TensegritySim Documentation

Release 1.0.0

Austin Brown

Mar 14, 2025

CONTENTS:

CHAPTER

ONE

TENSEGRITY SIM

This repo contains code to create a simulation for tensegrity structures. It allows for continuous cables and 2D, 2.5D and 3D structures.

1.1 Getting started

To get started with development, clone the repo and install the dependencies.

I reccommend using a venv to keep the libraries for this project separate from the main python interpreter. To create a venv, from the project's main directory run `python3 -m venv ./venv`. Now everytime you want to use this venv run `source venv/bin/activate`. To deactivate simply use the `deactivate` command.

This project uses Python3, in order to run it you will need some dependencies. To get them you can run `pip install -r requirements.txt` (with the venv active)

To run the project:

```
python3 main.py <path/to/yaml/config>
```

Sample yaml config files are provided in the `yaml` directory. To understand how to change the simulation to your needs, see the *simulation setup* documentation.

1.2 Installation

If instead of helping develop the project you want to use it as a library, you can install it using pip. To install the project, to use as a library, run `pip install .` from the project's main directory. This will install the `TensegritySim` module and allow you to import it in your own projects.

If you want to install from a different directory, you can run `pip install <path/to/project>` or `pip install -e <path/to/project>` to install in editable mode. Or without cloning the repo, you can run `pip install git+<git-repo-url>`.

1.3 Definitions and Conventions (as used in this project)

Strings - Strings are connection types that only carry tension, they lengthen as force is applied
Bar - A bar can carry either tension or compression, but does not change length
Forces - Tensions are positive values and compression forces in connections are negative.

1.4 Organization

`main.py` is the primary file to run the project. It take as an input a yaml file an allows the user to change the length of control strings

1.4.1 TensegritySim Module

The `TensegritySim` directory contains all the code TensegritySim python module.

- `data_structures.py` contains the `Node`, `Connection`, `Control`, and `Tensegrity` classes
- `yaml_parser.py` reads the yaml file and returns the Tensegrity object. See *YAML Reference* for how to format the yaml file
- `visualization.py` shows the tensegrity structure using matplotlib
- `tensegrity_solver.py` uses an optimizer to solve for an updated structure

1.4.2 yaml

The `yaml` directory contains sample yaml files for running the sim

CHAPTER

TWO

SIMULATION SETUP

The `main()` function in `main.py` can be changed to run the simulation as desired.

The primary parts to the simulations are:

- The YamlParser
- The Visualization
- The TensegritySolver

2.1 Parser

This should be the first step to running the simulation. The takes the input yaml file from the commandline and creates the Tensegrity object.

```
from TensegritySim import YamlParser

# Load the tensegrity system from the YAML file
tensegrity_system = YamlParser.parse(file)
```

2.2 Visualizer

The `Visualization` class takes in a `Tensegrity` object and can then be used to plot the tensegrity system using the `plot()` method.

```
from TensegritySim import Visualization as Viz

# Create the visualization object
viz = Viz(tensegrity_system)

viz.plot(label_nodes=True, label_connections=True)
```

2.3 TensegritySolver

The `TensegritySolver` class takes in a `Tensegrity` object and can then be used to solve the system using the `solve()` method.

```
from TensegritySim import TensegritySolver

# Create the TensegritySolver object
```

(continues on next page)

