anaStruct Documentation

Release 1.0

Ritchie Vink

Mar 02, 2019

Contents:

1 Indices and tables **1**

2 Code examples **3**
- 2.1 Installation . . . 3
- 2.2 Getting started . . . 3
  - 2.2.1 Structure object . . . 3
- 2.3 Elements . . . 10
  - 2.3.1 Standard elements . . . 10
  - 2.3.2 Truss elements . . . 14
  - 2.3.3 Discretization . . . 15
  - 2.3.4 Insert node . . . 15
- 2.4 Supports . . . 16
  - 2.4.1 add_support_hinged . . . 16
  - 2.4.2 add_support_roll . . . 17
  - 2.4.3 add_support_fixed . . . 17
  - 2.4.4 add_support_spring . . . 18
- 2.5 Loads . . . 19
  - 2.5.1 Node loads . . . 20
  - 2.5.2 Element loads . . . 21
  - 2.5.3 Remove loads . . . 22
- 2.6 Plotting . . . 22
  - 2.6.1 Structure . . . 23
  - 2.6.2 Bending moments . . . 23
  - 2.6.3 Axial forces . . . 23
  - 2.6.4 Shear forces . . . 24
  - 2.6.5 Reaction forces . . . 24
  - 2.6.6 Displacements . . . 24
  - 2.6.7 Save figure . . . 25
- 2.7 Calculation . . . 26
  - 2.7.1 Non linear . . . 26
  - 2.7.2 Geometrical non linear . . . 26
- 2.8 Load cases and load combinations . . . 26
  - 2.8.1 Load cases . . . 26
  - 2.8.2 Load combinations . . . 28
  - 2.8.3 Load case class . . . 32
  - 2.8.4 Load combination class . . . 33

2.9 Post processing . . . 34
2.9.1 Node results system . . . 35
2.9.2 Node displacements . . . 36
2.9.3 Range of node displacements . . . 36
2.9.4 Element results . . . 37
2.9.5 Range of element results . . . 38
2.10 Element/ node interaction . . . 39
2.10.1 Find node id based on coordinates . . . 39
2.10.2 Find nearest node id based on coordinates . . . 39
2.10.3 Query node coordinates . . . 39
2.11 Vertex . . . 39
2.12 Saving . . . 40
2.13 Examples . . . 41

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

CHAPTER 2

Code examples

- Example_1
- Example_2
- Example_3
- Real_world_use_case

2.1 Installation

You can install anaStruct with pip!

```
$ pip install anastruct
```

It takes a while before new features are added in the official PyPI index. So if you want the latest features, install from github.

```
$ pip install git+https://github.com/ritchie46/anaStruct.git
```

2.2 Getting started

anaStruct is a Python implementation of the 2D Finite Element method for structures. It allows you to do structural analysis of frames and frames. It helps you to compute the forces and displacements in the structural elements.

Besides linear calculations, there is also support for non-linear nodes and geometric non linearity.

2.2.1 Structure object

You start a model by instantiating a SystemElements object. All the models state, i.e. elements, materials and forces are kept by this object.

class anastruct.fem.system.**SystemElements**(*figsize=(12, 8), EA=15000.0, EI=5000.0, load_factor=1, mesh=50*)

Modelling any structure starts with an object of this class.

Variables

- **EA** – Standard axial stiffness of elements, default=15,000
- **EI** – Standard bending stiffness of elements, default=5,000
- **figsize** – (tpl) Matplotlibs standard figure size
- **element_map** – (dict) Keys are the element ids, values are the element objects
- **node_map** – (dict) Keys are the node ids, values are the node objects.
- **node_element_map** – (dict) maps node ids to element objects.
- **supports_fixed** – (list) All the fixed supports in the system.
- **supports_hinged** – (list) All the hinged supports in the system.
- **supports_roll** – (list) All the roll supports in the system.
- **supports_spring_x** – (list) All the spring supports in x-direction in the system.
- **supports_spring_z** – (list) All the spring supports in z-direction in the system.
- **supports_spring_y** – (list) All the spring supports in y-direction in the system.
- **supports_roll_direction** – (list) The directions of the rolling supports.
- **loads_point** – (dict) Maps node ids to point loads.
- **loads_q** – (dict) Maps element ids to q-loads.
- **loads_moment** – (dict) Maps node ids to moment loads.
- **loads_dead_load** – (set) Element ids that have a dead load applied.

__init__(*figsize=(12, 8), EA=15000.0, EI=5000.0, load_factor=1, mesh=50*)

- E = Young's modulus
- A = Area
- I = Moment of Inertia

Parameters

- **figsize** – (tpl) Set the standard plotting size.
- **EA** – (flt) Standard E * A. Set the standard values of EA if none provided when generating an element.
- **EI** – (flt) Standard E * I. Set the standard values of EA if none provided when generating an element.
- **load_factor** – (flt) Multiply all loads with this factor.
- **mesh** – (int) Plotting mesh. Has no influence on the calculation.

Example

```
from anastruct import SystemElements
ss = SystemElements()
```

This *ss* object now has access to several methods which modify the state of the model. We can for instance create a structure.

```
ss.add_element(location=[[0, 0], [3, 4]])
ss.add_element(location=[[3, 4], [8, 4]])
```

Now we have elements, we need to define the supporting conditions of our structure.

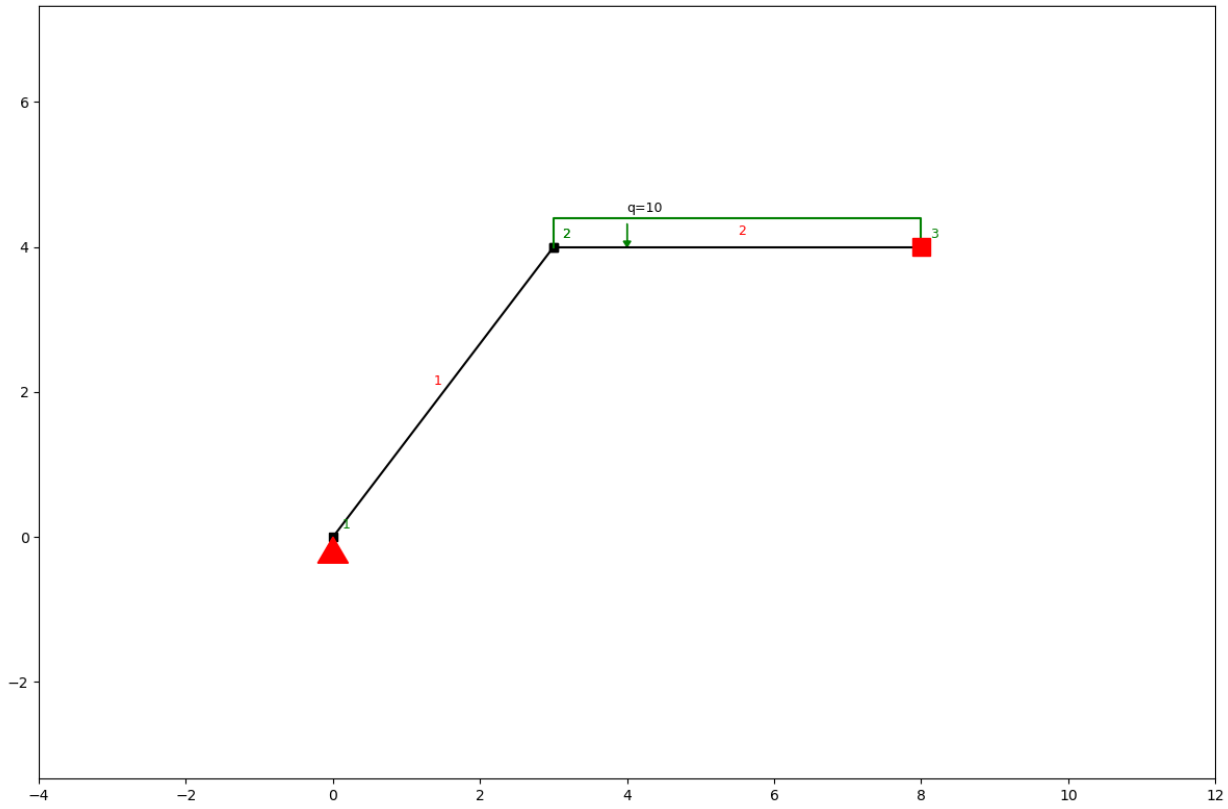
```
ss.add_support_hinged(node_id=1)
ss.add_support_fixed(node_id=3)
```

Finally we can add a load on the structure and compute the results.

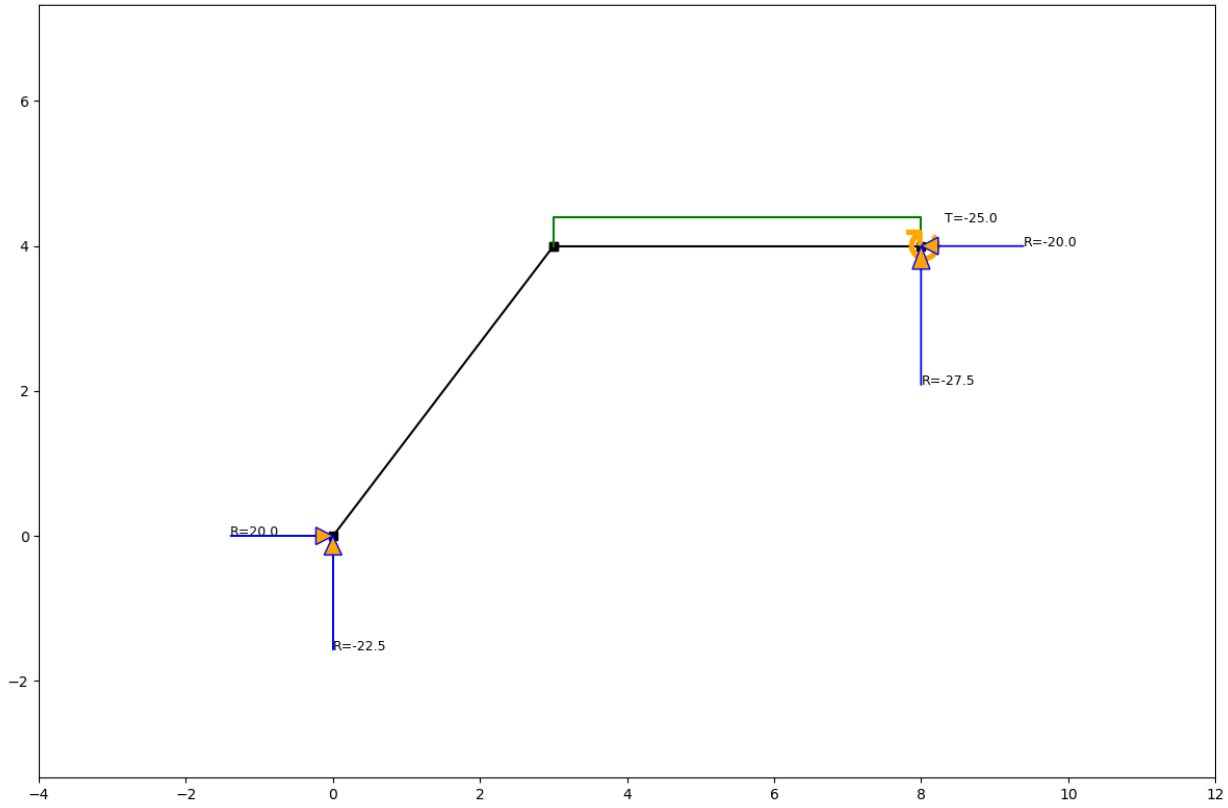
```
ss.q_load(element_id=2, q=-10)
ss.solve()
```

We can take a look at the results of the calculation by plotting different units we are interested in.

