uraeus.smbd

Release 0.0.1.dev3

Aug 17, 2020

Contents

Symbolic Multi-Body Dynamics in Python | A python package for the symbolic creation and analysis of constrained multi-body systems.

Note: *The documentation is still under construction . . .*

CHAPTER 1

Multi-Body Systems

In modern literature, multi-body systems refer to modern mechanical systems that are often very complex and consist of many components interconnected by joints and force elements such as springs, dampers, and actuators. Examples of multi-body systems are machines, mechanisms, robotics, vehicles, space structures, and bio-mechanical systems. The dynamics of such systems are often governed by complex relationships resulting from the relative motion and joint forces between the components of the system.[1]

Therefore, a multi-body system is hereby defined as *a finite number of material bodies connected in an arbitrary fashion by mechanical joints that limit the relative motion between pairs of bodies*. Practitioners of multi-body dynamics study the generation and solution of the equations governing the motion of such systems[2].

[1] Shabana, A.A., Computational Dynamics, Wiley, New York, 2010.
[2] McPhee, J.J. Nonlinear Dyn (1996) 9: 73. https://doi.org/10.1007/BF01833294

CHAPTER 2

Audience and Fields of Application

Initially, the main targeted audience was the Formula Student community. The motive was to encourage a deeper understanding of the modeling processes and the underlying theories used in other commercial software packages, which is a way of giving back to the community, and supporting the concept of "**knowledge share**" adopted there by exposing it to the open-source community as well.

Currently, the tool aims to serve a wider domain of users with different usage goals and different backgrounds, such as students, academic researchers and industry professionals.

Fields of application include any domain that deals with the study of interconnected bodies, such as:

- Ground Vehicles' Systems.
- Construction Equipment.
- Industrial Mechanisms.
- Robotics.
- Biomechanics.
- etc.

CHAPTER 3

Features

Currently, **uraeus.smbd** provides:

- Creation of symbolic **template-based** and **standalone** multi-body systems using minimal API via python scripting.
- Convenient and easy creation of complex multi-body assemblies.
- Convenient visualization of the system topology as a network graph.
- Viewing the system's symbolic equations in a natural mathematical format using Latex printing.
- Optimization of the system symbolic equations by performing common sub-expressions elimination.
- Creation of symbolic configuration files to facilitate the process of numerical simulation data entry.

CHAPTER 4

Guide

4.1 Installation

The package needs a valid python 3.6+ environment. If new to scientific computing in python, Anaconda is a recommended free python distribution from Continuum Analytics that includes SymPy, SciPy, NumPy, Matplotlib, and many more useful packages for scientific computing, which provides a nice coherent platform with most of the tools needed.

4.1.1 Pip

```
pip install uraeus.smbd
```

4.1.2 Git

As the package is still under continuous development, cloning this repository is a more versatile way to test and play with it, until a more stable first release is released. This can be done via the following **git** command using the terminal (Linux and Mac) or powershell (Windows).

```
git clone https://github.com/khaledghobashy/uraeus_smbd.git
```

This will download the repository locally on your machine. To install the package locally and use it as other python packages, using the same terminal/powershell run the following command:

```
pip install -e uraeus_smbd
```

4.2 Background

4.2.1 Background and Approach

The Problem

What is Multi-Body Dynamics?

As mentioned earlier, a multi-body system is hereby defined as *a finite number of material bodies connected in an arbitrary fashion by mechanical joints that limit the relative motion between pairs of bodies*, where practitioners of multi-body dynamics study the generation and solution of the equations governing the motion of such systems.

What is the problem to be solved?

One of the primary interests in multi-body dynamics is to analyze the behavior of a given multi-body system under the effect of some inputs. In analogy with control systems; a multi-body system can be thought as a system subjected to some inputs producing some outputs. These three parts of the problem are dependent on the analyst end goal of the analysis and simulation.

How is the system physics abstracted mathematically?

An unconstrained body in space is normally defined using 6 generalized coordinates defining its location and orientation in space. For example, a system of 10 bodies requires 60 generalized coordinates to be fully defined, which in turn requires 60 independent equations to be solved for these unknown generalized coordinates.

