centroid.

Examples

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A = symbols('l, E, G, I, A')
>>> b = Beam3D(l, E, G, I, A)
>>> b.polar_moment()
2*I
>>> I1 = [9, 15]
>>> b = Beam3D(l, E, G, I1, A)
>>> b.polar_moment()
24
```

property second_moment

Second moment of area of the Beam.

shear_force()

Returns a list of three expressions which represents the shear force curve of the Beam object along all three axes.

property shear_modulus

Young's Modulus of the Beam.

shear_stress()

Returns a list of three expressions which represents the shear stress curve of the Beam object along all three axes.

slope()

Returns a three element list representing slope of deflection curve along all the three axes.

solve_for_reaction_loads(**reaction*)

Solves for the reaction forces.

Examples

There is a beam of length 30 meters. It it supported by rollers at of its end. A constant distributed load of magnitude 8 N is applied from start till its end along y-axis. Another linear load having slope equal to 9 is applied along z-axis.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(30, E, G, I, A, x)
>>> b.apply_load(8, start=0, order=0, dir="y")
>>> b.apply_load(9*x, start=0, order=0, dir="z")
>>> b.bc_deflection = [(0, [0, 0, 0]), (30, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="y")
>>> b.apply_load(R2, start=30, order=-1, dir="y")
```

(continues on next page)

(continued from previous page)

```
>>> b.apply_load(R3, start=0, order=-1, dir="z")
>>> b.apply_load(R4, start=30, order=-1, dir="z")
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.reaction_loads
{R1: -120, R2: -120, R3: -1350, R4: -2700}
```

torsional_moment()

Returns expression of Torsional moment present inside the Beam object.

Solving Beam Bending Problems using Singularity Functions

To make this document easier to read, enable pretty printing:

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True, wrap_line=False)
```

Beam

A planar beam is a structural element that is capable of withstanding load through resistance to internal shear and bending. Beams are characterized by their length, constraints, cross-sectional second moment of area, and elastic modulus. In SymPy, 2D beam objects are constructed by specifying the following properties:

- Length
- Elastic Modulus
- Second Moment of Area
- Variable : A symbol representing the location along the beam's length. By default, this is set to `Symbol(x)`.
- **Boundary Conditions**
  - bc_slope : Boundary conditions for slope.
  - bc_deflection : Boundary conditions for deflection.
- Load Distribution

Once the above are specified, the following methods are used to compute useful information about the loaded beam:

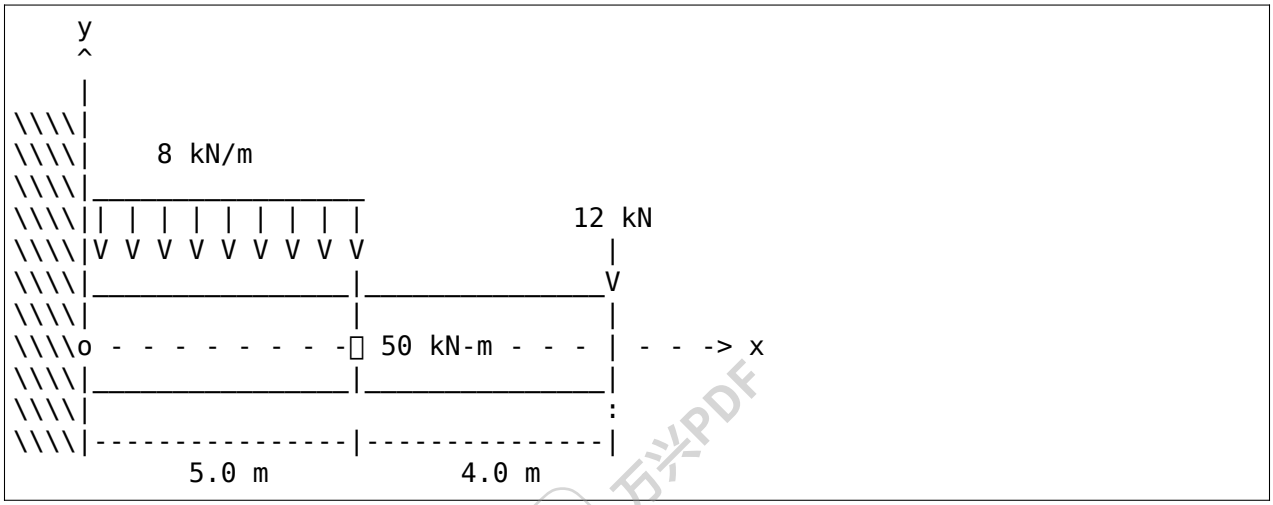
- `solve_for_reaction_loads()`
- `shear_force()`
- `bending_moment()`
- `slope()`

Examples

Below are examples of a variety two dimensional beam bending problems.

Example 1

A cantilever beam 9 meters in length has a distributed constant load of 8 kN/m applied downward from the fixed end over a 5 meter distance. A counterclockwise moment of 50 kN-m is applied 5 meters from the fixed end. Lastly, a downward point load of 12 kN is applied at the free end of the beam.



Note: The user is free to choose their own sign convention. In this case the downward forces and counterclockwise bending moment being positive.

The beam must be initialized with the length, modulus of elasticity, and the second moment of area. These quantities can be symbols or numbers.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> b = Beam(9, E, I)
```

The three loads are applied to the beam using the `apply_load()` method. This method supports point forces, point moments, and polynomial distributed loads of any order, i.e. $c, cx, cx^2, cx^3, \ldots$.

The 12 kN point load is in the negative direction, at the location of 9 meters, and the polynomial order is specified as -1:

```
>>> b.apply_load(12, 9, -1)
```

The `load` attribute can then be used to access the loading function in singularity function form:

```
>>> b.load
          -1
12·<x - 9>
```

Similarly, the positive moment can be applied with a polynomial order -2:

```
>>> b.apply_load(50, 5, -2)
```

The distributed load is of order 0 and spans x=0 to x=5:

```
>>> b.apply_load(8, 0, 0, end=5)
```

The fixed end imposes two boundary conditions: 1) no vertical deflection and 2) no rotation. These are specified by appending tuples of x values and the corresponding deflection or slope values:

```
>>> b.bc_deflection.append((0, 0))
>>> b.bc_slope.append((0, 0))
```

These boundary conditions introduce an unknown reaction force and moment which need to be applied to the beam to maintain static equilibrium:

```
>>> R, M = symbols('R, M')
>>> b.apply_load(R, 0, -1)
>>> b.apply_load(M, 0, -2)
>>> b.load
     -2         -1         0              -2             0              -1
M·<x>   + R·<x>   + 8·<x>  + 50·<x - 5>   - 8·<x - 5>  + 12·<x - 9>
```

These two variables can be solved for in terms of the applied loads and the final loading can be displayed:

```
>>> b.solve_for_reaction_loads(R, M)
>>> b.reaction_loads
    {M: 158, R: -52}
>>> b.load
           -2          -1         0              -2             0              -1
    158·<x>   - 52·<x>   + 8·<x>  + 50·<x - 5>   - 8·<x - 5>  + 12·<x - 9>
```

At this point, the beam is fully defined and the internal shear and bending moments are calculated:

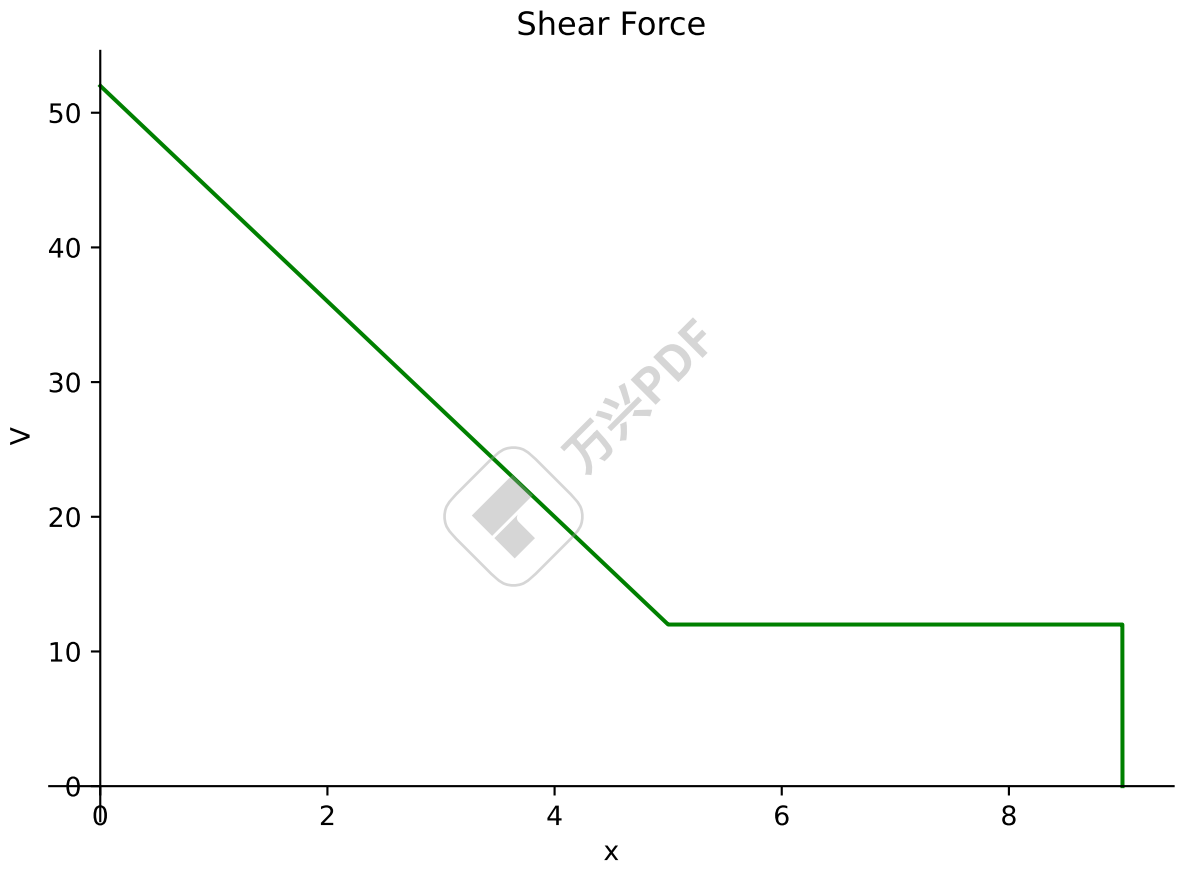
```
>>> b.shear_force()
          -1          0         1              -1             1              0
 - 158·<x>   + 52·<x>  - 8·<x>  - 50·<x - 5>   + 8·<x - 5>  - 12·<x - 9>
```