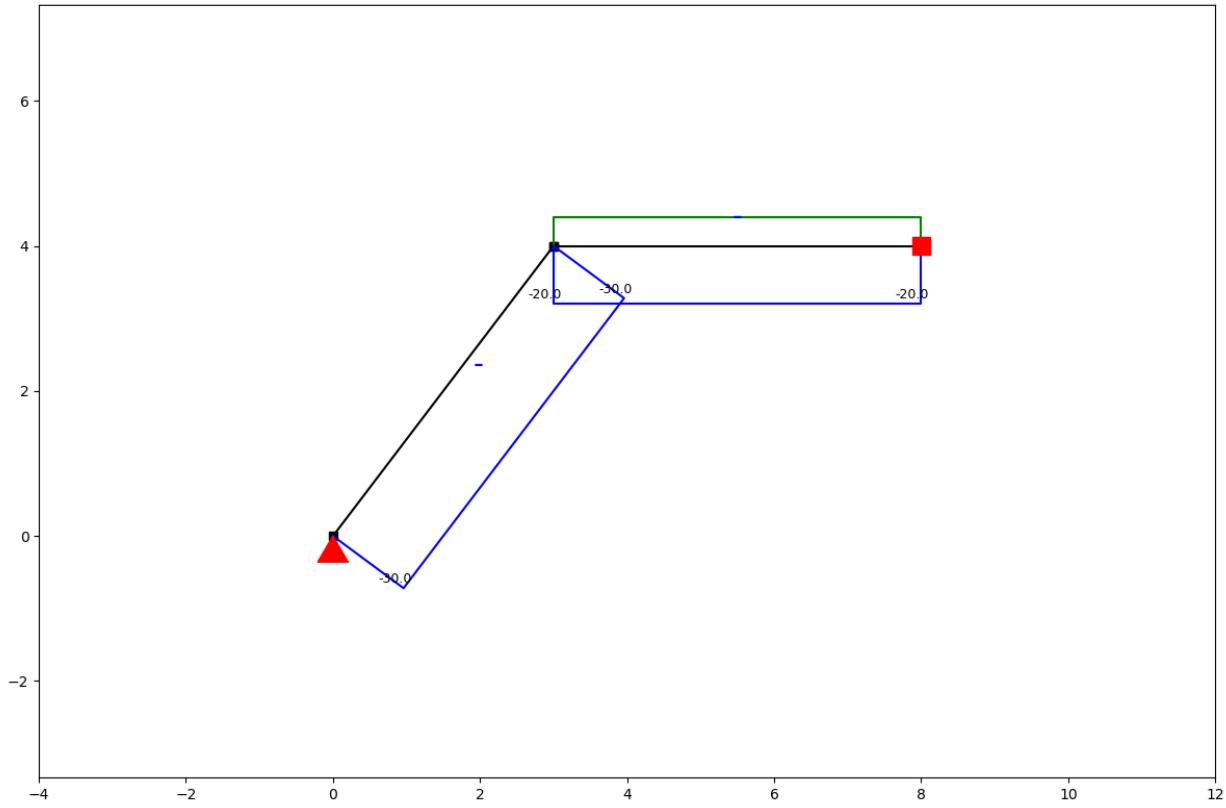
```
ss.show_structure()
```



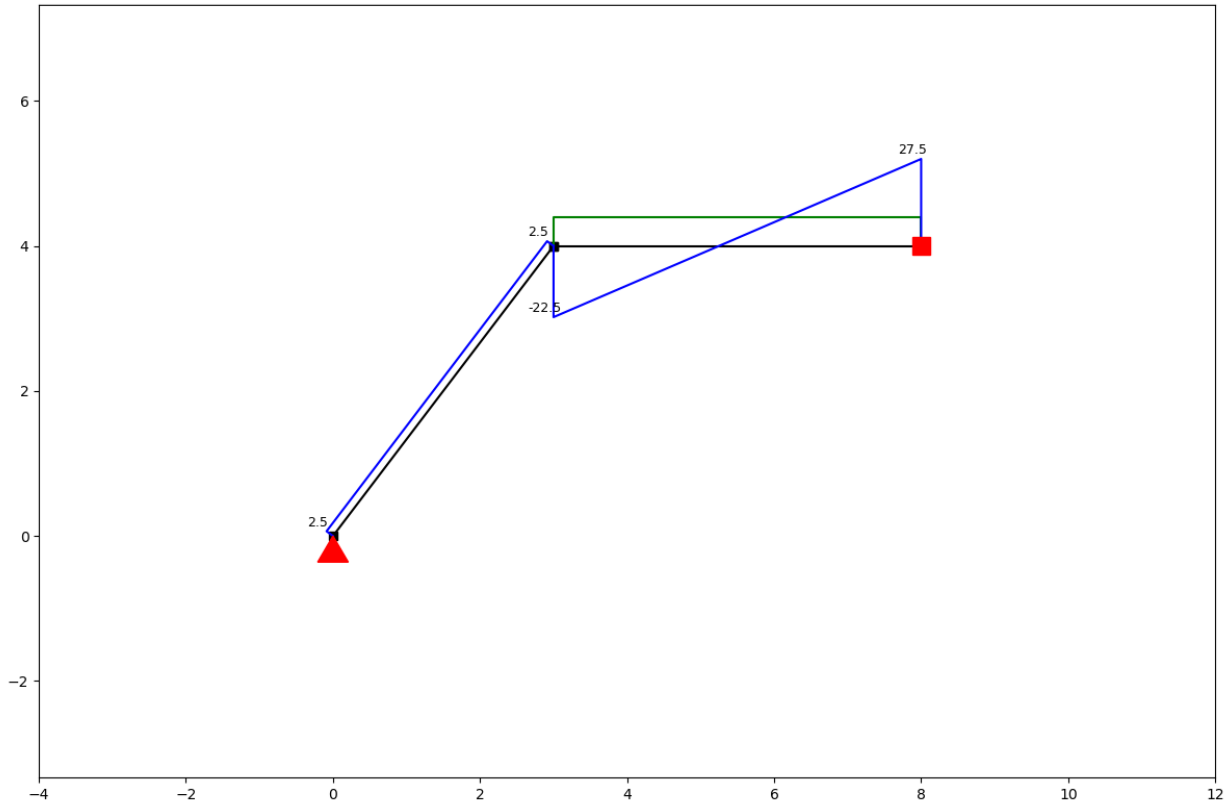

```
ss.show_reaction_force()
```



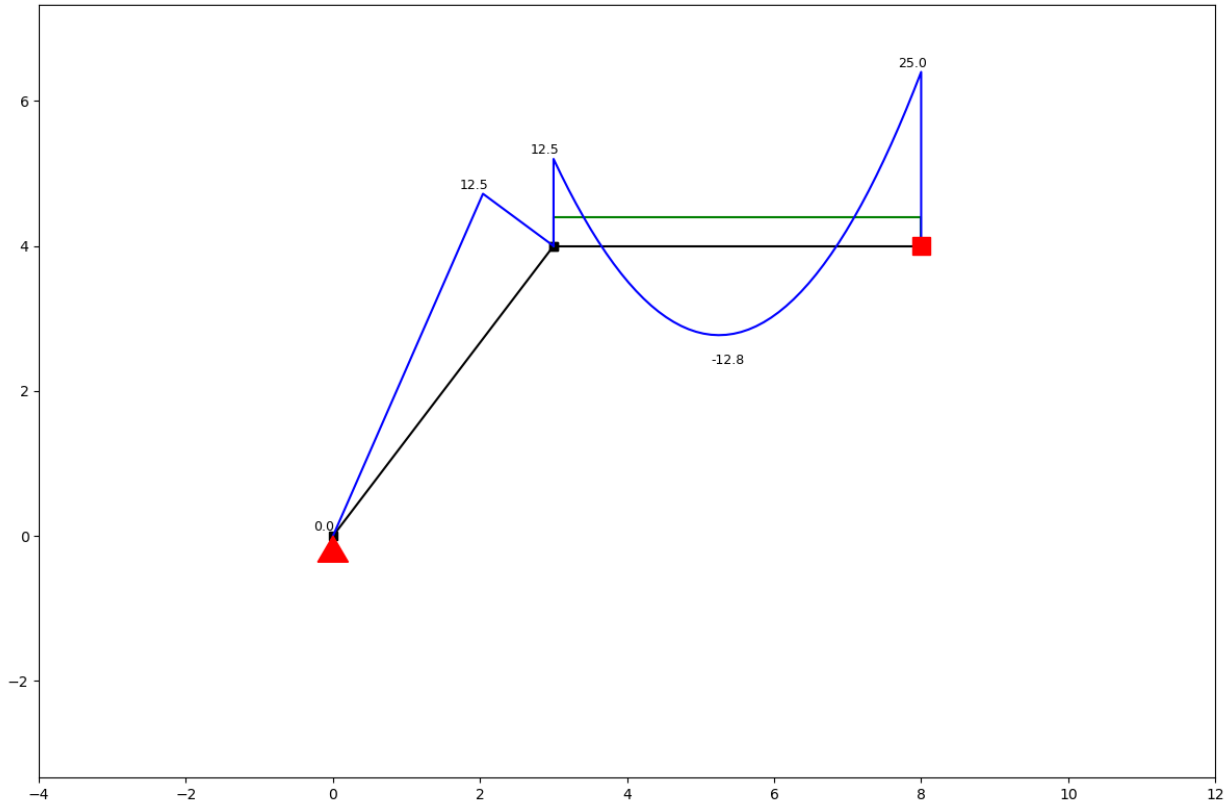

```
ss.show_axial_force()
```



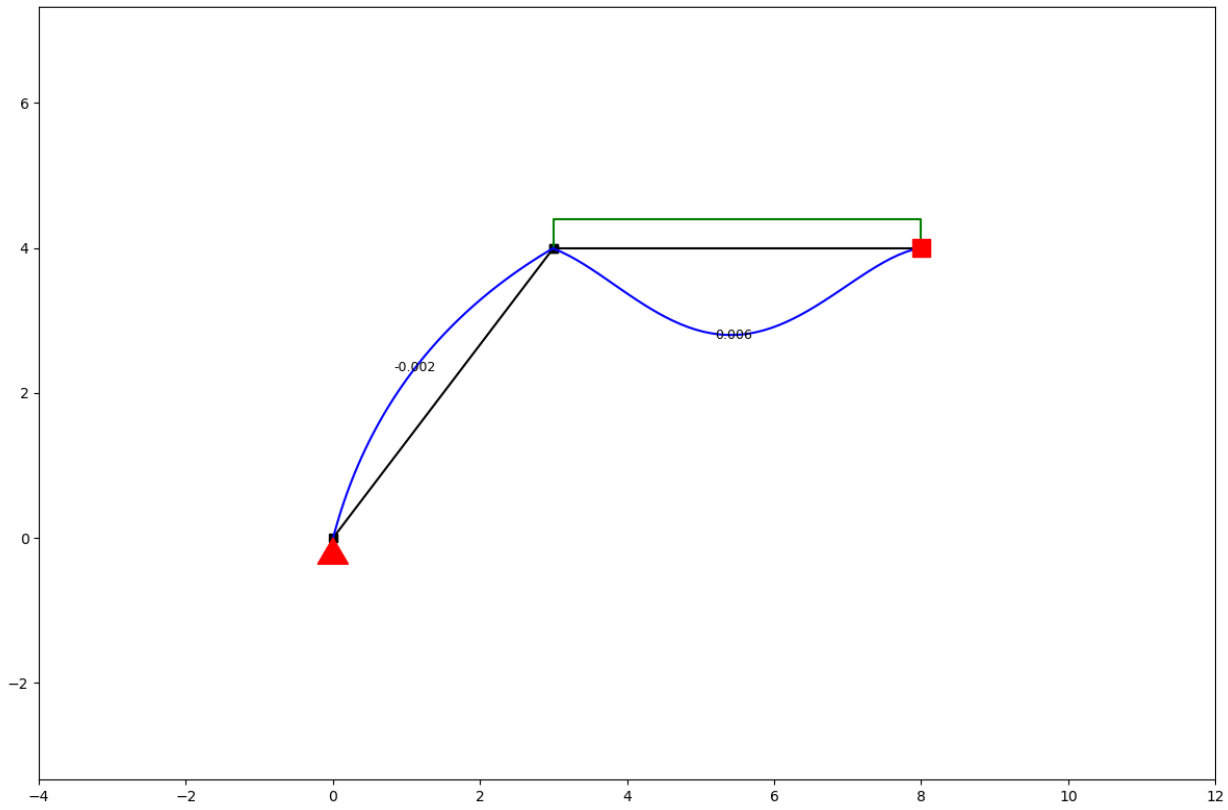

```
ss.show_shear_force()
```




```
ss.show_bending_moment()
```




```
ss.show_displacement()
```



2.3 Elements

The SystemElements class has several methods that help you model a structure. These methods are;

- add_truss_element
- add_element
- add_multiple_elements
- discretize

A structure is defined by elements, which have their own state.

The elements are stored in *SystemElement.element_map*. This is a dictionary with keys representing the element ids, and values being the element objects. The element objects ar implicitly created by the SystemElements object.

The state of an element can be interesting when post-processing results. For now we'll focus on the modelling part. Below you see the different methods for modelling a structure.

2.3.1 Standard elements

Standard elements have bending and axial stiffness and therefore will implement shear force, bending moment, axial force, extension, and deflection. Standard elements can be added with the following methods.

Add a single element

SystemElements.**add_element**(*location*, *EA=None*, *EI=None*, *g=0*, *mp=None*, *spring=None*, ***kwargs*)

Parameters

- **location** – (list/ Vertex) The two nodes of the element or the next node of the element.

  Example

```
location=[[x, y], [x, y]]
location=[Vertex, Vertex]
location=[x, y]
location=Vertex
```

- **EA** – (flt) EA
- **EI** – (flt) EI
- **g** – (flt) Weight per meter. [kN/m] / [N/m]
- **mp** –

  (dict) Set a maximum plastic moment capacity. Keys are integers representing the nodes. Values are the bending moment capacity.

  Example

```
mp={1: 210e3,
    2: 180e3}
```

- **spring** – (dict) Set a rotational spring or a hinge (k=0) at node 1 or node 2.

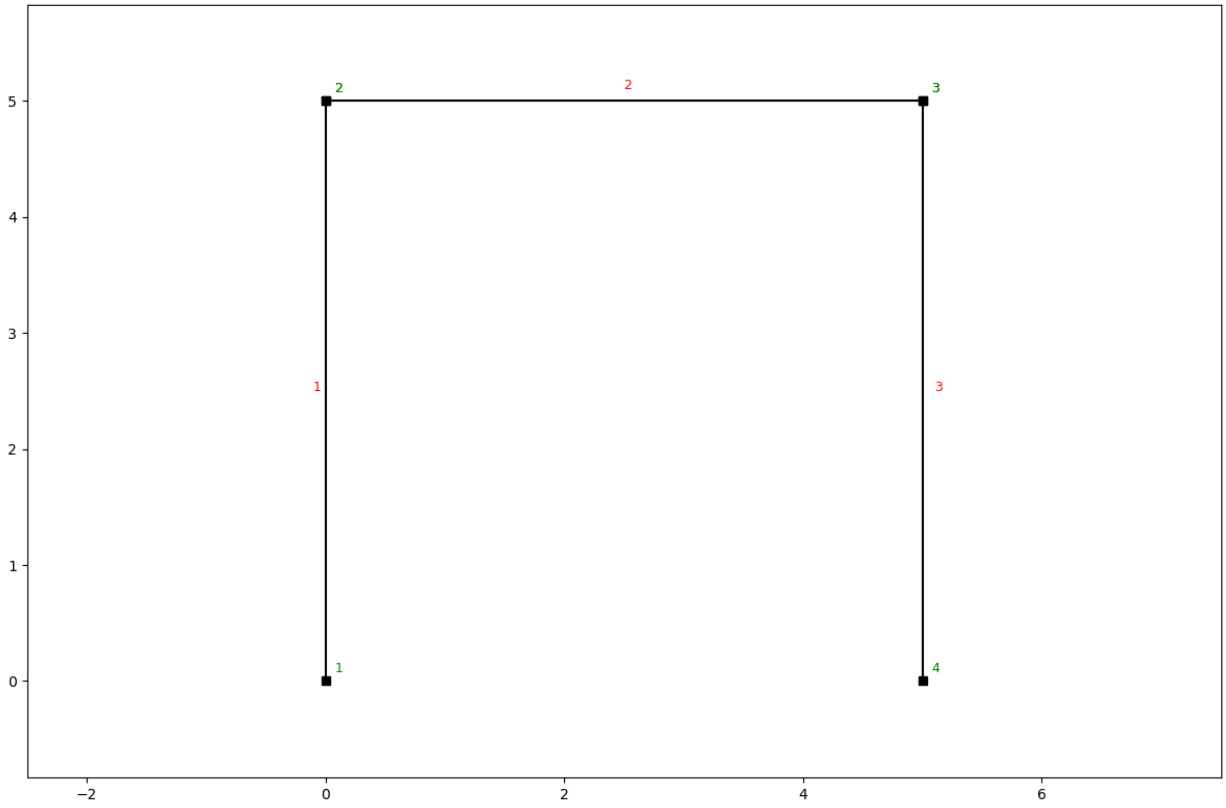
  Example

```
spring={1: k
        2: k}


# Set a hinged node:
spring={1: 0}
```

Returns (int) Elements ID.

Example

```
ss = SystemElements(EA=15000, EI=5000)
ss.add_element(location=[[0, 0], [0, 5]])
ss.add_element(location=[[0, 5], [5, 5]])
ss.add_element(location=[[5, 5], [5, 0]])
ss.show_structure()
```



Add multiple elements

`SystemElements.`**`add_multiple_elements`**(*location*, *n=None*, *dl=None*, *EA=None*, *EI=None*, *g=0*, *mp=None*, *spring=None*, ***kwargs*)

Add multiple elements defined by the first and the last point.