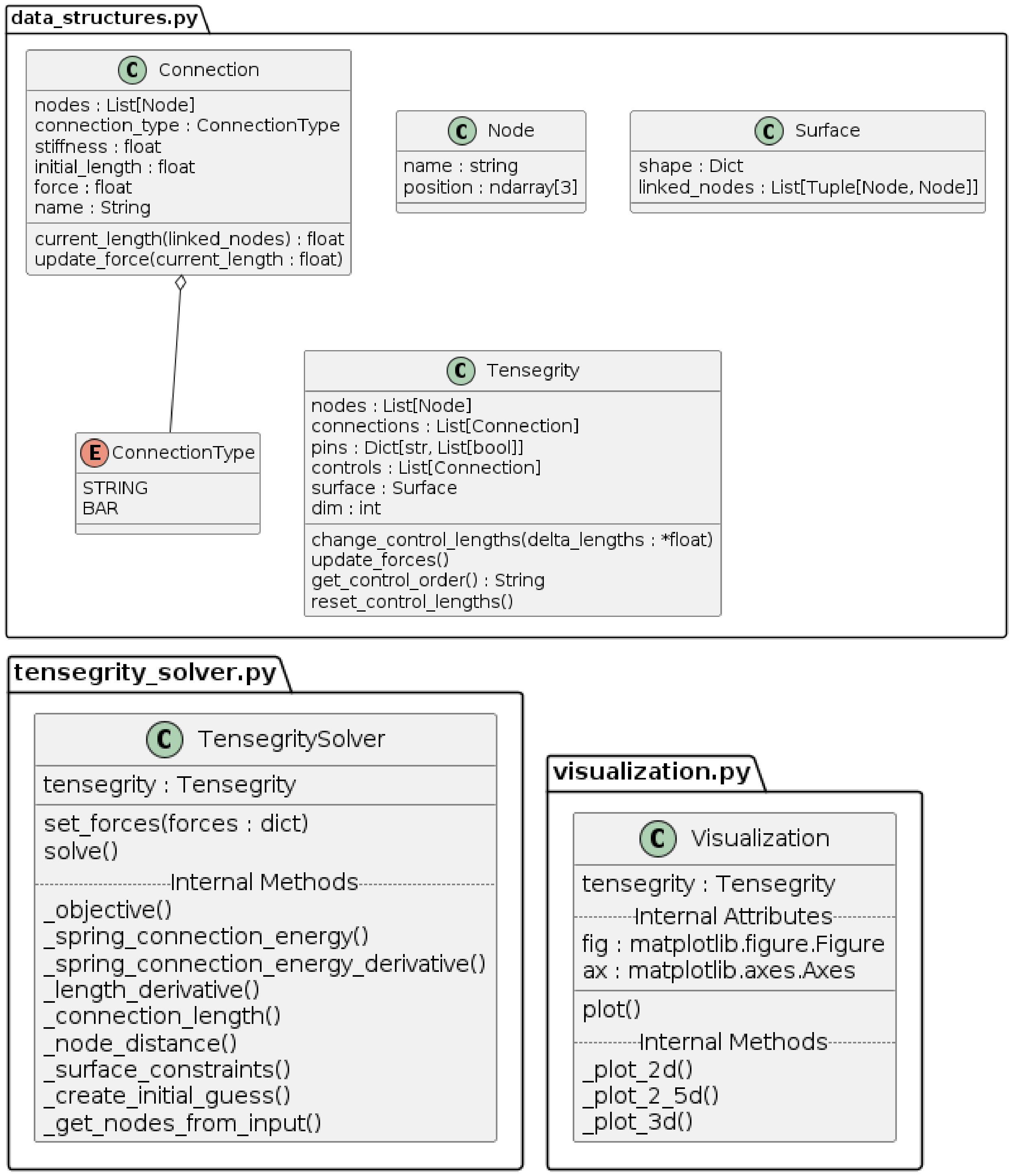
(continued from previous page)

```
solver = TensegritySolver(tensegrity_system)

# Manipulate the tensegrity system
tensegrity_system.change_control_lengths(-0.5)

solver.solve()
```

2.4 UML Class Diagrams



CHAPTER

THREE

YAML REFERENCE

The config file is a YAML file defining:

- *Nodes*
- *Connections*
- *Builders*
- *Pins*
- *Control*
- *Surface*
  - *Type*
  - *Linked Nodes*

These sections can be defined in any order in the YAML file, but it is easiest to logically go through them in the order defined above.