The way we achieve a solution for the system is dependent on the type of study we are performing. Mainly we have four types of analysis that are of interest for a given multi-body system. These are:

- **Kinematic Analysis** "How does the whole system move if we moved this particular body ?"
- **Inverse Dynamic Analysis** "What are the forces needed to achieve this motion we just did ?"
- **Equilibrium Analysis** "How does the system look if we did nothing ?"
- **Dynamic Analysis** "Now we gave it a force, how does it behave ?"

Each analysis type -or question- can be modeled by a set of algebraic and/or differential equations that can be solved for the system generalized states (positions, velocities and accelerations).

Note: A more detailed discussion of each analysis type will be provided in another part of the documentation and linked here.

The Approach

The philosophy of the **uraeus** framework is to isolate the model creation process form the actual numerical and computational representation of the system that will be used in the numerical simulation process. This is done through the concepts of **symbolic computing** and **code-generation**. The uraeus.smbd package is responsible for the symbolic creation of multi-body systems.

Symbolic Topology

The System Topology is a description of the connectivity relationships between the bodies in a given multi-body system. These relationships represent the system constraints that limit the relative motion between the system bodies and produce the desired kinematic behavior.

The package abstracts the topology of a given system as a multi-directed graph, where each **node** represents a **body** and each **edge** represents a **connection** between the end nodes, where this connection may represents a **joint**, an **actuator** or a **force element**.

No numerical inputs is needed at that step, the focus is only on the validity of the topological design of the system, **not** how it is configured in space.

This problem statement and approach leads to the following important landmarks:

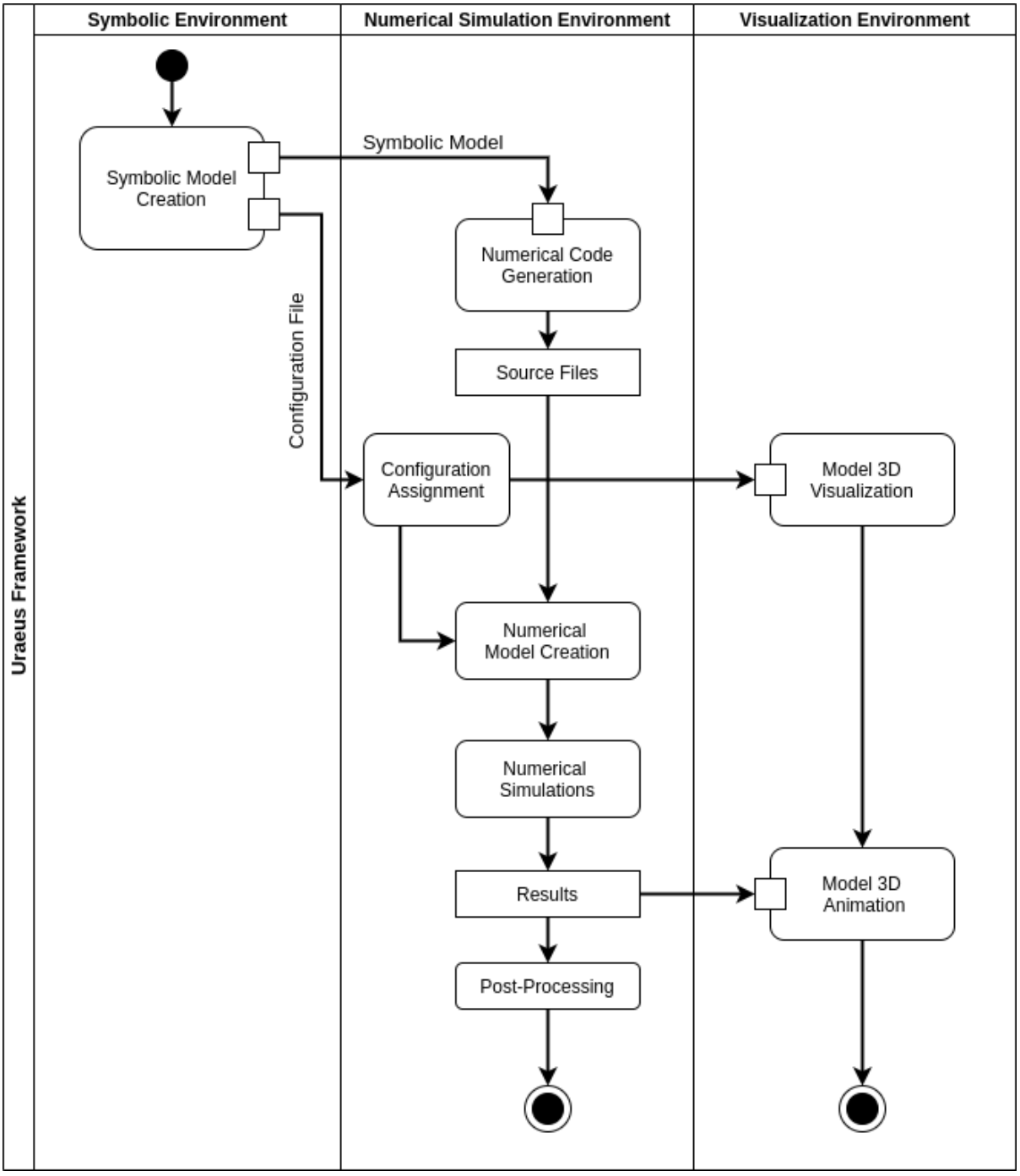
-

4.3 Tutorials

This section presents some tutorials illustrating the process of creating standalone and template-based symbolic models.

The figure below shows a high-level **activity diagram** of a typical usage flow of the **uraeus** framework, where we have three swim-lanes representing a main layer of the three activity layers of the framework.

Here we are concerned with the first *swim-lane* that represents the **symbolic environment** layer and the **symbolic model creation** activity.



4.3.1 Symboilc Models

The **uraeus.smbd** python package provides two main types of symbolic models to be constructed, **standalone models**, and **template-based** models.

Standalone models stands for symbolic models that are fully described in **one** topology graph, where all the bodies and connections are already defined in the model graph, and therefore, the model is fully independent and does not need any other topological information.

Template-based models, on the other hand, are not fully described in one topology, as they need to be assembled with other templates to form a complete assembled topology. For example, this is the case for creating a full vehicle assembly, where we model the vehicle subsystems as templates, then we assemble them together to form the desired vehicle assembly.

The creation of template-based models' database can be found in uraeus.fsae, which is an under-development multi-body systems database for formula-style vehicles implemented in the **uraeus** framework.

Guide

Standalone Topologies

Project Structure

An important step to do before the actual modeling step, is how can we structure and store our files and scripts to keep things organized some how. This is not a mandatory step but is highly recommended to keep things neat.

A recommended project structure for standalone projects is as follows:

```
standalone_project/
├── numenv
├── simenv
│   ├── configs
│   ├── python
│   │   └── simulation.py
│   └── results
└── symenv
    ├── data
    └── sym_model.py
```

On the root level, we have three main directories:

1. **numenv** Short for "**numerical environments**". A directory to store the numerical code files of the model that are generated by the uraeus code-generators.
2. **simenv** Short for "**simulation environments**". A directory to store the numerical simulation files, the numerical configurations .json files and the results of the simulations. The directory contains two main sub-directories, the results and the config directories, where additional directories are added for the desired simulation environments used.
3. **symenv** Short for "**symbolic environment**". A directory to store the symbolic model files. This directory has the sym_model.py script that contains the model script, and a data directory that is used to store the output files from the "Symbolic Model Creation" activity.

Note: It should be noted that this project structure may not be the optimal way to structure your project, and you are encouraged to restructure this as you see fit.

Examples

Below is a list of examples of multi-body models created as standalone models and project.

Pendulum

Model Description

A pendulum is a weight suspended from a pivot so that it can swing freely. It consists of two bodies, the pendulum and the ground, attached together through a pin/revolute joint. More general information about the pendulum can be found here.

Topology Layout

The mechanism consists of 1 Body + 1 Ground. Therefore, total system coordinates -including the ground- is $n = n_b \times 7 = 2 \times 7 = 14$, where n_b is the total number of bodies in the system.[1]

List of Bodies

The list of bodies is given below:

- Pendulum body $body$.

Connectivity

The system connectivity is as follows:

- Pendulum $body$ is connected to the **ground** by a revolute joint, resulting in constraint equations $n_{c,rev} = 5$.