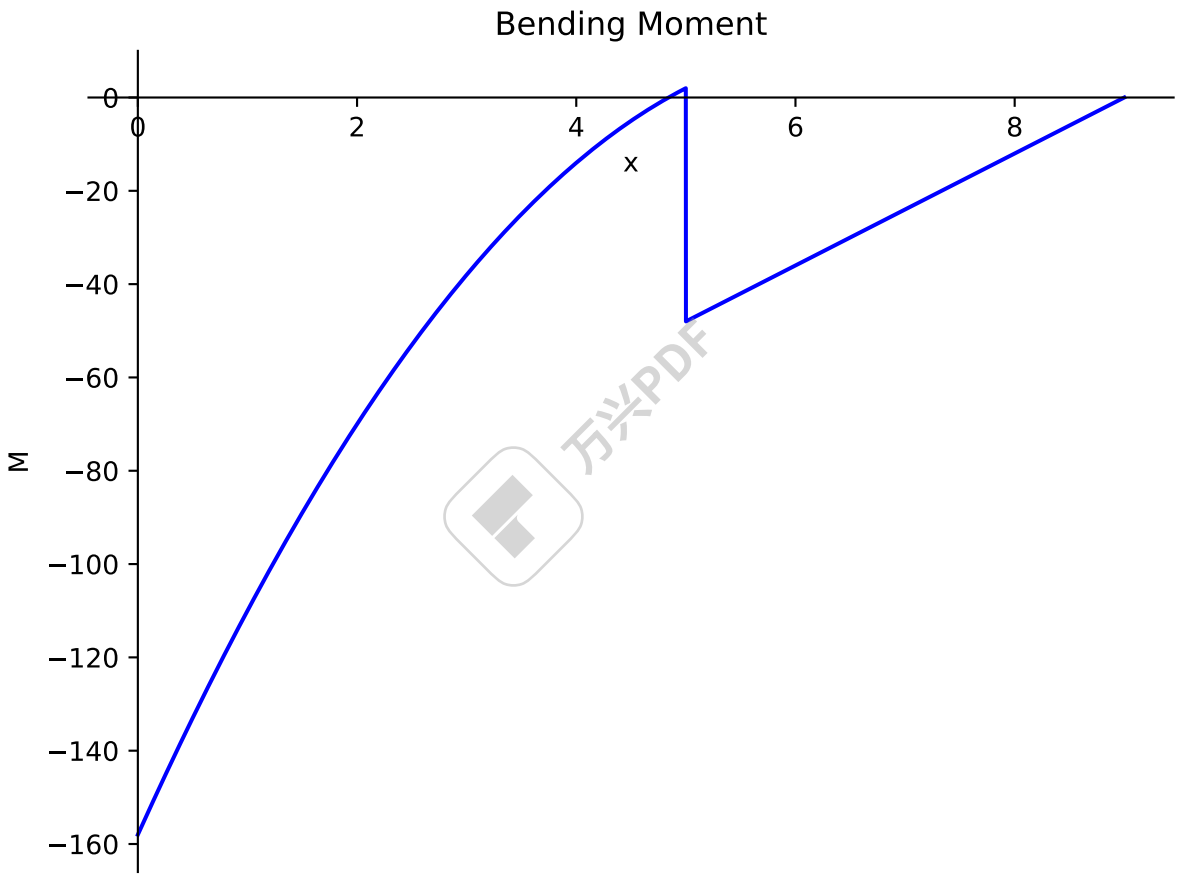
```
>>> b.bending_moment()
             0          1         2              0             2              1
    - 158·<x>  + 52·<x>  - 4·<x>  - 50·<x - 5>  + 4·<x - 5>  - 12·<x - 9>
```

These can be visualized by calling the respective plot methods:

```
>>> b.plot_shear_force()
>>> b.plot_bending_moment()
```

The beam will deform under load and the slope and deflection can be determined with:

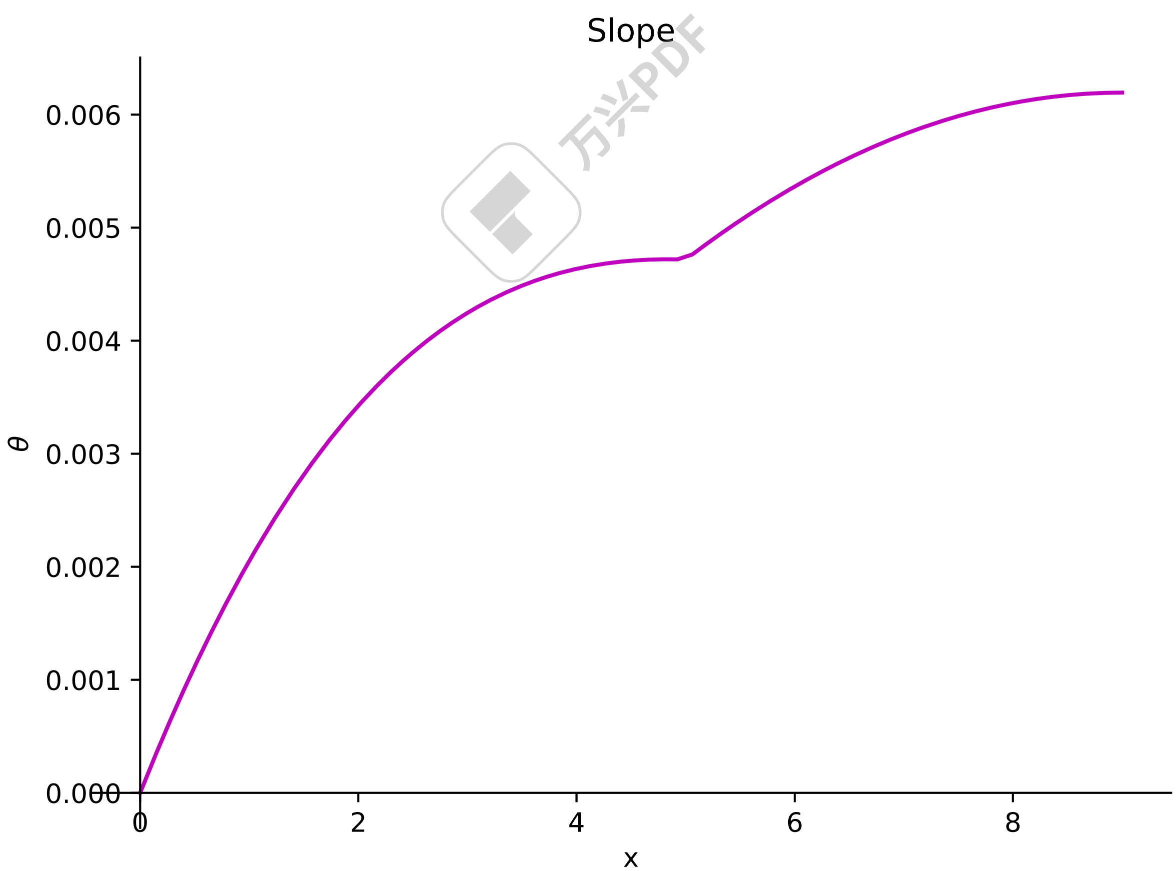

Shear Force
50
40
30
20
10
0
V
0
2
4
6
8
x


Bending Moment
0
0
2
4
6
8
x
−20
−40
−60
−80
−100
−120
−140
−160
M

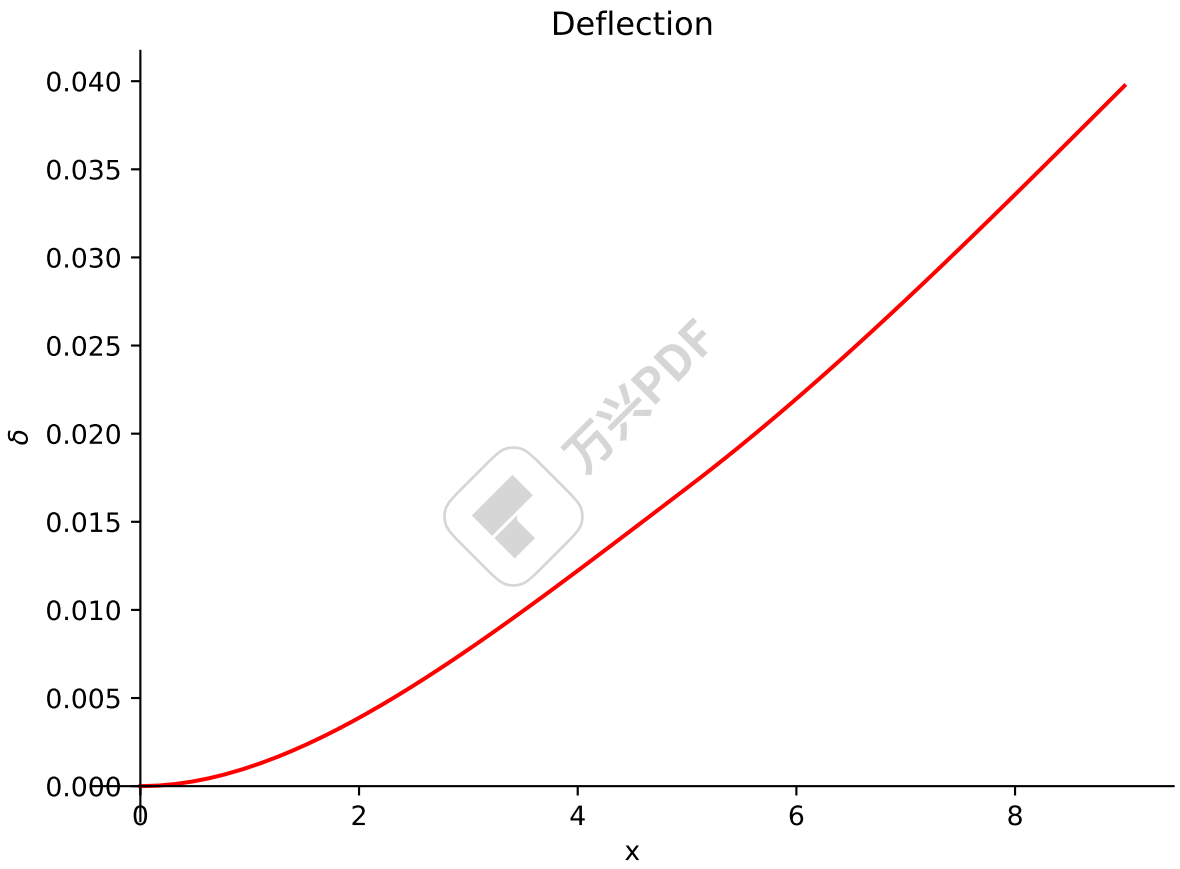
```
>>> b.slope()
 ⎛                            3                          3             ⎞
 ⎜         1         2   4⋅<x>              1   4⋅<x - 5>             2⎟
-⎜- 158⋅<x>  + 26⋅<x>  - ────── - 50⋅<x - 5>  + ────────── - 6⋅<x - 9> ⎟
 ⎝                         3                        3                  ⎠
─────────────────────────────────────────────────────────────────────────
                                   E⋅I
>>> b.deflection()
 ⎛                  3      4                        4             ⎞
 ⎜        2   26⋅<x>    <x>              2   <x - 5>             3⎟
-⎜- 79⋅<x>  + ─────── - ──── - 25⋅<x - 5>  + ──────── - 2⋅<x - 9> ⎟
 ⎝               3       3                      3                 ⎠
────────────────────────────────────────────────────────────────────
                                E⋅I
```

The slope and deflection of the beam can be plotted so long as numbers are provided for the modulus and second moment:

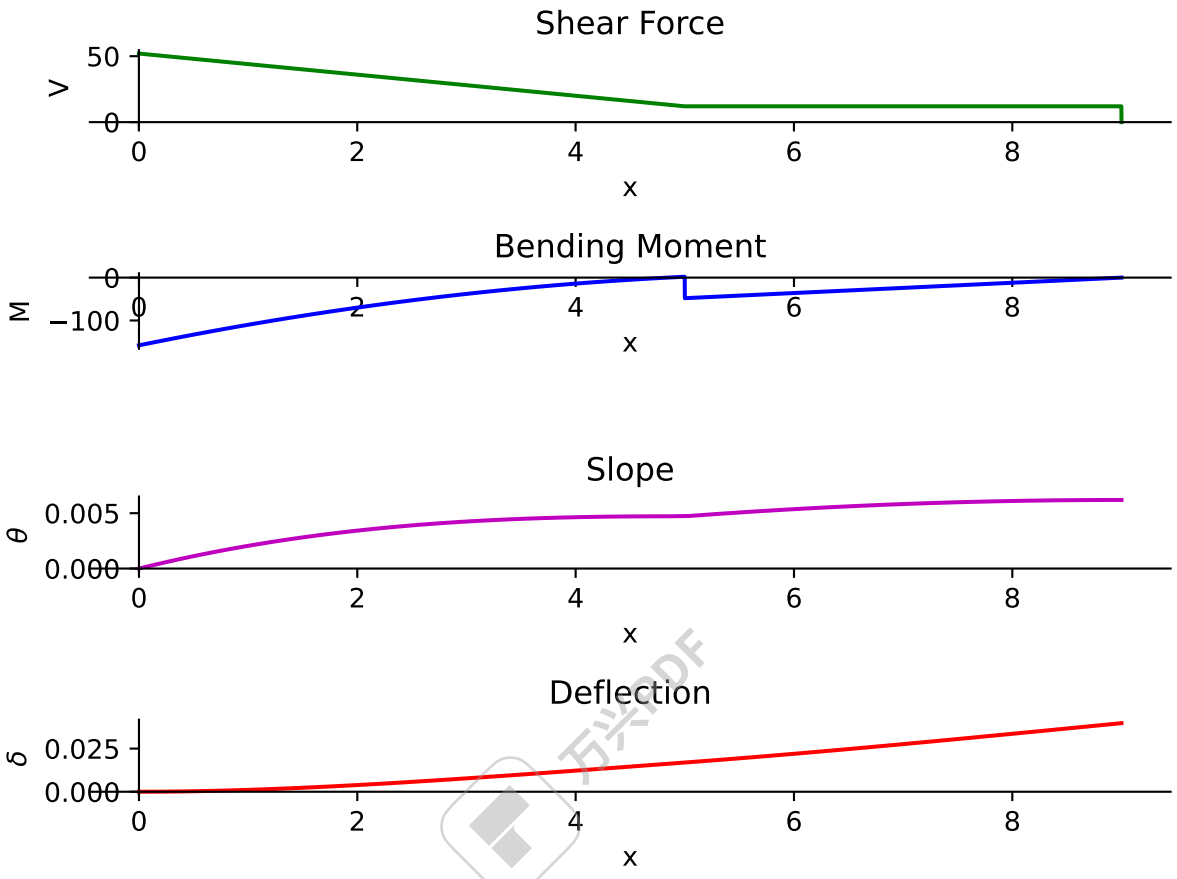
```
>>> b.plot_slope(subs={E: 20E9, I: 3.25E-6})
>>> b.plot_deflection(subs={E: 20E9, I: 3.25E-6})
```



All of the plots can be shown in one figure with:


Deflection
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
δ
0
2
4
6
8
x

```
>>> b.plot_loading_results(subs={E: 20E9, I: 3.25E-6})
```



Example 2

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the counter-clockwise direction at the end of the beam. A point load of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

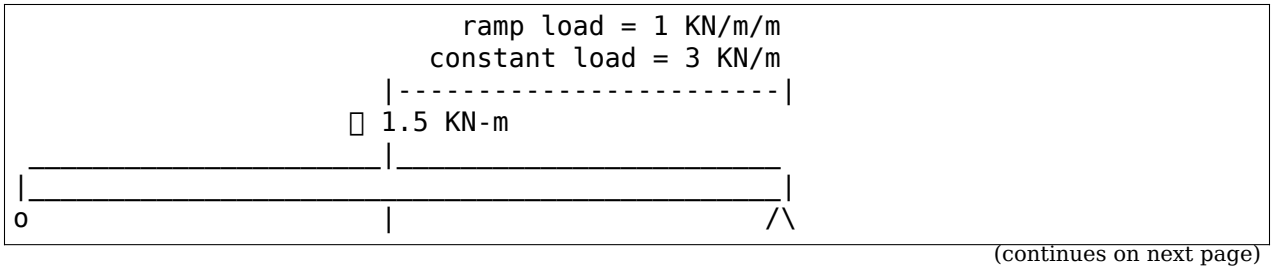
```
|| 8 N                                          ↺ 120 Nm
\/_____________________________________________|
|_____________________________________________|
           /\                                  /\
|------------|-----------------------------------|
    10 m                  20 m
```

Note: Using the sign convention of downward forces and counterclockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection.append((10, 0))
>>> b.bc_deflection.append((30, 0))
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
    {R_1: -18, R_2: 10}
>>> b.load
         -1              -1               -2               -1
    8⋅<x>   - 18⋅<x - 10>   + 120⋅<x - 30>   + 10⋅<x - 30>
>>> b.shear_force()
           0               0                -1               0
    - 8⋅<x>  + 18⋅<x - 10>  - 120⋅<x - 30>   - 10⋅<x - 30>
>>> b.bending_moment()
           1               1                0               1
    - 8⋅<x>  + 18⋅<x - 10>  - 120⋅<x - 30>  - 10⋅<x - 30>
>>> b.slope()
         2             2               1              2   1600
    4⋅<x>  - 9⋅<x - 10>  + 120⋅<x - 30>  + 5⋅<x - 30>  - ────
                                                        3
    ──────────────────────────────────────────────────────────
                                E⋅I
>>> b.deflection()
                   3                                        3
      1600⋅x   4⋅<x>              3               2   5⋅<x - 30>
    - ──────── + ──── - 3⋅<x - 10>  + 60⋅<x - 30>  + ────────── + 4000
         3        3                                      3
    ──────────────────────────────────────────────────────────────────
                                E⋅I
```


Example 3

A beam of length 6 meters is having a roller support at the start and a hinged support at the end. A counterclockwise moment of 1.5 kN-m is applied at the mid of the beam. A constant distributed load of 3 kN/m and a ramp load of 1 kN/m/m is applied from the mid till the end of the beam.



(continues on next page)

(continued from previous page)

```
|----------------------|----------------------|
        3.0 m                  3.0 m
```

Note: Using the sign convention of downward forces and counterclockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, plot, S
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(6, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(-S(3)/2, 3, -2)
>>> b.apply_load(3, 3, 0)
>>> b.apply_load(1, 3, 1)
>>> b.apply_load(R2, 6, -1)
>>> b.bc_deflection.append((0, 0))
>>> b.bc_deflection.append((6, 0))
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
   {R1: -11/4, R2: -43/4}
```

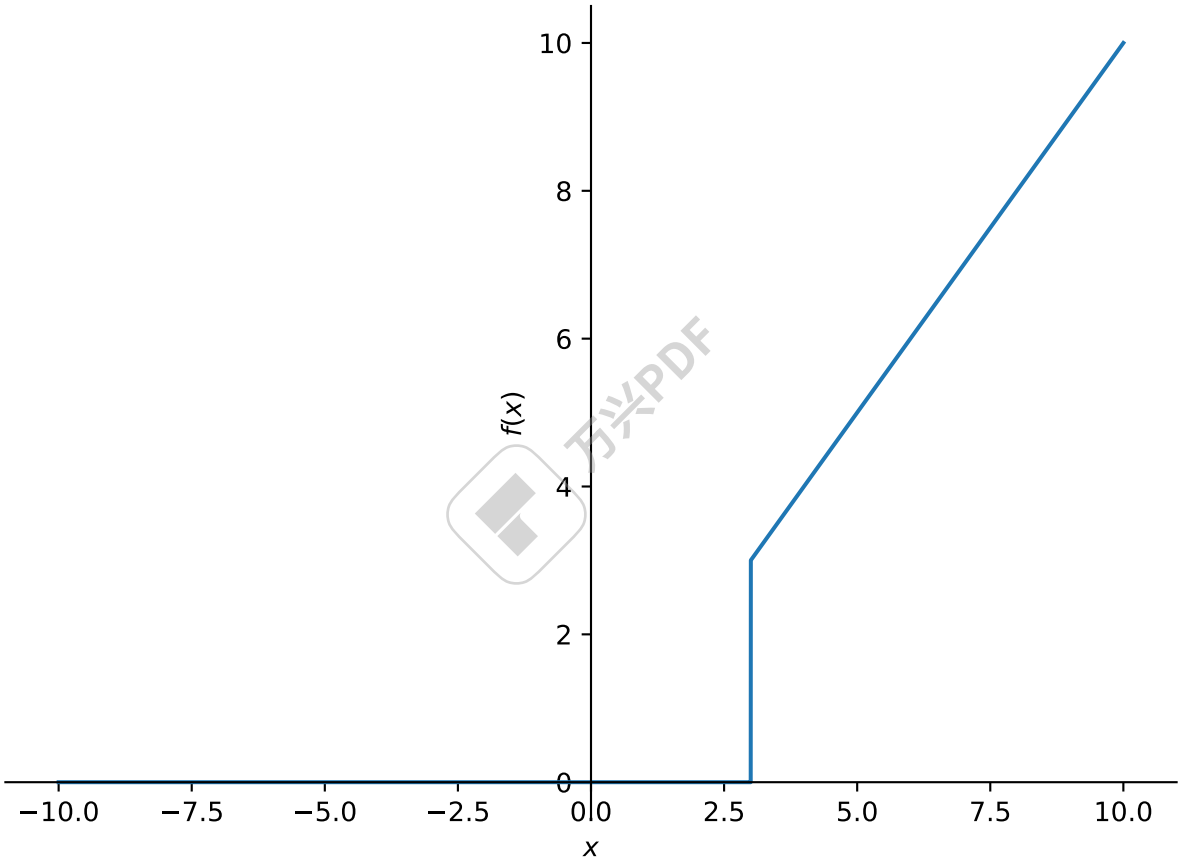
```
>>> b.load
            -1            -2                                  -1
      11·<x>      3·<x - 3>                0          1   43·<x - 6>
    - ──────── - ──────────── + 3·<x - 3>  + <x - 3>  - ──────────
         4            2                                     4
```

```
>>> plot(b.load)
```

```
>>> b.shear_force()
           0            -1                        2            0
      11·<x>    3·<x - 3>                1   <x - 3>    43·<x - 6>
      ──────── + ──────────── - 3·<x - 3>  - ──────── + ──────────
         4            2                        2           4
```

```
>>> b.bending_moment()
           1            0            2          3            1
      11·<x>    3·<x - 3>    3·<x - 3>    <x - 3>    43·<x - 6>
      ──────── + ──────────── - ──────────── - ──────── + ──────────
         4            2            2          6           4
```