There are sample config files in the `yaml` directory.

3.1 Nodes

Nodes are the points that bars and strings connect at.

Nodes have a name and initial x, y, z positions.
A node named `Node1` with x = 1, y = 2, and z = 0, it would look like:

```yaml
nodes:
    Node1: [1, 2, 0]
```

The Tensegrity class sets the number of dimensions to 2 or 3 based on the number of coordinates given for the nodes. If all nodes have 3 coordinates, the structure is solved in 3D space. If any nodes have only 2 coordinates, the structure is solved in 2D space.

3.2 Connections

Connections are how the nodes are connected to each other. There can be unlimited connection types, with each connection type having different properties as defined in the *Builders* section.

A connection type named `strings` with a connection between `Node1` and `Node2` looks like:

```yaml
connections:
    strings:
        - [Node1, Node2]
```

Connections can also optionally be named (for later specifying connections to control).

```yaml
connections:
    strings:
        - string1: [Node1, Node2] # Named connection
        - [Node2, Node3] # Unnamed connection
```

String connections (as defined in the builders section) can pass through multiple nodes. They are assumed to frictionlessly pass through nodes and therefore always have the same tension along it's entire length.

```yaml
connections:
    strings:
        - string1: [Node1, Node2, Node5, Node6]
```

3.3 Builders

Builders are the connection properties that define the strings or bars that hold the nodes together. A builder must have a name matching a connection type in the `Connections` section.

For the `string` connection type with a stiffness (k) of 100N/m (it is actually unitless, but it helps me to think of everything in terms of metric units) and the unstretched length of the string 95% of the currently defined length between nodes:

```yaml
builders:
    strings:
        stiffness: 100
        type: string
        initial_length_ratio: 0.95
```

If the unstretched length of the string is unknown but the tension is known, Hooke's Law can be used to calculate the initial length: $F = k * (l_s - l)$ where l_s is the stretched length of the string (distance between it's nodes) and l is the unstretched length.

Bars are defined in the builder's sections just like strings. The `initial_length_ratio` is normally left out because bars are usually significantly stiff enough it is assumed to be 1

```yaml
builders:
    bars:
        stiffness: 10000
        type: bar
```

Important: The name of each builder can be what ever the user desires (bars, strings, high_tension_strings, blue_string, red_springs, etc), but the defined type in each must be exactly `bar` or `string`

3.4 Pins

In 2D space the solved structure can float anywhere in the XY plane with any rotation unless we pin nodes (to define a place in XY space the structure is fixed to)

A pin needs a node name and a list of True/False values, with True indicating that the node is translationally pinned in that direction. To pin `Node1` in the x and y directions:

```
pin:
  Node1: [True, True, False]
```

3.5 Control

The `control` section defines which strings are able to be controlled (change length).

To define a control string the name of the connections need to be defined. For instance if the connection `String1` is being controlled:

```
control:
  - String1
```

3.6 Surface

3.6.1 Type

The only type of surface currently implemented is a cylinder. A radius must be specified for the radius of the cylinder to wrap the tensegrity around.

The structure is wrapped around a $\hat{k}$ axis. In other words the x-axis wraps the circumference of the cylinder with a set radius, r. In the future I hope we can define the radius to change as a function of the z height, either with an equation, or reading in points from a file and using interpolation.

3.6.2 Linked Nodes

The linked nodes section takes pairs of nodes to be connected to each other on opposite sides of the cylinder.

```
surface:
  cylinder:
    radius: 3.5

  linked_nodes:
    - [Node1, Node7]
    - [Node4, Node8]
    - [Node9, Node12]
```

The only currently defined surface for linking nodes around is a cylinder.

CHAPTER

FOUR

DATA STRUCTURES

4.1 Node

class TensegritySim.**Node**(*name: str*, *position: list*)

Represents a node in a 2D tensegrity structure.

name

The name of the node.