Parameters

- **`location`** – See 'add_element' method
- **`n`** – (int) Number of elements.
- **`dl`** – (flt) Distance between the elements nodes.
- **`EA`** – See 'add_element' method
- **`EI`** – See 'add_element' method
- **`g`** – See 'add_element' method
- **`mp`** – See 'add_element' method
- **`spring`** – See 'add_element' method

Keyword Args:

Parameters

- **`element_type`** – (str) See 'add_element' method
- **`first`** – (dict) Different arguments for the first element
- **`last`** – (dict) Different arguments for the last element

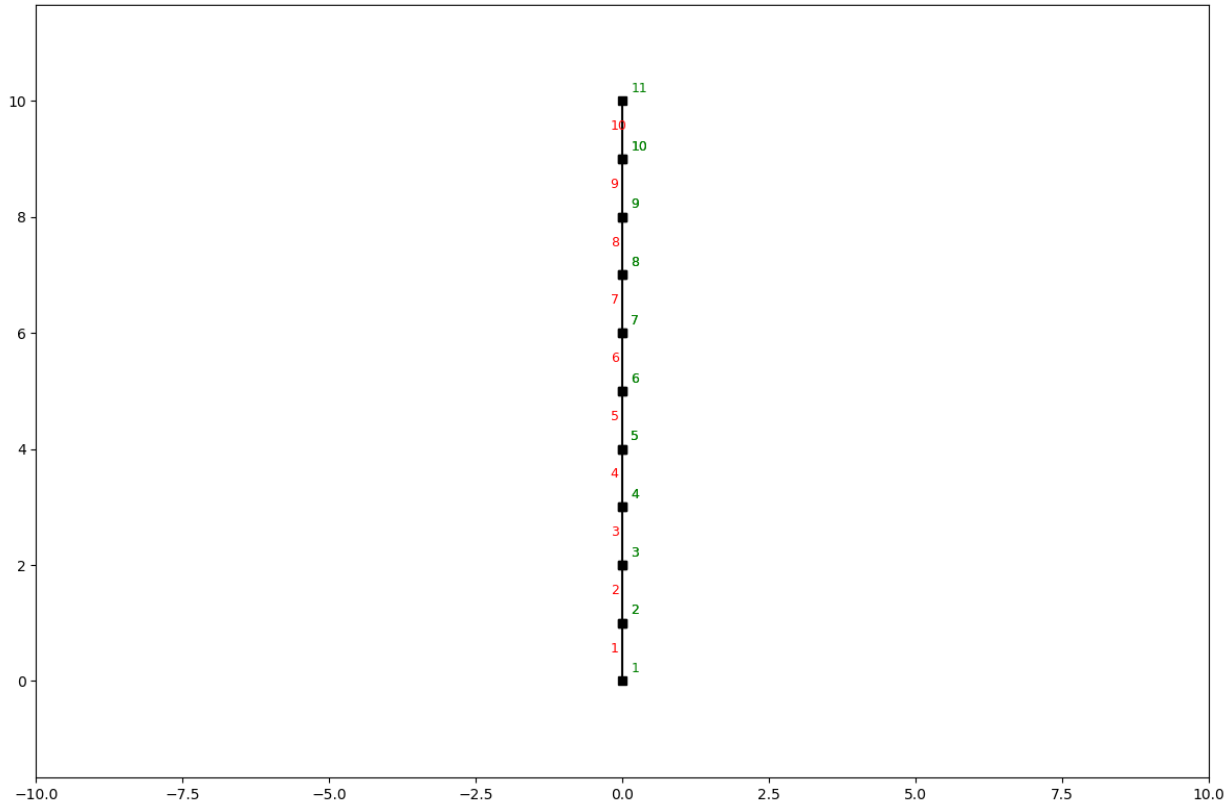
Example

```
last={'EA': 1e3, 'mp': 290}
```

Returns (list) Element IDs

Example add_multiple_elements

```
ss = SystemElements(EI=5e3, EA=1e5)
ss.add_multiple_elements([[0, 0], [0, 10]], 10)
ss.show_structure()
```



SystemElements.**add_element_grid**(*x*, *y*, *EA=None*, *EI=None*, *g=None*, *mp=None*, *spring=None*, ***kwargs*)

Add multiple elements defined by two containers with coordinates.

Parameters

- **x** – (list/ np.array) x coordinates.
- **y** – (list/ np.array) y coordinates.
- **EA** – See 'add_element' method
- **EI** – See 'add_element' method
- **g** – See 'add_element' method
- **mp** – See 'add_element' method

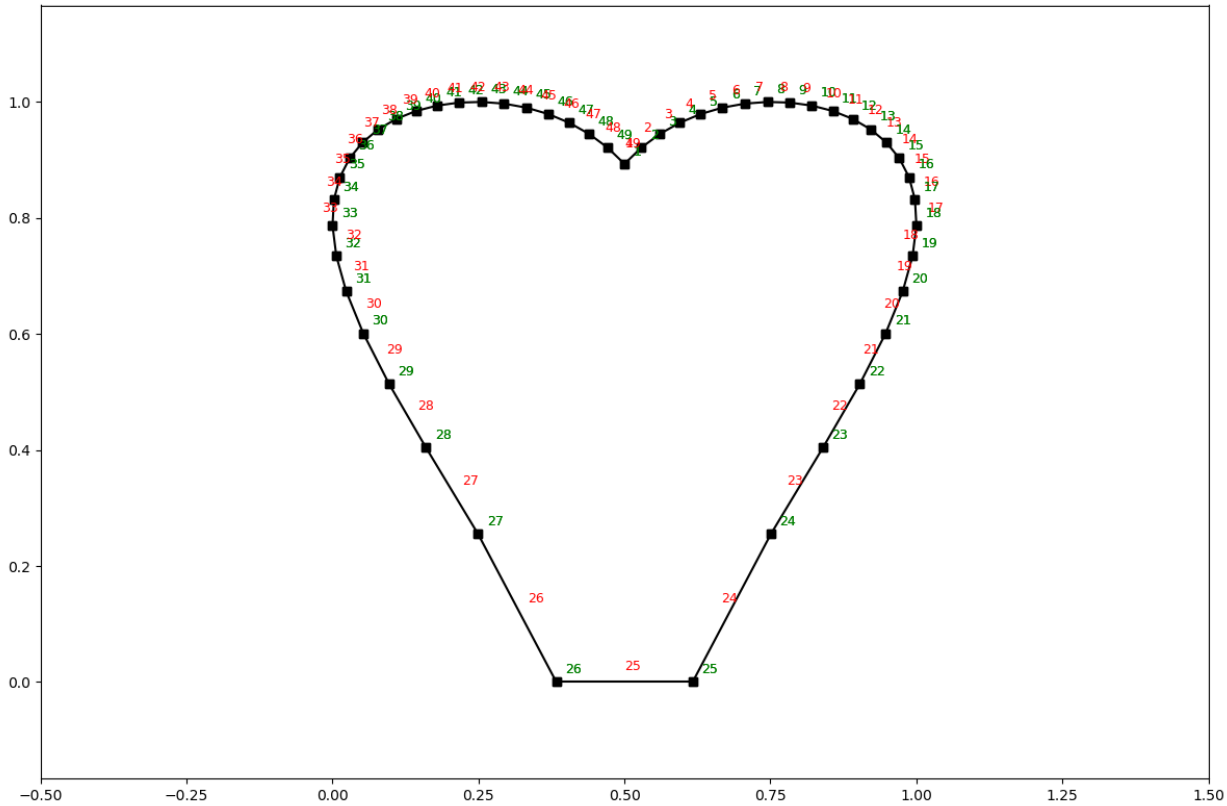
- **spring** – See 'add_element' method

Paramg **kwargs**kwargs** See 'add_element' method

Returns None

Example add_element_grid

```
from anastruct import SystemElements
import numpy as np

# <3
t = np.linspace(-1, 1)
x = np.sin(t) * np.cos(t) * np.log(np.abs(t))
y = np.abs(t)**0.3 * np.cos(t)**0.5 + 1
# Scaling to positive interval
x = (x - x.min()) / (x - x.min()).max()
y = (y - y.min()) / (y - y.min()).max()

ss = SystemElements()
ss.add_element_grid(x, y)
ss.show_structure()
```



2.3.2 Truss elements

Truss elements don't have bending stiffness and will therefore not implement shear force, bending moment and deflection. It does model axial force and extension.

add_truss_element

SystemElements.**add_truss_element**(*location*, *EA=None*)

Add an element that only has axial force.

Parameters

- **location** – (list/ Vertex) The two nodes of the element or the next node of the element.

  Example

```
location=[[x, y], [x, y]]
location=[Vertex, Vertex]
location=[x, y]
location=Vertex
```

- **EA** – (flt) EA

Returns (int) Elements ID.

2.3.3 Discretization

You can discretize an element in multiple smaller elements with the discretize method.

SystemElements.**discretize**(*n=10*)

Takes an already defined SystemElements object and increases the number of elements.

Parameters **n** – (int) Divide the elements into n sub-elements.

2.3.4 Insert node

Most of the nodes are defined when creating an element by passing the vertices (x, y coordinates) as the location parameter. It is also to add a node to elements that already exist via the insert_node method.

SystemElements.**insert_node**(*element_id*, *location=None*, *factor=None*)

Insert a node into an existing structure. This can be done by adding a new Vertex at any given location, or by setting a factor of the elements length. E.g. if you want a node at 40% of the elements length, you pass factor = 0.4.

Note: this method completely rebuilds the SystemElements object and is therefore slower then building a model with *add_element* methods.

Parameters

- **element_id** – (int) Id number of the element you want to insert the node.
- **location** – (list/ Vertex) The nodes of the element or the next node of the element.

  Example

```
location=[x, y]
location=Vertex
```

Param factor: (flt) Value between 0 and 1 to determine the new node location.

2.4 Supports

The following kinds of support conditions are possible.

- hinged (the node is able to rotate, but cannot translate)
- roll (the node is able to rotate and translation is allowed in one direction)
- fixed (the node cannot translate and not rotate)
- spring (translation and rotation are allowed but only with a linearly increasing resistance)

2.4.1 add_support_hinged

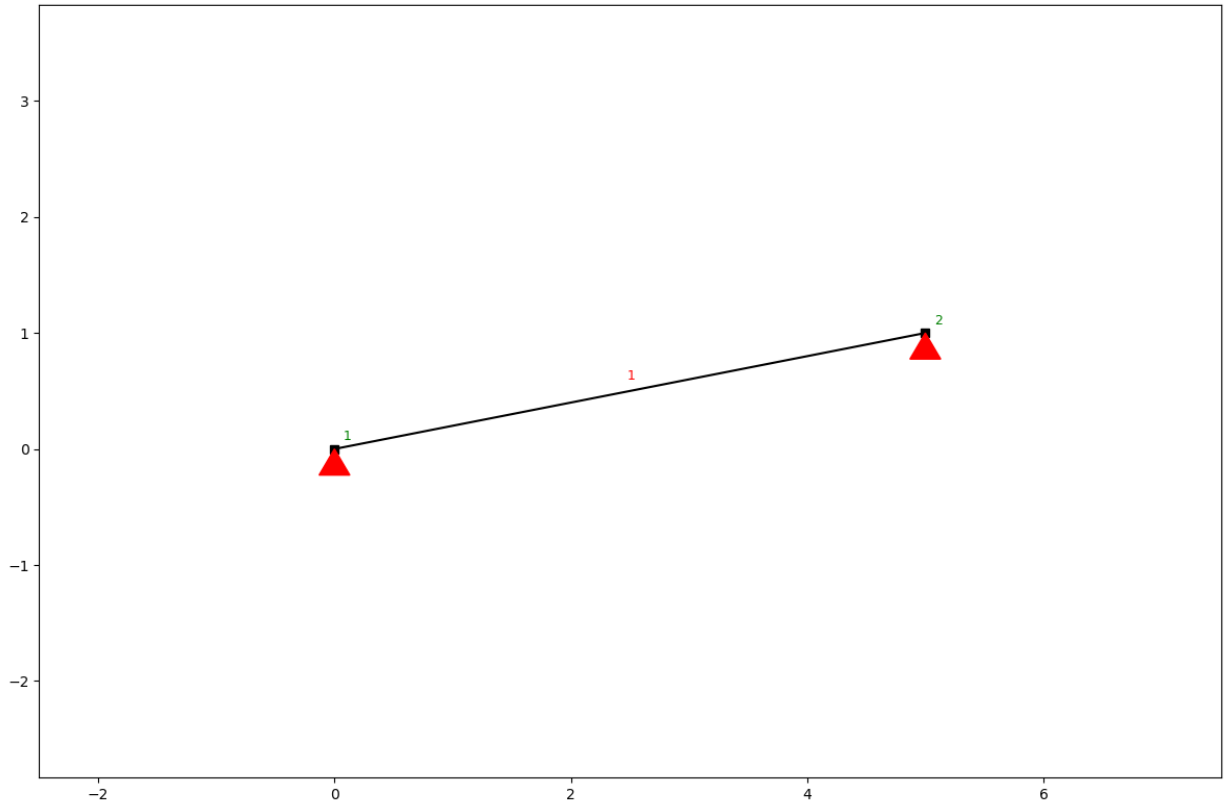
SystemElements.**add_support_hinged**(*node_id*)

Model a hinged support at a given node.

Parameters **node_id** – (int/ list) Represents the nodes ID

Example

```
ss.add_element(location=[5, 1])
ss.add_support_hinged(node_id=[1, 2])
ss.show_structure()
```



2.4.2 add_support_roll

SystemElements.**add_support_roll**(*node_id*, *direction=2*)

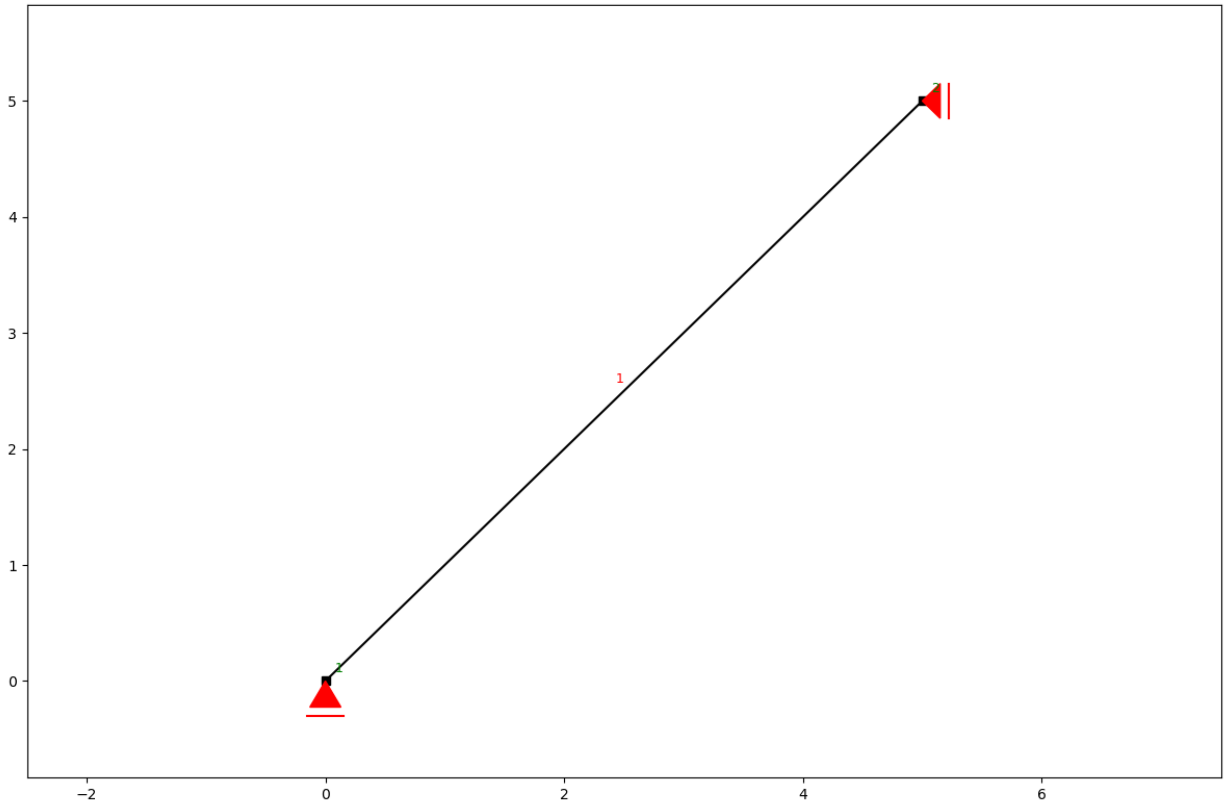
Adds a rolling support at a given node.

