anaStruct Documentation

Release 1.0

Ritchie Vink

Feb 24, 2023

CONTENTS:

CHAPTER

ONE

INDICES AND TABLES

- genindex
- modindex
- search

1.1 Installation

You will need Python and a few packages as pre-requisites of the anaStruct on your system.

1.1.1 Install the Python

Linux

Python is normally delivered on any Linux distribution. So you basically just need to call the python keyword which is stored on your operating system's path. To call version 3 of python on Linux you can use *python3* in the terminal. You can check installation status and version of the python on your system.

```
python3 --version
```

In case you are missing the python on your system, you can install it from the repositories of your system. For instance, on Ubuntu, you can easily install python 3.9 with the following commands:

```
sudo apt-get update
sudo apt-get install python3.9
```

Windows

On windows (and for other OS's too) you can download the installation source of the version you prefer from the Python's website. You can choose between the various versions and cpu architectures.

Mac

For Mac OS install Python 3 using homebrew

```
brew install python
```

1.1.2 Install the prerequisites

You will need the NumPy and SciPy. packages to be able to use the anaStruct package. However, if you are using the pip to install the package, it will take care of all dependencies and their versions.

1.1.3 Install the anaStruct

You can install anaStruct with pip! If you like to use the computational backend of the package without having the plotting features, simply run the code below in the terminal. Pip will install a headless version of anaStruct (with no plotting abilities).

```
python -m pip install anastruct
```

Otherwise you can have a full installation using the following code in your terminal.

```
python -m pip install anastruct[plot]
```

In case you need a specific version of the package, that's possible too. Simple declare the version condition over the code in terminal.

```
python -m pip install anastruct==1.4.1
```

Alternatively, you can build the package from the source by cloning the source from the git repository. Updates are made regularly released on PyPi, and if you'd like the bleeding edge newest features and fixes, or if you'd like to contribute to the development of anaStruct, then install from github.

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

1.2 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.

1.2.1 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=1.0*, *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.
- **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=1.0*, *mesh=50*)

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

Parameters

- **figsize** (Tuple[float, float]) – Set the standard plotting size.
- **EA** (float) – Standard E * A. Set the standard values of EA if none provided when generating an element.
- **EI** (float) – Standard E * I. Set the standard values of EA if none provided when generating an element.
- **load_factor** (float) – 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.

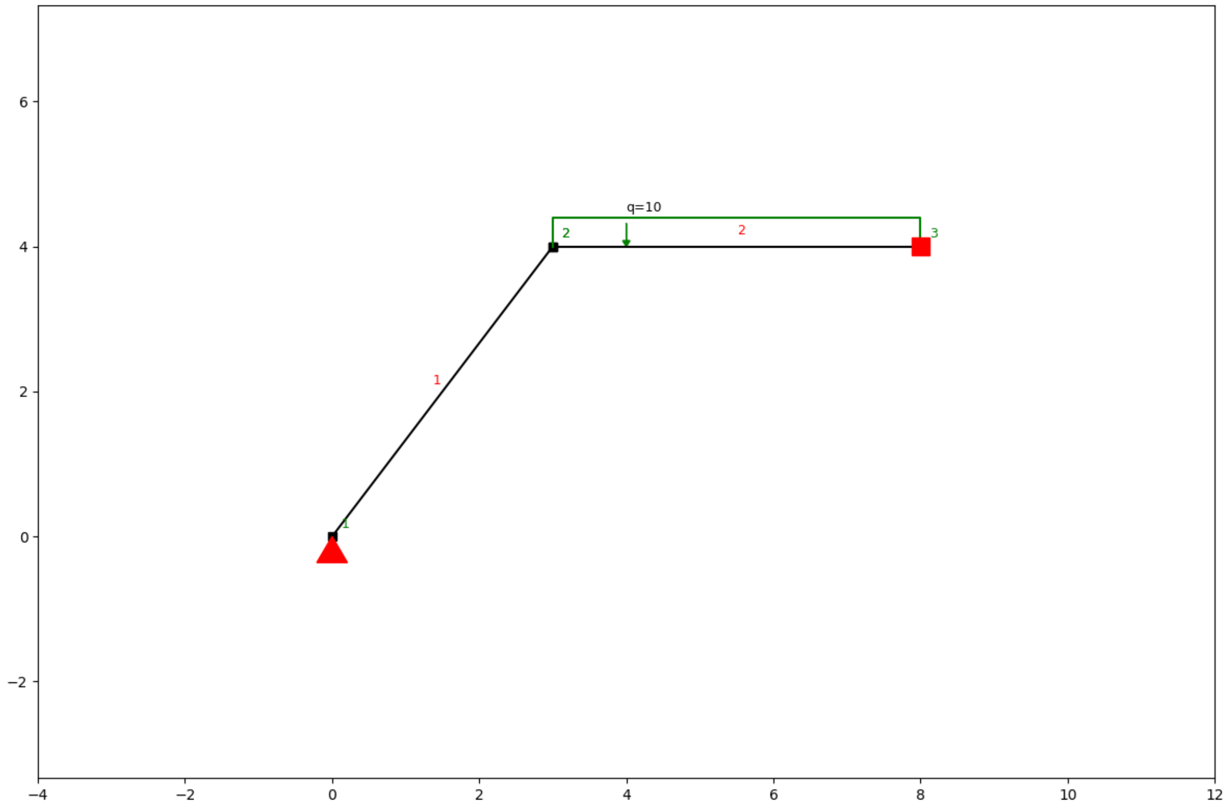
```
ss.add_support_hinged(node_id=1)
ss.add_support_fixed(node_id=3)
```

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

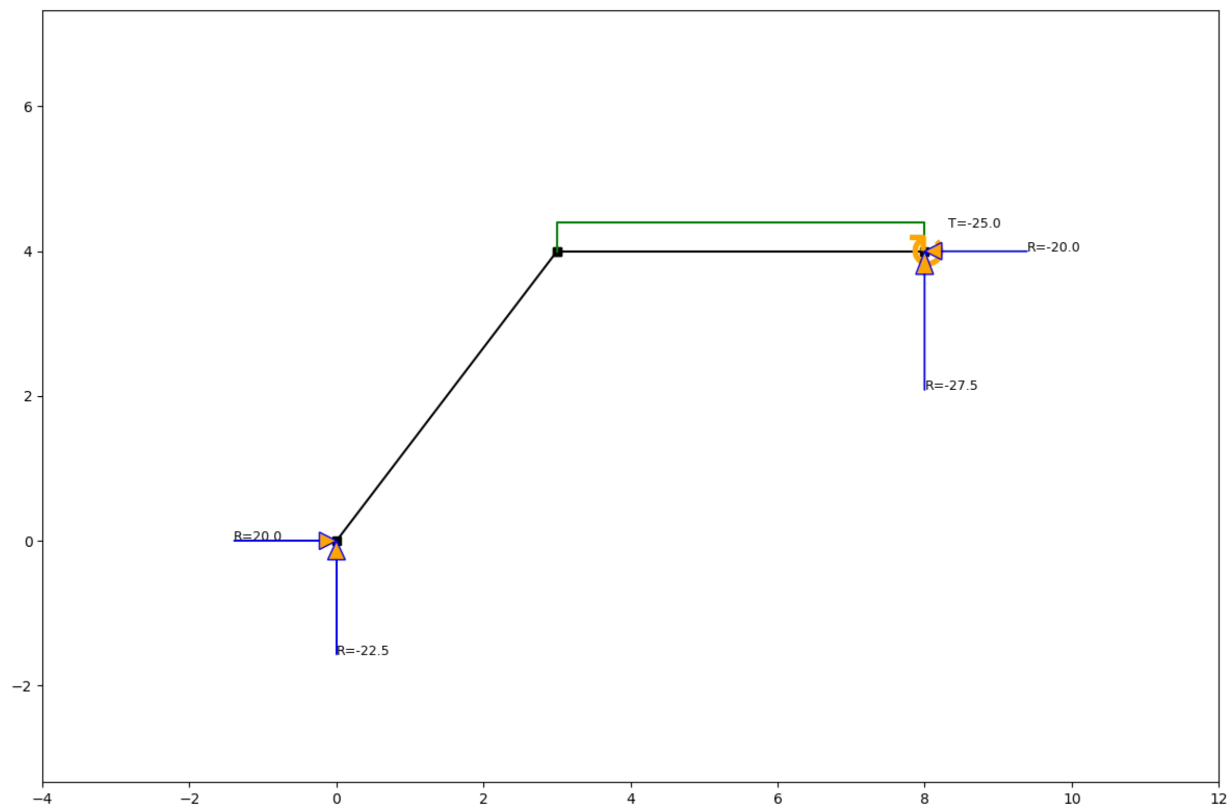
```
ss.q_load(element_id=2, q=-10)
ss.solve()
```

