pyelastica

Release 0.0.1

mattiaLab

Mar 22, 2020

MAIN PAGE:

CHAPTER

ONE

ELASTICA-PYTHON

Python version of elastica is located in this repository. Uploaded code is tested against the Timoshenko beam analytical solution. This branch contains ongoing implementation of friction and rod rod joints.

I am writting a test example

1.1 Just a simple test

Here I will write more things

1.2 Hello World

1.2.1 Installation

Install the package (or add it to your `requirements.txt` file):

```
$ pip install sphinx_rtd_theme
```

In your `conf.py` file:

1.3 Documentation

1.3.1 Rods

Rod base classes and implementation details that need to be hidden from the user

class `elastica.rod.cosserat_rod.`**`CosseratRod`**(*n_elements*, *shear_matrix*, *bend_matrix*, *rod*, **args*, ***kwargs*)

Rod constitutive model mixins

Data structure wrapper for rod components

1.3.2 Boundary Conditions

Boundary conditions for rod

class elastica.boundary_conditions.**FreeRod**
the base class for rod boundary conditions also the free rod class

class elastica.boundary_conditions.**HelicalBucklingBC**(*position_start*, *position_end*, *director_start*, *director_end*, *twisting_time*, *slack*, *number_of_rotations*)
boundary condition for helical buckling controlled twisting of the ends

class elastica.boundary_conditions.**OneEndFixedRod**(*fixed_position*, *fixed_directors*)
the end of the rod fixed x[0]

External forcing for rod

class elastica.external_forces.**EndpointForces**(*start_force*, *end_force*, *ramp_up_time=0.0*)
Applies constant forces on endpoints

apply_forces(*system*, *time=0.0*)
Apply forces to a system object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

class elastica.external_forces.**GravityForces**(*acc_gravity=array([0., -9.80665, 0.])*)
Applies a constant gravity on the entire rod

apply_forces(*system*, *time=0.0*)
Apply forces to a system object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

class elastica.external_forces.**MuscleTorques**(*base_length*, *b_coeff*, *period*, *wave_number*, *phase_shift*, *direction*, *ramp_up_time=0.0*, *with_spline=False*)
Applies muscle torques on the body. It can apply muscle torques as travelling wave with beta spline or only as travelling wave.

apply_torques(*system*, *time: float = 0.0*)
Apply torques to a Rod-like object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

class elastica.external_forces.**NoForces**

Base class for external forcing for Rods

Can make this an abstract class, but its inconvenient for the user to keep on defining apply_forces and apply_torques object over and over.

apply_forces (*system*, *time: float = 0.0*)

Apply forces to a system object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

apply_torques (*system*, *time: float = 0.0*)

Apply torques to a Rod-like object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

class elastica.external_forces.**UniformForces** (*force*, *direction=array([0., 0., 0.])*)

Applies uniform forces to entire rod

apply_forces (*system*, *time: float = 0.0*)

Apply forces to a system object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

class elastica.external_forces.**UniformTorques** (*torque*, *direction=array([0., 0., 0.])*)

Applies uniform torque to entire rod

apply_torques (*system*, *time: float = 0.0*)
Apply torques to a Rod-like object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

Interaction module

class elastica.interaction.**AnistropicFrictionalPlane** (*k*, *nu*, *plane_origin*, *plane_normal*, *slip_velocity_tol*, *static_mu_array*, *kinetic_mu_array*)

apply_forces (*system*, *time=0.0*)
Apply forces to a system object.

In NoForces, this routine simply passes.

Parameters

- **system** (*system that is Rod-like*) –
- **time** (*np.float, the time of simulation*) –

Returns

Return type None

class elastica.interaction.**SlenderBodyTheory** (*dynamic_viscosity*)

apply_forces (*system*, *time=0.0*)
This function applies hydrodynamic forces on body using the slender body theory given in Eq. 4.13 Gazzola et. al. RSoS 2018 paper

Parameters **system** –

elastica.interaction.**find_slipping_elements** (*velocity_slip*, *velocity_threshold*)
This function takes the velocity of elements and checks if they are larger than the threshold velocity. If velocity of elements are larger than threshold velocity, that means those elements are slipping, in other words kinetic friction will be acting on those elements not static friction. This function output an array called slip function, this array has a size of number of elements. If velocity of element is smaller than the threshold velocity slip function value for that element is 1, which means static friction is acting on that element. If velocity of element is larger than the threshold velocity slip function value for that element is between 0 and 1, which means kinetic friction is acting on that element.