Table 1: Connectivity Table

Joint Name	Body i	Body j	Joint Type	n_c
a	$ground$	$body$	Revolute	5

Degrees of Freedom

The degrees of freedom of the system can be calculated as:

$$n - (n_{c,rev} + n_{c,P} + n_{c,g}) = 14 - (5 + (1 \times 1) + 7) = 14 - 13 = 1$$

Where:

- $n_{c,P}$ represents the constraints due to euler-parameters normalization equations.
- $n_{c,g}$ represents the constraints due to the ground constraints.
- $n_{c,rev}$ represents the constraints due to the revolute joint.

---

[1] **uraeus.smbd** uses euler-parameters -which is a 4D unit quaternion- to represents bodies orientation in space. This makes the generalized coordinates used to fully define a body in space to be **7,** instead of **6**, it also adds an algebraic equation to the constraints that ensures the unity/normalization of the body quaternion. This is an important remark as the calculations of the degrees-of-freedom depends on it.

Symbolic Topology

In this section, we create the symbolic topology that captures the topological layout that we discussed earlier. Defining the topology is very simple. We start by importing the `standalone_topology` class and create a new instance that represents our symbolic model, let it be `sym_model`. Then we start adding the components we discussed earlier, starting by the bodies, then the joints, actuators and forces, and thats it. These components will be represented symbolically, and therefore there is no need for any numerical inputs at this step.

The system is stored in a form of a network graph that stores all the data needed for the assemblage of the system equations later. But even before the assemblage process, we can gain helpful insights about our system as well be shown.

Directories Construction

```
# standard library imports
import os

# getting directory of current file and specifying the directory
# where data will be saved
os.makedirs(os.path.join("model", "symenv", "data"), exist_ok=True)

data_dir = os.path.abspath("model/symenv/data")
```

Topology Construction

```
# uraeus imports
from uraeus.smbd.systems import standalone_topology, configuration

# ============================================================ #
#                        Symbolic Topology
# ============================================================ #

# Creating the symbolic topology as an instance of the
# standalone_topology class
project_name = 'pendulum'
sym_model = standalone_topology(project_name)

# Adding Bodies
# =============
sym_model.add_body('body')

# Adding Joints
# =============
sym_model.add_joint.revolute('a', 'ground', 'rbs_body')
```

Symbolic Characteristics

Now, we can gain some insights about our topology using our `sym_model` instance. By accessing the `topology` attribute of the `sym_model`, we can visualize the connectivity of the model as a network graph using the `sym_model.topology.draw_constraints_topology()` method, where the nodes represent the bodies, and the edges represent the joints, forces and/or actuators between the bodies.

```
sym_model.topology.draw_constraints_topology()
```

Also, we can check the system's number of generalized coordinates n and number of constraints nc.

```
print(sym_model.topology.n, sym_model.topology.nc)
```

Assembling

This is the last step of the symbolic building process, where we ask the system to assemble the governing equations, which will be used then in the code generation for the numerical simulation, as well as further symbolic manipulations.

Also, we can export/save a *pickled* version of the model.

```
# Assembling and Saving model
sym_model.save(data_dir)
sym_model.assemble()
```

Note: The equations' notations will be discussed in another part of the documentation.

Symbolic Configuration

In this step we define a symbolic configuration of our symbolic topology. As you may have noticed in the symbolic topology building step, we only cared about the **topology**, thats is the system bodies and their connectivity, and we did not care explicitly with how these components are configured in space.

In order to create a valid numerical simulation session, we have to provide the system with its numerical configuration needed, for example, the joints' locations and orientations. The symbolic topology in its raw form will require you to manually enter all these numerical arguments, which can be cumbersome even for smaller systems. This can be checked by checking the configuration inputs of the symbolic configuration as `sym_config.config.input_nodes`

Here we start by stating the symbolic inputs we wish to use instead of the default inputs set, and then we define the relation between these newly defined arguments and the original ones.

```
# ============================================================ #
#                      Symbolic Configuration
# ============================================================ #

# Symbolic configuration name.
config_name = "%s_cfg"%project_name

# Symbolic configuration instance.
sym_config = configuration(config_name, sym_model)

# Adding the desired set of UserInputs
# ====================================
sym_config.add_point.UserInput('p1')
sym_config.add_point.UserInput('p2')

sym_config.add_vector.UserInput('v')
```

(continues on next page)

(continued from previous page)

```
# Defining Relations between original topology inputs
# and our desired UserInputs.
# ===================================================

# Revolute Joint (a) location and orientation
sym_config.add_relation.Equal_to('pt1_jcs_a', ('hps_p1',))
sym_config.add_relation.Equal_to('ax1_jcs_a', ('vcs_v',))

# Creating Geometries
# ===================
sym_config.add_scalar.UserInput('radius')

sym_config.add_geometry.Sphere_Geometry('body', ('hps_p2', 's_radius'))
sym_config.assign_geometry_to_body('rbs_body', 'gms_body')

# Exporting the configuration as a JSON file
sym_config.export_JSON_file(data_dir)
```

Note: The details of this process will be discussed in another part of the documentation.