Parameters

- **node_id** – (int/ list) Represents the nodes ID
- **direction** – (int/ list) Represents the direction that is fixed: x = 1, y = 2

Example

```
ss.add_element(location=[5, 5])
ss.add_support_roll(node_id=2, direction=1)
ss.add_support_roll(node_id=1, direction=2)
ss.show_structure()
```



2.4.3 add_support_fixed

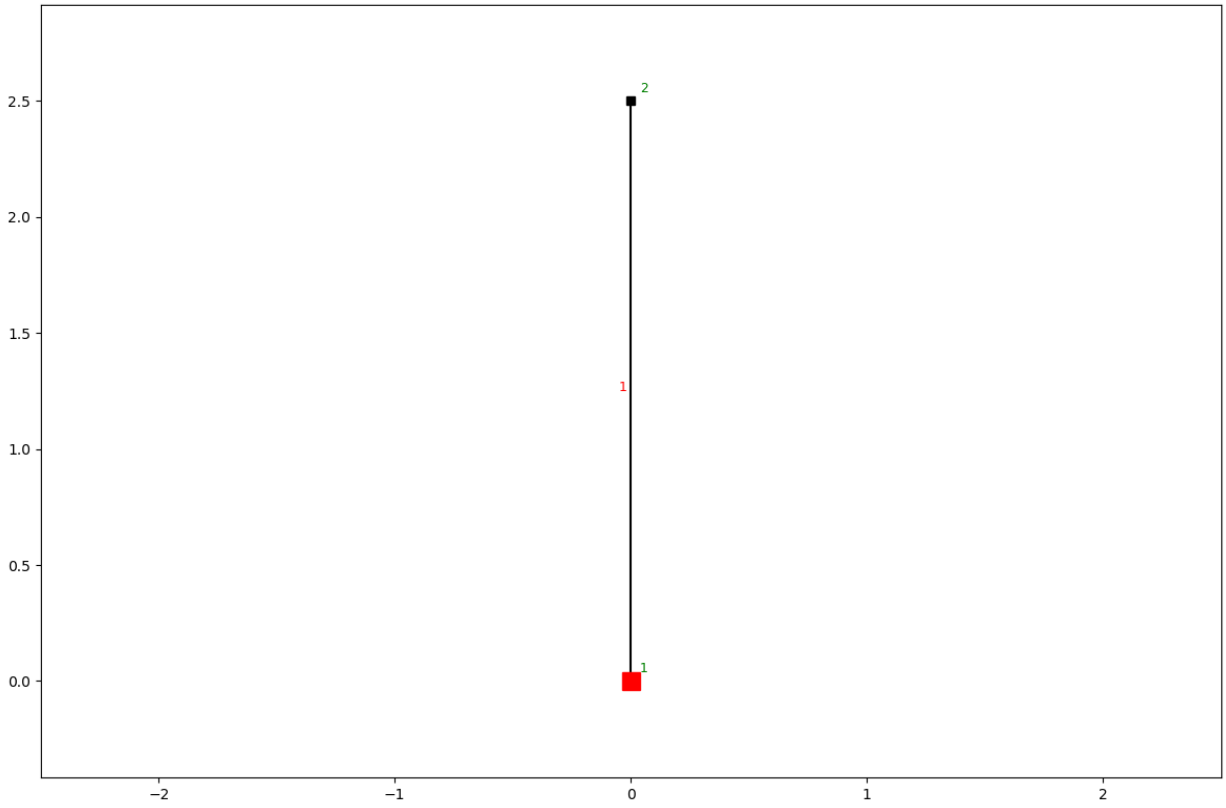
SystemElements.**add_support_fixed**(*node_id*)

Add a fixed support at a given node.

Parameters **node_id** – (int/ list) Represents the nodes ID

Example

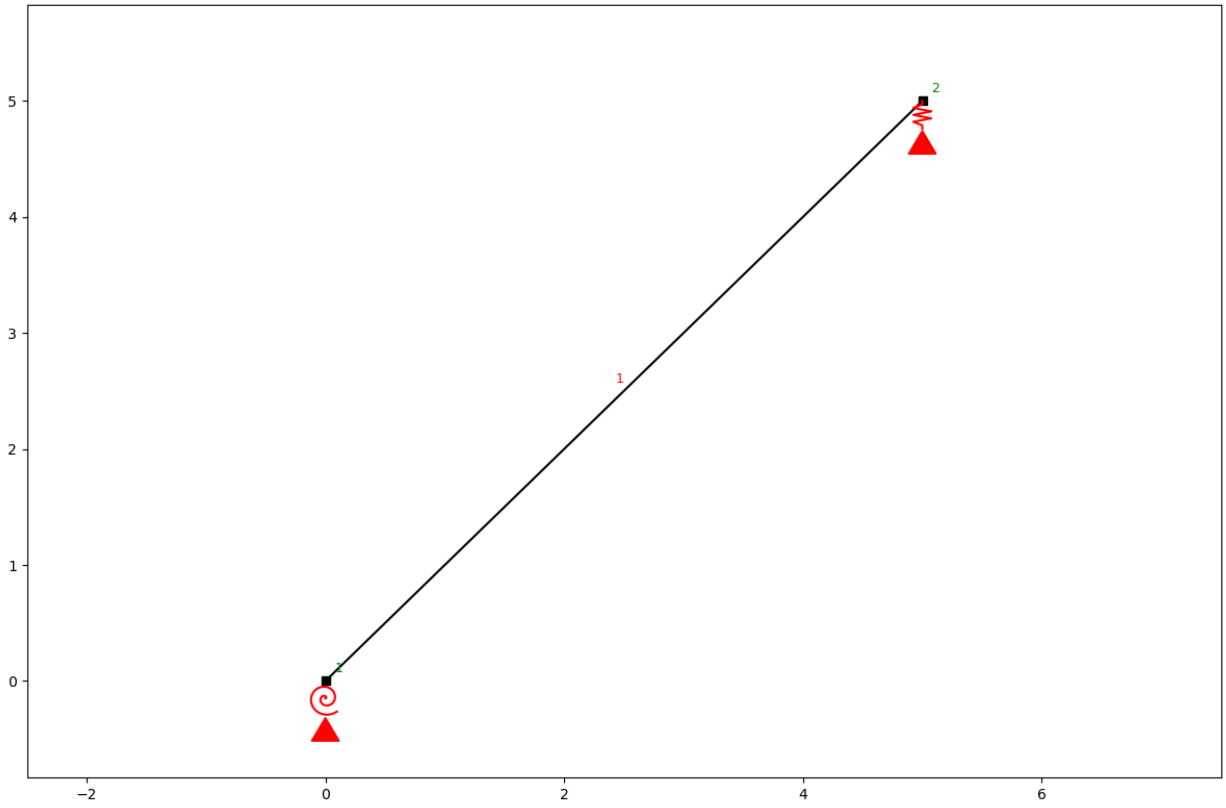
```
ss.add_element(location=[0, 2.5])
ss.add_support_fixed(node_id=1)
ss.show_structure()
```



2.4.4 add_support_spring

Example

```
ss.add_element(location=[5, 5])
ss.add_support_spring(node_id=1, translation=3, k=1000)
ss.add_support_spring(node_id=-1, translation=2, k=1000)
ss.show_structure()
```



`SystemElements.`**`add_support_spring`**(*node_id*, *translation*, *k*, *roll=False*)

Add a translational support at a given node.

Parameters

- **`translation`** – (int/ list) Represents the prevented translation.

  Note

  1 = translation in x
  2 = translation in z
  3 = rotation in y

- **`node_id`** – (int/ list) Integer representing the nodes ID.
- **`k`** – (flt) Stiffness of the spring
- **`roll`** – (bool) If set to True, only the translation of the spring is controlled.

2.5 Loads

anaStruct allows the following loads on a structure. There are loads on nodes and loads on elements. Element loads are implicitly placed on the loads and recalculated during post processing.

2.5.1 Node loads

Point loads

Point loads are defined in x- and/ or y-direction, or by defining a load with an angle.

`SystemElements.`**`point_load`**(*node_id*, *Fx=0*, *Fy=0*, *rotation=0*)

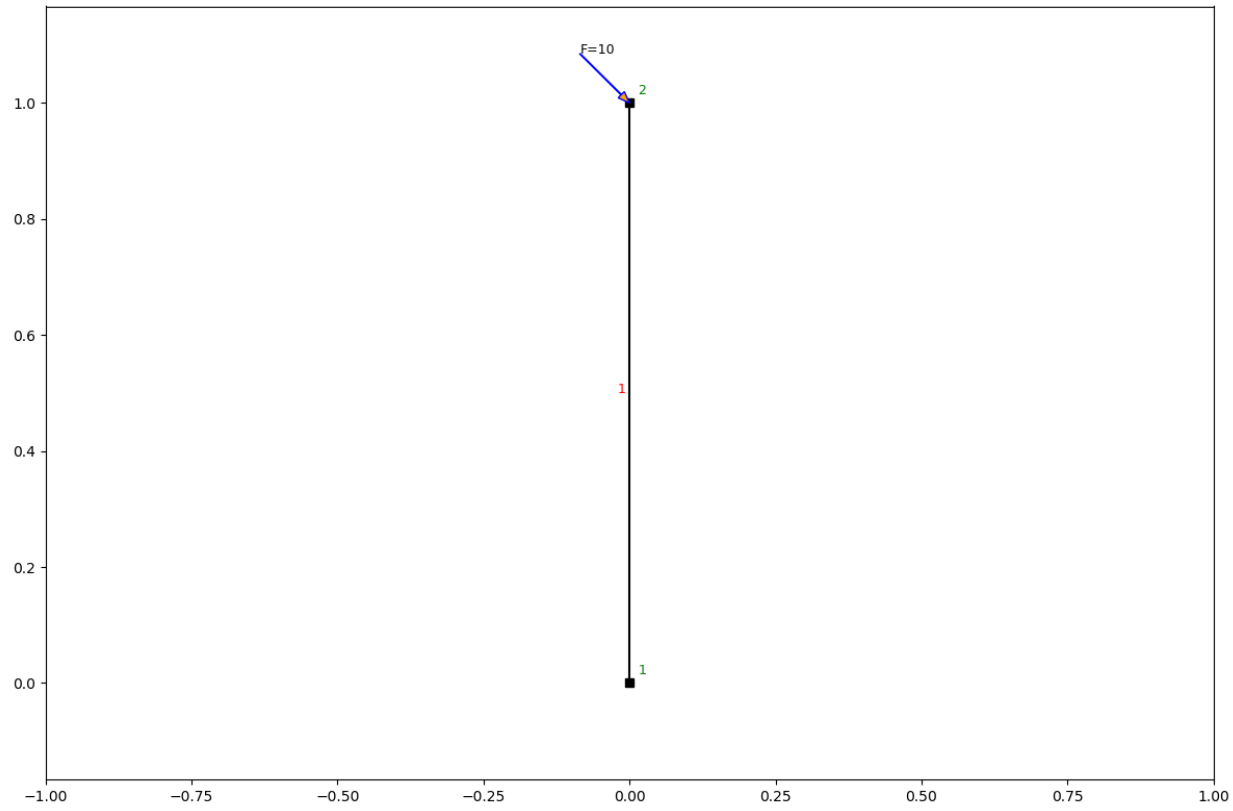
Apply a point load to a node.

Parameters

- **`node_id`** – (int/ list) Nodes ID.
- **`Fx`** – (flt/ list) Force in global x direction.
- **`Fy`** – (flt/ list) Force in global x direction.
- **`rotation`** – (flt/ list) Rotate the force clockwise. Rotation is in degrees.

Example

```
ss.add_element(location=[0, 1])
ss.point_load(ss.id_last_node, Fx=10, rotation=45)
ss.show_structure()
```



Bending moments

Moment loads apply a rotational force on the nodes.

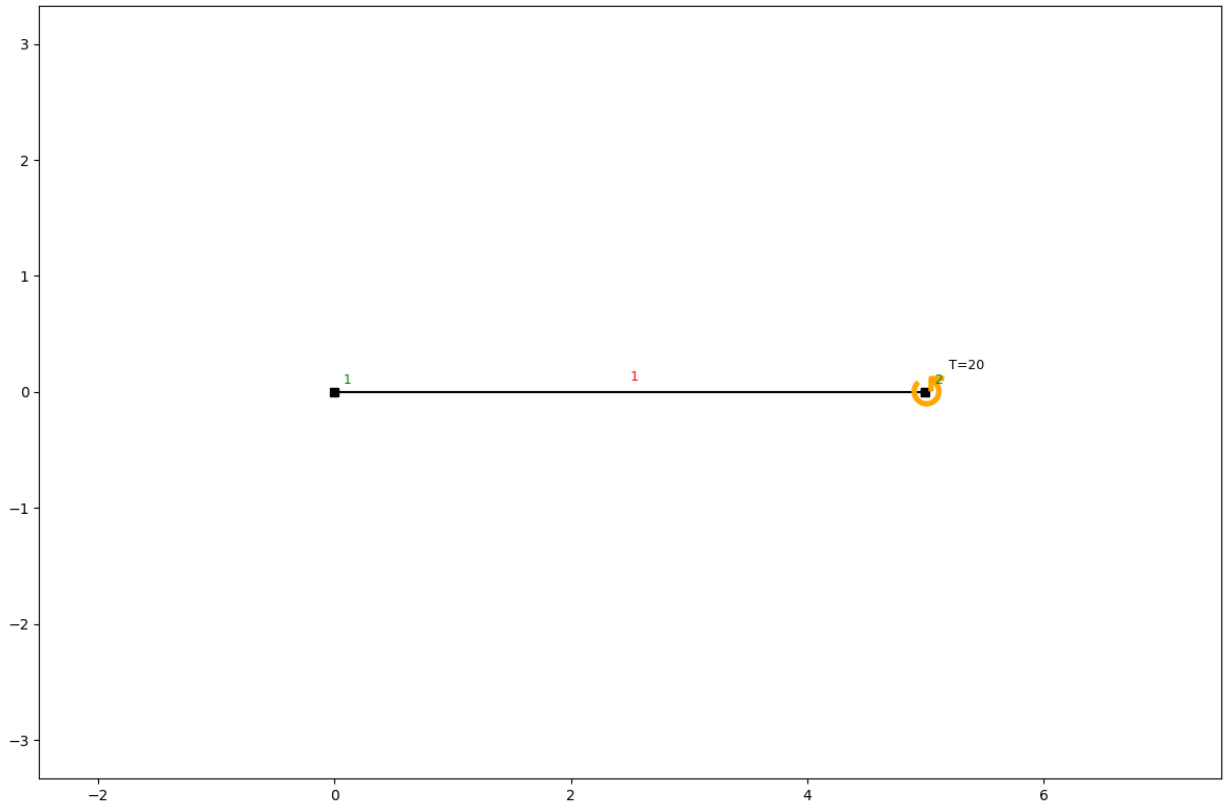
SystemElements.**moment_load**(*node_id*, *Ty*)
Apply a moment on a node.