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

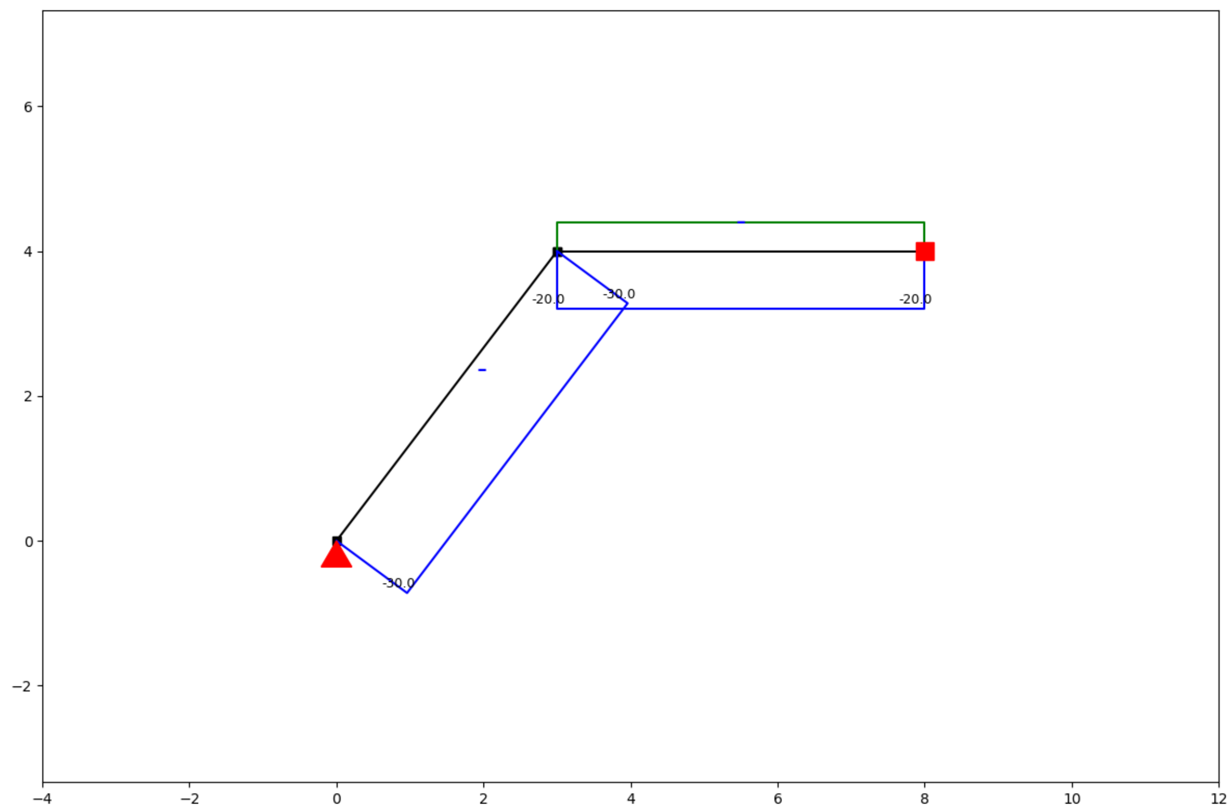
```
ss.show_structure()
```



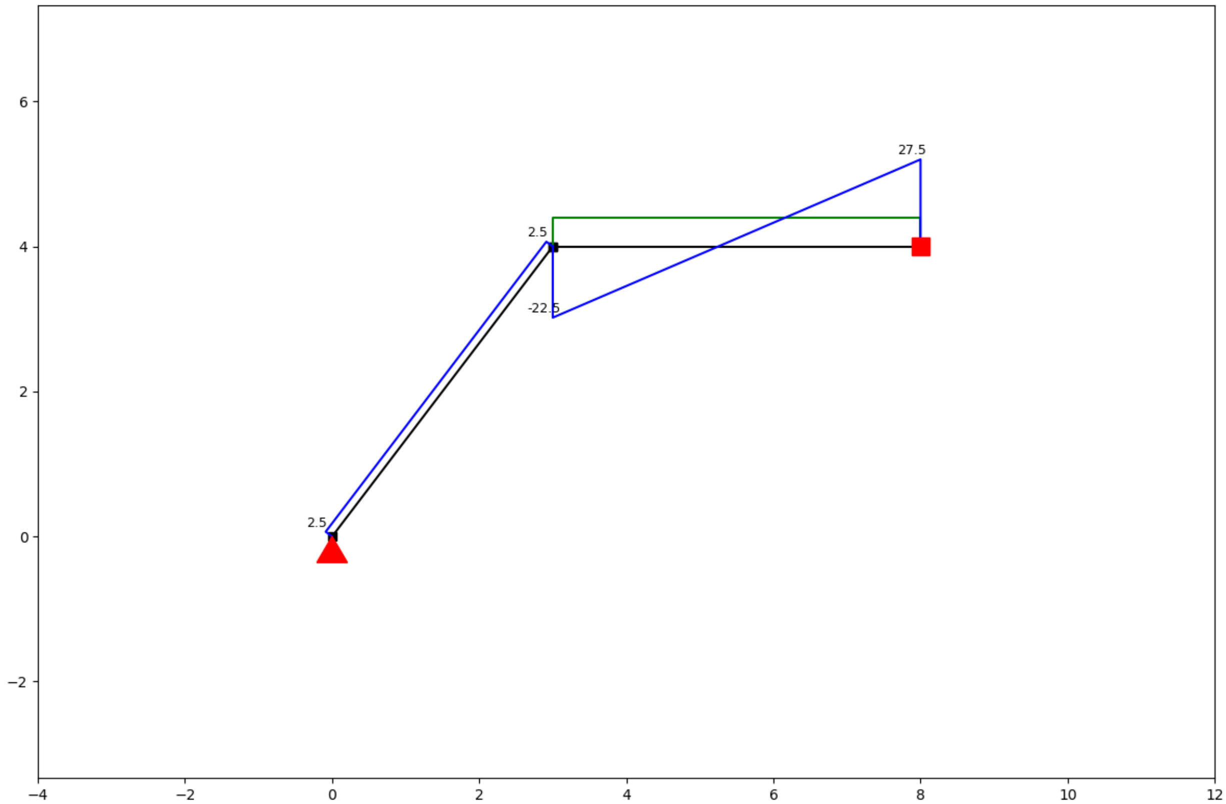

```
ss.show_reaction_force()
```



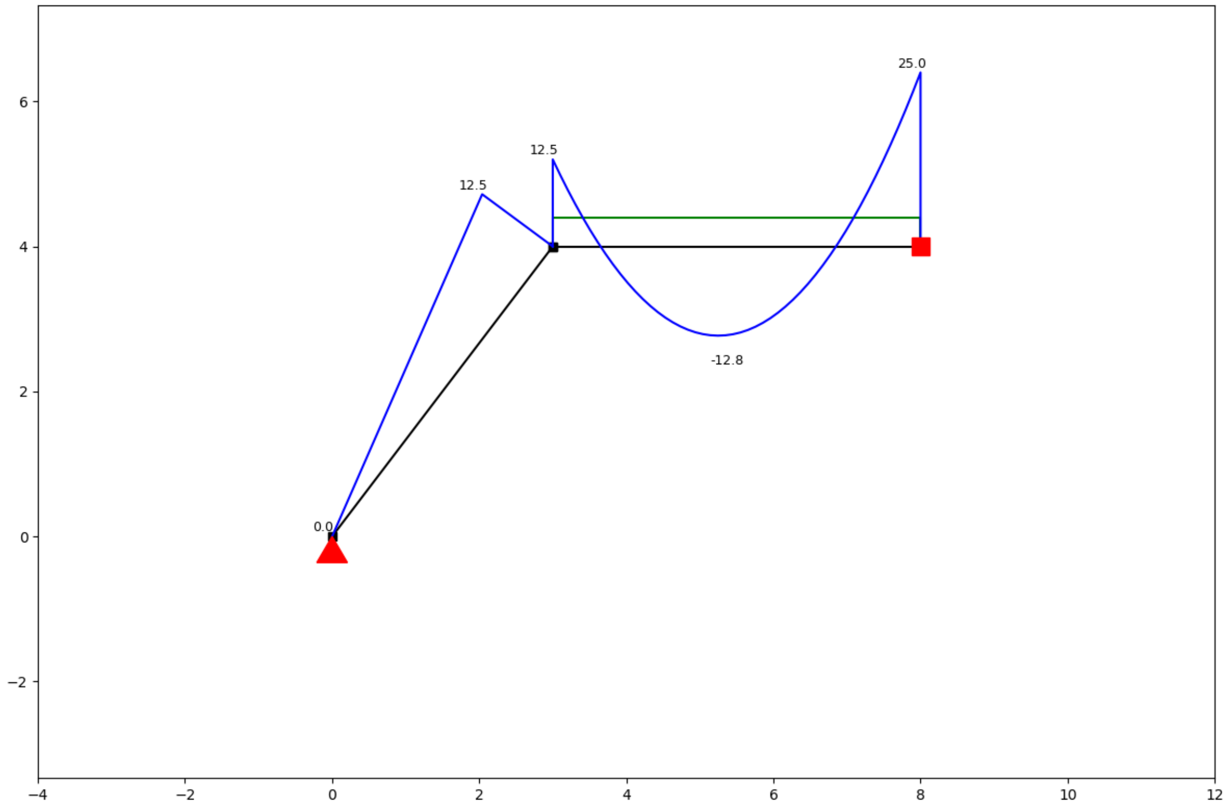

```
ss.show_axial_force()
```



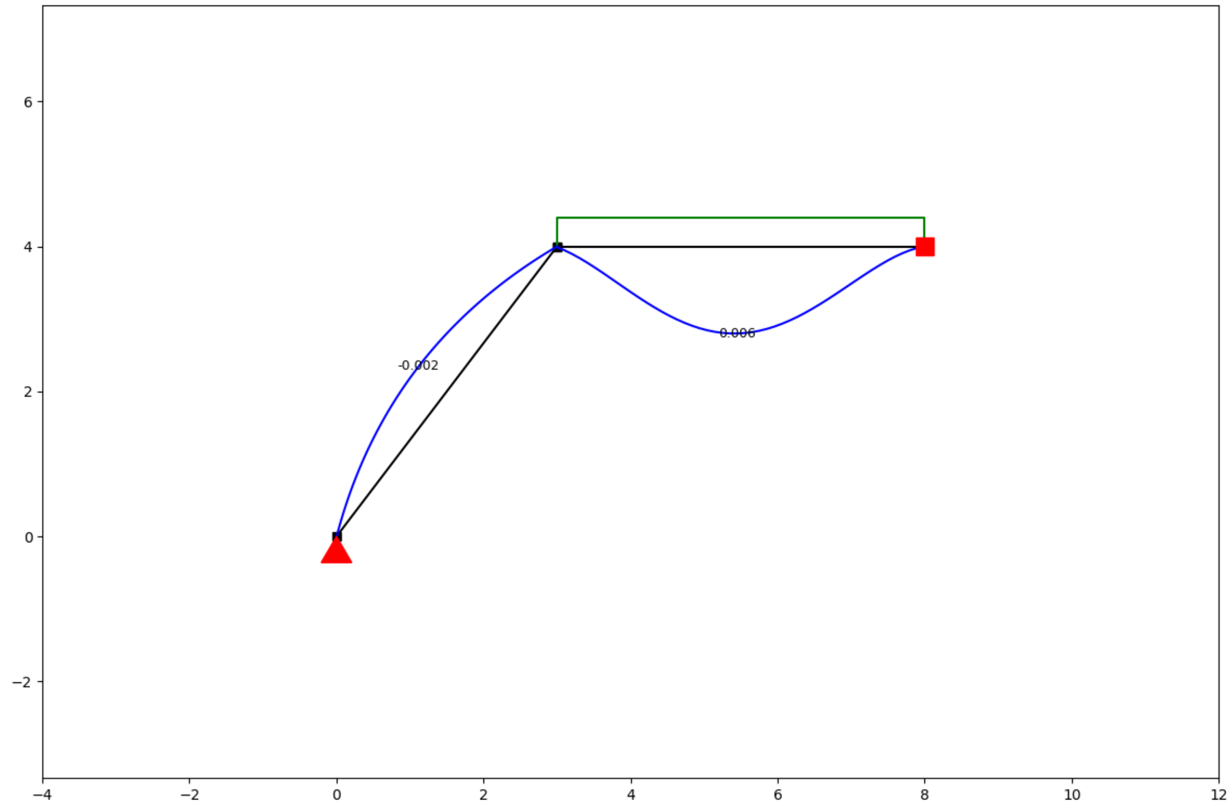

```
ss.show_shear_force()
```




```
ss.show_bending_moment()
```




```
ss.show_displacement()
```



1.3 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 are 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.

1.3.1 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** (`Union[Sequence[Sequence[float]], Sequence[Vertex], Sequence[float], 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** (`Optional[float]`) – EA
- **EI** (`Optional[float]`) – EI
- **g** (`float`) – Weight per meter. [kN/m] / [N/m]
- **mp** (`Optional[Dict[int, float]]`) –

  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** (`Optional[Dict[int, float]]`) – Set a rotational spring or a hinge (k=0) at node 1 or node 2.

  Example

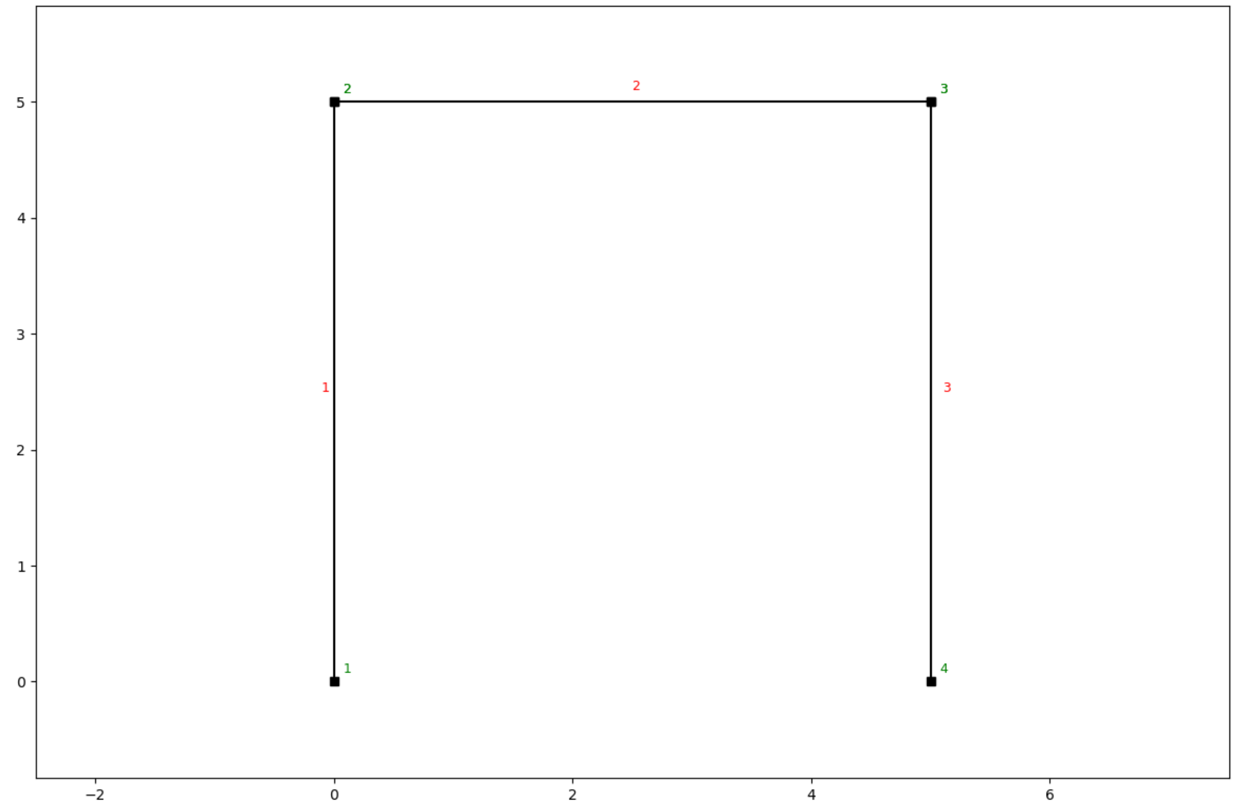
```
spring={1: k
        2: k}


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

Return type
`int`

Returns
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** (Union[Sequence[Sequence[float]], Sequence[Vertex], Sequence[float], Vertex]) – See 'add_element' method
- **n** (Optional[int]) – Number of elements.
- **dl** (Optional[float]) – Distance between the elements nodes.
- **EA** (Optional[float]) – See 'add_element' method
- **EI** (Optional[float]) – See 'add_element' method
- **g** (float) – See 'add_element' method

- **mp** (Optional[Dict[int, float]]) – See 'add_element' method
- **spring** (Optional[Dict[int, float]]) – See 'add_element' method

Keyword Args:

Parameters

- **element_type** – See 'add_element' method
- **first** – Different arguments for the first element
- **last** – Different arguments for the last element
- **steelsection** – Steel section name like IPE 300
- **orient** – Steel section axis for moment of inertia - 'y' and 'z' possible
- **b** – Width of generic rectangle section
- **h** – Height of generic rectangle section
- **d** – Diameter of generic circle section
- **sw** – If true self weight of section is considered as dead load
- **E** – Modulus of elasticity for section material
- **gamma** – Weight of section material per volume unit. [kN/m3] / [N/m3]s

Example

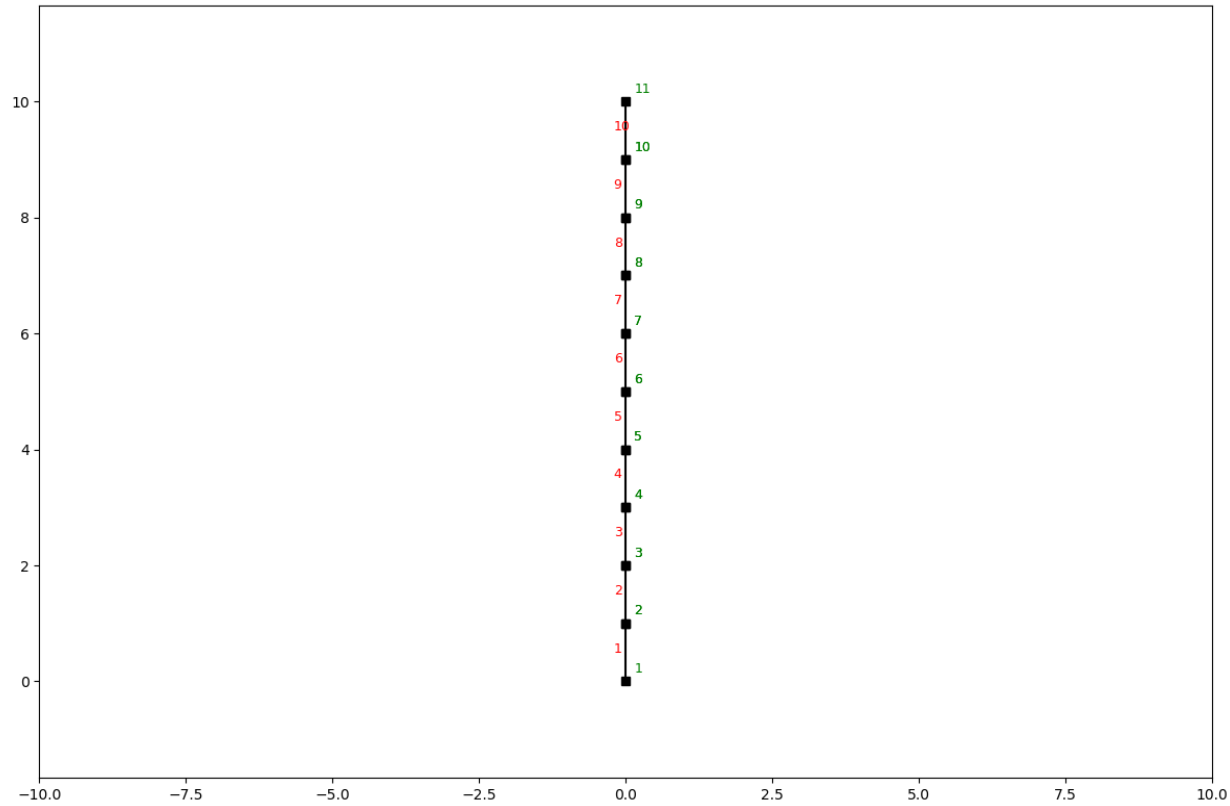
```
last={'EA': 1e3, 'mp': 290}
```

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** (Union[List[float], ndarray]) – x coordinates.
- **y** (Union[List[float], ndarray]) – y coordinates.
- **EA** (Union[List[float], ndarray, None]) – See 'add_element' method
- **EI** (Union[List[float], ndarray, None]) – See 'add_element' method
- **g** (Union[List[float], ndarray, None]) – See 'add_element' method
- **mp** (Optional[Dict[int, float]]) – See 'add_element' method
- **spring** (Optional[Dict[int, float]]) – See 'add_element' method

Paramg **kwargskwargs**

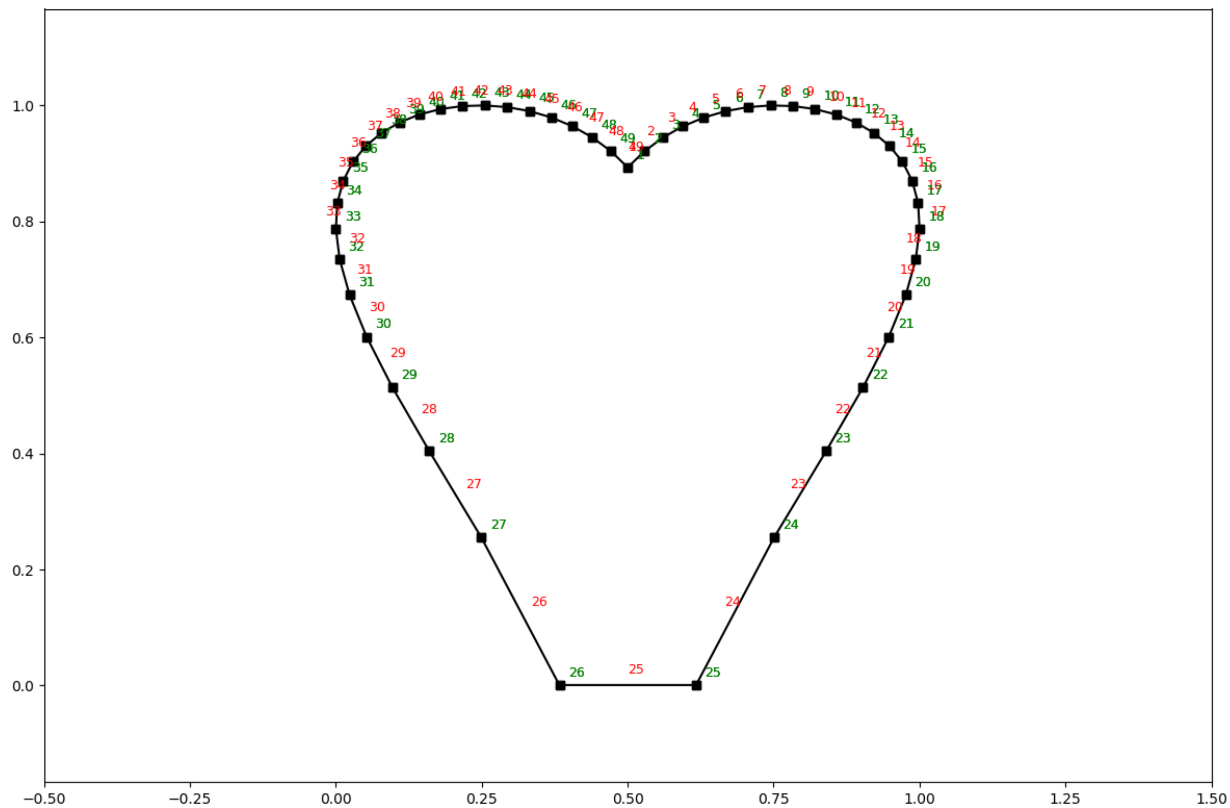
See 'add_element' method

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()
```



1.3.2 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, **kwargs)`

Add an element that only has axial force.