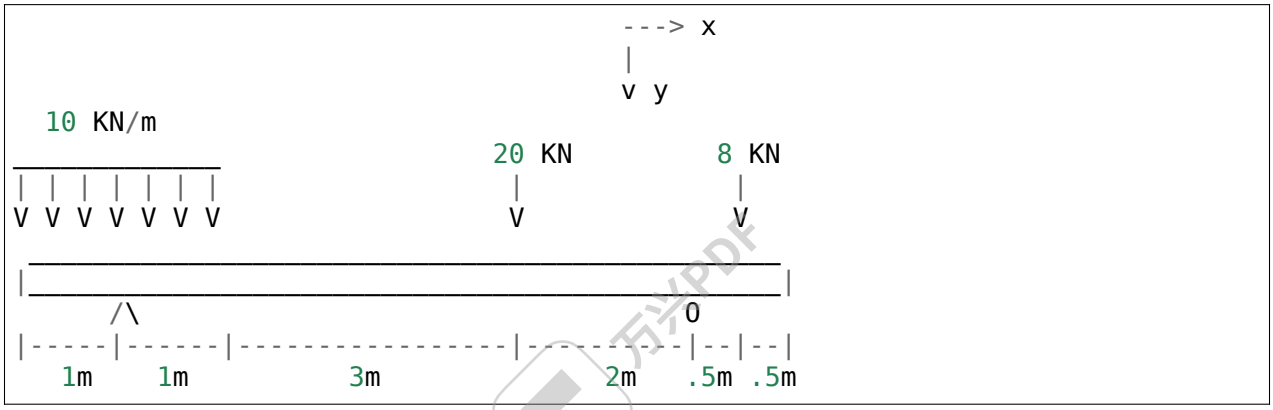
```
>>> b.slope()
           2            1          3          4            2
      11·<x>    3·<x - 3>    <x - 3>    <x - 3>    43·<x - 6>    78
    - ──────── - ──────────── + ──────── + ──────── - ────────── + ──
         8            2          2          24          8         5
    ──────────────────────────────────────────────────────────────────
                                  E·I
```


10
8
6
4
2
0
f(x)
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

```
>>> b.deflection()
                  3            2          4          5             3
    78·x   11·<x>    3·<x - 3>    <x - 3>    <x - 3>    43·<x - 6>
    ──── - ──────── - ────────── + ──────── + ──────── - ──────────
     5        24          4          8         120          24
    ──────────────────────────────────────────────────────────────
                                E·I
```

Example 4

An overhanging beam of length 8 meters is pinned at 1 meter from starting point and supported by a roller 1 meter before the other end. It is subjected to a distributed constant load of 10 KN/m from the starting point till 2 meters away from it. Two point loads of 20KN and 8KN are applied at 5 meters and 7.5 meters away from the starting point respectively.




```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E,I,M,V = symbols('E I M V')
>>> b = Beam(8, E, I)
>>> E,I,R1,R2 = symbols('E I R1 R2')
>>> b.apply_load(R1, 1, -1)
>>> b.apply_load(R2, 7, -1)
>>> b.apply_load(10, 0, 0, end=2)
>>> b.apply_load(20, 5, -1)
>>> b.apply_load(8, 7.5, -1)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R₁: -26, R₂: -22}
>>> b.load
      0             -1             0             -1             -1
↪   -1
10·<x>  - 26·<x - 1>  - 10·<x - 2>  + 20·<x - 5>  - 22·<x - 7>  + 8·<x - 7.
↪5>

>>> b.shear_force()
       1             0             1             0             0
↪ 0
- 10·<x>  + 26·<x - 1>  + 10·<x - 2>  - 20·<x - 5>  + 22·<x - 7>  - 8·<x - 7.
```

(continues on next page)

(continued from previous page)

```
 ↪5>

>>> b.bending_moment()
       2             1             2             1             1             1
- 5·<x>  + 26·<x - 1>  + 5·<x - 2>  - 20·<x - 5>  + 22·<x - 7>  - 8·<x - 7.5>

>>> b.bc_deflection = [(1, 0), (7, 0)]
>>> b.slope()
     3                           3
5·<x>              2   5·<x - 2>               2             2             2
 ↪ 679
──────  - 13·<x - 1>  - ──────────── + 10·<x - 5>  - 11·<x - 7>  + 4·<x - 7.5>
 ↪+ ───
  3                         3
 ↪  24
──────────────────────────────────────────────────────────────────────────────
                                                E·I

>>> b.deflection()
             4             3            4             3             3
 ↪      3
679·x   5·<x>    13·<x - 1>    5·<x - 2>    10·<x - 5>    11·<x - 7>    4·<x -
 ↪ 7.5>    689
─────  + ──────  - ──────────  - ──────────  + ──────────  - ──────────  +
 ↪──────────  - ───
  24       12          3            12            3             3
 ↪3          24
──────────────────────────────────────────────────────────────────────────────
                                                E·I
```

Example 5

A cantilever beam of length 6 meters is under downward distributed constant load with magnitude of 4.0 KN/m from starting point till 2 meters away from it. A ramp load of 1 kN/m/m applied from the mid till the end of the beam. A point load of 12KN is also applied in same direction 4 meters away from start.

```
  ---> x                                .
  |                                   . |
  v y                       12 KN     . | |
                              |     . | | |
                              V   . | | | |
\\\\|    4 KN/m                 . | | | | |
\\\\|_____________             . 1 KN/m/m| |
\\\\|| | | | | |              . V V V V V V V
\\\\|V V V V V V       |----------------|
\\\\|_____________________________________
\\\\|_____________________________________|
\\\\|           :           :           :
\\\\|-----------|-----|-----|-----------|
         2.0 m      1m    1m       2.0 m
```

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E,I,M,V = symbols('E I M V')
>>> b = Beam(6, E, I)
>>> b.apply_load(V, 0, -1)
>>> b.apply_load(M, 0, -2)
>>> b.apply_load(4, 0, 0, end=2)
>>> b.apply_load(12, 4, -1)
>>> b.apply_load(1, 3, 1, end=6)
>>> b.solve_for_reaction_loads(V, M)
>>> b.reaction_loads
{M: 157/2, V: -49/2}
>>> b.load
       -2         -1
157⋅<x>     49⋅<x>          0            0          1             -1
 ↪  0          1
──────── - ──────── + 4⋅<x>  - 4⋅<x - 2>  + <x - 3>  + 12⋅<x - 4>   - 3⋅<x -
 ↪6>  - <x - 6>
   2          2
>>> b.shear_force()
          -1         0                                  2
 ↪              2
   157⋅<x>     49⋅<x>        1            1    <x - 3>             0
 ↪  1    <x - 6>
- ──────── + ──────── - 4⋅<x>  + 4⋅<x - 2>  - ──────── - 12⋅<x - 4>  + 3⋅<x -
 ↪6>  + ────────
     2          2                                 2
 ↪          2
>>> b.bending_moment()
         0         1                                  3
 ↪ 2          3
  157⋅<x>    49⋅<x>         2             2   <x - 3>              1   3⋅<x -
 ↪6>    <x - 6>
- ──────── + ──────── - 2⋅<x>  + 2⋅<x - 2>  - ──────── - 12⋅<x - 4>  +
 ↪──────── + ────────
    2          2                                  6                         2
 ↪         6
>>> b.bc_deflection = [(0, 0)]
>>> b.bc_slope = [(0, 0)]
>>> b.slope()
⎛         1         2        3           3          4
 ↪3          4⎞
⎜  157⋅<x>    49⋅<x>    2⋅<x>    2⋅<x - 2>    <x - 3>             2   <x - 6>
 ↪    <x - 6> ⎟
-⎜- ──────── + ──────── - ──────── + ──────── - ──────── - 6⋅<x - 4>  +
 ↪──────── + ────────⎟
 ⎝     2          4        3          3           24                        2
 ↪       24    ⎠
──────────────────────────────────────────────────────────────────────────────
                                      E⋅I
>>> b.deflection()
⎛         2         3      4          4          5                        4
```

(continues on next page)

(continued from previous page)

```
↪       5⎞
⎜  157·<x>    49·<x>    <x>    <x - 2>    <x - 3>             3   <x - 6>
↪<x - 6> ⎟
-⎜- ──────── + ──────── - ──── + ──────── - ──────── - 2·<x - 4>  + ──────── +␣
↪────────⎟
⎝     4          12      6        6         120                    8       ␣
↪  120   ⎠
──────────────────────────────────────────────────────────────────────────────
                                  E·I
```

Example 6

An overhanging beam of length 11 meters is subjected to a distributed constant load of 2 KN/m from 2 meters away from the starting point till 6 meters away from it. It is pinned at the starting point and is resting over a roller 8 meters away from that end. Also a counter-clockwise moment of 5 KN-m is applied at the overhanging end.

```
                2 KN/m                          ---> x
           ________________                     |
           | | | | | | | | |                    v y
           V V V V V V V V V                         5 KN-m
 ____________________________________________________|
 O___________________________________________________|
/ \                                  /\
 |--------|----------------|----------|---------------|
     2m           4m              2m          3m
```

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> E, I = symbols('E, I')
>>> b = Beam(11, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(2, 2, 0, end=6)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(5, 11, -2)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: -37/8, R2: -27/8}
>>> b.load
        -1                                          -1
  37·<x>              0             0   27·<x - 8>              -2
- ──────── + 2·<x - 2>  - 2·<x - 6>  - ──────────── + 5·<x - 11>
     8                                      8
>>> b.shear_force()
        0                                        0
  37·<x>             1             1   27·<x - 8>              -1
  ──────── - 2·<x - 2>  + 2·<x - 6>  + ──────────── - 5·<x - 11>
     8                                      8
```

(continues on next page)

(continued from previous page)

```
>>> b.bending_moment()
        1                                   1
  37·<x>            2           2   27·<x - 8>               0
  ──────── - <x - 2>  + <x - 6>  + ────────── - 5·<x - 11>
     8                                 8
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.slope()
        2           3           3            2
  37·<x>    <x - 2>    <x - 6>    27·<x - 8>               1
- ──────── + ──────── - ──────── - ────────── + 5·<x - 11>  + 36
     16         3          3           16
──────────────────────────────────────────────────────────────────
                                E·I
>>> b.deflection()
              3           4           4            3             2
       37·<x>    <x - 2>    <x - 6>    9·<x - 8>    5·<x - 11>
36·x - ──────── + ──────── - ──────── - ────────── + ──────────
          48         12          12          16            2
──────────────────────────────────────────────────────────────────
                                E·I
```

Example 7

There is a beam of length `l`, fixed at both ends. A concentrated point load of magnitude `F` is applied in downward direction at mid-point of the beam.

```
                                    ^ y
                                    |
                                    ---> x
\\\\|                    F                    |\\\\
\\\\|                    |                    |\\\\
\\\\|                    V                    |\\\\
\\\\|_________________________________________|\\\\
\\\\|_________________________________________|\\\\
\\\\|                    :                    |\\\\
\\\\|                    :                    |\\\\
\\\\|--------------------|--------------------|\\\\
              l/2                  l/2
```