Parameters

- **node_id** – (int/ list) Nodes ID.
- **Ty** – (flt/ list) Moments acting on the node.

Example

```
ss.add_element([5, 0])
ss.moment_load(node_id=ss.id_last_node, Ty=20)
ss.show_structure()
```



2.5.2 Element loads

Q-loads are distributed loads. They can act perpendicular to the elements direction, parallel to the elements direction, and in global x and y directions.

q-loads

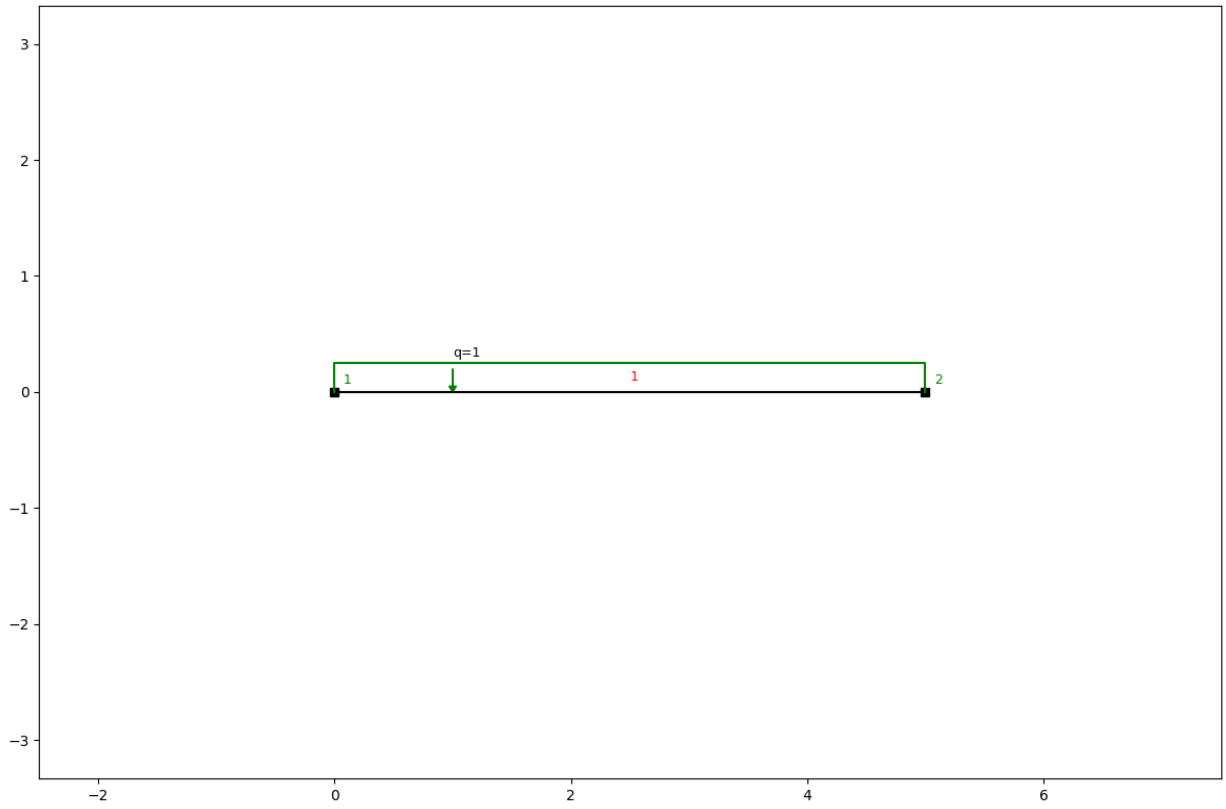
SystemElements.**q_load**(*q*, *element_id*, *direction='element'*)
Apply a q-load to an element.

Parameters

- **element_id** – (int/ list) representing the element ID
- **q** – (flt) value of the q-load
- **direction** – (str) "element", "x", "y"

Example

```
ss.add_element([5, 0])
ss.q_load(q=-1, element_id=ss.id_last_element, direction='element')
ss.show_structure()
```



2.5.3 Remove loads

SystemElements.**remove_loads**(*dead_load=False*)

Remove all the applied loads from the structure.

Parameters **dead_load** – (bool) Remove the dead load.

2.6 Plotting

The SystemElements object implements several plotting methods for retrieving standard plotting results. Every plotting method has got the same parameters. The plotter is based on a Matplotlib backend and it is possible to get the figure and do modifications of your own. The x and y coordinates of the model should all be positive value for the plotter to work properly.

2.6.1 Structure

SystemElements.**show_structure**(*verbosity=0*, *scale=1.0*, *offset=(0, 0)*, *figsize=None*, *show=True*, *supports=True*, *values_only=False*)

Plot the structure.

Parameters

- **verbosity** – (int) 0: All information, 1: Suppress information.
- **scale** – (flt) Scale of the plot.
- **offset** – (tpl) Offset the plots location on the figure.
- **figsize** – (tpl) Change the figure size.
- **show** – (bool) Plot the result or return a figure.
- **supports** – (bool) Show the supports.
- **values_only** – (bool) Return the values that would be plotted as tuple containing two arrays: (x, y)

Returns (figure)

2.6.2 Bending moments

SystemElements.**show_bending_moment**(*factor=None*, *verbosity=0*, *scale=1*, *offset=(0, 0)*, *figsize=None*, *show=True*, *values_only=False*)

Plot the bending moment.

Parameters

- **factor** – (flt) Influence the plotting scale.
- **verbosity** – (int) 0: All information, 1: Suppress information.
- **scale** – (flt) Scale of the plot.
- **offset** – (tpl) Offset the plots location on the figure.
- **figsize** – (tpl) Change the figure size.
- **show** – (bool) Plot the result or return a figure.
- **values_only** – (bool) Return the values that would be plotted as tuple containing two arrays: (x, y)

Returns (figure)

2.6.3 Axial forces

SystemElements.**show_axial_force**(*factor=None*, *verbosity=0*, *scale=1*, *offset=(0, 0)*, *figsize=None*, *show=True*, *values_only=False*)

Plot the axial force.

Parameters

- **factor** – (flt) Influence the plotting scale.
- **verbosity** – (int) 0: All information, 1: Suppress information.
- **scale** – (flt) Scale of the plot.

- **offset** – (tpl) Offset the plots location on the figure.
- **figsize** – (tpl) Change the figure size.
- **show** – (bool) Plot the result or return a figure.
- **values_only** – (bool) Return the values that would be plotted as tuple containing two arrays: (x, y)

Returns (figure)

2.6.4 Shear forces

SystemElements.**show_shear_force**(*factor=None*, *verbosity=0*, *scale=1*, *offset=(0, 0)*, *figsize=None*, *show=True*, *values_only=False*)

Plot the shear force. :param factor: (flt) Influence the plotting scale. :param verbosity: (int) 0: All information, 1: Suppress information. :param scale: (flt) Scale of the plot. :param offset: (tpl) Offset the plots location on the figure. :param figsize: (tpl) Change the figure size. :param show: (bool) Plot the result or return a figure. :param values_only: (bool) Return the values that would be plotted as tuple containing two arrays: (x, y) :return: (figure)

2.6.5 Reaction forces

SystemElements.**show_reaction_force**(*verbosity=0*, *scale=1*, *offset=(0, 0)*, *figsize=None*, *show=True*)

Plot the reaction force.

Parameters

- **verbosity** – (int) 0: All information, 1: Suppress information.
- **scale** – (flt) Scale of the plot.
- **offset** – (tpl) Offset the plots location on the figure.
- **figsize** – (tpl) Change the figure size.
- **show** – (bool) Plot the result or return a figure.

Returns (figure)

2.6.6 Displacements

SystemElements.**show_displacement**(*factor=None*, *verbosity=0*, *scale=1*, *offset=(0, 0)*, *figsize=None*, *show=True*, *linear=False*, *values_only=False*)

Plot the displacement.

Parameters

- **factor** – (flt) Influence the plotting scale.
- **verbosity** – (int) 0: All information, 1: Suppress information.
- **scale** – (flt) Scale of the plot.
- **offset** – (tpl) Offset the plots location on the figure.
- **figsize** – (tpl) Change the figure size.
- **show** – (bool) Plot the result or return a figure.

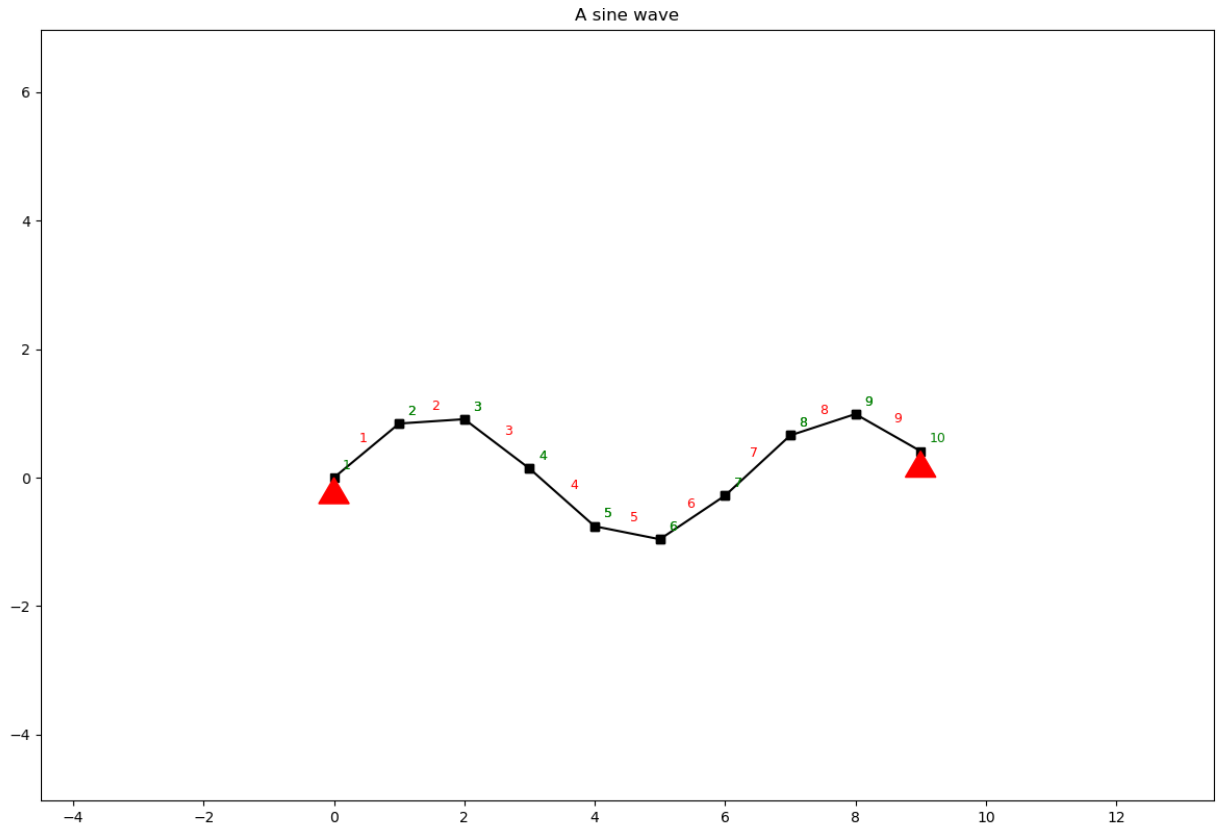
- **linear** – (bool) Don't evaluate the displacement values in between the elements
- **values_only** – (bool) Return the values that would be plotted as tuple containing two arrays: (x, y)

Returns (figure)

2.6.7 Save figure

When the *show* parameter is set to *False* a Matplotlib figure is returned and the figure can be saved with proper titles.

```
from anastruct import SystemElements
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10)
y = np.sin(x)

ss = SystemElements()
ss.add_element_grid(x, y)
ss.add_support_hinged(node_id=[1, -1])

fig = ss.show_structure(show=False)
plt.title('A sine wave')
plt.savefig('my-figure.png')
```



2.7 Calculation

Once all the elements, supports and loads are in place, solving the calculation is as easy as calling the *solve* method.

`SystemElements.`**`solve`**(*force_linear=False*, *verbosity=0*, *max_iter=200*, *geometrical_non_linear=False*, ***kwargs*)

Compute the results of current model.

Parameters

- **`force_linear`** – (bool) Force a linear calculation. Even when the system has non linear nodes.
- **`verbosity`** – (int) 0. Log calculation outputs. 1. silence.
- **`max_iter`** – (int) Maximum allowed iterations.
- **`geometrical_non_linear`** – (bool) Calculate second order effects and determine the buckling factor.

Returns (array) Displacements vector.

Development **kwargs:

param naked (bool) Whether or not to run the solve function without doing post processing.

param discretize_kwargs When doing a geometric non linear analysis you can reduce or increase the number of elements created that are used for determining the buckling_factor

2.7.1 Non linear

The model will automatically do a non linear calculation if there are non linear nodes present in the SystemElements state. You can however force the model to do a linear calculation with the *force_linear* parameter.