Parameters

- **location** (Union[Sequence[Sequence[float]], Sequence[Vertex], Sequence[float], 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** (Optional[float]) – EA

Return type

int

Returns

Elements ID.

1.3.3 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.

1.3.4 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** (Union[Sequence[float], Vertex, None]) – The nodes of the element or the next node of the element.

  Example

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

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

1.4 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)

1.4.1 add_support_hinged

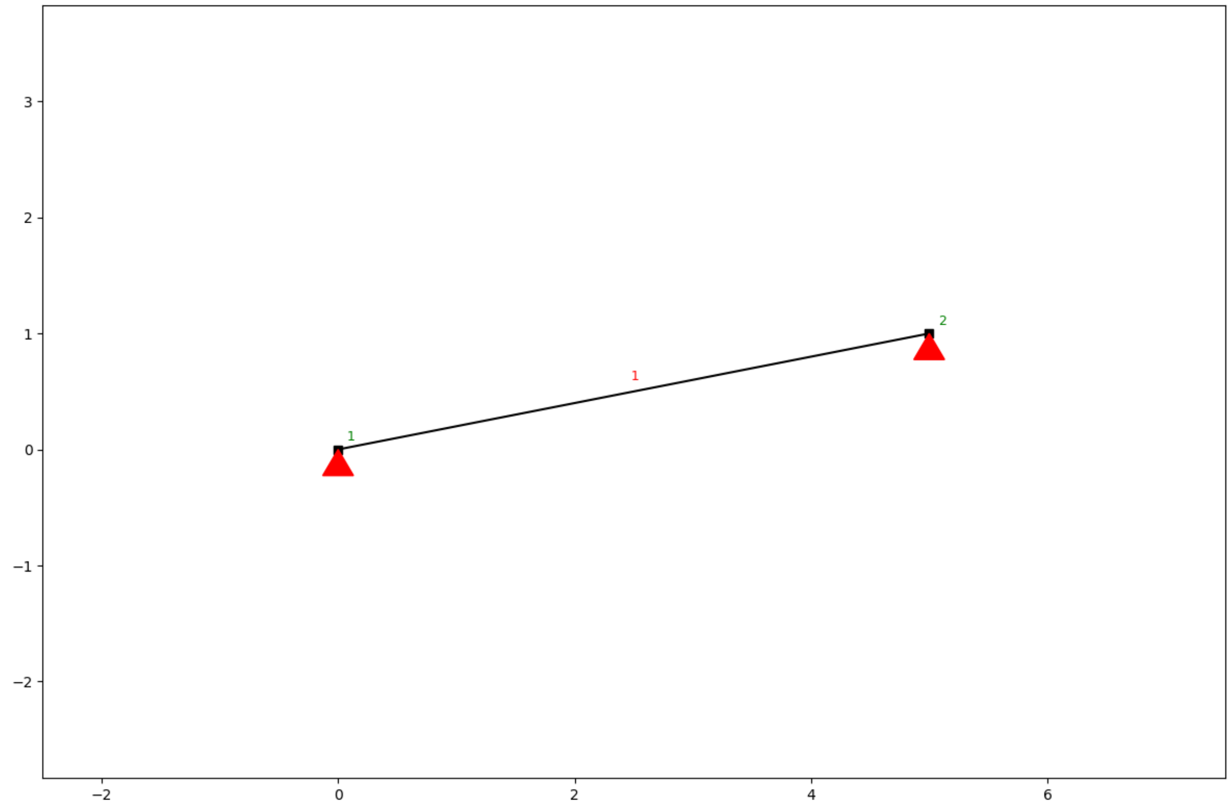
SystemElements.**add_support_hinged**(*node_id*)

Model a hinged support at a given node.

Parameters
node_id (Union[int, Sequence[int]]) – Represents the nodes ID

Example

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



1.4.2 add_support_roll

`SystemElements.`**`add_support_roll`**`(`*node_id*, *direction='x'*, *angle=None*, *rotate=True*`)`

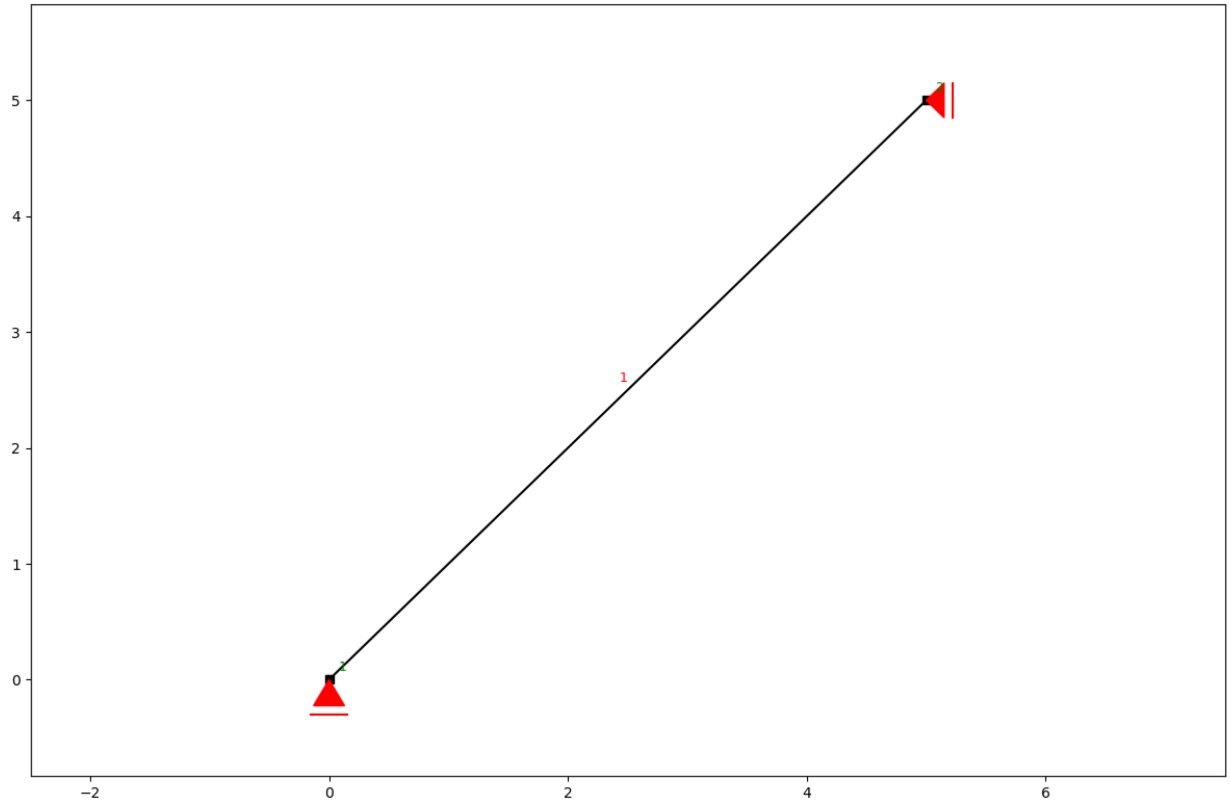
Adds a rolling support at a given node.

Parameters

- **`node_id`** (`Union[Sequence[int], int]`) – Represents the nodes ID
- **`direction`** (`Union[Sequence[Union[str, int]], str, int]`) – Represents the direction that is free: 'x', 'y'
- **`angle`** (`Union[Sequence[Optional[float]], float, None]`) – Angle in degrees relative to global x-axis. If angle is given, the support will be inclined.
- **`rotate`** (`Union[Sequence[bool], bool]`) – If set to False, rotation at the roller will also be restrained.

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()
```



1.4.3 add_support_fixed

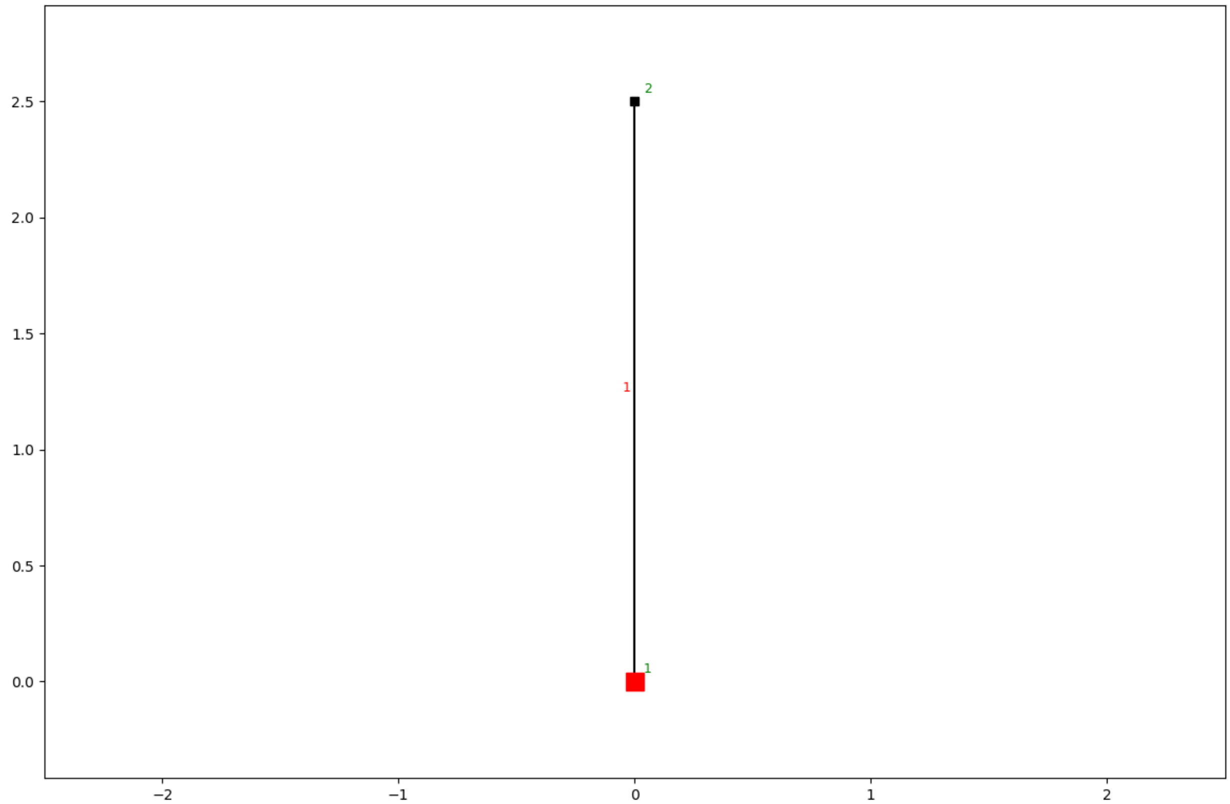
SystemElements.**add_support_fixed**(*node_id*)

Add a fixed support at a given node.

Parameters
node_id (Union[Sequence[int], int]) – Represents the nodes ID

Example

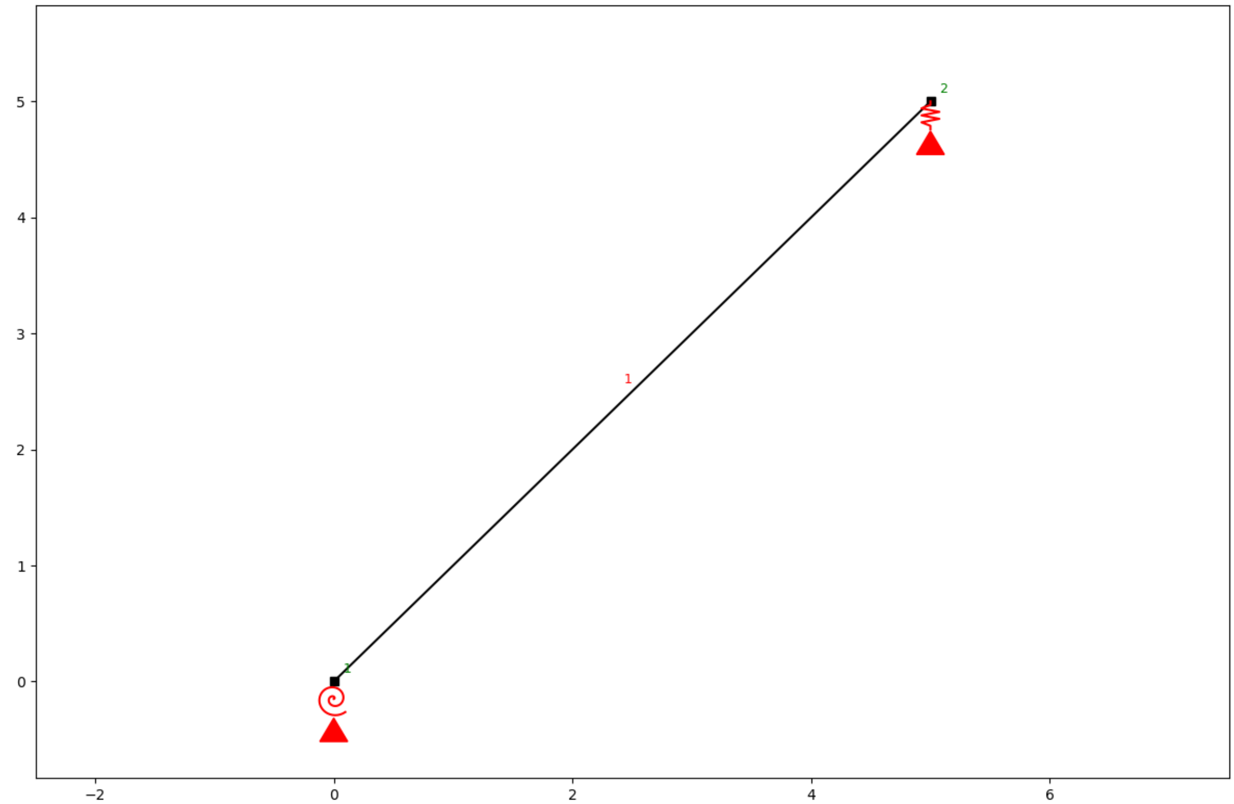
```
ss.add_element(location=[0, 2.5])
ss.add_support_fixed(node_id=1)
ss.show_structure()
```



1.4.4 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`** (`Union[Sequence[int], int]`) – Represents the prevented translation.

  Note

  1 = translation in x

  2 = translation in z

  3 = rotation in y

- **`node_id`** (`Union[Sequence[int], int]`) – Integer representing the nodes ID.
- **`k`** (`Union[Sequence[float], float]`) – Stiffness of the spring
- **`roll`** (`Union[Sequence[bool], bool]`) – If set to True, only the translation of the spring is controlled.

1.5 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.

1.5.1 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=0.0*, *Fy=0.0*, *rotation=0*`)`

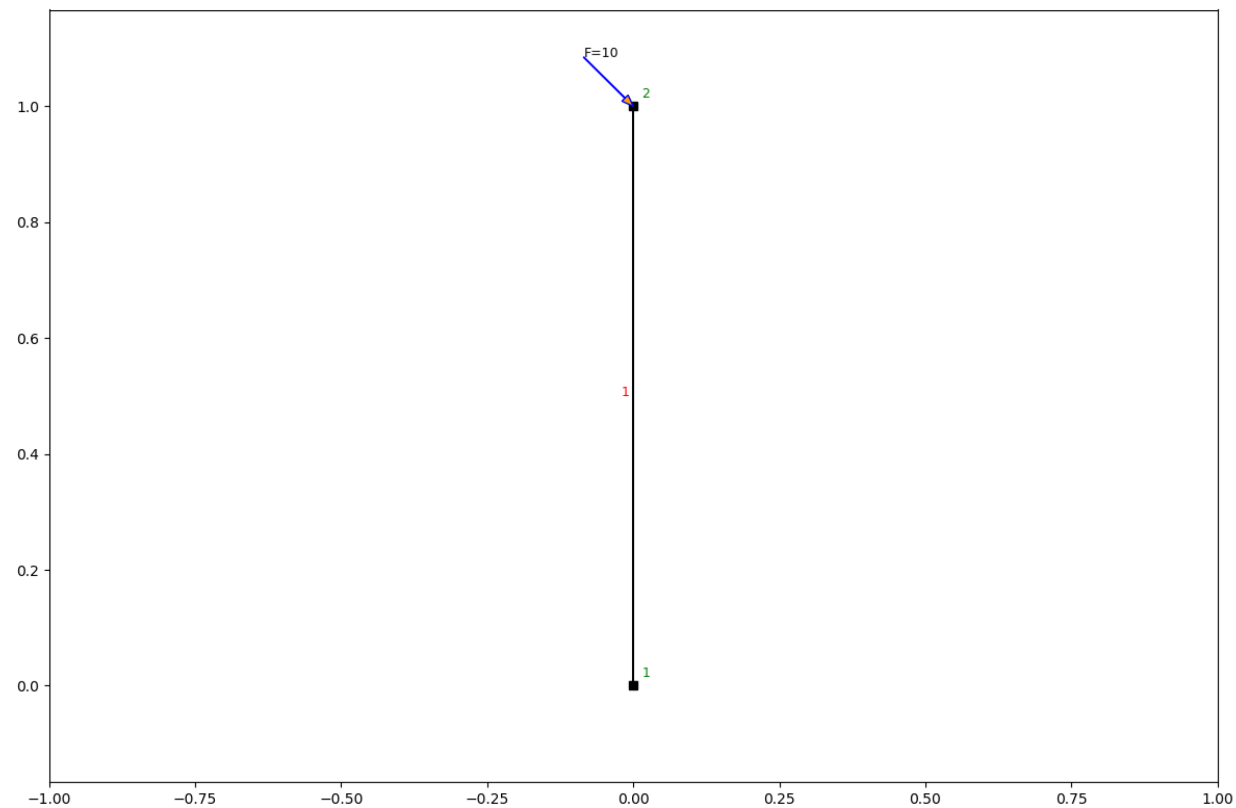
Apply a point load to a node.

Parameters

- **`node_id`** (`Union[int, Sequence[int]]`) – Nodes ID.
- **`Fx`** (`Union[float, Sequence[float]]`) – Force in global x direction.
- **`Fy`** (`Union[float, Sequence[float]]`) – Force in global x direction.
- **`rotation`** (`Union[float, Sequence[float]]`) – 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*)

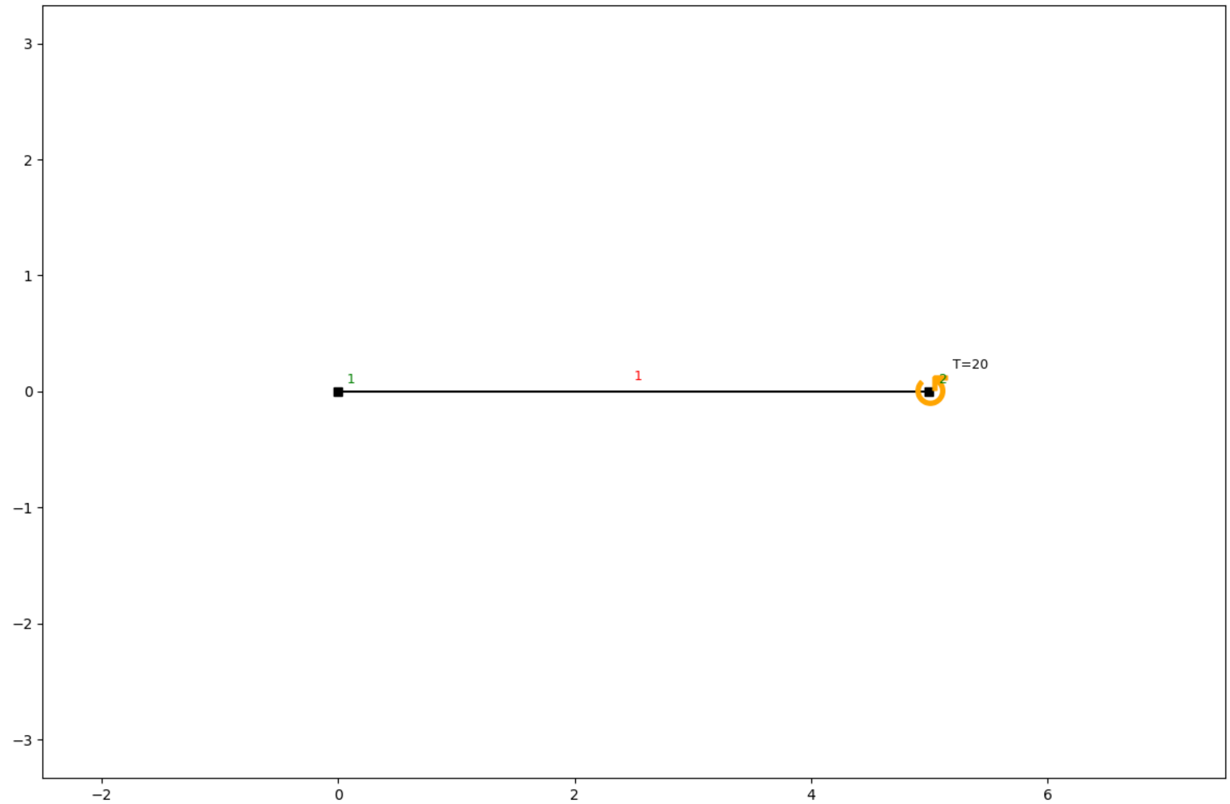
Apply a moment on a node.

Parameters

- **`node_id`** (`Union[int, Sequence[int]]`) – Nodes ID.
- **`Ty`** (`Union[float, Sequence[float]]`) – 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()
```