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I, F = symbols('E I F')
>>> l = symbols('l', positive=True)
>>> b = Beam(l, E, I)
>>> R1,R2 = symbols('R1  R2')
>>> M1, M2 = symbols('M1, M2')
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(M1, 0, -2)
>>> b.apply_load(R2, l, -1)
>>> b.apply_load(M2, l, -2)
```

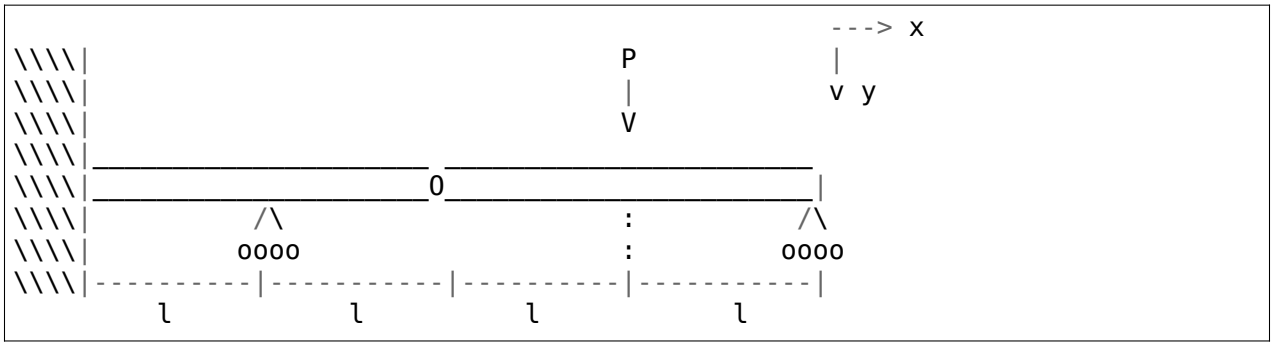
(continues on next page)

(continued from previous page)

```
>>> b.apply_load(-F, l/2, -1)
>>> b.bc_deflection = [(0, 0),(l, 0)]
>>> b.bc_slope = [(0, 0),(l, 0)]
>>> b.solve_for_reaction_loads(R1, R2, M1, M2)
>>> b.reaction_loads
⎡    -F·l       F·l      F      F⎤
⎨M₁: ─────, M₂: ───, R₁: ─, R₂: ─⎬
⎣      8         8       2      2⎦

>>> b.load
          -2                -2        -1                -1              -1
  F·l·⟨x⟩     F·l·⟨-l + x⟩     F·⟨x⟩          l           F·⟨-l + x⟩
- ─────────── + ──────────────── + ──────── - F·⟨- ─ + x⟩   + ─────────────
      8               8             2           2                2

>>> b.shear_force()
          -1                -1        0               0             0
  F·l·⟨x⟩     F·l·⟨-l + x⟩     F·⟨x⟩          l           F·⟨-l + x⟩
  ─────────── - ──────────────── - ──────── + F·⟨- ─ + x⟩  - ─────────────
      8               8             2           2                2

>>> b.bending_moment()
          0                 0         1                 1              1
  F·l·⟨x⟩     F·l·⟨-l + x⟩     F·⟨x⟩          l           F·⟨-l + x⟩
  ─────────── - ──────────────── - ──────── + F·⟨- ─ + x⟩  - ─────────────
      8               8             2           2                2

>>> b.slope()
⎛                                                  2              ⎞
⎜                                                 l               ⎟
⎜         1                 1         2   F·⟨- ─ + x⟩             2⎟
⎜  F·l·⟨x⟩     F·l·⟨-l + x⟩     F·⟨x⟩          2           F·⟨-l + x⟩ ⎟
-⎜─────────── - ──────────────── - ──────── + ──────────── - ──────────⎟
⎝     8               8             4           2                4     ⎠
──────────────────────────────────────────────────────────────────────
                                E·I

>>> b.deflection()
⎛                                                  3              ⎞
⎜                                                 l               ⎟
⎜         2                 2         3   F·⟨- ─ + x⟩             3⎟
⎜  F·l·⟨x⟩     F·l·⟨-l + x⟩     F·⟨x⟩          2           F·⟨-l + x⟩ ⎟
-⎜─────────── - ──────────────── - ──────── + ──────────── - ──────────⎟
⎝    16              16            12          6                12    ⎠
──────────────────────────────────────────────────────────────────────
                                E·I
```

Example 8

There is a beam of length 4*l, having a hinge connector at the middle. It is having a fixed support at the start and also has two rollers at a distance of l and 4*l from the starting point. A concentrated point load P is also applied at a distance of 3*l from the starting point.




```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E I')
>>> l = symbols('l', positive=True)
>>> R1, M1, R2, R3, P = symbols('R1 M1 R2 R3 P')
>>> b1 = Beam(2*l, E, I)
>>> b2 = Beam(2*l, E, I)
>>> b = b1.join(b2, "hinge")
>>> b.apply_load(M1, 0, -2)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, l, -1)
>>> b.apply_load(R3, 4*l, -1)
>>> b.apply_load(P, 3*l, -1)
>>> b.bc_slope = [(0, 0)]
>>> b.bc_deflection = [(0, 0), (l, 0), (4*l, 0)]
>>> b.solve_for_reaction_loads(M1, R1, R2, R3)
>>> b.reaction_loads
⎡    -P·l       3·P       -5·P       -P ⎤
⎨M₁: ────, R₁: ───, R₂: ────, R₃: ───⎬
⎣     4         4         4         2  ⎦

>>> b.load
          -2           -1                -1                                  -1
  P·l·<x>     3·P·<x>     5·P·<-l + x>                  -1   P·<-4·l + x>
- ──────── + ──────── - ──────────── + P·<-3·l + x>    - ──────────────
     4           4            4                                  2

>>> b.shear_force()
          -1          0               0                                    0
  P·l·<x>     3·P·<x>    5·P·<-l + x>                  0   P·<-4·l + x>
  ──────── - ──────── + ──────────── - P·<-3·l + x>  + ──────────────
     4           4            4                                  2

>>> b.bending_moment()
         0          1               1                                    1
```

(continues on next page)

(continued from previous page)

```
  P·l·<x>     3·P·<x>     5·P·<-l + x>                  1   P·<-4·l + x>
  ─────── - ─────── + ──────────── - P·<-3·l + x>  + ──────────── 
     4          4            4                          2

>>> b.slope()
⎛     2               2              2              2⎞              ⎛
 ↪    1          2               2              2⎞       ⎛        1
 ↪ 2              2              2⎞
⎜5·P·l    P·<-2·l + x>    P·<-3·l + x>    P·<-4·l + x> ⎟          0   ⎜  P·l·
 ↪<x>    3·P·<x>    5·P·<-l + x>    P·<-2·l + x> ⎟    0   ⎜  P·l·<x>    3·P·
 ↪<x>    5·P·<-l + x>    P·<-2·l + x> ⎟          0
⎜────── - ──────────── + ──────────── - ────────────⎟·<-2·l + x>    ⎟-
 ↪──────── + ──────── - ──────────── + ────────────⎟·<x>    ⎟- ──────── +
 ↪──────── - ──────────── + ────────────⎟·<-2·l + x>
⎝  48          4              2              4     ⎠              ⎝
 ↪4          8              8              4     ⎠       ⎝     4          8
 ↪            8              4     ⎠
──────────────────────────────────────────────────────────────────────── +
 ↪──────────────────────────────────────────────────────────────────── -
 ↪──────────────────────────────────────────────────────────────────────
                                E·I
 ↪                    E·I
 ↪            E·I
>>> b.deflection()
⎛         2       3              3              3⎞       ⎛        2
 ↪  3              3              3⎞              ⎛     3       2
 ↪                 3              3              3⎞
⎜  P·l·<x>    P·<x>    5·P·<-l + x>    P·<-2·l + x> ⎟    0   ⎜  P·l·<x>    P·
 ↪<x>    5·P·<-l + x>    P·<-2·l + x> ⎟          0   ⎜7·P·l    5·P·l ·(-2·l
 ↪+ x)    P·<-2·l + x>    P·<-3·l + x>    P·<-4·l + x> ⎟          0
⎜- ──────── + ─────── - ──────────── + ────────────⎟·<x>    ⎟- ──────── +
 ↪─────── - ──────────── + ────────────⎟·<-2·l + x>    ⎟─────── +
 ↪──────────────── - ──────────── + ──────────── - ────────────⎟·<-2·l +
 ↪x>
⎝     8         8           24             12     ⎠       ⎝     8
 ↪8           24             12     ⎠             ⎝  24            48
 ↪            12             6              12     ⎠
──────────────────────────────────────────────────────────────────── -
 ↪──────────────────────────────────────────────────────────────────── +
 ↪──────────────────────────────────────────────────────────────────────
                           E·I
 ↪              E·I
 ↪                 E·I
```

Example 9

There is a cantilever beam of length 4 meters. For first 2 meters its moment of inertia is `1.5*I` and `I` for the rest. A pointload of magnitude 20 N is applied from the top at its free end.

```
                                           ---> x
\\\\|                                      |
\\\\|                               20 N   v y
\\\\|________________                |
\\\\|                |_______________V
\\\\|      1.5*I      _______I_______|
\\\\|________________|
\\\\|                                :
\\\\|----------------|---------------|
           2.0 m            2.0 m
```

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b1 = Beam(2, E, 1.5*I)
>>> b2 = Beam(2, E, I)
>>> b = b1.join(b2, "fixed")
>>> b.apply_load(20, 4, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 0, -2)
>>> b.bc_slope = [(0, 0)]
>>> b.bc_deflection = [(0, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.load
      -2          -1             -1
80⋅<x>   - 20⋅<x>   + 20⋅<x - 4>
>>> b.shear_force()
        -1          0              0
- 80⋅<x>   + 20⋅<x>   - 20⋅<x - 4>
>>> b.bending_moment()
        0          1              1
- 80⋅<x>  + 20⋅<x>  - 20⋅<x - 4>
>>> b.slope()
⎛          1         2              2           ⎞
⎜  - 80⋅<x>  + 10⋅<x>  - 10⋅<x - 4>    120      ⎟
⎜  ────────────────────────────────── + ───      ⎟                            ␣
↪ ⎛         1         2              2⎞    0                      ⎛         1 ␣
↪      2             2⎞         0
⎜                  I                    I    80.0⎟        0    0.
↪666666666666667⋅⎝- 80⋅<x>  + 10⋅<x>  - 10⋅<x - 4> ⎠⋅<x>     0.
↪666666666666667⋅⎝- 80⋅<x>  + 10⋅<x>  - 10⋅<x - 4> ⎠⋅<x - 2>
⎜- ─────────────────────────────────────── + ───────⎟⋅<x - 2>   -␣
↪────────────────────────────────────────────────────────── +␣
↪────────────────────────────────────────────────────────────
⎝                    E                      E⋅I ⎠                           ␣
↪           E⋅I                                                            ␣
```

