):
                    load_args = self.load.args
                else:
                    load_args = (self.load,)
                load_eq = [i.subs(l) for i in load_args if list(i.atoms(SingularityFunction))[0].args[2] >= 0]

            load_eq = Add(*load_eq)

            # filling higher order loads with colour
            y = numpy.arange(0, float(length), 0.001)
            expr = height + load_eq.rewrite(Piecewise)
            y1 = lambdify(x, expr, 'numpy')
            y2 = float(height)
            fill = {'x': y, 'y1': y1(y), 'y2': y2, 'color':'darkkhaki'}

        return annotations, markers, load_eq, fill

```

_draw_supports() : A helper function to draw()

```
    def _draw_supports(self, length, l):
        height = float(length/10)

        support_markers = []
        support_rectangles = []
        for support in self._applied_supports:
            if l:
                pos =  support[0].subs(l)
            else:
                pos = support[0]

            if support[1] == "pin":
                support_markers.append({'args':[pos, [0]], 'marker':6, 'markersize':13, 'color':"black"})

            elif support[1] == "roller":
                support_markers.append({'args':[pos, [-height/2.5]], 'marker':'o', 'markersize':11, 'color

            elif support[1] == "fixed":
                if pos == 0:
                    support_rectangles.append({'xy':(0, -3*height), 'width':-length/20, 'height':6*height -
                else:
                    support_rectangles.append({'xy':(length, -3*height), 'width':length/20, 'height': 6*he

        return support_markers, support_rectangles

```

Future Scope

Here is a list that comprises of all the ideas (which were a part of my GSoC Proposal and/or thought over during the summer) which can extend my project.

- Calculating the first moment of area for ellipses class.
- Determining the critical load for the pinned-fixed end condition of Column.
- Making a class for symbolic Truss analysis.
- Making composite shapes using Boolean operations on basic shapes in the geometry module.

Conclusion

This summer has been a great learning experience and has helped me get good knowledge on the subject. I plan to actively review the work that went into this project and continue contributing to SymPy. I am grateful to my mentors, Jason, Jashanpreet and Yathartha for reviewing my work, giving me valuable suggestions, and being readily available for discussions.