1.5.2 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'*, *rotation=None*, *q_perp=None*)

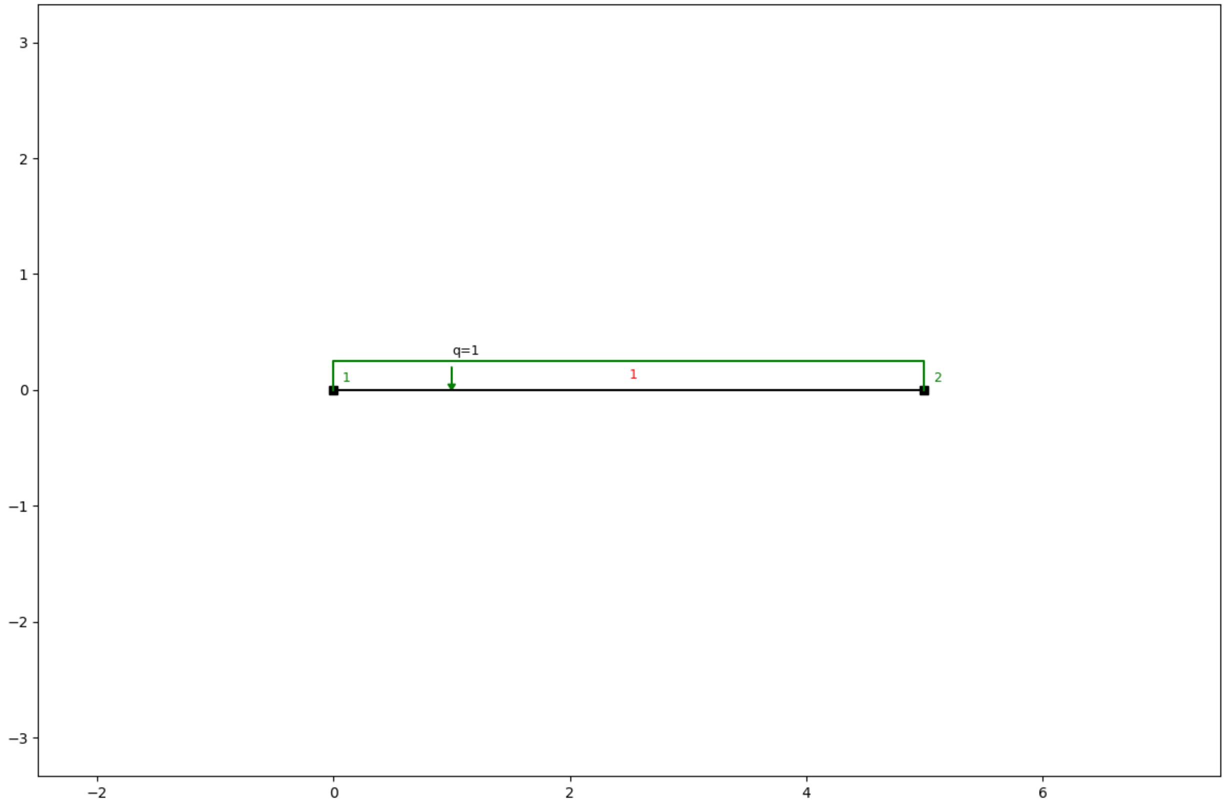
Apply a q-load to an element.

Parameters

- **`element_id`** (`Union[int, Sequence[int]]`) – representing the element ID
- **`q`** (`Union[float, Sequence[float]]`) – value of the q-load
- **`direction`** (`Union[str, Sequence[str]]`) – "element", "x", "y", "parallel"
- **`rotation`** (`Union[float, Sequence[float], None]`) – Rotate the force clockwise. Rotation is in degrees
- **`q_perp`** (`Union[float, Sequence[float], None]`) – value of any q-load perpendicular to the indication direction/rotation

Example

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



1.5.3 Remove loads

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

Remove all the applied loads from the structure.

Parameters

dead_load (bool) – Remove the dead load.

1.6 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.

Note that plotting capabilities do require that anaStruct be installed with the "plot" sub-module (e.g. *pip install anastruct[plot]*)

1.6.1 Structure

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

Plot the structure.

Parameters

- **factor** – Influence the plotting scale.
- **verbosity** (int) – 0: All information, 1: Suppress information.
- **scale** (float) – Scale of the plot.
- **offset** (Tuple[float, float]) – Offset the plots location on the figure.
- **figsize** (Optional[Tuple[float, float]]) – 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)
- **annotations** (bool) – if True, structure annotations are plotted. It includes section name. Note: only works when verbosity is equal to 0.

1.6.2 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** (Optional[float]) – Influence the plotting scale.
- **verbosity** (int) – 0: All information, 1: Suppress information.
- **scale** (float) – Scale of the plot.
- **offset** (Tuple[float, float]) – Offset the plots location on the figure.
- **figsize** (Optional[Tuple[float, float]]) – 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)

1.6.3 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** (Optional[float]) – Influence the plotting scale.
- **verbosity** (int) – 0: All information, 1: Suppress information.

- **scale** (float) – Scale of the plot.
- **offset** (Tuple[float, float]) – Offset the plots location on the figure.
- **figsize** (Optional[Tuple[float, float]]) – 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)

1.6.4 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.

Parameters

- **factor** (Optional[float]) – Influence the plotting scale.
- **verbosity** (int) – 0: All information, 1: Suppress information.
- **scale** (float) – Scale of the plot.
- **offset** (Tuple[float, float]) – Offset the plots location on the figure.
- **figsize** (Optional[Tuple[float, float]]) – 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)

1.6.5 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** (float) – Scale of the plot.
- **offset** (Tuple[float, float]) – Offset the plots location on the figure.
- **figsize** (Optional[Tuple[float, float]]) – Change the figure size.
- **show** (bool) – Plot the result or return a figure.

1.6.6 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.

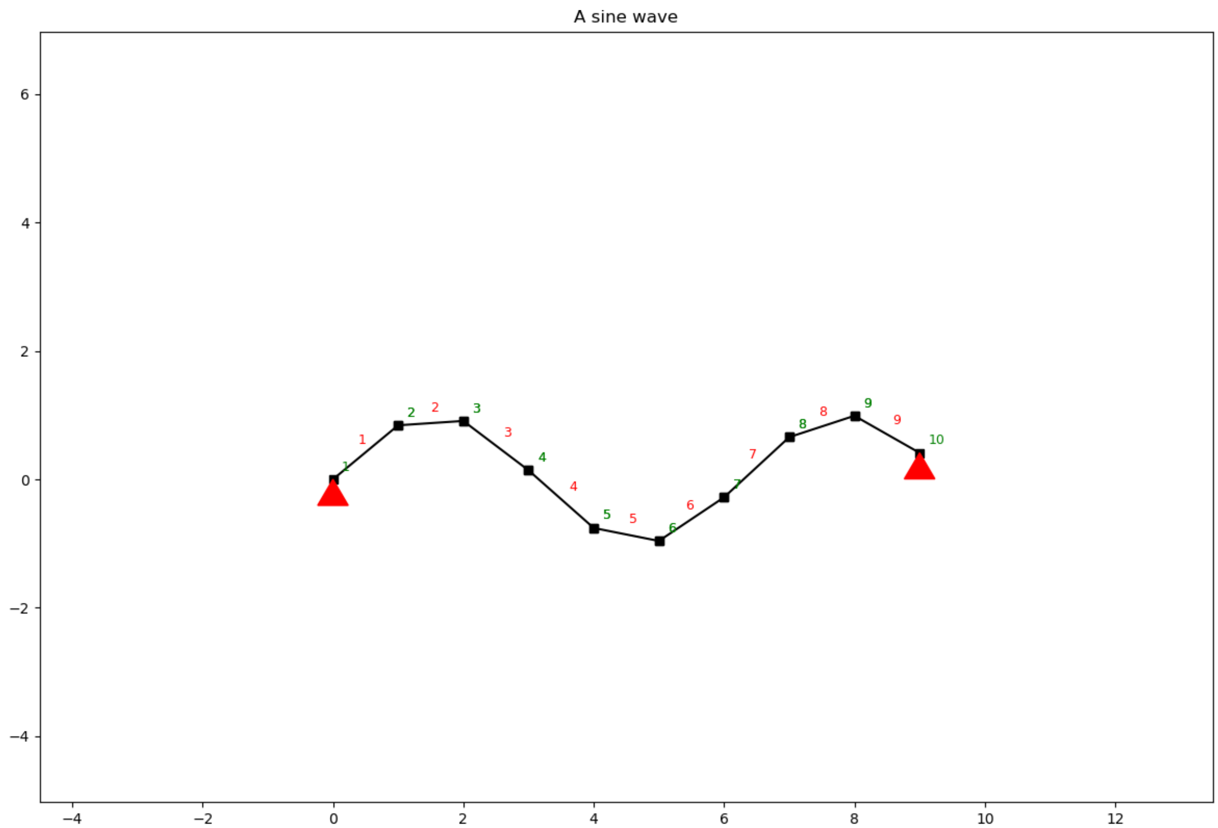
Parameters

- **factor** (Optional[float]) – Influence the plotting scale.
- **verbosity** (int) – 0: All information, 1: Suppress information.
- **scale** (float) – Scale of the plot.
- **offset** (Tuple[float, float]) – Offset the plots location on the figure.
- **figsize** (Optional[Tuple[float, float]]) – 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)

1.6.7 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')
```



1.7 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** (int) – Calculate second order effects and determine the buckling factor.

Returns
Displacements vector.

Development **kwargs:

param naked
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

1.7.1 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.

1.7.2 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))
```