Type

str

position

The position of the node in 2D or 3D space as a numpy array.

Type

numpy.ndarray

copy()

4.2 Connection

class TensegritySim.**Connection**(*nodes: List[Node]*, *connection_type: ConnectionType*, *stiffness: float = 0*, *initial_length: float | None = None*, *name: str | None = None*)

Represents a connection between nodes in a tensegrity structure.

nodes

A list of nodes that are part of the connection.

Type

List[*Node*]

nodes_original

A list of the original positions of the nodes.

Type

List[*Node*]

connection_type

The type of connection.

Type

ConnectionType

`stiffness`

The stiffness of the connection.

Type

float

`initial_length`

The initial length of the connection.

Type

float

`force`

The force in the connection.

Type

float

`name`

The name of the connection.

Type

str, optional

`initial_length_ratio`

The initial length ratio of the connection.

Type

float, optional

`class` **`ConnectionType`**(*value*)

An enumeration.

`BAR = 2`

`STRING = 1`

`current_length`(*linked_nodes: List[Tuple[Node, Node]] | None = None*)

Calculates the current length of the connection, considering linked nodes if provided.

Parameters

`linked_nodes` (*`List[Tuple[Node, Node]], optional`*) – A list of tuples representing linked nodes. Defaults to None.

Returns

The current length of the connection.

Return type

float

`update_force`(*current_length: float*)

Updates the force in the connection using the provided current length.

Parameters

`current_length` (*`float`*) – The current length of the connection.

4.3 Surface

`class TensegritySim.`**`Surface`**(*shape: Dict*, *linked_nodes: List[Tuple[*Node, Node*]]*)

Represents a surface in the simulation.

`shape`

The shape of the surface. Contains 'surface_type' and 'properties'.

Type
dict

`linked_nodes`

A list of tuples representing the linked nodes that form the seam on the surface.

Type
List[Tuple[*Node*, *Node*]]

4.4 Tensegrity

`class TensegritySim.`**`Tensegrity`**(*nodes: List[*Node*]*, *connections: List[*Connection*]*, *pins: Dict[str, List[bool]] | None = None*, *controls: List[*Connection*] | None = None*, *surface:* Surface *| None = None*, *dim: int | None = None*)

Represents a tensegrity structure.

`nodes`

A list of nodes in the tensegrity structure.

Type
List[*Node*]

`connections`

A list of connections between the nodes.

Type
List[*Connection*]

`pins`

A dictionary representing the pinned nodes. Defaults to an empty dictionary.

Type
Dict[str, List[bool]], optional

`controls`

A list of control connections. Defaults to an empty list.

Type
List[*Connection*], optional

`surface`

The surface on which the tensegrity structure is placed. Defaults to None.

Type
Surface, optional

`dim`

The dimension of the tensegrity structure. Should be 2, 2.5, or 3. Defaults to None (will automatically be set).

Type
int, optional

`update_forces()`

Updates the forces in all connections. Call after updating the positions of the nodes.

`get_control_order()`

Returns a comma-separated string of the names of the control connections.

Returns
A comma-separated string of the names of the control connections.

Return type
str

`change_control_lengths`(**delta_lengths*)

Changes the lengths of the control connections by the given delta lengths.

Parameters
`*delta_lengths` (*`float`*) – Variable number of arguments representing the changes in lengths for each control connection.

Raises
`ValueError` – If the number of delta lengths provided does not match the number of control connections.

`reset_control_lengths()`

Resets the lengths of the control connections to their original lengths.

CHAPTER

FIVE

YAMLPARSER

class TensegritySim.**YamlParser**

A class for parsing YAML files and creating a Tensegrity object.

parse(file

str) -> Tensegrity: Parses the YAML file and returns a Tensegrity object.

static parse(*file: str*) → *Tensegrity*

Parses the YAML file and returns a Tensegrity object.