Parameters

- **velocity_slip** –
- **velocity_threshold** –

Returns

Return type slip function

`elastica.interaction.`**`node_to_element_velocity`**

This function computes to velocity on the elements. Here we define a seperate function because benchmark results showed that using numba, we get almost 3 times faster calculation

Parameters **`node_velocity`** –

Returns

Return type element_velocity

Note: Faster than pure python for blocksize 100 python: 3.81 µs ± 427 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each) this version: 1.11 µs ± 19.3 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

`elastica.interaction.`**`slender_body_forces`**

This function computes hydrodynamic forces on body using slender body theory. Below implementation is from the Eq. 4.13 in Gazzola et. al. RSoS 2018 paper.

Fh = - 4*pi*mu/ln(L/r) * ((I - 0.5 * t`t) * v)

Parameters

- **`tangents`** –
- **`velocity_collection`** –
- **`dynamic_viscosity`** –
- **`length`** –
- **`radius`** –

Returns

- *Faster than numpy einsum implementation for blocksize 100*
- **numpy** (*39.5 µs ± 6.78 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)*)
- **this version** (*3.91 µs ± 310 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)*)
- *Unrolling loops show better performance. Also since we are working in 3D everything is*
- *3 dimensional.*

`elastica.interaction.`**`sum_over_elements`**

This function sums all elements of input array, using a numba jit decorator shows better performance compared to python sum(), .sum() and np.sum()

Parameters **`input`** –

Returns

- *Faster than sum(), .sum() and np.sum()*
- *For blocksize = 200*
- **sum()** (*36.9 µs ± 3.99 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)*)
- **.sum()** (*3.17 µs ± 90.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)*)
- **np.sum()** (*5.17 µs ± 364 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)*)
- **This version** (*513 ns ± 24.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)*)

1.3.3 Multiple Rod Connections

Joint between rods module

class elastica.joint.**HingeJoint**(*k*, *nu*, *kt*, *normal_direction*)

this joint currently keeps rod one fixed and moves rod two how couples act needs to be reconfirmed

1.3.4 Callback Functions

Call back functions for rod

class elastica.callback_functions.**CallBackBaseClass**

Base call back class, user has to derive new call back classes from this class

make_callback(*system*, *time*, *current_step: int*)

This function will be called every time step, user can define which parameters at which time-step to be called back in derived call back class :param system: :type system: system is rod :param time: :type time: simulation time :param current_step: :type current_step: current simulation time step

class elastica.callback_functions.**ContinuumSnakeCallBack**(*step_skip: int*, *callback_params*)

Call back function for continuum snake

make_callback(*system*, *time*, *current_step: int*)

This function will be called every time step, user can define which parameters at which time-step to be called back in derived call back class :param system: :type system: system is rod :param time: :type time: simulation time :param current_step: :type current_step: current simulation time step

class elastica.callback_functions.**MyCallBack**(*step_skip: int*, *callback_params*)

My call back class it is derived from the base call back class. This is an example, user can use this class as an example to write new call back classes

make_callback(*system*, *time*, *current_step: int*)

This function will be called every time step, user can define which parameters at which time-step to be called back in derived call back class :param system: :type system: system is rod :param time: :type time: simulation time :param current_step: :type current_step: current simulation time step