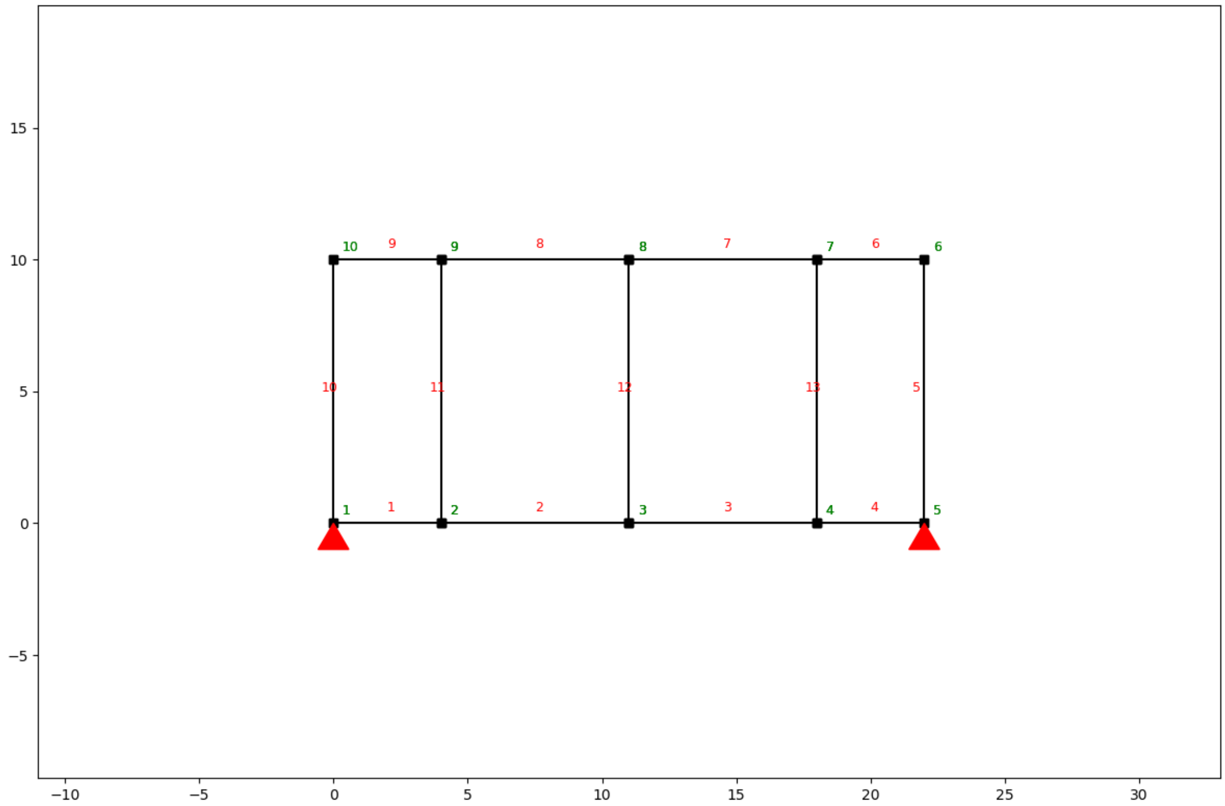
With this dictionary you can set the amount of discretization elements generated during the geometrical non linear calculation. This calculation is an approximation and gets more accurate with more discretization elements.

1.8 Load cases and load combinations

1.8.1 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.

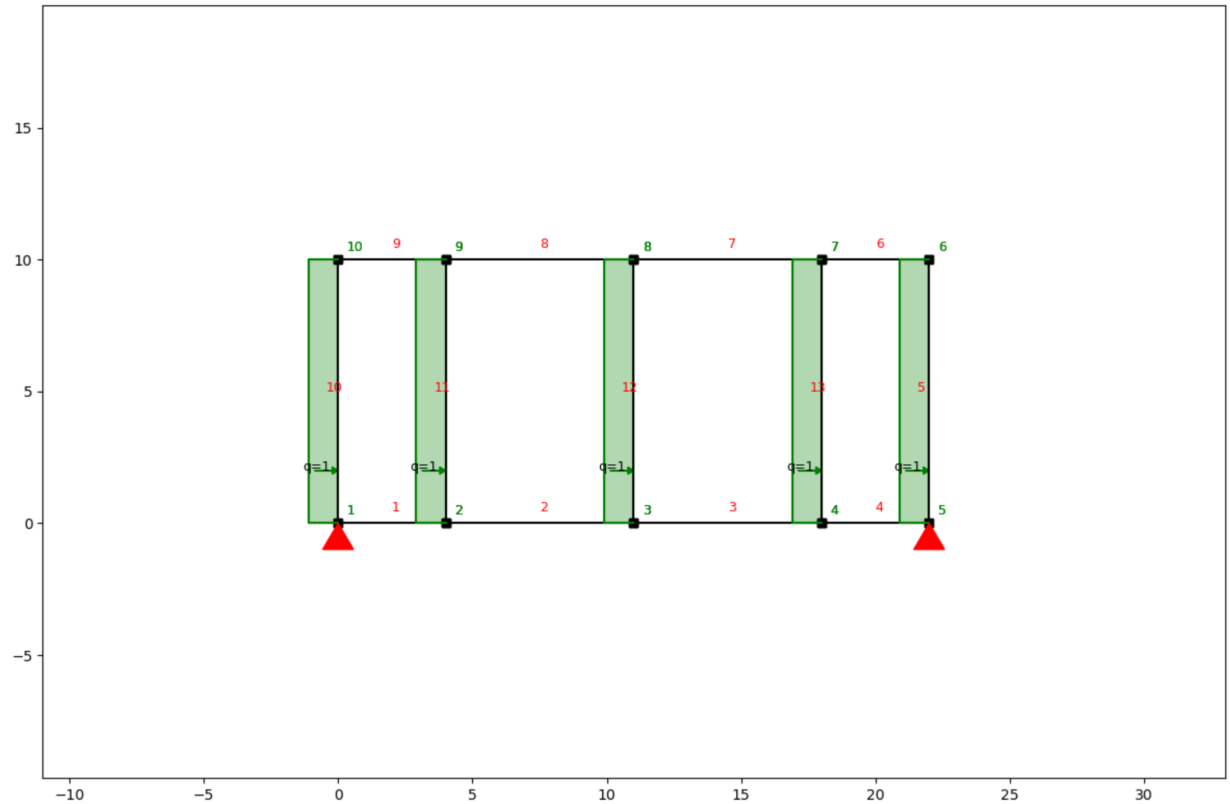
```
lc_wind = LoadCase('wind')
lc_wind.q_load(q=-1, element_id=[10, 11, 12, 13, 5])

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()
```



1.8.2 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')
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.

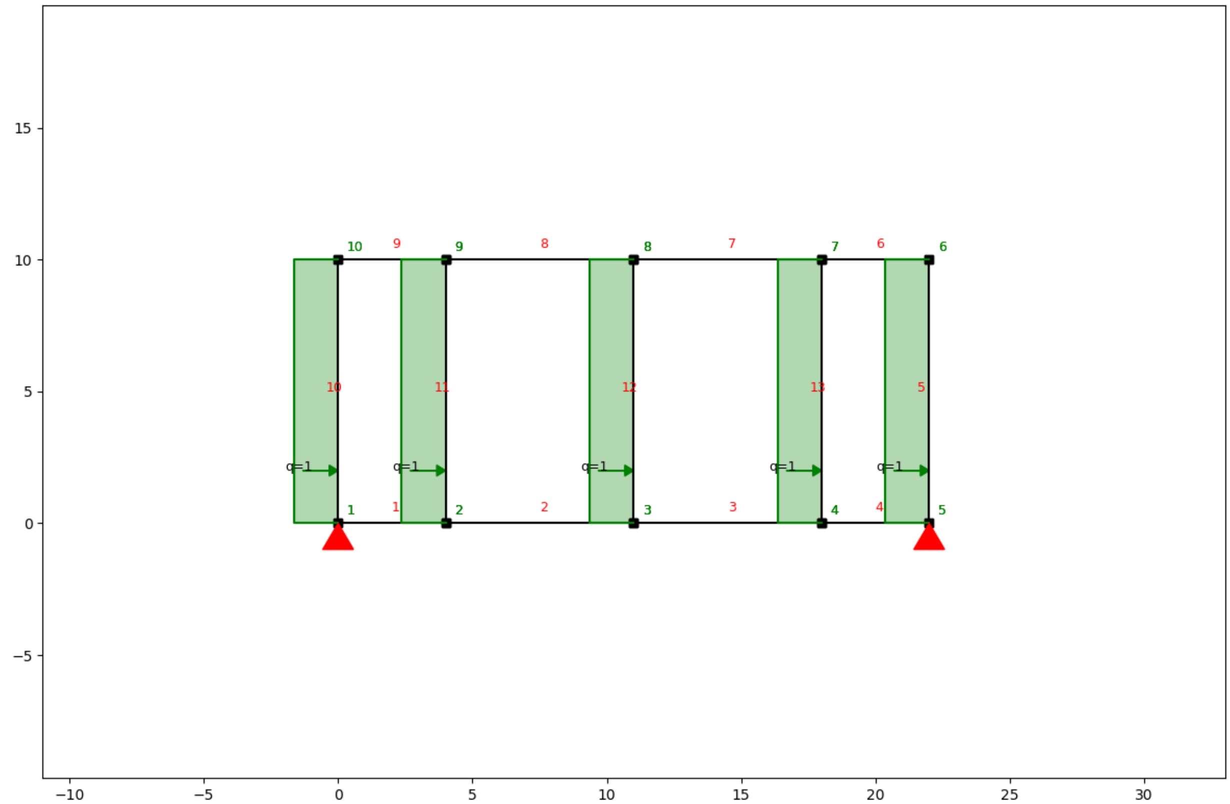
```
results = combination.solve(ss)

for k, ss in results.items():
```

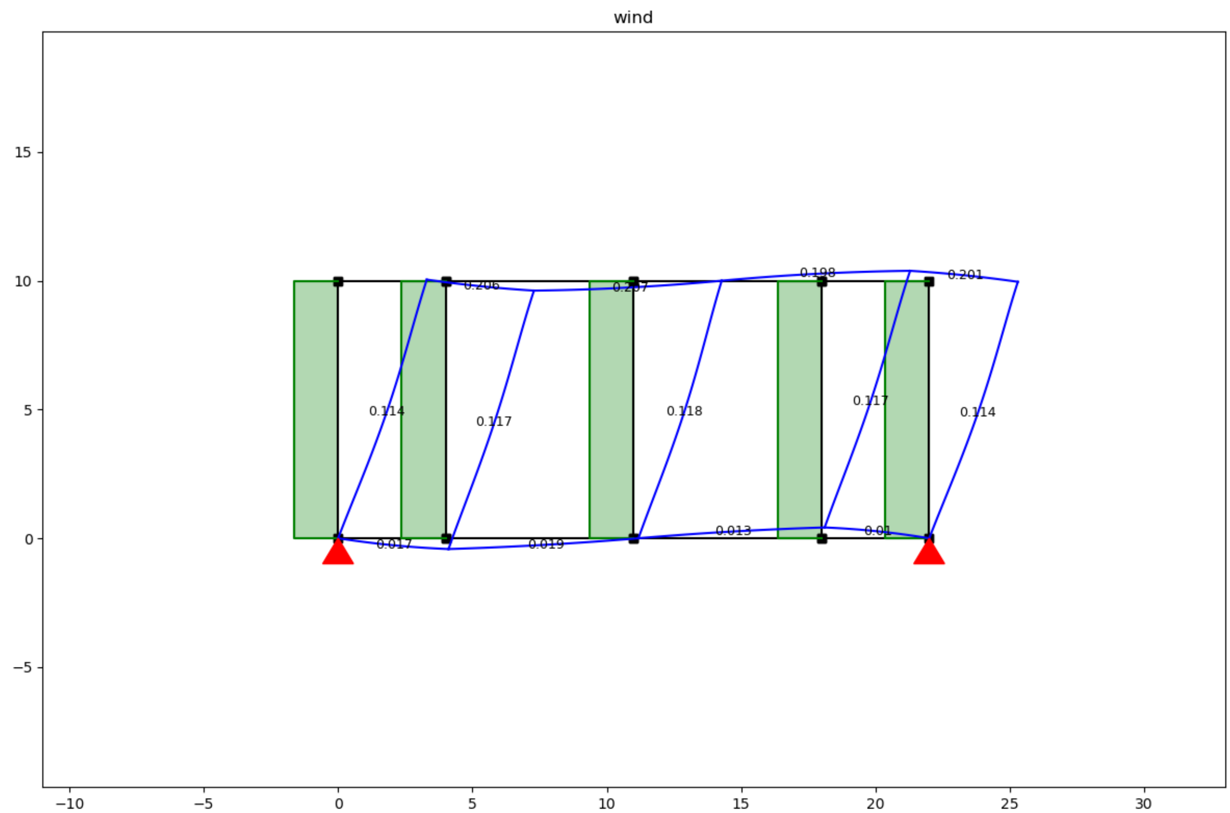
(continues on next page)

(continued from previous page)

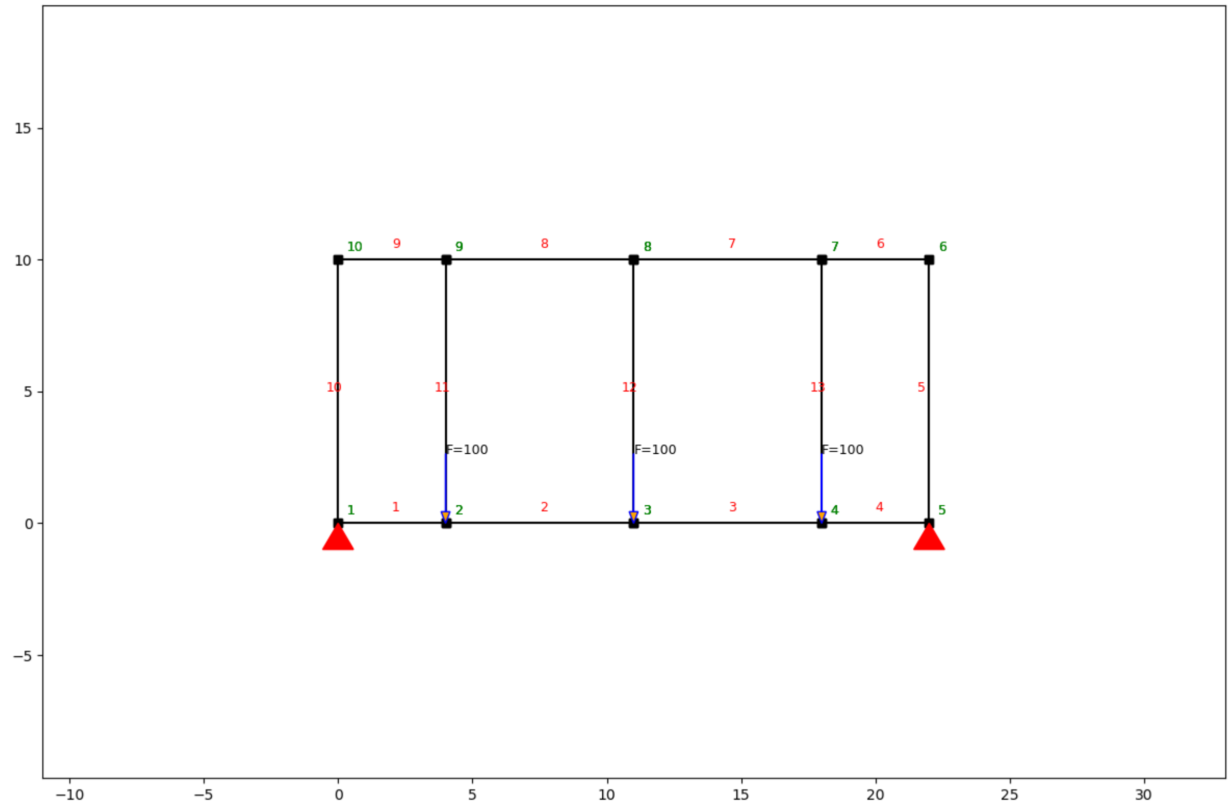
```
    results[k].show_structure()
    results[k].show_displacement(show=False)
    plt.title(k)
    plt.show()
```