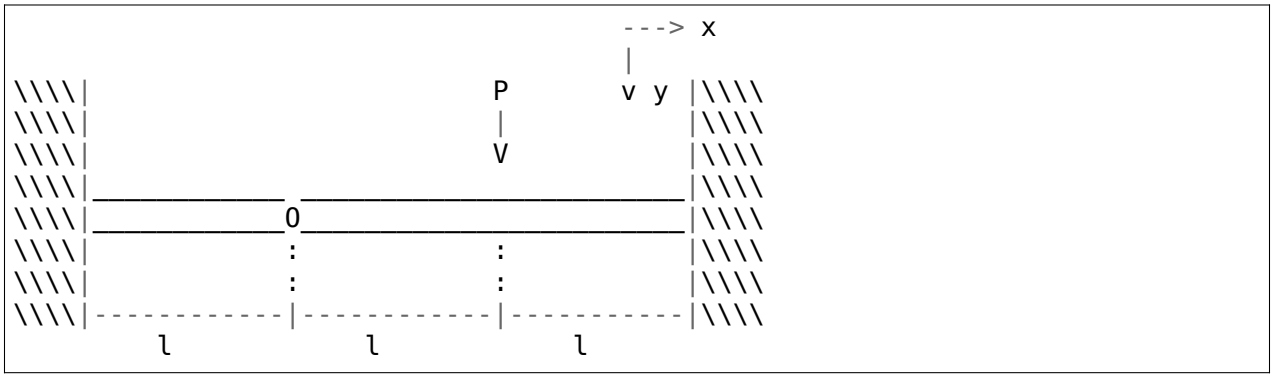
(continues on next page)

(continued from previous page)

```
↪E·I
```

Example 10

A combined beam, with constant flexural rigidity `E*I`, is formed by joining a Beam of length `2*l` to the right of another Beam of length `l`. The whole beam is fixed at both of its ends. A point load of magnitude `P` is also applied from the top at a distance of `2*l` from starting point.




```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> l = symbols('l', positive=True)
>>> b1 = Beam(l ,E,I)
>>> b2 = Beam(2*l ,E,I)
>>> b = b1.join(b2,"hinge")
>>> M1, A1, M2, A2, P = symbols('M1 A1 M2 A2 P')
>>> b.apply_load(A1, 0, -1)
>>> b.apply_load(M1, 0 ,-2)
>>> b.apply_load(P, 2*l, -1)
>>> b.apply_load(A2, 3*l, -1)
>>> b.apply_load(M2, 3*l, -2)
>>> b.bc_slope=[(0, 0), (3*l, 0)]
>>> b.bc_deflection=[(0, 0), (3*l, 0)]
>>> b.solve_for_reaction_loads(M1, A1, M2, A2)
>>> b.reaction_loads
⎡    -5·P       -13·P       5·P·l      -4·P·l ⎤
⎨A₁: ─────, A₂: ──────, M₁: ─────, M₂: ──────⎬
⎣      18         18          18          9   ⎦

>>> b.load
         -2                    -2          -1                                  ␣
↪ -1
5·P·l·<x>     4·P·l·<-3·l + x>     5·P·<x>                  -1   13·P·<-3·l +␣
↪x>
────────── - ──────────────── - ────────── + P·<-2·l + x>   -␣
↪────────────────
    18              9               18                                    18
```

(continues on next page)

(continued from previous page)

```
>>> b.shear_force()
          -1                   -1          0                                  ␣
↪ 0
  5·P·l·<x>     4·P·l·<-3·l + x>     5·P·<x>                0   13·P·<-3·l + ␣
↪x>
- ──────────── + ──────────────────── + ──────── - P·<-2·l + x>  + ␣
↪──────────────────
      18                9                18                                18

>>> b.bending_moment()
          0                   0          1                                  1
  5·P·l·<x>     4·P·l·<-3·l + x>     5·P·<x>                1   13·P·<-3·l + x>
- ──────────── + ──────────────────── + ──────── - P·<-2·l + x>  + ──────────────
      18                9                18                                18

>>> b.slope()
⎛         1          2                2⎞       ⎛         1          2
↪      2⎞             ⎛   2                  1              2
↪ 2                2⎞
⎜5·P·l·<x>    5·P·<x>    5·P·<-l + x> ⎟    0   ⎜5·P·l·<x>    5·P·<x>    5·P·<-
↪l + x> ⎟         0   ⎜P·l    4·P·l·<-3·l + x>    5·P·<-l + x>    P·<-2·l + ␣
↪x>    13·P·<-3·l + x> ⎟         0
⎜──────────── - ──────── + ──────────────⎟·<x>⎜──────────── - ──────── + ␣
↪──────────────⎟·<-l + x>    ⎜──── - ──────────────── - ──────────────── + ␣
↪──────────────── - ──────────────⎟·<-l + x>
⎝    18          36             36     ⎠       ⎝    18          36
↪36     ⎠             ⎝ 18          9                36              2
↪             36       ⎠
──────────────────────────────────────────── - ␣
↪──────────────────────────────────────────── + ␣
↪──────────────────────────────────────────────────────────────────────────────
                 E·I                                                E·I
↪                                                                E·I

>>> b.deflection()
⎛         2          3                3⎞       ⎛         2          3
↪      3⎞             ⎛     3      2                          2
↪    3                3                   3⎞
⎜5·P·l·<x>    5·P·<x>    5·P·<-l + x> ⎟    0   ⎜5·P·l·<x>    5·P·<x>    5·P·<-
↪l + x> ⎟         0   ⎜5·P·l    P·l ·(-l + x)   2·P·l·<-3·l + x>    5·P·<-l ␣
↪+ x>    P·<-2·l + x>    13·P·<-3·l + x> ⎟         0
⎜──────────── - ──────── + ──────────────⎟·<x>    ⎜──────────── - ──────── + ␣
↪──────────────⎟·<-l + x>    ⎜──────── + ──────────────── - ──────────────── - ␣
↪──────────────── + ──────────────── - ──────────────⎟·<-l + x>
⎝    36          108            108    ⎠       ⎝    36          108
↪108     ⎠             ⎝  54            18                9
↪108              6               108      ⎠
──────────────────────────────────────────── - ␣
↪──────────────────────────────────────────── + ␣
↪──────────────────────────────────────────────────────────────────────────────
```

(continues on next page)

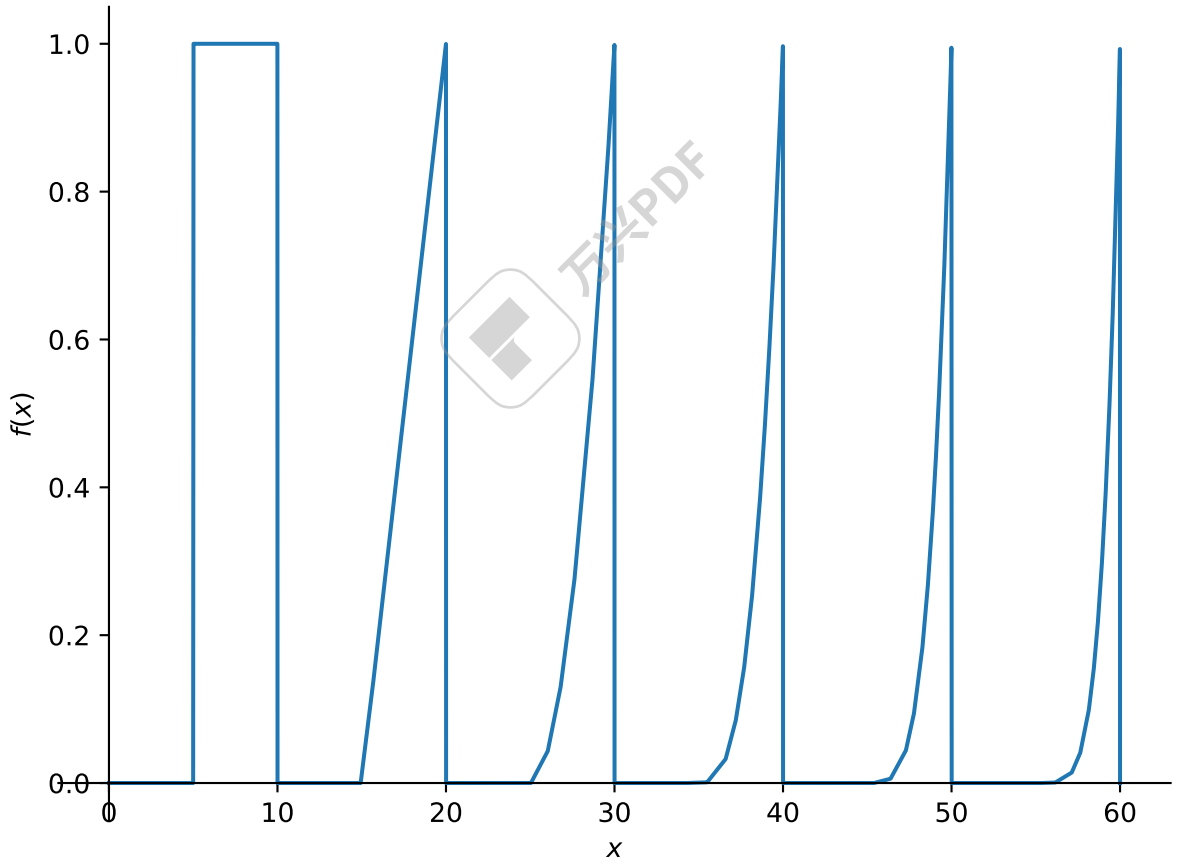
(continued from previous page)

```
                    E·I                                                E·I
                                                                          E·I
↪
```

Example 11

Any type of load defined by a polynomial can be applied to the beam. This allows approximation of arbitrary load distributions. The following example shows six truncated polynomial loads across the surface of a beam.

```
>>> n = 6
>>> b = Beam(10*n, E, I)
>>> for i in range(n):
...     b.apply_load(1 / (5**i), 10*i + 5, i, end=10*i + 10)
>>> plot(b.load, (x, 0, 10*n))
```



Truss

Truss (Docstrings)

Truss

This module can be used to solve problems related to 2D Trusses.

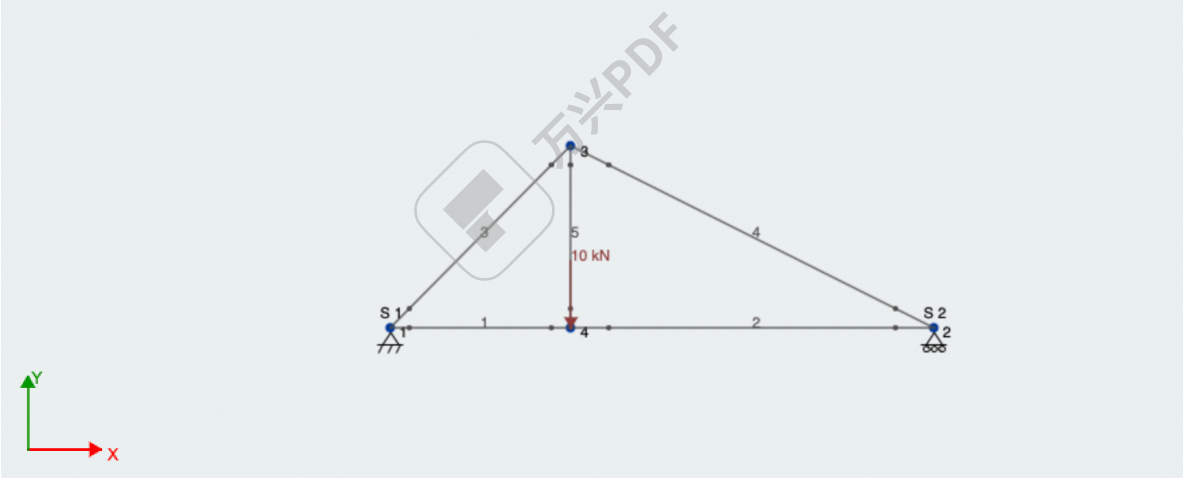
class sympy.physics.continuum_mechanics.truss.**Truss**

A Truss is an assembly of members such as beams, connected by nodes, that create a rigid structure. In engineering, a truss is a structure that consists of two-force members only.