1.3.5 Time steppers

Symplectic timesteppers and concepts

class elastica.timestepper.symplectic_steppers.**PEFRL**

Position Extended Forest-Ruth Like Algorithm of I.M. Omelyan, I.M. Mryglod and R. Folk, Computer Physics Communications 146, 188 (2002), http://arxiv.org/abs/cond-mat/0110585

class elastica.timestepper.symplectic_steppers.**PositionVerlet**

class elastica.timestepper.symplectic_steppers.**SymplecticLinearExponentialIntegrator**

class elastica.timestepper.symplectic_steppers.**SymplecticStepper**(*cls=None*)

1.3.6 Wrappers

base_system

basic coordinating multiple, smaller systems that have an independently integrable interface (ie. works with symplectic or explicit routines *timestepper.py*.)

class `elastica.wrappers.base_system.`**`BaseSystemCollection`**

Base System

Technical note : We can directly subclass a list for the most part, but this is a bad idea, as List is non abstract https://stackoverflow.com/q/3945940

`finalize`()

Finalizes all feature class methods

`insert`(*idx*, *system*)

S.insert(index, value) – insert value before index

callback

Provides the CallBack interface to collect data in time (see *callback_functions.py*).

connect

Provides the Connections interface to connect entities (rods, rigid bodies) using Joints (see *joints.py*).

constraints

Provides the Constraints interface to enforce boundary conditions (see *boundary_conditions.py*).

forcing

Add forces and torques to rod (external point force, b-spline torques etc).

1.3.7 Utility Functions

Rotation interface functions

`elastica.transformations.`**`inv_skew_symmetrize`**(*matrix_collection*)

Safe wrapper around inv_skew_symmetrize that does checking and formatting on type of matrix_collection using format_matrix_shape function.

Handy utilities

`elastica.utils.`**`isqrt`**

Efficiently computes sqrt for integer values

Dropin replacement for python3.8's isqrt function Credits : https://stackoverflow.com/a/53983683

Parameters **`num`** (*`int, input`*) –

Returns

- **sqrt_num** (*int, rounded down sqrt of num*)

- *Caveats*
- ——- –
  - Doesn't handle edge-cases of negative numbers by design
  - Doesn't type-check for integers by design, although it is hinted at

Examples

Quadrature and difference kernels

`elastica._calculus.`**`difference_kernel`**`(`*array_collection*`)`

Parameters **`array_collection`** –

Note: Not using numpy.pad, numpy.diff, numpy.hstack for performance reasons with pad : 23.3 µs ± 1.65 µs per loop without pad (previous version, see git history) : 8.3 µs ± 195 ns per loop without pad, hstack (this version) : 5.73 µs ± 216 ns per loop

- Getting the array shape and ndim seems to add ±0.5 µs difference
- Diff also seems to add only ±3.0 µs
- As an added bonus, this works for n-dimensions as long as last dimension

is preserved

`elastica._calculus.`**`quadrature_kernel`**`(`*array_collection*`)`

Simple trapezoidal quadrature rule with zero at end-points, in a dimension agnostic way

Parameters **`array_collection`** –

Note: Not using numpy.pad, numpy.hstack for performance reasons with pad : 23.3 µs ± 1.65 µs per loop without pad (previous version, see git history) : 9.73 µs ± 168 ns per loop without pad and hstack (this version) : 6.52 µs ± 118 ns per loop

- Getting the array shape and manipulating them seems to add ±0.5 µs difference
- As an added bonus, this works for n-dimensions as long as last dimension

is preserved

Convenient linear algebra kernels

`elastica._linalg.`**`levi_civita_tensor`**

param dim:

Rotation kernels

Spline for muscle torques acting on rod

1.3.8 Systems

Analytically integrable systems, used primarily for testing time-steppers

class elastica.systems.analytical.**SecondOrderHybridSystem**(*init_x=5.0*, *init_f=3.0*, *init_v=1.0*, *init_w=1.0*)

Integrate a simple, non-linear ODE: dx/dt = v df/dt = -f * (f is short for frame, for lack of better notation)
dv/dt = -v**2 d/dt = -**2

Dofs: [x, f, v,], with the convention that

_state in this case are [x, v,] linear_states are [f] _kin_state are [x], taken as a slice _dyn_state are [v,], taken as a slice

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e