Parameters

`file` (*str*) – The path to the YAML file.

Returns

The tensegrity system containing Nodes and Connections.

Return type

Tensegrity

Raises

- `FileNotFoundError` – If the specified file does not exist.
- `yaml.YAMLError` – If the YAML file is invalid.
- `KeyError` – If a builder type does not have a builder.
- `ValueError` – If a connection type is not recognized.

CHAPTER

SIX

TENSEGRITYSOLVER

class `TensegritySim.`**`TensegritySolver`**(*tensegrity: Tensegrity*)

TensegritySolver class for solving the positions of nodes in a tensegrity structure. This class provides methods to initialize the solver, set forces on nodes, and solve the positions of nodes using optimization techniques. It also includes internal functions to compute the objective function, spring connection energy, and its derivatives.

`tensegrity`

The tensegrity object containing nodes and connections.

Type

Tensegrity

`dim`

The dimension of the optimization problem (defaults to tensegrity's dim).

Type

int

`set_forces`(*forces: Dict[str, ndarray]*) → None

Sets the forces on the nodes in the tensegrity structure.

Parameters

`forces` (*`Dict[str, np.ndarray]`*) – A dictionary containing the forces on each node.

Raises

`ValueError` – If the force vector does not have the same dimension as the optimization problem.

`solve`(*method: str = 'lm'*) → None

Solves the position of nodes in the tensegrity structure.

This method uses the root function from the scipy.optimize module to find the positions of the nodes in the tensegrity structure with Virtual Work.

Parameters

`method` (*`str`*) – The method to use for the root function (default is "lm").

Returns

None. Changes are made internally to the Tensegrity object.

`_objective`(*x: ndarray*) → ndarray

Computes the objective function for the optimization problem. This function calculates the virtual work from potential energy and external forces, and optionally includes surface constraints if a surface is defined.

Parameters

`x` (*`np.ndarray`*) – Input array representing the generalized coordinates.

Returns
The objective function value, which is the virtual work with optional surface constraints.

Return type
np.ndarray

`_spring_connection_energy`(*connection: Connection*, *N: ndarray*) → float

Calculates the energy stored in a spring connection.

Parameters

- `connection` (`Connection`) – The spring connection object.
- `N` (*np.ndarray*) – The current positions of all nodes.

Returns
The energy stored in the spring connection.

Return type
float

`_spring_connection_energy_derivative`(*connection: Connection*, *N: ndarray*) → ndarray

Calculates the derivative of the spring connection energy with respect to node positions.

The energy of a spring connection is given by:
V = 0.5 * k * (l - l0)^2

where k is the stiffness, l is the current length of the connection, and l0 is the rest length.

The derivative of the energy with respect to the position of node i is:

dV/dq_i = -k * (l - l0) * dl/dq_i
= C * dl/dq_i

where C is a constant factor.

Parameters

- `connection` (`Connection`) – The connection object representing the spring.
- `N` (*np.ndarray*) – The array of node positions.

Returns
The derivative of the spring connection energy with respect to the node positions.

Return type
np.ndarray

`_length_derivative`(*connection: Connection*, *N: ndarray*) → ndarray

Calculates the derivative of the length of a connection with respect to the node positions.

Parameters

- `connection` (`Connection`) – The connection object containing the nodes.
- `N` (*np.ndarray*) – The array of node positions.

Returns
The derivative of the length with respect to the node positions.

Return type
np.ndarray

`_connection_length`(*connection: Connection*, *N: ndarray*) → float

Calculates the current length of a connection based on the node positions.

Parameters

- **`connection`** (`Connection`) – The connection object.
- **`N`** (*`np.ndarray`*) – The current positions of all nodes.

Returns

The current length of the connection.

Return type

float

`_node_distance`(*node1: str*, *node2: str*, *N: ndarray*) → float

Calculates the distance between two nodes based on their positions.