Trusses are extremely important in engineering applications and can be seen in numerous real-world applications like bridges.

Examples

There is a Truss consisting of four nodes and five members connecting the nodes. A force P acts downward on the node D and there also exist pinned and roller joints on the nodes A and B respectively.




```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node("node_1", 0, 0)
>>> t.add_node("node_2", 6, 0)
>>> t.add_node("node_3", 2, 2)
>>> t.add_node("node_4", 2, 0)
>>> t.add_member("member_1", "node_1", "node_4")
>>> t.add_member("member_2", "node_2", "node_4")
>>> t.add_member("member_3", "node_1", "node_3")
>>> t.add_member("member_4", "node_2", "node_3")
>>> t.add_member("member_5", "node_3", "node_4")
>>> t.apply_load("node_4", magnitude=10, direction=270)
>>> t.apply_support("node_1", type="fixed")
>>> t.apply_support("node_2", type="roller")
```

add_member(*label, start, end*)

This method adds a member between any two nodes in the given truss.

Parameters

label: String or Symbol

The label for a member. It is the only way to identify a particular member.

start: String or Symbol

The label of the starting point/node of the member.

end: String or Symbol

The label of the ending point/node of the member.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.add_node('C', 2, 2)
>>> t.add_member('AB', 'A', 'B')
>>> t.members
{'AB': ['A', 'B']}
```

add_node(*label, x, y*)

This method adds a node to the truss along with its name/label and its location.

Parameters

label: String or a Symbol

The label for a node. It is the only way to identify a particular node.

x: Sympifyable

The x-coordinate of the position of the node.

y: Sympifyable

The y-coordinate of the position of the node.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.nodes
[('A', 0, 0)]
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
```

apply_load(*location*, *magnitude*, *direction*)

This method applies an external load at a particular node

Parameters

location: String or Symbol

Label of the Node at which load is applied.

magnitude: Sympifyable

Magnitude of the load applied. It must always be positive and any changes in the direction of the load are not reflected here.

direction: Sympifyable

The angle, in degrees, that the load vector makes with the horizontal in the counter-clockwise direction. It takes the values 0 to 360, inclusive.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> from sympy import symbols
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> P = symbols('P')
>>> t.apply_load('A', P, 90)
>>> t.apply_load('A', P/2, 45)
>>> t.apply_load('A', P/4, 90)
>>> t.loads
{'A': [[P, 90], [P/2, 45], [P/4, 90]]}
```

apply_support(*location*, *type*)

This method adds a pinned or roller support at a particular node

Parameters

location: String or Symbol

Label of the Node at which support is added.

type: String

Type of the support being provided at the node.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.apply_support('A', 'pinned')
>>> t.supports
{'A': 'pinned'}
```

change_member_label(*label*, *new_label*)

This method changes the label of a member.

Parameters

label: String or Symbol

The label of the member for which the label has to be changed.

new_label: String or Symbol

The new label of the member.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
>>> t.change_node_label('A', 'C')
>>> t.nodes
[('C', 0, 0), ('B', 3, 0)]
>>> t.add_member('BC', 'B', 'C')
>>> t.members
{'BC': ['B', 'C']}
>>> t.change_member_label('BC', 'BC_new')
>>> t.members
{'BC_new': ['B', 'C']}
```

change_node_label(*label*, *new_label*)

This method changes the label of a node.

Parameters

label: String or Symbol

The label of the node for which the label has to be changed.

new_label: String or Symbol

The new label of the node.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
>>> t.change_node_label('A', 'C')
>>> t.nodes
[('C', 0, 0), ('B', 3, 0)]
```

property internal_forces
Returns the internal forces for all members which are all initialized to 0.

property loads
Returns the loads acting on the truss.

property member_labels
Returns the members of the truss along with the start and end points.

property members
Returns the members of the truss along with the start and end points.

property node_labels
Returns the node labels of the truss.

property node_positions
Returns the positions of the nodes of the truss.

property nodes
Returns the nodes of the truss along with their positions.

property reaction_loads
Returns the reaction forces for all supports which are all initialized to 0.

remove_load(*location*, *magnitude*, *direction*)
This method removes an already present external load at a particular node

Parameters
location: String or Symbol
Label of the Node at which load is applied and is to be removed.

magnitude: Sympifyable
Magnitude of the load applied.

direction: Sympifyable
The angle, in degrees, that the load vector makes with the horizontal in the counter-clockwise direction. It takes the values 0 to 360, inclusive.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> from sympy import symbols
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> P = symbols('P')
>>> t.apply_load('A', P, 90)
>>> t.apply_load('A', P/2, 45)
>>> t.apply_load('A', P/4, 90)
>>> t.loads
{'A': [[P, 90], [P/2, 45], [P/4, 90]]}
>>> t.remove_load('A', P/4, 90)
```

(continues on next page)

(continued from previous page)

```
>>> t.loads
{'A': [[P, 90], [P/2, 45]]}
```

remove_member(*label*)

This method removes a member from the given truss.

Parameters
label: String or Symbol

The label for the member to be removed.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.add_node('C', 2, 2)
>>> t.add_member('AB', 'A', 'B')
>>> t.add_member('AC', 'A', 'C')
>>> t.add_member('BC', 'B', 'C')
>>> t.members
{'AB': ['A', 'B'], 'AC': ['A', 'C'], 'BC': ['B', 'C']}
>>> t.remove_member('AC')
>>> t.members
{'AB': ['A', 'B'], 'BC': ['B', 'C']}
```

remove_node(*label*)

This method removes a node from the truss.

Parameters
label: String or Symbol

The label of the node to be removed.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.nodes
[('A', 0, 0)]
>>> t.add_node('B', 3, 0)
>>> t.nodes
[('A', 0, 0), ('B', 3, 0)]
>>> t.remove_node('A')
>>> t.nodes
[('B', 3, 0)]
```

remove_support(*location*)

This method removes support from a particular node

Parameters
location: String or Symbol

Label of the Node at which support is to be removed.

Examples

```
>>> from sympy.physics.continuum_mechanics.truss import Truss
>>> t = Truss()
>>> t.add_node('A', 0, 0)
>>> t.add_node('B', 3, 0)
>>> t.apply_support('A', 'pinned')
>>> t.supports
{'A': 'pinned'}
>>> t.remove_support('A')
>>> t.supports
{}
```

property supports

Returns the nodes with provided supports along with the kind of support provided i.e. pinned or roller.

5.8.7 Utilities

Contents

Testing

This module contains code for running the tests in SymPy.

Contents:

pytest

py.test hacks to support XFAIL/XPASS

`sympy.testing.pytest.`**`SKIP`**(*reason*)

Similar to `skip()`, but this is a decorator.

`sympy.testing.pytest.`**`nocache_fail`**(*func*)

Dummy decorator for marking tests that fail when cache is disabled

`sympy.testing.pytest.`**`raises`**(*expectedException*, *code=None*)

Tests that `code` raises the exception `expectedException`.

`code` may be a callable, such as a lambda expression or function name.

If `code` is not given or None, `raises` will return a context manager for use in `with` statements; the code to execute then comes from the scope of the `with`.

`raises()` does nothing if the callable raises the expected exception, otherwise it raises an AssertionError.

Examples

```
>>> from sympy.testing.pytest import raises
```

```
>>> raises(ZeroDivisionError, lambda: 1/0)
<ExceptionInfo ZeroDivisionError(...)>
>>> raises(ZeroDivisionError, lambda: 1/2)
Traceback (most recent call last):
...
Failed: DID NOT RAISE
```

```
>>> with raises(ZeroDivisionError):
...     n = 1/0
>>> with raises(ZeroDivisionError):
...     n = 1/2
Traceback (most recent call last):
...
Failed: DID NOT RAISE
```

Note that you cannot test multiple statements via `with raises`:

```
>>> with raises(ZeroDivisionError):
...     n = 1/0    # will execute and raise, aborting the ``with``
...     n = 9999/0 # never executed
```

This is just what `with` is supposed to do: abort the contained statement sequence at the first exception and let the context manager deal with the exception.

To test multiple statements, you'll need a separate `with` for each:

```
>>> with raises(ZeroDivisionError):
...     n = 1/0    # will execute and raise
>>> with raises(ZeroDivisionError):
...     n = 9999/0 # will also execute and raise
```

sympy.testing.pytest.**warns**(*warningcls*, ***, *match=''*, *test_stacklevel=True*)

Like raises but tests that warnings are emitted.

```
>>> from sympy.testing.pytest import warns
>>> import warnings
```

```
>>> with warns(UserWarning):
...     warnings.warn('deprecated', UserWarning, stacklevel=2)
```

```
>>> with warns(UserWarning):
...     pass
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type UserWarning    was emitted.␣
↪The list of emitted warnings is: [].
```

`test_stacklevel` makes it check that the `stacklevel` parameter to `warn()` is set so that the warning shows the user line of code (the code under the warns() context manager).

Set this to False if this is ambiguous or if the context manager does not test the direct user code that emits the warning.

If the warning is a `SymPyDeprecationWarning`, this additionally tests that the `active_deprecations_target` is a real target in the `active-deprecations.md` file.

`sympy.testing.pytest.`**`warns_deprecated_sympy`**`()`

Shorthand for `warns(SymPyDeprecationWarning)`

This is the recommended way to test that `SymPyDeprecationWarning` is emitted for deprecated features in SymPy. To test for other warnings use `warns`. To suppress warnings without asserting that they are emitted use `ignore_warnings`.

Note: `warns_deprecated_sympy()` is only intended for internal use in the SymPy test suite to test that a deprecation warning triggers properly. All other code in the SymPy codebase, including documentation examples, should not use deprecated behavior.

If you are a user of SymPy and you want to disable SymPyDeprecationWarnings, use `warnings` filters (see *Silencing SymPy Deprecation Warnings* (page 164)).