2.7.2 Geometrical non linear

To start a geometrical non linear calculation you'll need to set the *geometrical_non_linear* to True. It is also wise to pass a *discretize_kwargs* dictionary.

```
ss.solve(geometrical_non_linear=True, discretize_kwargs=dict(n=20))
```

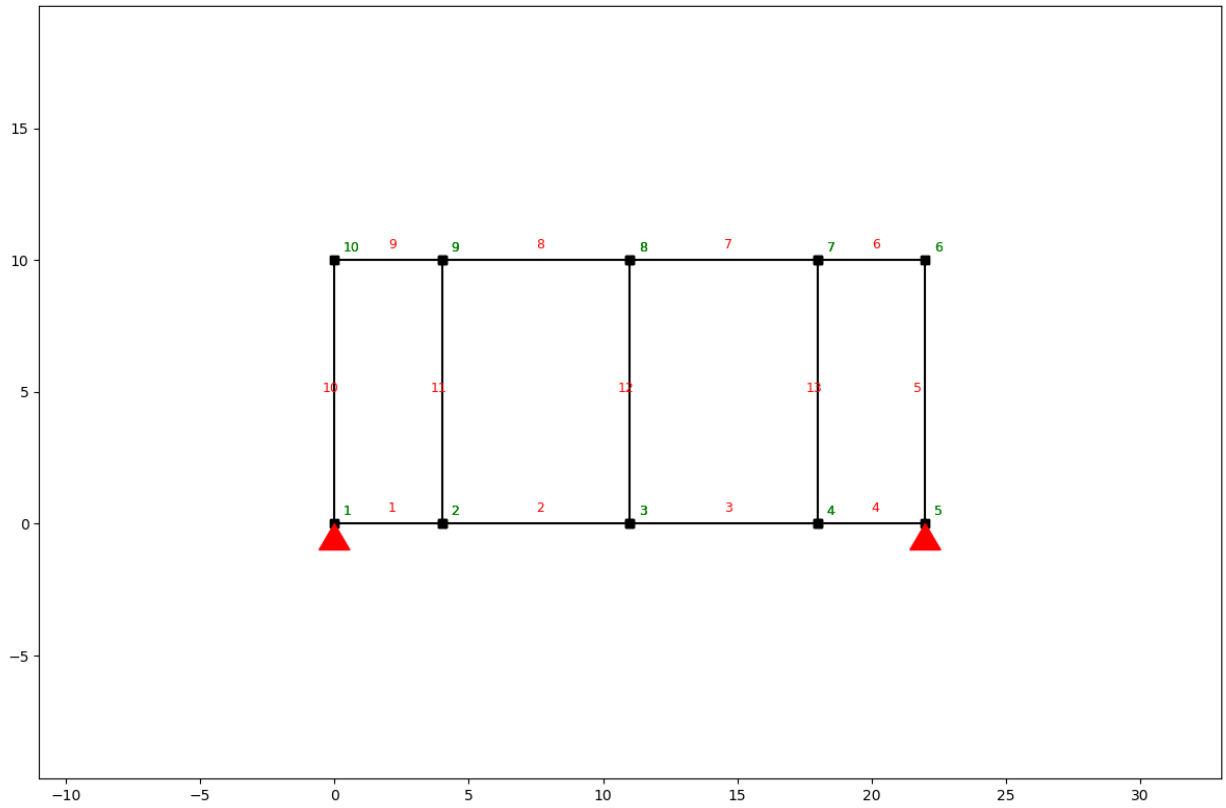
With this dictionary you can set the amount of discretization elements generated during the geometrical non linear calculation. This calculation is an approximation and gets more accurate with more discretization elements.

2.8 Load cases and load combinations

2.8.1 Load cases

You can group different loads in a single load case and add these to a SystemElements object. Let's look at an example. First we create a frame girder.

```
from anastruct import SystemElements
from anastruct import LoadCase, LoadCombination
import numpy as np

ss = SystemElements()
height = 10

x = np.cumsum([0, 4, 7, 7, 4])
y = np.zeros(x.shape)
x = np.append(x, x[::-1])
y = np.append(y, y + height)

ss.add_element_grid(x, y)
ss.add_element([[0, 0], [0, height]])
ss.add_element([[4, 0], [4, height]])
ss.add_element([[11, 0], [11, height]])
ss.add_element([[18, 0], [18, height]])
ss.add_support_hinged([1, 5])
ss.show_structure()
```



Now we can add a loadcase for all the wind loads.

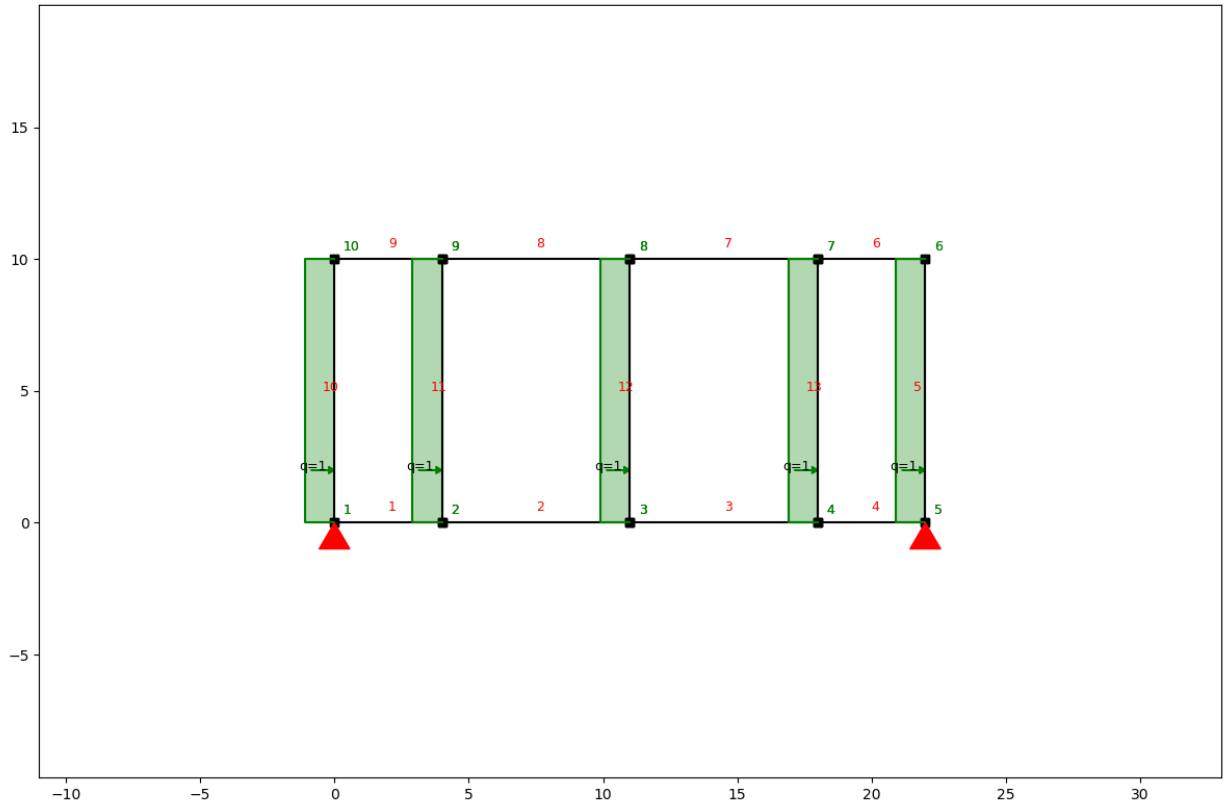
```
lc_wind = LoadCase('wind')
lc_wind.q_load(q=-1, element_id=[10, 11, 12, 13, 5])

print(lc_wind)
```

output

```
Loadcase wind:
{'q_load-1': {'direction': 'element',
              'element_id': [10, 11, 12, 13, 5],
              'q': -1}}
```

And apply to the load case to our system.

```
# add the load case to the SystemElements object
ss.apply_load_case(lc_wind)
ss.show_structure()
```



2.8.2 Load combinations

We can also combine load cases in a load combination with the *LoadCombination* class. First remove the previous load case from the system, create a *LoadCombination* object and add the *LoadCase* objects to the *LoadCombination* object.

```
# reset the structure
ss.remove_loads()


# create another load case
lc_cables = LoadCase('cables')
lc_cables.point_load(node_id=[2, 3, 4], Fy=-100)

combination = LoadCombination('ULS')
```

(continues on next page)

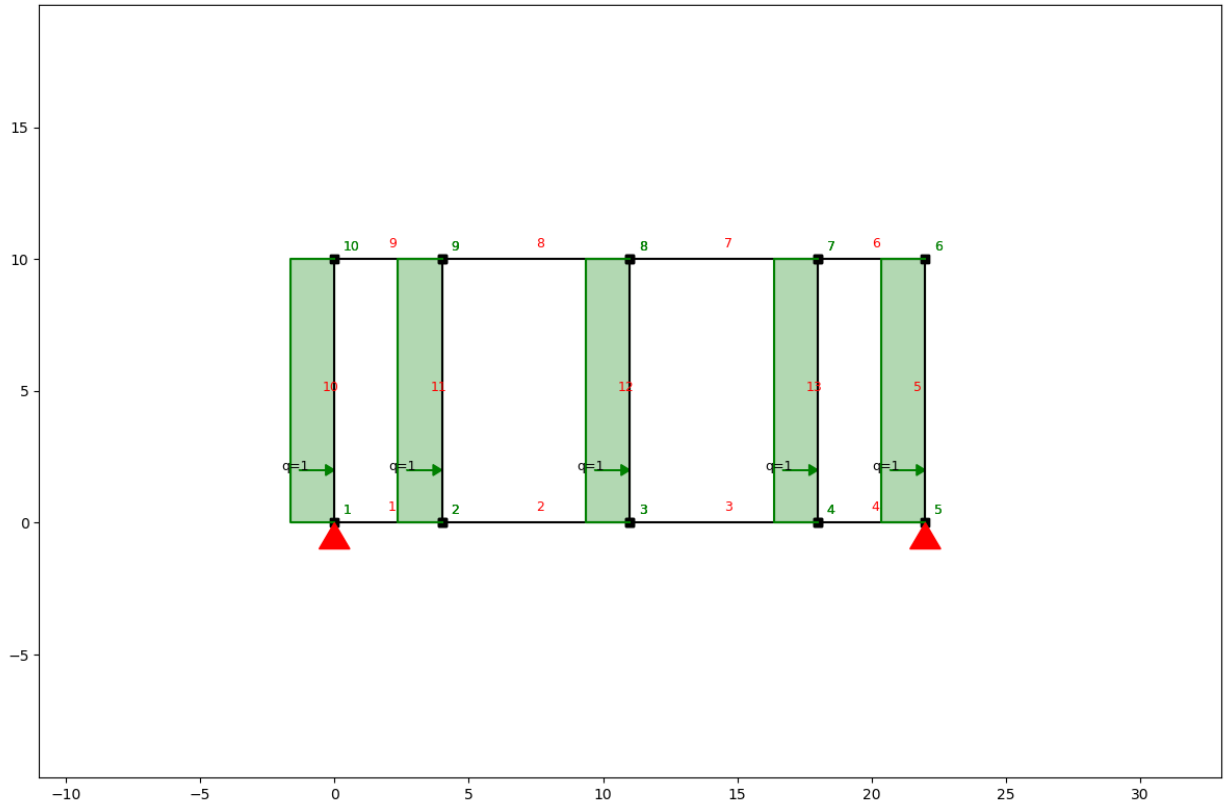
(continued from previous page)

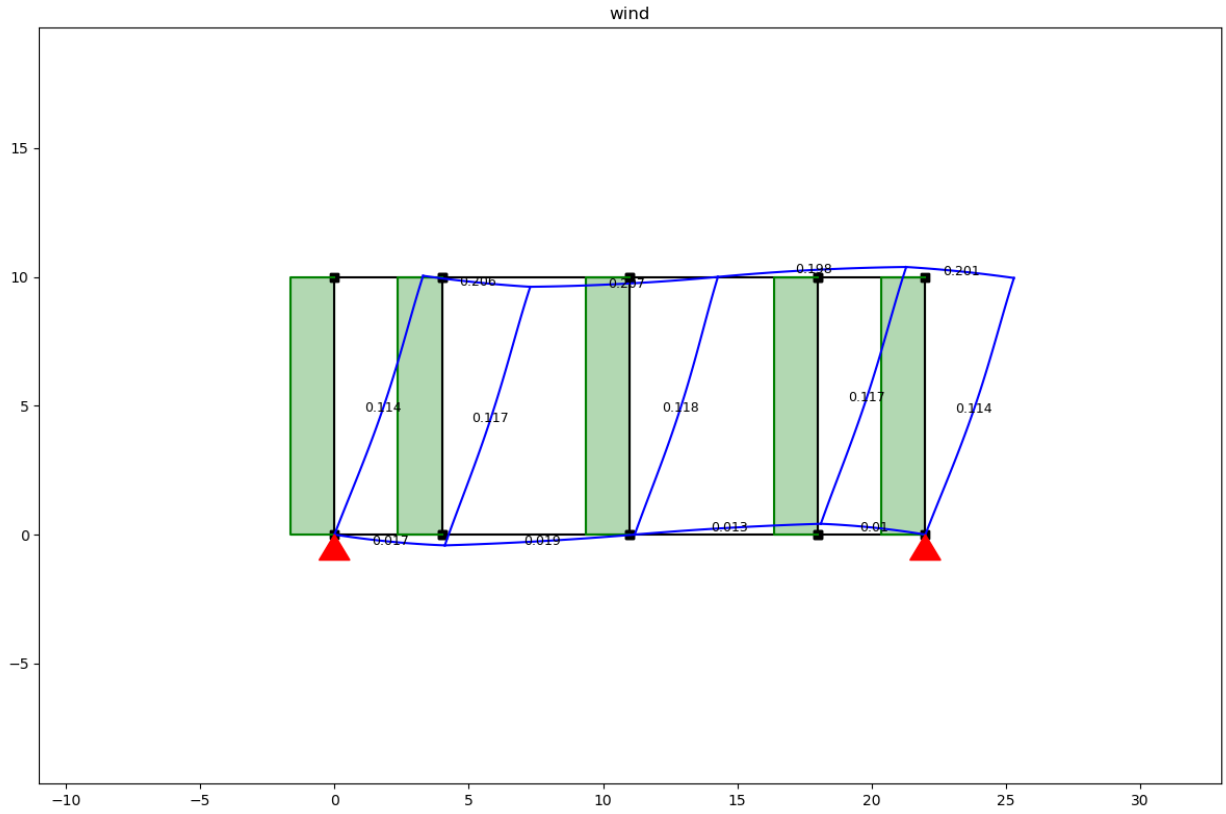
```
combination.add_load_case(lc_wind, 1.5)
combination.add_load_case(lc_cables, factor=1.2)
```

Now we can make a separate calculation for every load case and for the whole load combination. We solve the combination by calling the *solve* method and passing our *SystemElements* model. The *solve* method returns a dictionary where the keys are the load cases and the values are the unique *SystemElement* objects for every load case. There is also a key *combination* in the results dictionary.