Parameters

- **`node1`** (*`str`*) – The name of the first node.
- **`node2`** (*`str`*) – The name of the second node.
- **`N`** (*`np.ndarray`*) – The current positions of all nodes.

Returns

The distance between the two nodes.

Return type

float

`_surface_constraints`(*x: ndarray*) → ndarray

Computes the surface constraints for the optimization problem.

Parameters

`x` (*`np.ndarray`*) – The input array representing the current state of the nodes.

Returns

An array of constraints that must be satisfied. For a cylindrical surface,
the constraints ensure that: - The y-coordinates of linked nodes are equal. - The x-coordinates of linked nodes are exactly the circumference of the cylinder apart.

Return type

np.ndarray

`_create_initial_guess`() → ndarray

Creates the input vector x0 for the optimization problem (node positions - pinned nodes).

Returns

The input vector x0.
length = d*len(nodes) - pins, Elements are the node positions (except those that are pinned)

Return type

np.ndarray

`_get_nodes_from_input`(*x: ndarray*) → ndarray

Extracts node positions input vector (adding the pinned nodes back in).

Parameters

`x` (*`np.ndarray`*) – The input vector containing node positions.

Returns
The extracted node positions including those removed from the input because they were pinned.

Return type
np.ndarray

CHAPTER

SEVEN

VISUALIZATION

class TensegritySim.**Visualization**(*tensegrity: Tensegrity*, *dim: int | None = None*)

Visualization class for tensegrity structures.

This class provides methods to visualize 2D and 3D tensegrity structures using matplotlib.

tensegrity

The tensegrity structure to visualize.

Type
Tensegrity

fig

Matplotlib figure object.

Type
Figure

ax

Matplotlib axes object.

Type
Axes

dim

Dimension of the visualization (2, 2.5 or 3), set by the dim of the tensegrity.

Type
int

plot(*label_nodes: bool = False*, *label_connections: bool = False*, *label_forces: bool = False*)

Plots the visualization of the tensegrity structure.

Parameters

- **label_nodes** (*bool, optional*) – Whether to label the node names in the plot. Defaults to False.
- **label_connections** (*bool, optional*) – Whether to label the connection names in the plot. Defaults to False.
- **label_forces** (*bool, optional*) – Whether to label the forces on the connections. Defaults to False.

_plot_2d(*label_nodes: bool = False*, *label_connections: bool = False*, *label_forces: bool = False*)

Plots the 2D visualization of the tensegrity structure.

Parameters

- `label_nodes` (*bool, optional*) – Whether to label the node names in the plot. Defaults to False.
- `label_connections` (*bool, optional*) – Whether to label the connection names in the plot. Defaults to False.
- `label_forces` (*bool, optional*) – Whether to label the forces on the connections. Defaults to False.

`_plot_3d`(*label_nodes: bool = False*, *label_connections: bool = False*, *label_forces: bool = False*)

Plots the 3D visualization of the tensegrity structure.

Parameters

- `label_nodes` (*bool, optional*) – Whether to label the node names in the plot. Defaults to False.
- `label_connections` (*bool, optional*) – Whether to label the connection names in the plot. Defaults to False.
- `label_forces` (*bool, optional*) – Whether to label the forces on the connections. Defaults to False.

`_plot_2_5d`(*label_nodes: bool = False*, *label_connections: bool = False*, *label_forces: bool = False*)

Plots the 2.5D visualization of the tensegrity structure.

Parameters

- `label_nodes` (*bool, optional*) – Whether to label the node names in the plot. Defaults to False.
- `label_connections` (*bool, optional*) – Whether to label the connection names in the plot. Defaults to False.
- `label_forces` (*bool, optional*) – Whether to label the forces on the connections. Defaults to False.

`set_3d_equal_scaling`(*ax*)

Sets equal scaling for the 3D plot.

Parameters

`ax` (*Axes3D*) – The 3D axes object.

INDEX