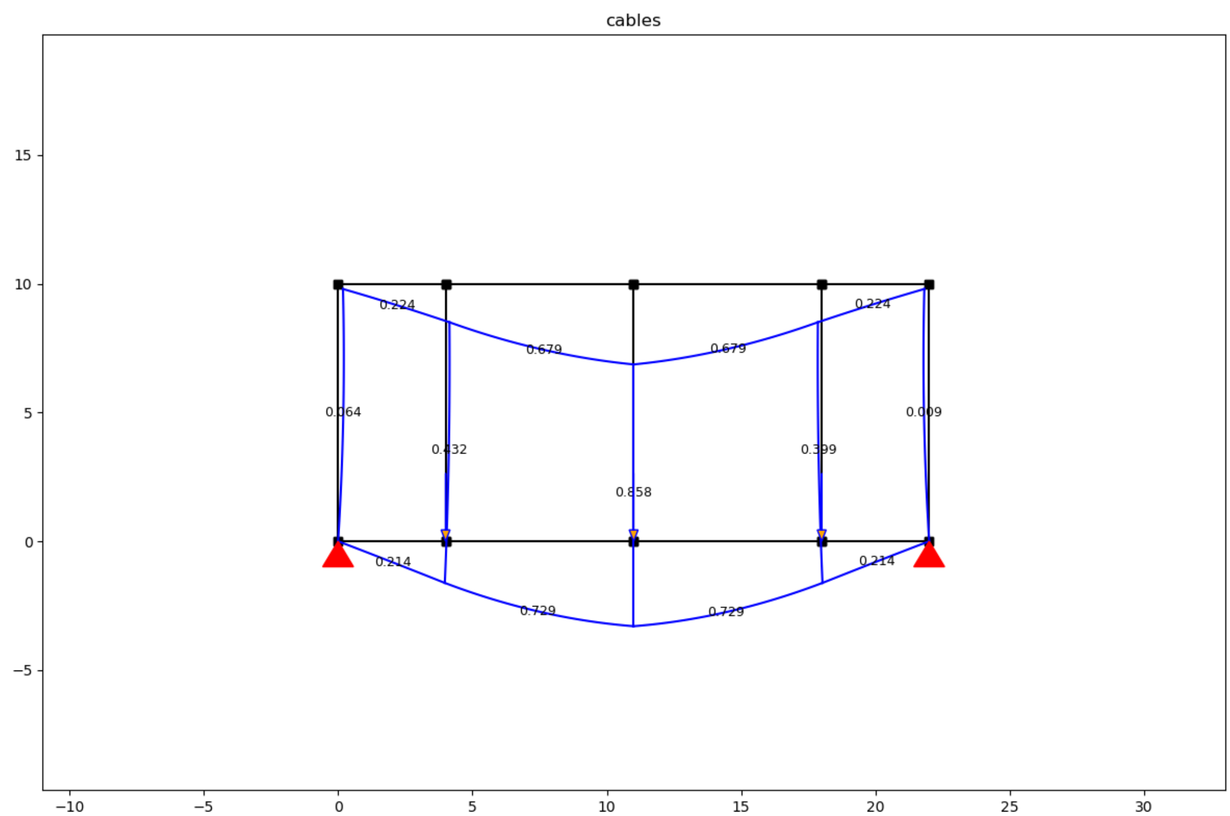
Load case wind



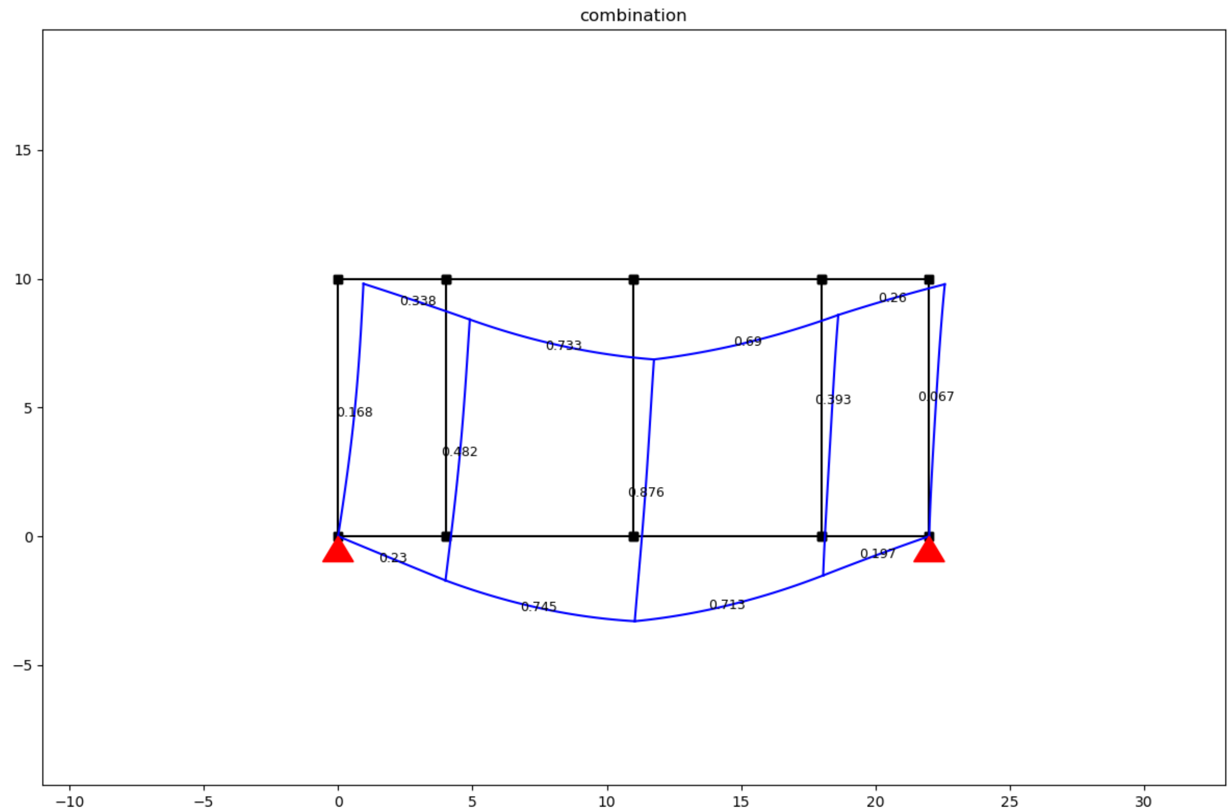


Load case cables


F=100
F=100
F=100


cables

Combination



1.8.3 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'*, *rotation=None*, *q_perp=None*)

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", "parallel"

1.8.4 Load combination class

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

__init__(*name*)

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

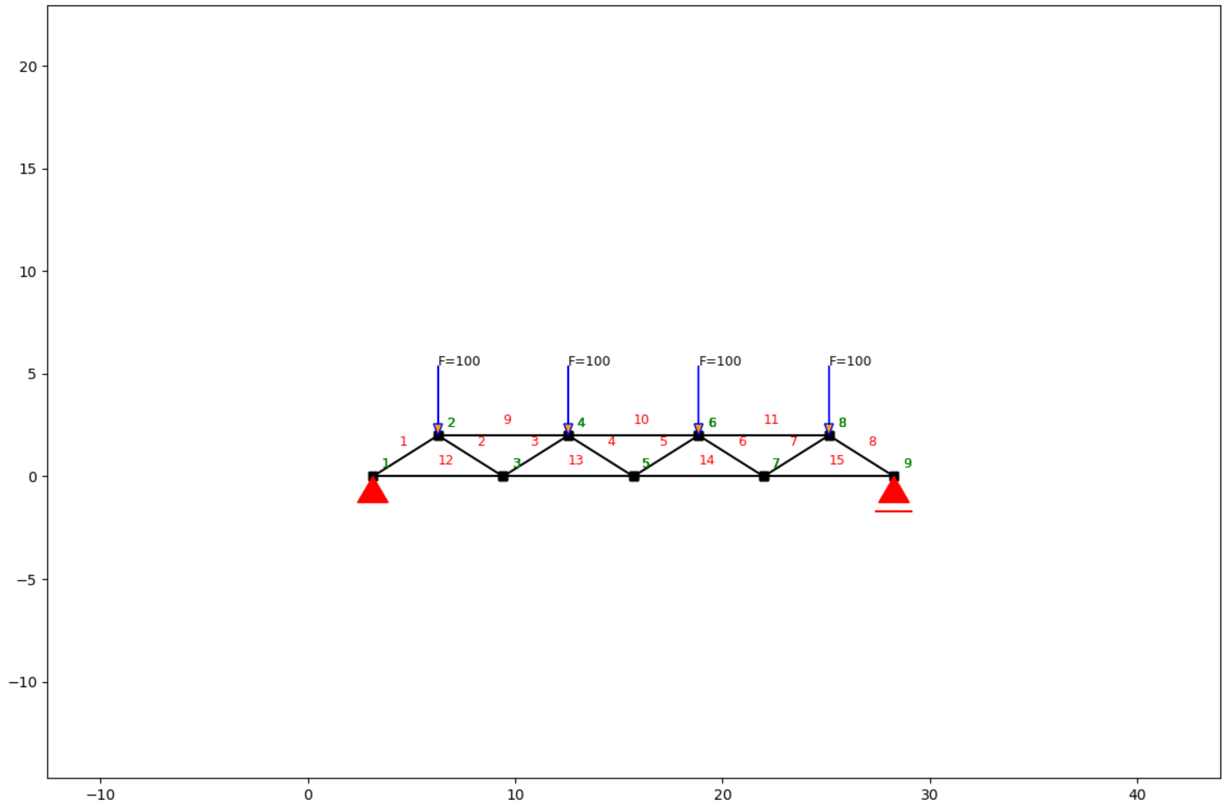
(bool) Whether or not to run the solve function without doing post processing.

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

1.9 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()
```



1.9.1 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 (int) – representing the node's ID. If integer = 0, the results of all nodes are returned

Return type
Union[List[Tuple[Any, Any, Any, Any, Any, Any, Any]], Dict[str, Union[int, float]]]

Returns

if node_id == 0:

Returns a list containing tuples with the results:

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

if node_id > 0:

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
```

1.9.2 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.

Return type
`Union[List[Tuple[Any, Any, Any, Any]], Dict[str, Any]]`

Returns

if node_id == 0:

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}
```

1.9.3 Range of node displacements

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

Query a list with node results.

Return type
`List[float]`

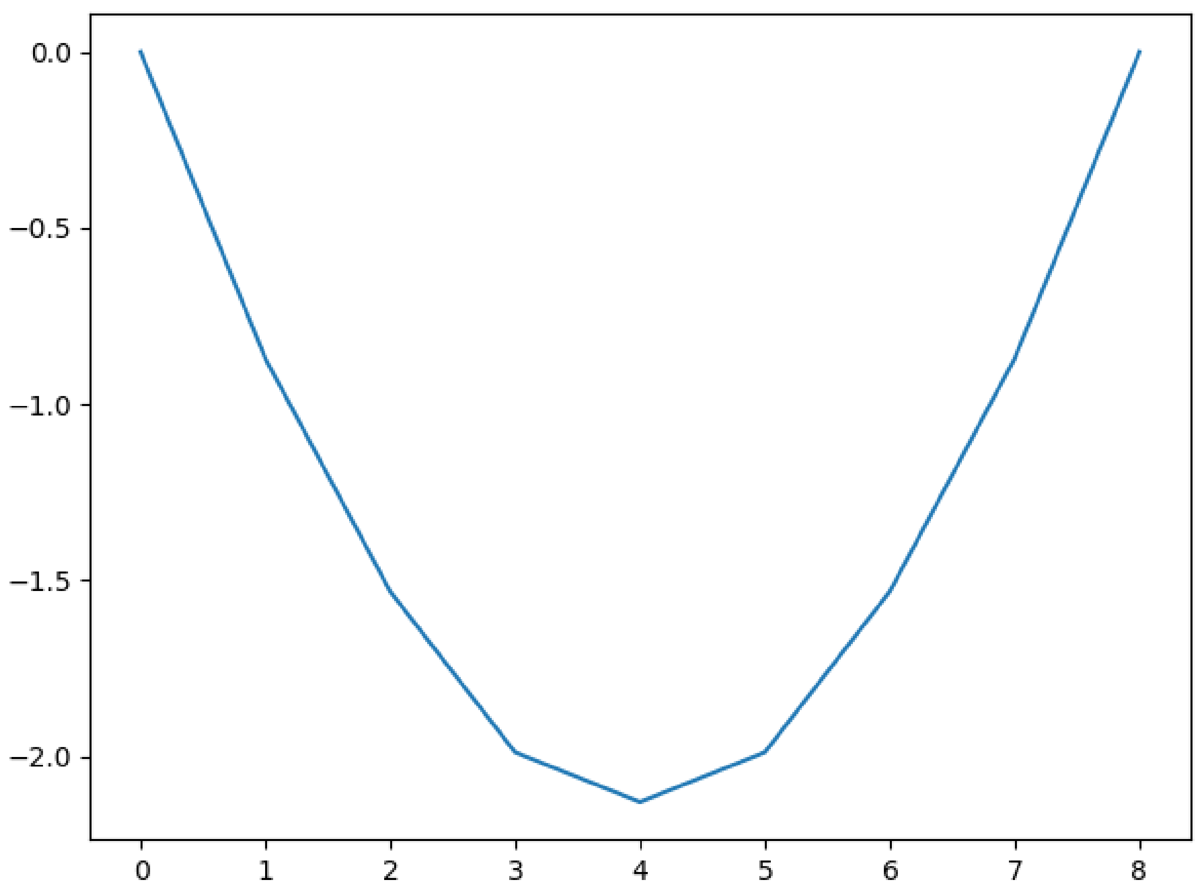
Example

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]
```



1.9.4 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.

Return type

Union[List[Dict[str, Any]], Dict[str, Any]]

Returns

if node_id == 0:

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
```

1.9.5 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'

Return type
List[float]

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
```

1.10 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.

1.10.1 Find node id based on coordinates

`SystemElements.find_node_id(vertex)`

Retrieve the ID of a certain location.

Parameters
vertex (Union[Vertex, Sequence[float]]) – Vertex_xz, [x, y], (x, y)

Return type
Optional[int]

Returns
id of the node at the location of the vertex

1.10.2 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`** (`Union[float, Sequence[float]]`) – Value of the dimension.

Return type

`Optional[int]`

Returns

ID of the node.