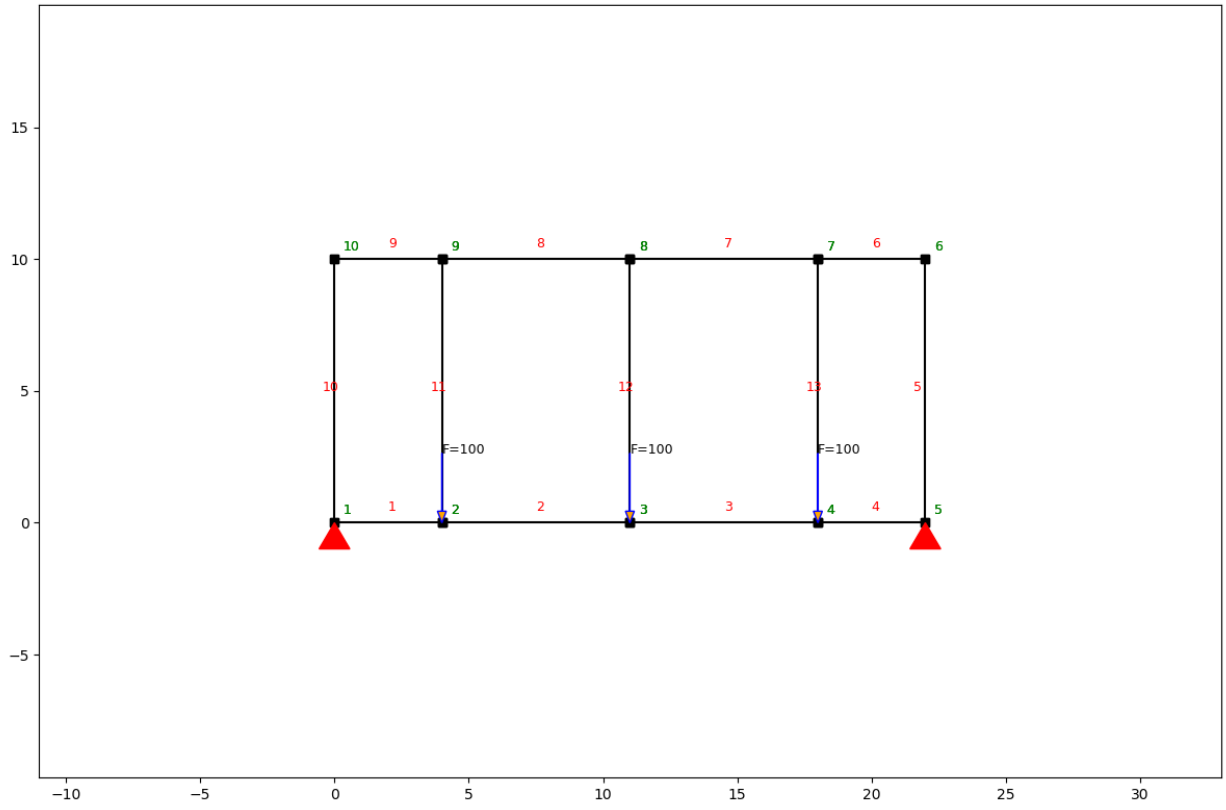
```
results = combination.solve(ss)

for k, ss in results.items():
    results[k].show_structure()
    results[k].show_displacement(show=False)
    plt.title(k)
    plt.show()
```

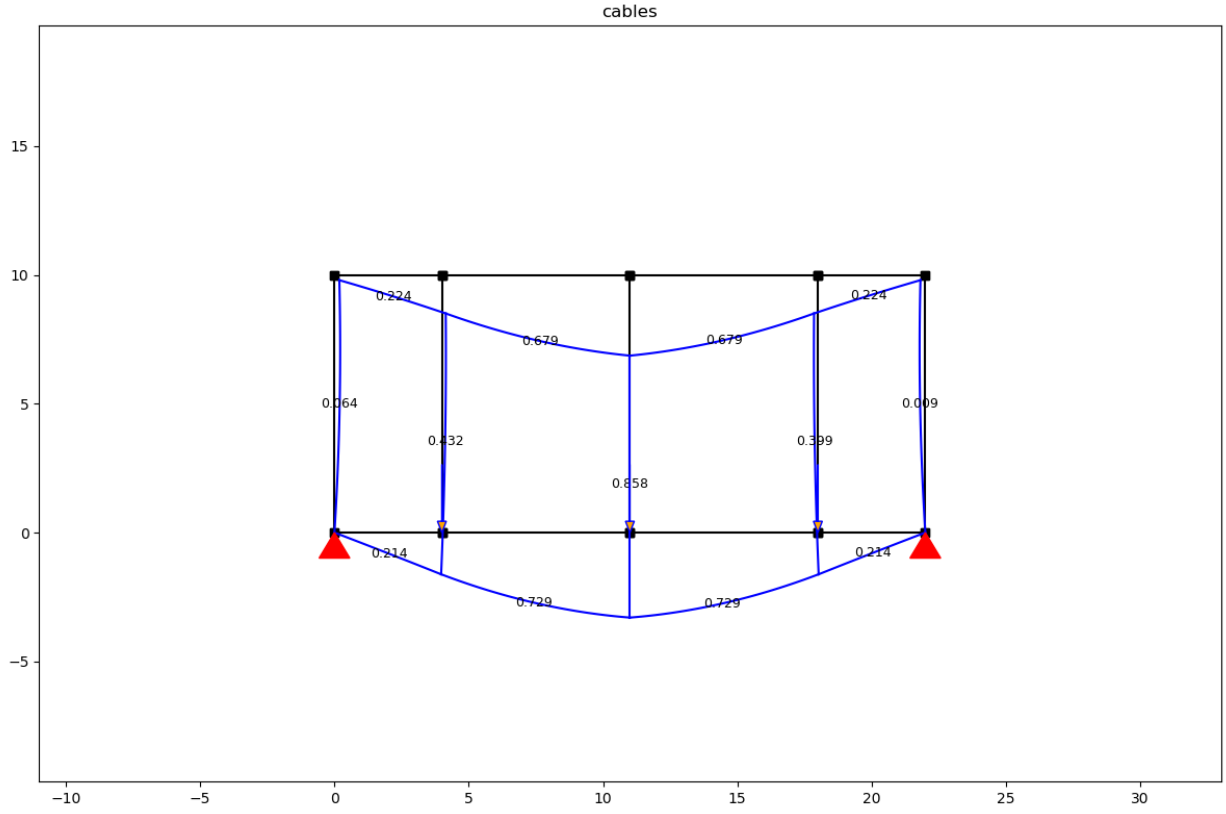
Load case wind



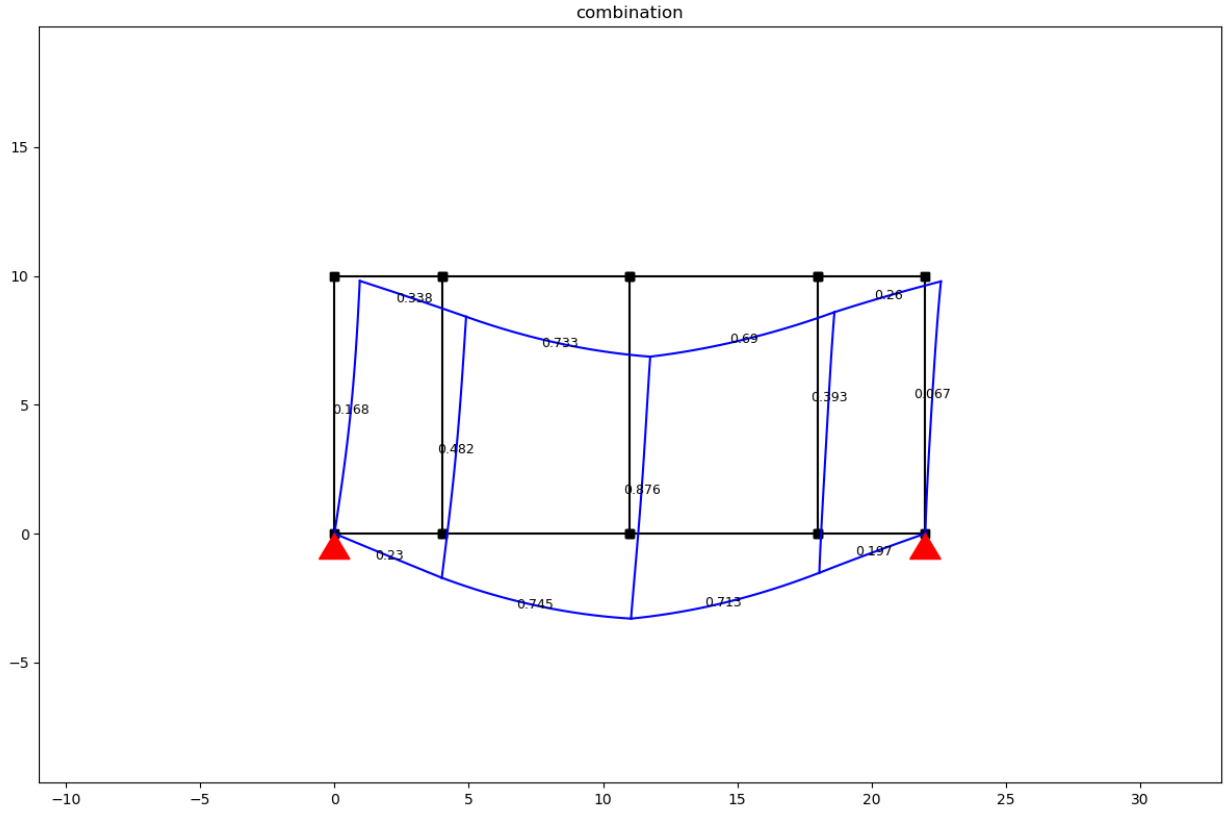


Load case cables


F=100
F=100
F=100


cables
0.224
0.679
0.679
0.224
0.364
0.009
0.432
0.339
0.658
0.214
0.729
0.729
0.214

Combination



2.8.3 Load case class

class anastruct.fem.util.load.**LoadCase**(*name*)

Group different loads in a load case

__init__(*name*)

Parameters **name** – (str) Name of the load case

dead_load(*element_id*, *g*)

Apply a dead load in kN/m on elements.

Parameters

- **element_id** – (int/ list) representing the element ID
- **g** – (flt/ list) Weight per meter. [kN/m] / [N/m]

moment_load(*node_id*, *Ty*)

Apply a moment on a node.

Parameters

- **node_id** – (int/ list) Nodes ID.
- **Ty** – (flt/ list) Moments acting on the node.

point_load(*node_id*, *Fx=0*, *Fy=0*, *rotation=0*)

Apply a point load to a node.

Parameters

- **node_id** – (int/ list) Nodes ID.
- **Fx** – (flt/ list) Force in global x direction.
- **Fy** – (flt/ list) Force in global x direction.
- **rotation** – (flt/ list) Rotate the force clockwise. Rotation is in degrees.

q_load(*q*, *element_id*, *direction='element'*)
Apply a q-load to an element.

Parameters

- **element_id** – (int/ list) representing the element ID
- **q** – (flt) value of the q-load
- **direction** – (str) "element", "x", "y"

2.8.4 Load combination class

class anastruct.fem.util.load.**LoadCombination**(*name*)

__init__(*name*)
Initialize self. See help(type(self)) for accurate signature.

add_load_case(*lc*, *factor*)
Add a load case to the load combination.

Parameters

- **lc** – (anastruct.fem.util.LoadCase)
- **factor** – (flt) Multiply all the loads in this LoadCase with this factor.

solve(*system*, *force_linear=False*, *verbosity=0*, *max_iter=200*, *geometrical_non_linear=False*, ***kwargs*)
Evaluate the Load Combination.

Parameters

- **system** – (anastruct.fem.system.SystemElements) Structure to apply loads on.
- **force_linear** – (bool) Force a linear calculation. Even when the system has non linear nodes.
- **verbosity** – (int) 0: Log calculation outputs. 1: silence.
- **max_iter** – (int) Maximum allowed iterations.
- **geometrical_non_linear** – (bool) Calculate second order effects and determine the buckling factor.

Returns (ResultObject)

Development ****kwargs:

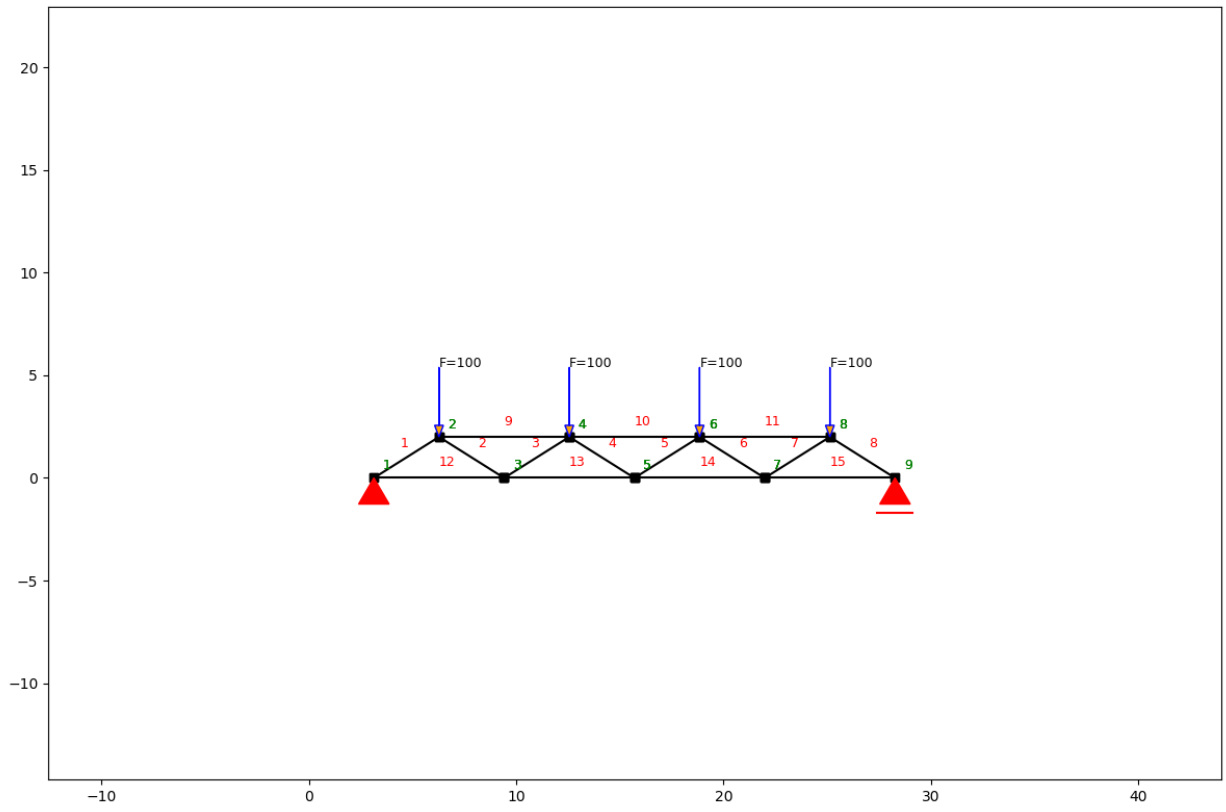
param naked (bool) Whether or not to run the solve function without doing post processing.

param discretize_kwargs When doing a geometric non linear analysis you can reduce or increase the number of elements created that are used for determining the buckling_factor

2.9 Post processing

Besides plotting the result, it is also possible to query numerical results. We'll go through them with a simple example.

```
from anastruct import SystemElements
import matplotlib.pyplot as plt
import numpy as np

ss = SystemElements()
element_type = 'truss'

# create triangles
x = np.arange(1, 10) * np.pi
y = np.cos(x)
y -= y.min()
ss.add_element_grid(x, y, element_type=element_type)

# add top girder
ss.add_element_grid(x[1:-1][::2], np.ones(x.shape) * y.max(), element_type=element_
↪type)

# add bottom girder
ss.add_element_grid(x[::2], np.ones(x.shape) * y.min(), element_type=element_type)

# supports
ss.add_support_hinged(1)
ss.add_support_roll(-1, 2)

# loads
ss.point_load(node_id=np.arange(2, 9, 2), Fy=-100)

ss.solve()
ss.show_structure()
```



2.9.1 Node results system

`SystemElements.`**`get_node_results_system`**(*node_id=0*)

These are the node results. These are the opposite of the forces and displacements working on the elements and may seem counter intuitive.