Template-Based Topologies

4.4 Implementation Details

4.5 Reference

4.5.1 Symbolic Components

Bodies

Rigid Body

class uraeus.smbd.symbolic.components.bodies.**body**(*name*)

A class that represents an un-constrained rigid body object in 3D in a symbolic form, where all the body parameters and equations are generated automatically in a symbolic format.

Parameters **name** (*str*) – Name of the body instance. Should mimic a valid python variable name.

Note: An un-constraied body in space is typically defined using 6 generalized coordinates representing its' location and orientation. In cartesian coordinate system, body location is simply defined by the (x, y, z) coordinates of a reference point on the body -normally the center-of-mass - , where the body orientation can be defined in various ways, such as the directional cosines matrix, euler-angles and euler-parameters.

The package uses euler-parameters -which is a 4D unit quaternion- to represents a given body orientation in space. This makes the generalized coordinates used to fully define a body in space to be 7, instead of 6, it also adds an algebraic equation to the constraints that ensures the unity/normalization of the body quaternion.

Attributes

n
Number of generalized coordinates used to define the body configuration. Equals 7.

Type int

nc
Number of scalar constraint equations (euler-parameters normalization).

Type int

nve
Number of vetor constraint equations (euler-parameters normalization).

Type int

A
The directional cosines matrix that represents the symbolic orientation of the body relative to the global_frame. This matrix is function of the body orientation parameters e.g. euler-parameters

Type A

R
A symbolic matrix that represents the location of the body's reference point relative to the global origin.

Type vector

Rd
A symbolic matrix that represents the translational velocity of the body's reference point relative to the global origin.

Type vector

P
A symbolic matrix that represents the orientation of the body's reference frame relative to the global frame in terms of euler-parameters.

Type quatrenion

Pd
A symbolic matrix that represents the rotaional velocity of the body's reference frame relative to the global frame in terms of euler_parameters time-derivatives.

Type quatrenion

q
Blockmatrix containing the position-level coordinates of the body

Type sympy.BlockMatrix

qd
Blockmatrix containing the velocity-level coordinates of the body

Type sympy.BlockMatrix

normalized_pos_equation
The normalization equation of the euler-parameters quatrenion at the position level.

Type sympy.MatrixExpr

normalized_vel_equation
The normalization equation of the euler-parameters quatrenion at the velocity level.

Type sympy.MatrixExpr

normalized_acc_equation
The normalization equation of the euler-parameters quatrenion at the acceleration level.

Type sympy.MatrixExpr

normalized_jacobian
The jacobian of the normalization equation of the euler-parameters quatrenion relative to the vector of euler-parameters.

Type list (of sympy.MatrixExpr)

arguments_symbols
A list containing the symbolic mathematical objects that should be nuemrically defined by the user in a numerical simulation session.

Type list (of symbolic objects)

runtime_symbols
A list containing the symbolic mathematical objects that changes during the run-time of a nuemric simulation's "solve" method.

Type list (of symbolic objects)

constants_symbolic_expr
A list containing sympy equalities representing the values of internal class symbolic constants that are evaluated from other symbolic expressions.

Type list (of sympy.Equality)

constants_numeric_expr
A list containing sympy equalities representing the values of internal class symbolic constants that are evaluated directly from numerical expressions.

Type list (of sympy.Equality)

constants_symbols
A list containing all the symbolic mathematical objects that represent constants for the given body instance.

Type list (of symbolic objects)

Joints

Mechanical Joints

Spherical Joint

class uraeus.smbd.symbolic.components.joints.**spherical**(*name*, *body_i=None*, *body_j=None*)
The spherical joint prevents the relative translational movement between the two connected bodies at a given common location, where the two bodies are free to rotate relative to each-other in all directions.