```
>>> from sympy.testing.pytest import warns_deprecated_sympy
>>> from sympy.utilities.exceptions import sympy_deprecation_warning
>>> with warns_deprecated_sympy():
...     sympy_deprecation_warning("Don't use",
...         deprecated_since_version="1.0",
...         active_deprecations_target="active-deprecations")
```

```
>>> with warns_deprecated_sympy():
...     pass
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type     SymPyDeprecationWarning␣
↪was emitted. The list of emitted warnings is: [].
```

Note: Sometimes the stacklevel test will fail because the same warning is emitted multiple times. In this case, you can use *sympy.utilities.exceptions.ignore_warnings()* (page 2067) in the code to prevent the `SymPyDeprecationWarning` from being emitted again recursively. In rare cases it is impossible to have a consistent `stacklevel` for deprecation warnings because different ways of calling a function will produce different call stacks.. In those cases, use `warns(SymPyDeprecationWarning)` instead.

See also:

sympy.utilities.exceptions.SymPyDeprecationWarning (page 2066), *sympy.utilities.exceptions.sympy_deprecation_warning* (page 2067), *sympy.utilities.decorator.deprecated* (page 2057)

Randomised Testing

Deprecated since version 1.10: `sympy.testing.randtest` functions have been moved to *sympy.core.random* (page 1077).

Run Tests

This is our testing framework.

Goals:

- it should be compatible with py.test and operate very similarly (or identically)
- does not require any external dependencies
- preferably all the functionality should be in this file only
- no magic, just import the test file and execute the test functions, that's it
- portable

class sympy.testing.runtests.**PyTestReporter**(*verbose=False*, *tb='short'*, *colors=True*, *force_colors=False*, *split=None*)

Py.test like reporter. Should produce output identical to py.test.

write(*text*, *color=''*, *align='left'*, *width=None*, *force_colors=False*)

Prints a text on the screen.

It uses sys.stdout.write(), so no readline library is necessary.

Parameters

color : choose from the colors below, "" means default color

align : "left"/"right", "left" is a normal print, "right" is aligned on the right-hand side of the screen, filled with spaces if necessary

width : the screen width

class sympy.testing.runtests.**Reporter**

Parent class for all reporters.

class sympy.testing.runtests.**SymPyDocTestFinder**(*verbose=False*, *parser=<doctest.DocTestParser object>*, *recurse=True*, *exclude_empty=True*)

A class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. Doctests can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

Modified from doctest's version to look harder for code that appears comes from a different module. For example, the @vectorize decorator makes it look like functions come from multidimensional.py even though their code exists elsewhere.

class sympy.testing.runtests.**SymPyDocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A class used to run DocTest test cases, and accumulate statistics. The `run` method is used to process a single DocTest case. It returns a tuple `(f, t)`, where `t` is the number of test cases tried, and `f` is the number of test cases that failed.

Modified from the doctest version to not reset the sys.displayhook (see issue 5140).

See the docstring of the original DocTestRunner for more information.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

Run the examples in `test`, and display the results using the writer function `out`.

The examples are run in the namespace `test.globs`. If `clear_globs` is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use `clear_globs=False`.

`compileflags` gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to `globs`.

The output of each example is checked using `SymPyDocTestRunner.check_output`, and the results are formatted by the `SymPyDocTestRunner.report_*` methods.

class sympy.testing.runtests.**SymPyOutputChecker**

Compared to the OutputChecker from the stdlib our OutputChecker class supports numerical comparison of floats occurring in the output of the doctest examples

check_output(*want*, *got*, *optionflags*)

Return True iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See the documentation for *TestRunner* for more information about option flags.

class sympy.testing.runtests.**SymPyTestResults**(*failed*, *attempted*)

attempted

Alias for field number 1

failed

Alias for field number 0

sympy.testing.runtests.**convert_to_native_paths**(*lst*)

Converts a list of '/' separated paths into a list of native (os.sep separated) paths and converts to lowercase if the system is case insensitive.

sympy.testing.runtests.**doctest**(**paths*, *subprocess=True*, *rerun=0*, ***kwargs*)

Runs doctests in all *.py files in the SymPy directory which match any of the given strings in `paths` or all tests if paths=[].

Notes:

- Paths can be entered in native system format or in unix, forward-slash format.
- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.doctest()
```

Run one file:

```
>>> sympy.doctest("sympy/core/basic.py")
>>> sympy.doctest("polynomial.rst")
```

Run all tests in sympy/functions/ and some particular file:

```
>>> sympy.doctest("/functions", "basic.py")
```

Run any file having polynomial in its name, doc/src/modules/polynomial.rst, sympy/functions/special/polynomials.py, and sympy/polys/polynomial.py:

```
>>> sympy.doctest("polynomial")
```

The `split` option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. `split` should be a string of the form 'a/b', which will run part `a` of `b`. Note that the regular doctests and the Sphinx doctests are split independently. For instance, to run the first half of the test suite:

```
>>> sympy.doctest(split='1/2')
```

The `subprocess` and `verbose` options are the same as with the function `test()`. See the docstring of that function for more information.

sympy.testing.runtests.**get_sympy_dir**()

Returns the root SymPy directory and set the global value indicating whether the system is case sensitive or not.

sympy.testing.runtests.**raise_on_deprecated**()

Context manager to make DeprecationWarning raise an error

This is to catch SymPyDeprecationWarning from library code while running tests and doctests. It is important to use this context manager around each individual test/doctest in case some tests modify the warning filters.

sympy.testing.runtests.**run_all_tests**(*test_args=()*, *test_kwargs=None*, *doctest_args=()*, *doctest_kwargs=None*, *examples_args=()*, *examples_kwargs=None*)

Run all tests.

Right now, this runs the regular tests (bin/test), the doctests (bin/doctest), and the examples (examples/all.py).

This is what `setup.py test` uses.

You can pass arguments and keyword arguments to the test functions that support them (for now, test, doctest, and the examples). See the docstrings of those functions for a description of the available options.

For example, to run the solvers tests with colors turned off:

```
>>> from sympy.testing.runtests import run_all_tests
>>> run_all_tests(test_args=("solvers",),
... test_kwargs={"colors:False"})
```

sympy.testing.runtests.**run_in_subprocess_with_hash_randomization**(*function*, *function_args=()*, *function_kwargs=None*, *command='/opt/hostedtoolcache/Python/3.10.8/x64/bin/python'*, *module='sympy.testing.runtests'*, *force=False*)

Run a function in a Python subprocess with hash randomization enabled.

If hash randomization is not supported by the version of Python given, it returns False. Otherwise, it returns the exit value of the command. The function is passed to sys.exit(), so the return value of the function will be the return value.

The environment variable PYTHONHASHSEED is used to seed Python's hash randomization. If it is set, this function will return False, because starting a new subprocess is unnecessary in that case. If it is not set, one is set at random, and the tests are run. Note that if this environment variable is set when Python starts, hash randomization is automatically enabled. To force a subprocess to be created even if PYTHONHASHSEED is set, pass `force=True`. This flag will not force a subprocess in Python versions that do not support hash randomization (see below), because those versions of Python do not support the `-R` flag.

`function` should be a string name of a function that is importable from the module `module`, like "_test". The default for `module` is "sympy.testing.runtests". `function_args` and `function_kwargs` should be a repr-able tuple and dict, respectively. The default Python command is sys.executable, which is the currently running Python command.

This function is necessary because the seed for hash randomization must be set by the environment variable before Python starts. Hence, in order to use a predetermined seed for tests, we must start Python in a separate subprocess.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

Examples

```
>>> from sympy.testing.runtests import (
... run_in_subprocess_with_hash_randomization)
>>> # run the core tests in verbose mode
>>> run_in_subprocess_with_hash_randomization("_test",
... function_args=("core",),
... function_kwargs={'verbose': True})
# Will return 0 if sys.executable supports hash randomization and tests
# pass, 1 if they fail, and False if it does not support hash
# randomization.
```

sympy.testing.runtests.**split_list**(*l*, *split*, *density=None*)

Splits a list into part a of b

split should be a string of the form 'a/b'. For instance, '1/3' would give the split one of three.

If the length of the list is not divisible by the number of splits, the last split will have more items.

density may be specified as a list. If specified, tests will be balanced so that each split has as equal-as-possible amount of mass according to *density*.

```
>>> from sympy.testing.runtests import split_list
>>> a = list(range(10))
>>> split_list(a, '1/3')
[0, 1, 2]
>>> split_list(a, '2/3')
[3, 4, 5]
>>> split_list(a, '3/3')
[6, 7, 8, 9]
```

sympy.testing.runtests.**sympytestfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=<doctest.DocTestParser object>*, *encoding=None*)

Test examples in the given file. Return (#failures, #tests).

Optional keyword arg `module_relative` specifies how filenames should be interpreted:

- If `module_relative` is True (the default), then `filename` specifies a module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure os-independence, `filename` should use "/" characters to separate path segments, and should not be an absolute path (i.e., it may not begin with "/").
- If `module_relative` is False, then `filename` specifies an os-specific path. The path may be absolute or relative (to the current working directory).

Optional keyword arg `name` gives the name of the test; by default use the file's basename.

Optional keyword argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module relative filename. If no package is specified, then the calling module's directory is used as the base directory for module relative filenames. It is an error to specify `package` if `module_relative` is False.

Optional keyword arg `globs` gives a dict to be used as the globals when executing examples; by default, use {}. A copy of this dict is actually used for each docstring, so that each docstring's examples start with a clean slate.

Optional keyword arg `extraglobs` gives a dictionary that should be merged into the globals that are used to execute examples. By default, no extra globals are used.

Optional keyword arg `verbose` prints lots of stuff if true, prints only failures if false; by default, it's true iff "-v" is in sys.argv.

Optional keyword arg `report` prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else very brief (in fact, empty if all tests passed).

Optional keyword arg `optionflags` or's together module constants, and defaults to 0. Possible values (see the docs for details):

- DONT_ACCEPT_TRUE_FOR_1
- DONT_ACCEPT_BLANKLINE
- NORMALIZE_WHITESPACE
- ELLIPSIS
- SKIP
- IGNORE_EXCEPTION_DETAIL
- REPORT_UDIFF
- REPORT_CDIFF
- REPORT_NDIFF
- REPORT_ONLY_FIRST_FAILURE

Optional keyword arg `raise_on_error` raises an exception on the first unexpected exception or failure. This allows failures to be post-mortem debugged.

Optional keyword arg `parser` specifies a DocTestParser (or subclass) that should be used to extract tests from the files.

Optional keyword arg `encoding` specifies an encoding that should be used to convert the file to unicode.

Advanced tomfoolery: testmod runs methods of a local instance of class doctest.Tester, then merges the results into (or creates) global Tester instance doctest.master. Methods of doctest.master can be called directly too, if you want to do something unusual. Passing report=0 to testmod is especially useful then, to delay displaying a summary. Invoke doctest.master.summarize(verbose) when you're done fiddling.

`sympy.testing.runtests.`**`test`**(**paths*, *subprocess=True*, *rerun=0*, ***kwargs*)

Run tests in the specified test_*.py files.

Tests in a particular test_*.py file are run if any of the given strings in `paths` matches a part of the test file's path. If `paths=[]`, tests in all test_*.py files are run.

Notes:

- If sort=False, tests are run in random order (not default).
- Paths can be entered in native system format or in unix, forward-slash format.
- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

Explanation of test results

Out-put	Meaning
.	passed
F	failed
X	XPassed (expected to fail but passed)
f	XFAILed (expected to fail and indeed failed)
s	skipped
w	slow
T	timeout (e.g., when `--timeout` is used)
K	KeyboardInterrupt (when running the slow tests with `--slow`, you can interrupt one of them without killing the test runner)

Colors have no additional meaning and are used just to facilitate interpreting the output.

Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.test()
```

Run one file:

```
>>> sympy.test("sympy/core/tests/test_basic.py")
>>> sympy.test("_basic")
```

Run all tests in sympy/functions/ and some particular file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
...        "sympy/functions")
```

Run all tests in sympy/core and sympy/utilities:

```
>>> sympy.test("/core", "/util")
```

Run specific test from a file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
...        kw="test_equality")
```

Run specific test from any file:

```
>>> sympy.test(kw="subs")
```

Run the tests with verbose mode on:

```
>>> sympy.test(verbose=True)
```

Do not sort the test output:

```
>>> sympy.test(sort=False)
```