Parameters **`node_id`** – (integer) representing the node's ID. If integer = 0, the results of all nodes are returned

Returns

if node_id == 0: (list)

Returns a list containing tuples with the results:

```
[(id, Fx, Fy, Ty, ux, uy, phi_y), (id, Fx, Fy...), () .. ]
```

if node_id > 0: (dict)

Example

We can use this method to query the reaction forces of the supports.

```
print(ss.get_node_results_system(node_id=1)['Fy'], ss.get_node_results_system(node_
↪id=-1)['Fy'])
```

output

```
199.9999963370603 200.00000366293816
```

2.9.2 Node displacements

SystemElements.**get_node_displacements**(*node_id=0*)

Parameters **node_id** – (int) Represents the node's ID. If integer = 0, the results of all nodes are returned.

Returns

if node_id == 0: (list)

Returns a list containing tuples with the results:

```
[(id, ux, uy, phi_y), (id, ux, uy, phi_y),  ... (id, ux, uy, phi_y) ]
```

if node_id > 0: (dict)

Example

We can also query node displacements on a node level (So not opposite, as with the system node results.) To get the maximum displacements at node 5 (the middle of the girder) we write.

```
print(ss.get_node_displacements(node_id=5))
```

output

```
{'id': 5, 'ux': 0.25637068208810526, 'uy': -2.129555426623823, 'phi_y': 7.
↪11561178433554e-09}
```

2.9.3 Range of node displacements

SystemElements.**get_node_result_range**(*unit*)

Query a list with node results.

param unit (str) - 'uy' - 'ux' - 'phi_y'

Returns (list)

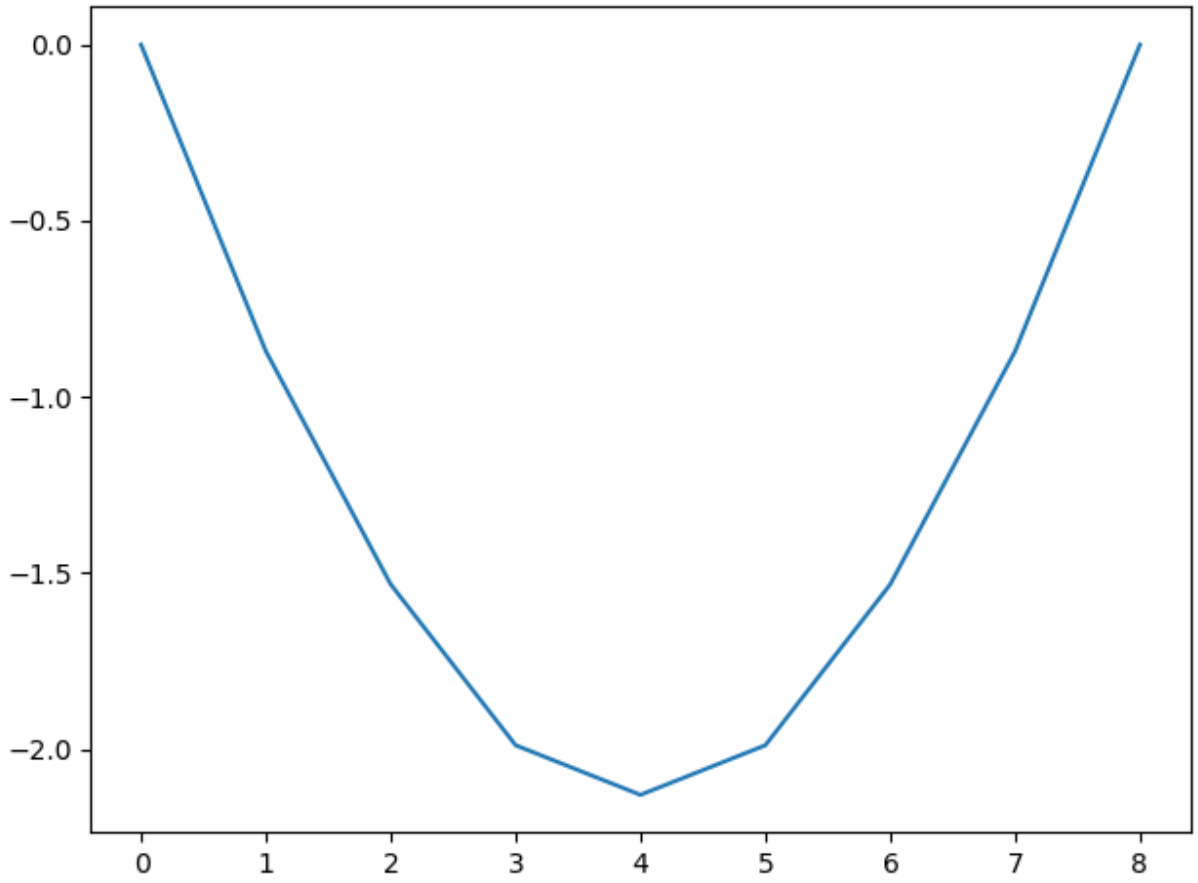
Example

To get the deflection of all nodes in the girder, we use the *get_node_result_range* method.

```
deflection = ss.get_node_result_range('uy')
print(deflection)
plt.plot(deflection)
plt.show()
```

output

```
[-0.0, -0.8704241688181067, -1.5321803865868588, -1.9886711039126856, -2.
↪129555426623823, -1.9886710728856773, -1.5321805004461058, -0.8704239570876975, -0.
↪0]
```



2.9.4 Element results

`SystemElements.`**`get_element_results`**(*element_id=0*, *verbose=False*)

Parameters

- **`element_id`** – (int) representing the elements ID. If elementID = 0 the results of all elements are returned.

- **verbose** – (bool) If set to True the numerical results for the deflection and the bending moments are returned.

Returns

if node_id == 0: (list)

Returns a list containing tuples with the results:

```
[(id, length, alpha, u, N_1, N_2), (id, length, alpha, u, N_1, N_2), ... (id,␣
↪length, alpha, u, N_1, N_2)]
```

if node_id > 0: (dict)

Example

Axial force, shear force and extension are properties of the elements and not of the nodes. To get this information, we need to query the results from the elements.

Let's find the value of the maximum axial compression force, which is in element 10.

```
print(ss.get_element_results(element_id=10)['N'])
```

output

```
-417.395490645013
```

2.9.5 Range of element results

SystemElements.**get_element_result_range**(*unit*)

Useful when added lots of elements. Returns a list of all the queried unit.

Parameters **unit** – (str) - 'shear' - 'moment' - 'axial'

Returns (list)

Example

We can of course think of a structure where we do not know where the maximum axial compression force will occur. So let's check if our assumption is correct and that the maximum force is indeed in element 10.

We query all the axial forces. The returned item is an ordered list. Because Python starts counting from zero, and our elements start counting from one, we'll need to add one to get the right element. Here we'll see that the minimum force (compression is negative) is indeed in element 10.

```
print(np.argmin(ss.get_element_result_range('axial')) + 1)
```

output

```
10
```

2.10 Element/ node interaction

Once you structures will get more and more complex, it will become harder to keep count of element id and node ids. The *SystemElements* class therefore has several methods that help you:

- Find a node id based on a x- and y-coordinate
- Find the nearest node id based on a x- and y-coordinate
- Get all the coordinates of all nodes.

2.10.1 Find node id based on coordinates

SystemElements.**find_node_id**(*vertex*)

Retrieve the ID of a certain location.

Parameters **vertex** – (Vertex/ list/ tpl) Vertex_xz, [x, y], (x, y)

Returns (int/ None) id of the node at the location of the vertex

2.10.2 Find nearest node id based on coordinates

SystemElements.**nearest_node**(*dimension*, *val*)

Retrieve the nearest node ID.

Parameters

- **dimension** – (str) "both", 'x', 'y' or 'z'
- **val** – (flt) Value of the dimension.

Returns (int) ID of the node.

2.10.3 Query node coordinates

SystemElements.**nodes_range**(*dimension*)

Retrieve a list with coordinates x or z (y).

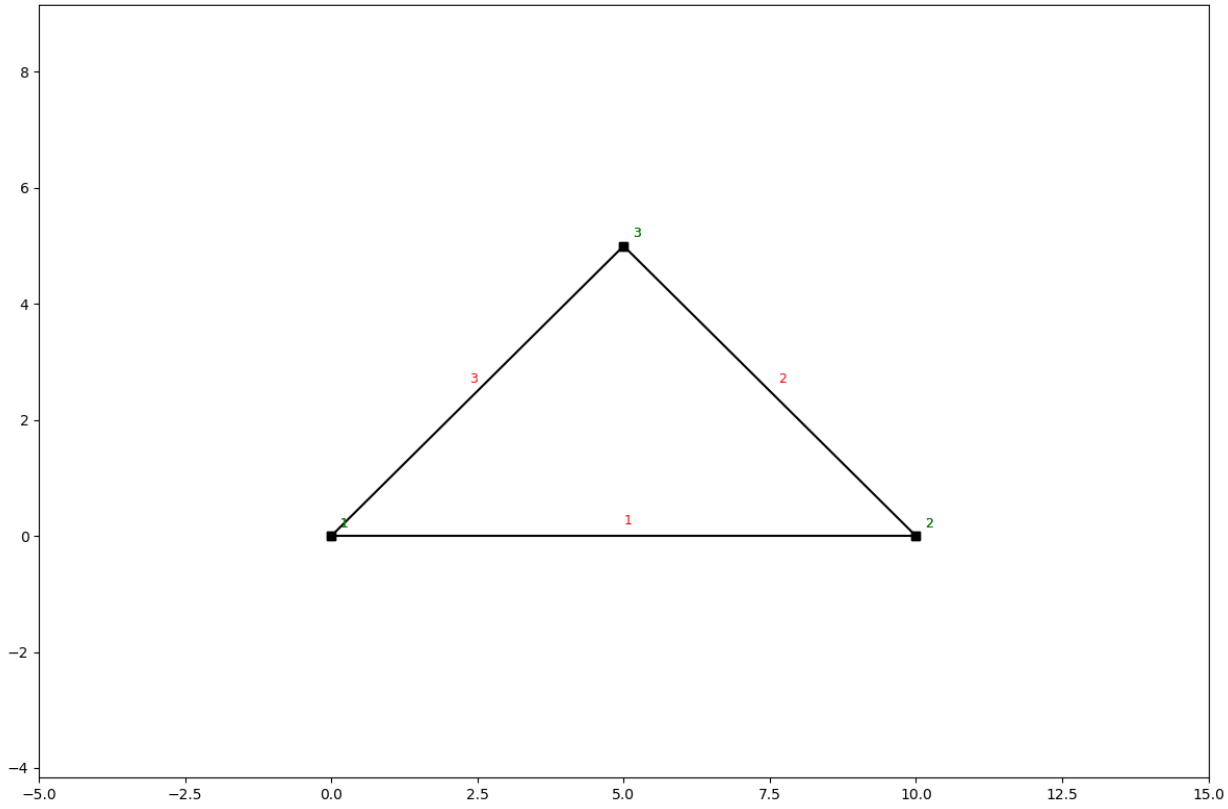
Parameters **dimension** – (str) "both", 'x', 'y' or 'z'

Returns (list)

2.11 Vertex

Besides coordinates as a list such as *[[x1, y1], [x2, y2]]* anaStruct also has a utility node class called *Vertex* Objects from this class can used to model elements and allow simple arithmetic on coordinates. Modelling with *Vertex* objects can make it easier to model structures.

```
from anastruct import SystemElements, Vertex

point_1 = Vertex(0, 0)
point_2 = point_1 + [10, 0]
point_3 = point_2 + [-5, 5]

ss = SystemElements()
ss.add_element([point_1, point_2])
ss.add_element(point_3)
ss.add_element(point_1)

ss.show_structure()
```



2.12 Saving

What do you need to save? You've got a script that represents your model. Just run it!

If you do need to save a model, you can save it with standard python object pickling.

```
import pickle
from anastruct import SystemElements

ss = SystemElements()

# save
with open('my_structure.pkl', 'wb') as f:
```

(continues on next page)

(continued from previous page)

```
    pickle.dump(ss, f)

# load
with open('my_structure.pkl', 'rb') as f:
    ss = pickle.load(f)
```

2.13 Examples

Take a look at this blog post. Here anaStruct was used to do a non linear water accumulation analysis.

Water accumulation blog post.

Index

Symbols

__init__() (anastruct.fem.system.SystemElements method), 4
__init__() (anastruct.fem.util.load.LoadCase method), 32
__init__() (anastruct.fem.util.load.LoadCombination method), 33

A

add_element() (anastruct.fem.system.SystemElements method), 11
add_element_grid() (anastruct.fem.system.SystemElements method), 13
add_load_case() (anastruct.fem.util.load.LoadCombination method), 33
add_multiple_elements() (anastruct.fem.system.SystemElements method), 12
add_support_fixed() (anastruct.fem.system.SystemElements method), 17
add_support_hinged() (anastruct.fem.system.SystemElements method), 16
add_support_roll() (anastruct.fem.system.SystemElements method), 17
add_support_spring() (anastruct.fem.system.SystemElements method), 19
add_truss_element() (anastruct.fem.system.SystemElements method), 15

D

dead_load() (anastruct.fem.util.load.LoadCase method), 32
discretize() (anastruct.fem.system.SystemElements method), 15

F

find_node_id() (anastruct.fem.system.SystemElements method), 39

G

get_element_result_range() (anastruct.fem.system.SystemElements method), 38
get_element_results() (anastruct.fem.system.SystemElements method), 37
get_node_displacements() (anastruct.fem.system.SystemElements method), 36
get_node_result_range() (anastruct.fem.system.SystemElements method), 36
get_node_results_system() (anastruct.fem.system.SystemElements method), 35

I

insert_node() (anastruct.fem.system.SystemElements method), 15

L

LoadCase (class in anastruct.fem.util.load), 32
LoadCombination (class in anastruct.fem.util.load), 33

M

moment_load() (anastruct.fem.system.SystemElements method), 20
moment_load() (anastruct.fem.util.load.LoadCase method), 32

N

nearest_node() (anastruct.fem.system.SystemElements method), 39

nodes_range() (anastruct.fem.system.SystemElements method), 39

P

point_load() (anastruct.fem.system.SystemElements method), 20
point_load() (anastruct.fem.util.load.LoadCase method), 32

Q

q_load() (anastruct.fem.system.SystemElements method), 21
q_load() (anastruct.fem.util.load.LoadCase method), 33

R

remove_loads() (anastruct.fem.system.SystemElements method), 22

S

show_axial_force() (anastruct.fem.system.SystemElements method), 23
show_bending_moment() (anastruct.fem.system.SystemElements method), 23
show_displacement() (anastruct.fem.system.SystemElements method), 24
show_reaction_force() (anastruct.fem.system.SystemElements method), 24
show_shear_force() (anastruct.fem.system.SystemElements method), 24
show_structure() (anastruct.fem.system.SystemElements method), 23
solve() (anastruct.fem.system.SystemElements method), 26
solve() (anastruct.fem.util.load.LoadCombination method), 33
SystemElements (class in anastruct.fem.system), 3