1.10.3 Query node coordinates

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

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

Parameters

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

Return type

`List[Union[float, Tuple[float, float], None]]`

1.11 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()
```

1.12 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:
    pickle.dump(ss, f)

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

1.13 Examples

Examples below a side variety of the structures which aim to show capabilities of the package. The same as any other packages, anaStruct should be called and imported.

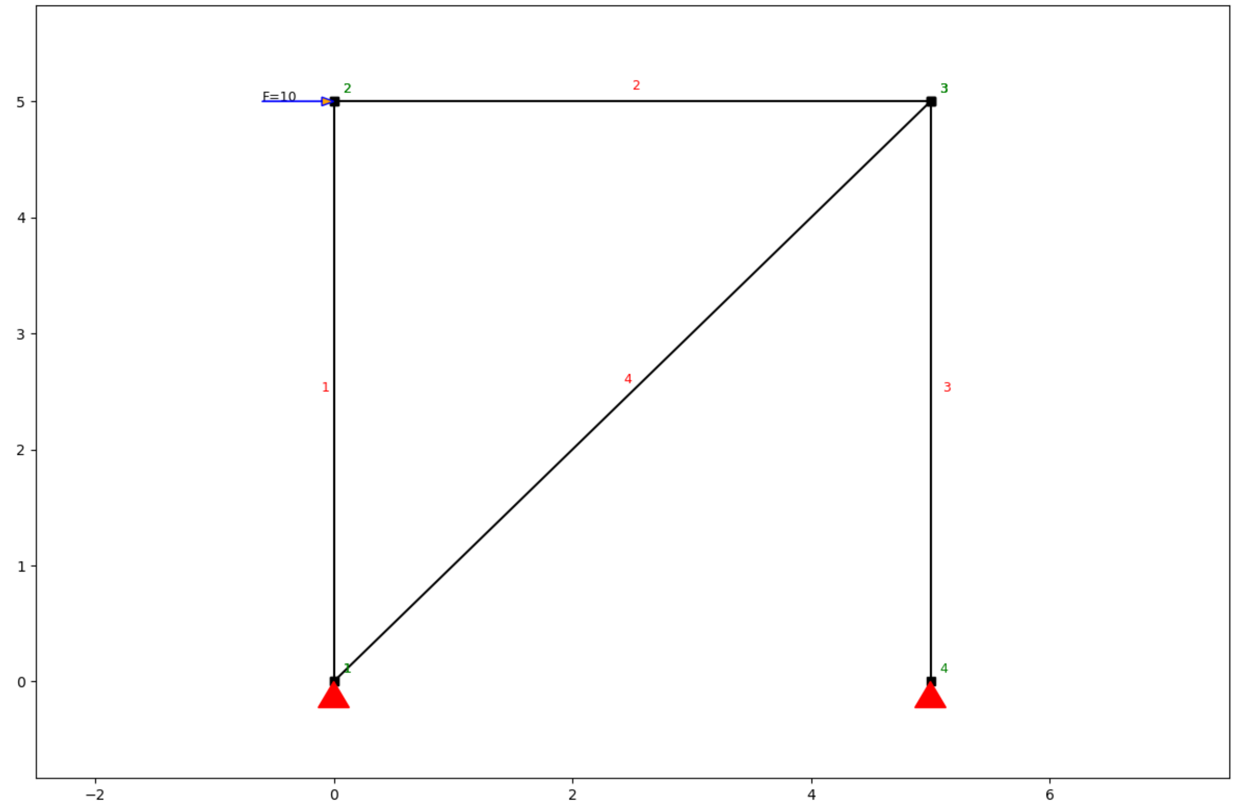
```
import anastruct as anas
```

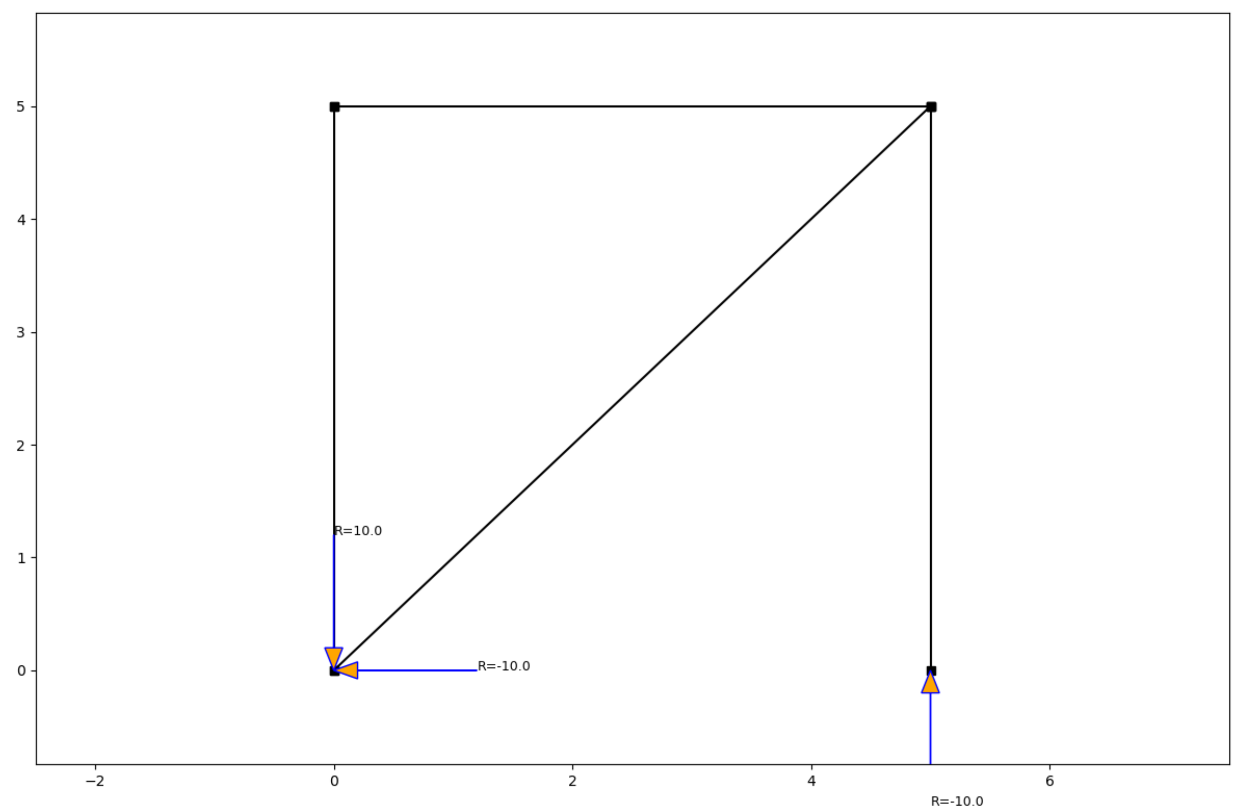
And for a mater of minimalism and making calls and coding more efficient, different classes can be called separately.

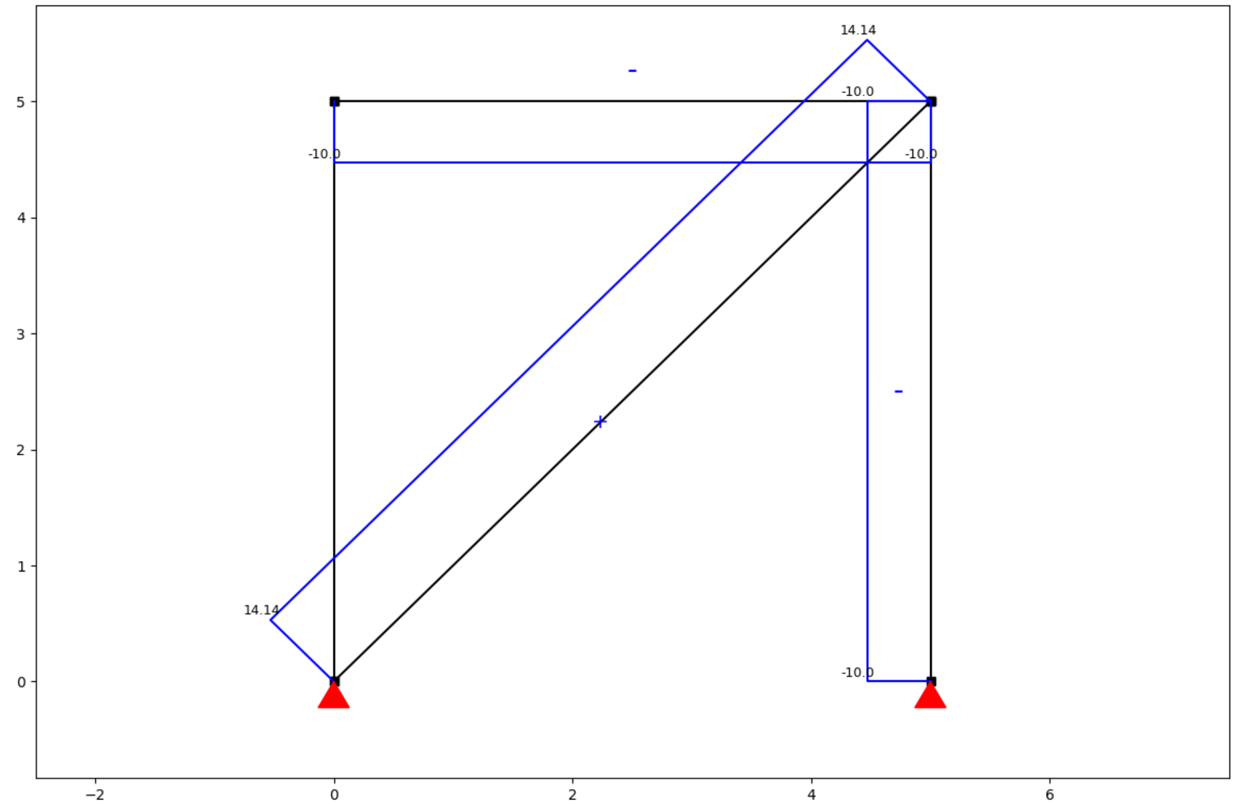
```
anas.LoadCase
anas.LoadCombination
anas.SystemElements
anas.Vertex
```

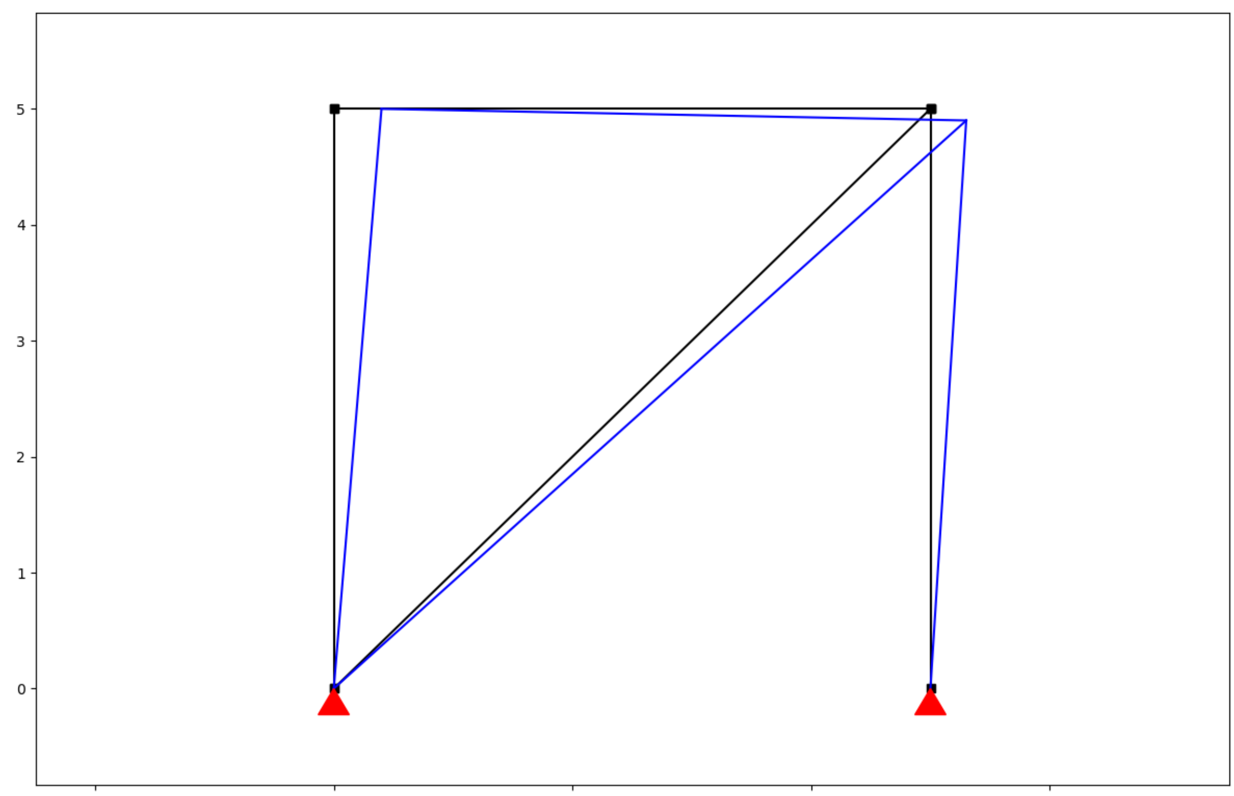
1.13.1 Simple example - Truss

```
ss = SystemElements(EA=5000)
ss.add_truss_element(location=[[0, 0], [0, 5]])
ss.add_truss_element(location=[[0, 5], [5, 5]])
ss.add_truss_element(location=[[5, 5], [5, 0]])
ss.add_truss_element(location=[[0, 0], [5, 5]], EA=5000 * math.sqrt(2))

ss.add_support_hinged(node_id=1)
ss.add_support_hinged(node_id=4)

ss.point_load(Fx=10, node_id=2)

ss.solve()
ss.show_structure()
ss.show_reaction_force()
ss.show_axial_force()
ss.show_displacement(factor=10)
```


F=10
2
2
3
1
4
3
1
4
5
4
3
2
1
0
−2
0
2
4
6


R=10.0
R=-10.0
R=-10.0
5
4
3
2
1
0
−2
0
2
4
6


14.14
-10.0
-10.0
-10.0
14.14
-10.0
-
-
+
5
4
3
2
1
0
−2
0
2
4
6


5
4
3
2
1
0
−2
0
2
4
6

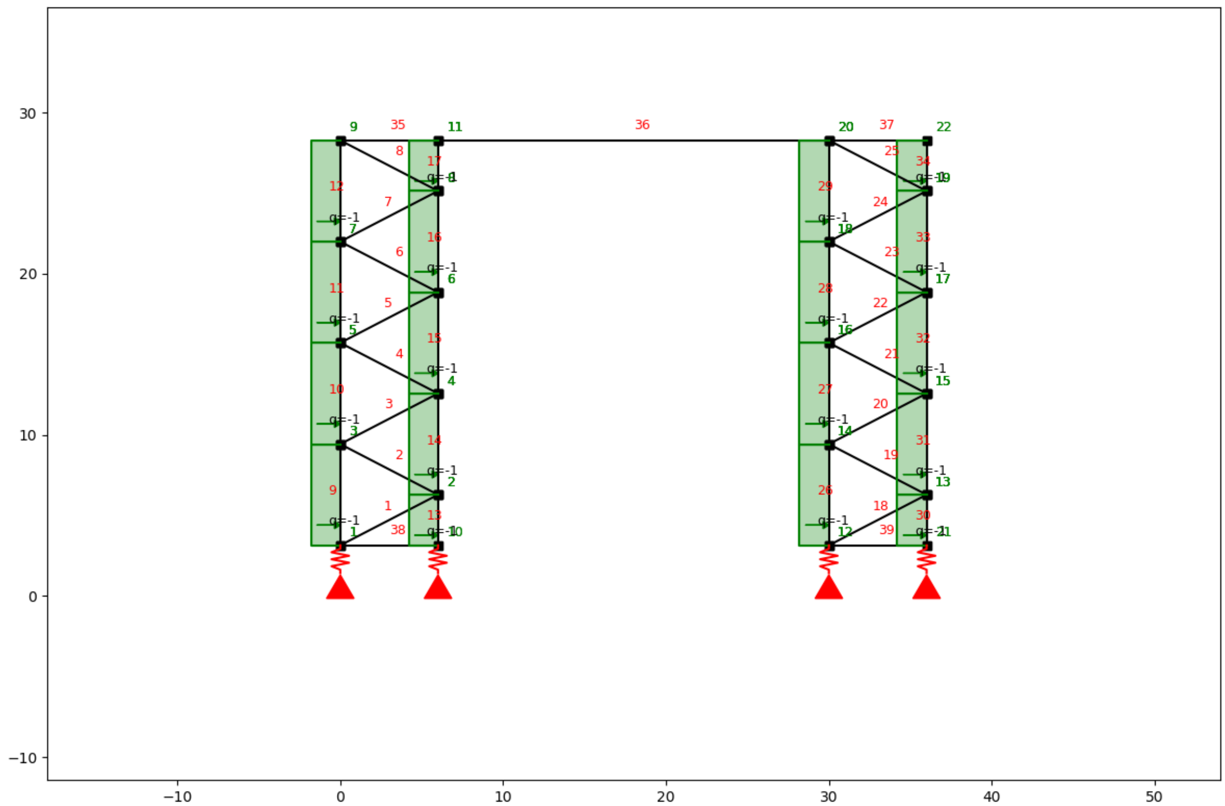
1.14 Intermediate

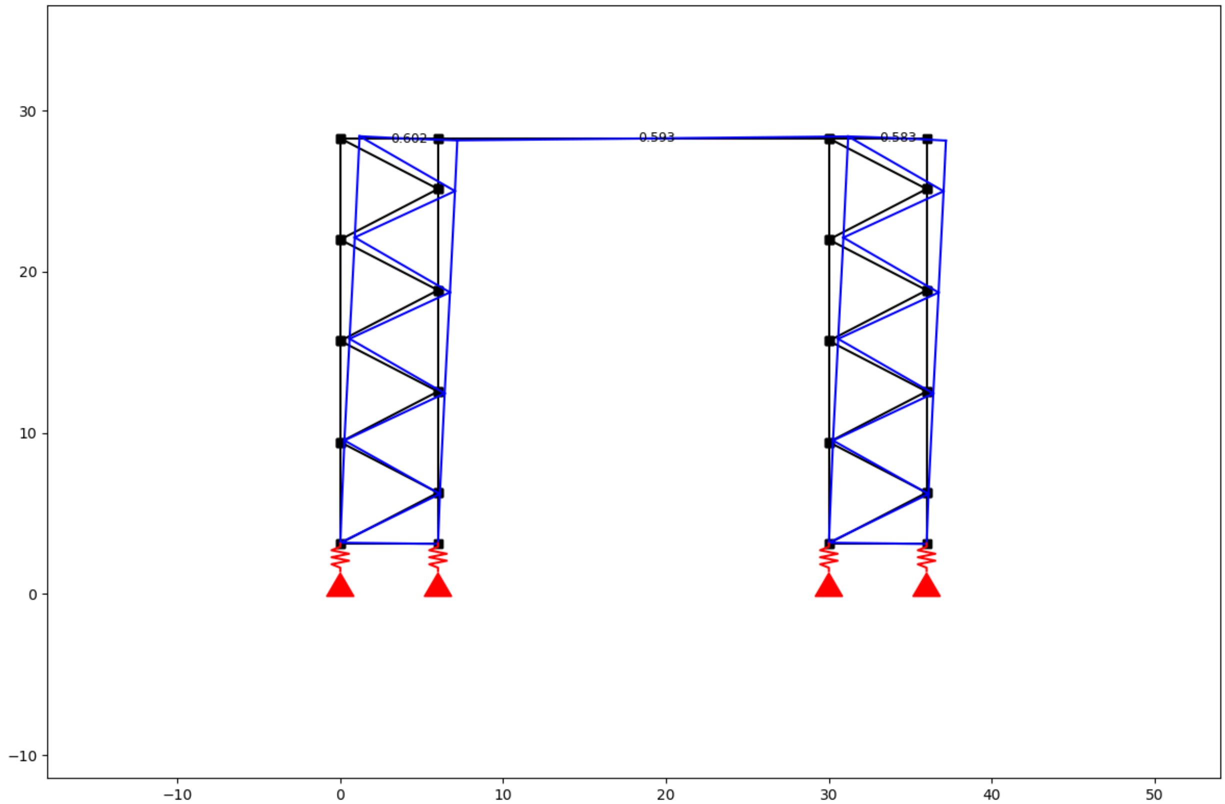
```python
from anastruct import SystemElements
import numpy as np