The joint definition requires one defintion point and one defintion axis.

Parameters

- **name** (*str*) – Name of the joint instance. Should mimic a valid python variable name.
- **body_i** (body) – The 1st body isntance. Should be an instance of the *body* class.
- **body_j** (body) – The 2nd body isntance. Should be an instance of the *body* class.

Revolute Joint

class uraeus.smbd.symbolic.components.joints.**revolute**(*name*, *body_i=None*, *body_j=None*)

The revolute joint allows only one rotation freedome between the connected bodies around a common axis, thus it fully prevents the relative translation between the bodies at the joint definition location, as well as any rotation other-than around the joint definition axis.

The joint definition requires one defintion point and one defintion axis.

Parameters

- **name** (*str*) – Name of the joint instance. Should mimic a valid python variable name.
- **body_i** (body) – The 1st body isntance. Should be an instance of the *body* class.
- **body_j** (body) – The 2nd body isntance. Should be an instance of the *body* class.

Cylinderical Joint

class uraeus.smbd.symbolic.components.joints.**cylinderical**(*name*, *body_i=None*, *body_j=None*)

The cylinderical joint allows only one relative rotation freedome and one relative translation freedome between the connected bodies along a common axis, thus it prevents any relative translation and rotation along any other direction, other-than around the joint definition axis.

The joint definition requires one defintion point and one defintion axis.

Parameters

- **name** (*str*) – Name of the joint instance. Should mimic a valid python variable name.
- **body_i** (body) – The 1st body isntance. Should be an instance of the *body* class.
- **body_j** (body) – The 2nd body isntance. Should be an instance of the *body* class.

Fixed Joint

class uraeus.smbd.symbolic.components.joints.**fixed**(*name*, *body_i=None*, *body_j=None*)

A joint that constraints two bodies to be fixed relative to each-other, by imposing six algebraic constraints equations to deminish the relative six degrees-of-freedom between the constrained bodies.

The joint definition requires one defintion point and one defintion axis.

Parameters

- **name** (*str*) – Name of the joint instance. Should mimic a valid python variable name.
- **body_i** (body) – The 1st body isntance. Should be an instance of the *body* class.
- **body_j** (body) – The 2nd body isntance. Should be an instance of the *body* class.

Fixed Orientation Joint

class uraeus.smbd.symbolic.components.joints.**fixed_orientation**(*name*, *body_i=None*, *body_j=None*)

A joint that constraints two bodies to have fixed relative orientation w.r.t each-other, by imposing three algebraic constraints equations to deminish the relative three relative orientation degrees-of-freedom between the constrained bodies.

The joint definition requires only one defintion axis.

Parameters

- **name** (*str*) – Name of the joint instance. Should mimic a valid python variable name.
- **body_i** (body) – The 1st body isntance. Should be an instance of the *body* class.
- **body_j** (body) – The 2nd body isntance. Should be an instance of the *body* class.

Translational Joint

class uraeus.smbd.symbolic.components.joints.**translational**(*name*, *body_i=None*, *body_j=None*)

The translational joint allows only one relative translation freedome between the connected bodies along a common axis, thus it prevents all relative rotations between the connected bodies, and any relative translation along any other direction, other-than around the joint definition axis.

The joint definition requires one defintion point and one defintion axis.

Parameters

- **name** (*str*) – Name of the joint instance. Should mimic a valid python variable name.
- **body_i** (body) – The 1st body isntance. Should be an instance of the *body* class.
- **body_j** (body) – The 2nd body isntance. Should be an instance of the *body* class.

Universal Joint

class uraeus.smbd.symbolic.components.joints.**universal**(*name*, *body_i=None*, *body_j=None*)

The universal joint prevents the relative translational movements between the connected bodies just like the spherical joint, but it also prevents the relative rotation/spin too, so, the connected body pair is only allowed to rotate around two common axes.

The joint definition requires one defintion point and two defintion axis.

Parameters

- **name** (*str*) – Name of the joint instance. Should mimic a valid python variable name.
- **body_i** (body) – The 1st body isntance. Should be an instance of the *body* class.
- **body_j** (body) – The 2nd body isntance. Should be an instance of the *body* class.

Force Elements

4.5.2 Symbolic Systems

4.6 License

CHAPTER 5

References

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

T

U