ss = SystemElements()
element_type = 'truss'

# Create 2 towers
width = 6
span = 30
k = 5e3

# create triangles
y = np.arange(1, 10) * np.pi
x = np.cos(y) * width * 0.5
x -= x.min()

for length in [0, span]:
    x_left_column = np.ones(y[::2].shape) * x.min() + length
    x_right_column = np.ones(y[::2].shape[0] + 1) * x.max() + length

    # add triangles
    ss.add_element_grid(x + length, y, element_type=element_type)
    # add vertical elements
    ss.add_element_grid(x_left_column, y[::2], element_type=element_type)
    ss.add_element_grid(x_right_column, np.r_[y[0], y[1::2], y[-1]], element_
↪type=element_type)

    ss.add_support_spring(
        node_id=ss.find_node_id(vertex=[x_left_column[0], y[0]]),
        translation=2,
        k=k)
    ss.add_support_spring(
        node_id=ss.find_node_id(vertex=[x_right_column[0], y[0]]),
        translation=2,
        k=k)

# add top girder
ss.add_element_grid([0, width, span, span + width], np.ones(4) * y.max(), EI=10e3)

# Add stability elements at the bottom.
ss.add_truss_element([[0, y.min()], [width, y.min()]])
ss.add_truss_element([[span, y.min()], [span + width, y.min()]])

for el in ss.element_map.values():
    # apply wind load on elements that are vertical
    if np.isclose(np.sin(el.angle), 1):
        ss.q_load(
            q=1,
            element_id=el.id,
            direction='x'
```

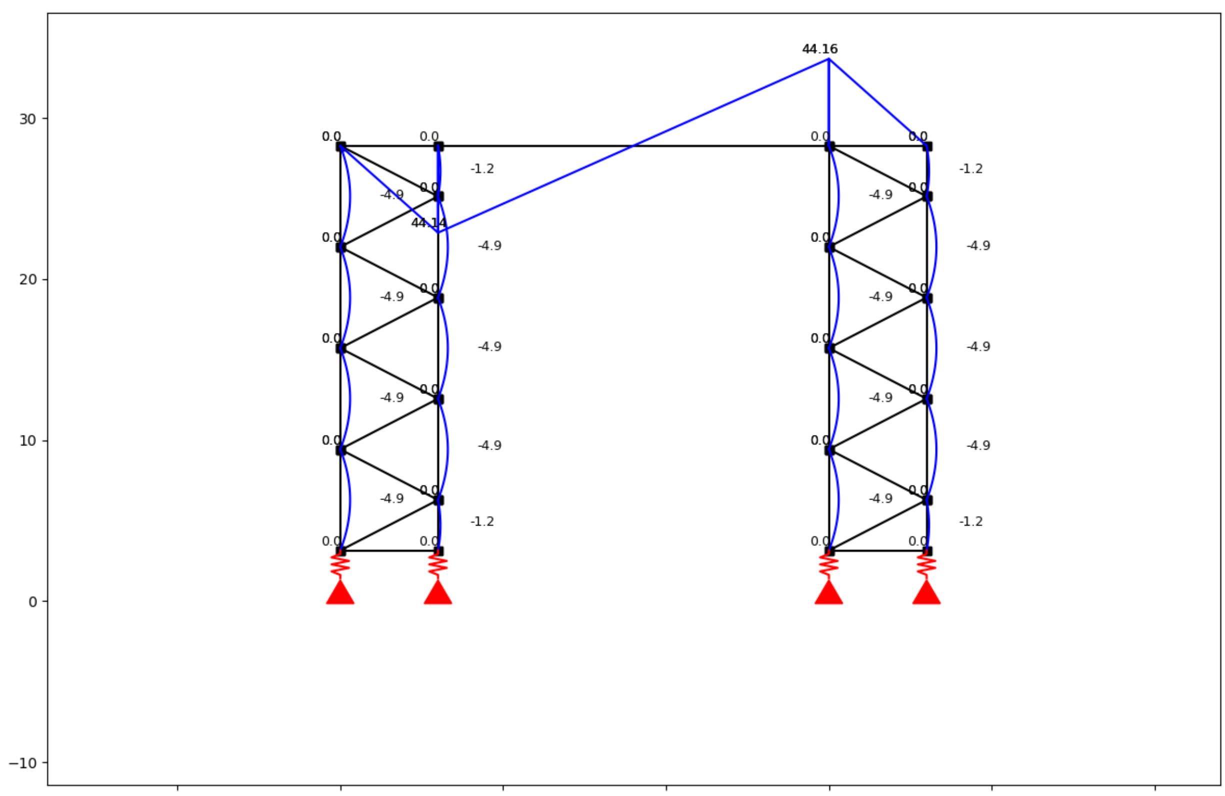
(continues on next page)

(continued from previous page)

```
        )

ss.show_structure()
ss.solve()
ss.show_displacement(factor=2)
ss.show_bending_moment()
```




0.602
0.593
0.583


44.16
44.14
-1.2
-4.9

1.15 Advanced

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

```python
# import dependencies
import matplotlib.pyplot as plt
from anastruct.basic import converge
from anastruct.material.profile import HEA, IPE
from anastruct.fem.system import SystemElements, Vertex
from anastruct.material.units import to_kNm2, to_kN

# constants
E = 2.1e5  # Construction steels Young's modulus
b = 5  # c.t.c distance portals
q_water = 10

# axes height levels
h_1 = 0
h_2 = 0.258
h_3 = 0.046
h_4 = 0.274
h_5 = 0.032
h_6 = 0.15

# beam spans
span_1 = span_2 = 21.9
span_3 = 8.9

# Vertices at the axes
p1 = Vertex(0, h_1)
p2 = Vertex(span_1 * 0.5, h_2)
p3 = Vertex(span_1, h_3)
p4 = Vertex(span_1 + span_2 * 0.5, h_4)
p5 = Vertex(span_1 + span_2, h_5)
p6 = Vertex(span_1 + span_2 + span_3, h_6)

def structure():
    """
    Build the structure from left to right, starting at axis 1.

    variables:
    EA = Young's modulus * Area
    EI = Young's modulus * moment of Inertia
    g = Weight [kN/ m]
    elements = reference of the element id's that were created
    dl = c.t.c distance different nodes.
    """

    dl = 0.2


    ## SPAN 1 AND 2
```

(continues on next page)

(continued from previous page)

```python

    # The elements between axis 1 and 3 are an IPE 450 member.
    EA = to_kN(E * IPE[450]['A'])  # Y
    EI = to_kNm2(E * IPE[450]["Iy"])
    g = IPE[450]['G'] / 100

    # New system.
    ss = SystemElements(mesh=3, plot_backend="mpl")

    # span 1
    first = dict(
        spring={1: 9e3},
        mp={1: 70},
    )

    elements = ss.add_multiple_elements(location=[p1, p2], dl=dl, first=first, EA=EA, EI=EI, g=g)
    elements += ss.add_multiple_elements(location=p3, dl=dl, EA=EA, EI=EI, g=g)

    # span 2
    first = dict(
        spring={1: 40e3},
        mp={1: 240}
    )
    elements += ss.add_multiple_elements(location=p4, dl=dl, first=first, EA=EA, EI=EI, g=g)
    elements += ss.add_multiple_elements(location=p5, dl=dl, EA=EA, EI=EI, g=g)


    ## SPAN 3

    # span 3
    # different IPE
    g = IPE[240]['G'] / 100
    EA = to_kN(E * IPE[240]['A'])
    EI = to_kNm2(E * IPE[240]["Iy"])
    first = dict(
        spring={1: 15e3},
        mp={1: 25},
    )

    elements += ss.add_multiple_elements(location=p6, first=first, dl=dl, EA=EA, EI=EI, g=g)

    # Add a dead load of -2 kN/m to all elements.
    ss.q_load(-2, elements, direction="y")


    ## COLUMNS

    # column height
    h = 7.2
```

(continues on next page)

(continued from previous page)

```

    # left column
    EA = to_kN(E * IPE[220]['A'])
    EI = to_kNm2(E * HEA[220]["Iy"])
    left = ss.add_element([[0, 0], [0, -h]], EA=EA, EI=EI)

    # right column
    EA = to_kN(E * IPE[180]['A'])
    EI = to_kNm2(E * HEA[180]["Iy"])
    right = ss.add_element([p6, Vertex(p6.x, -h)], EA=EA, EI=EI)


    ## SUPPORTS

    # node ids for the support
    id_left = max(ss.element_map[left].node_map.keys())
    id_top_right = min(ss.element_map[right].node_map.keys())
    id_btm_right = max(ss.element_map[right].node_map.keys())

    # Add supports. The location of the supports is defined with the nodes id.
    ss.add_support_hinged((id_left, id_btm_right))

    # Retrieve the node ids at axis 2 and 3
    id_p3 = ss.find_node_id(p3)
    id_p5 = ss.find_node_id(p5)

    ss.add_support_roll(id_top_right, direction=1)

    # Add translational spring supports at axes 2 and 3
    ss.add_support_spring(id_p3, translation=2, k=2e3, roll=True)
    ss.add_support_spring(id_p5, translation=2, k=3e3, roll=True)
    return ss

ss = structure()
ss.show_structure(verbosity=1, scale=0.6)

def water_load(ss, water_height, deflection=None):
    """
    :param ss: (SystemElements) object.
    :param water_height: (flt) Water level.
    :param deflection: (array) Computed deflection.
    :return (flt) The cubic meters of water on the structure
    """

    # The horizontal distance between the nodes.
    dl = np.diff(ss.nodes_range('x'))

    if deflection is None:
        deflection = np.zeros(len(ss.node_map))

    # Height of the nodes
    y = np.array(ss.nodes_range('y'))
```

(continues on next page)

(continued from previous page)

```python

    # An array with point loads.
    # cubic meters * weight water
    force_water = (water_height - y[:-3] - deflection[:-3]) * q_water * b * dl[:-2]

    cubics = 0
    n = force_water.shape[0]
    for k in ss.node_map:
        if k > n:
            break
        point_load = force_water[k - 1]

        if point_load > 0:
            ss.point_load(k, Fx=0, Fz=-point_load)
            cubics += point_load / q_water

    return cubics

def det_water_height(c, deflection=None):
    """
    :param c: (flt) Cubic meters.
    :param deflection: (array) Node deflection values.
    :return (SystemElement, flt) The structure and the redistributed water level is returned.
    """
    wh = 0.1

    while True:
        ss = structure()
        cubics = water_load(ss, wh, deflection)

        factor = converge(cubics, c)
        if 0.9999 <= factor <= 1.0001:
            return ss, wh

        wh *= factor

cubics = [0]
water_heights = [0]

a = 0
deflection = None
max_water_level = 0

# Iterate from 8 m3 to 15 m3 of water.

for cubic in range(80, 150, 5):  # This loop computes the results per m3 of storaged water.
    wh = 0.05
    lastwh = 0.2
    cubic /= 10

```

(continues on next page)

(continued from previous page)

```
    print(f"Starting analysis of {cubic} m3")

    c = 1
    for _ in range(100):  # This loop redistributes the water until the water level converges.

        # redistribute the water
        ss, wh = det_water_height(cubic, deflection)

        # Do a non linear calculation!!
        ss.solve(max_iter=100, verbosity=1)
        deflection = ss.get_node_result_range("uy")

        # Some breaking conditions
        if min(deflection) < -1:
            print(min(deflection), "Breaking due to exceeding max deflection")
            break
        if 0.9999 < lastwh / wh < 1.001:
            print(f"Convergence in {c} iterations.")
            cubics.append(cubic)
            water_heights.append(wh)
            break

        lastwh = wh
        c += 1

    if wh > max_water_level:
        max_water_level = wh
    else:
        a += 1
        if a >= 2:
            print("Breaking. Water level isn't rising.")
            break

plt.plot(ss.nodes_range('x')[:-2], [el.bending_moment[0] for el in list(ss.element_map.values())[:-1]])
a = 0
plt.plot([0, p6.x], [a, a], color="black")

c = "red"
a = 240
plt.plot([p3.x - 5, p3.x + 5], [a, a], color=c)
a = 25
plt.plot([p5.x - 5, p5.x + 5], [a, a], color=c)
a = 70
plt.plot([p1.x - 5, p1.x + 5], [a, a], color=c)

plt.ylabel("Bending moment [kNm]")
plt.xlabel("Span [m]")
plt.show()

plt.plot(ss.nodes_range('x')[:-2], ss.nodes_range('y')[:-2])
```

(continues on next page)

(continued from previous page)

```
plt.plot(ss.nodes_range('x')[:-2], [a + b for a, b in zip(ss.nodes_range('y')[:-2], ss.
↪get_node_result_range("uy")[:-2])])

plt.ylabel("Height level roof when accumulating [m]")
plt.xlabel("Span [m]")
plt.show()
```

INDEX

P

Q

R

S