PyDy Distribution Documentation

Release 0.7.1

PyDy Authors

Mar 04, 2023

CONTENTS

PyDy, short for Python Dynamics, is a tool kit written in the Python programming language that utilizes an array of scientific programs to enable the study of multibody dynamics. The goal is to have a modular framework that can provide the user with their desired workflow, including:

- Model specification
- Equation of motion generation
- Simulation
- Visualization
- Benchmarking
- Publication

We started by building the SymPy mechanics package which provides an API for building models and generating the symbolic equations of motion for complex multibody systems. More recently we developed two packages, *pydy.codegen* and *pydy.viz*, for simulation and visualization of the models, respectively. This Python package contains these two packages and other tools for working with mathematical models generated from SymPy mechanics. The remaining tools currently used in the PyDy workflow are popular scientific Python packages such as NumPy, SciPy, IPython, Jupyter, ipywidgets, pythreejs, and matplotlib which provide additional code for numerical analyses, simulation, and visualization.

If you make use of PyDy in your work or research, please cite us in your publications or on the web. This citation can be used:

> Gilbert Gede, Dale L Peterson, Angadh S Nanjangud, Jason K Moore, and Mont Hubbard, "Constrained Multibody Dynamics With Python: From Symbolic Equation Generation to Publication", ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, 2013, 10.1115/DETC2013-13470.

If you have any question about installation, usage, etc, feel free send a message to our public mailing list.

If you think there's a bug or you would like to request a feature, please open an issue on Github.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation

We recommend the conda package manager and the Anaconda or Miniconda distributions for easy cross platform installation.

Once Anaconda (or Miniconda) is installed type:

```
$ conda install -c conda-forge pydy
```

Also, a simple way to install all of the optional dependencies is to install the `pydy-optional` metapackage using conda:

```
$ conda install -c conda-forge pydy-optional
```

Note that `pydy-optional` currently enforces the use of Jupyter 4.0, so you may not want to install into your root environment. Create a new environment for working with PyDy examples that use the embedded Jupyter visualizations:

```
$ conda create -n pydy -c conda-forge pydy-optional
$ conda activate pydy
(pydy)$ python -c "import pydy; print(pydy.__version__)"
```

1.1.1 Other installation options

If you have the pip package manager installed you can type:

```
$ pip install pydy
```

Installing from source is also supported. The latest stable version of the package can be downloaded from PyPi[1]:

```
$ wget https://pypi.python.org/packages/source/p/pydy/pydy-X.X.X.tar.gz
```

and extracted and installed[2]:

```
$ tar -zxvf pydy-X.X.X.tar.gz
$ cd pydy-X.X.X
$ python setup.py install
```

[1] Change `X.X.X` to the latest version number.

[2] For system wide installs you may need root permissions (perhaps prepend commands with `sudo`).

1.1.2 Dependencies

PyDy has hard dependencies on the following software[3]:

- Python >= 3.7
- setuptools >= 44.0.0
- **NumPy_** >= 1.16.5
- **SciPy_** >= 1.3.3
- **SymPy_** >= 1.5.1
- PyWin32 >= 219 (Windows Only)

PyDy has optional dependencies for extended code generation on:

- Cython >= 0.29.14
- Theano >= 1.0.4

and animated visualizations with `Scene.display_jupyter()` on:

- Jupyter Notebook >= 6.0.0 or *Jupyter Lab* >= 1.0.0
- **ipywidgets_** >= 6.0.0
- **pythreejs_** >= 2.1.1

or interactive animated visualizations with `Scene.display_ipython()` on:

- 4.0.0 <= Jupyter Notebook < 5.0.0
- 4.0.0 <= **ipywidgets_** < 5.0.0

The examples may require these dependencies:

- **matplotlib_** >= 3.1.2
- version_information

1.2 Usage

This is an example of a simple one degree of freedom system: a mass under the influence of a spring, damper, gravity and an external force:

[3] We only test PyDy with these minimum dependencies; these module versions are provided in the Ubuntu 20.04 packages. Previous versions may work.

Derive the system:

```
from sympy import symbols
import sympy.physics.mechanics as me

mass, stiffness, damping, gravity = symbols('m, k, c, g')

position, speed = me.dynamicsymbols('x v')
positiond = me.dynamicsymbols('x', 1)
force = me.dynamicsymbols('F')

ceiling = me.ReferenceFrame('N')

origin = me.Point('origin')
origin.set_vel(ceiling, 0)

center = origin.locatenew('center', position * ceiling.x)
center.set_vel(ceiling, speed * ceiling.x)

block = me.Particle('block', center, mass)

kinematic_equations = [speed - positiond]

force_magnitude = mass * gravity - stiffness * position - damping * speed + force
forces = [(center, force_magnitude * ceiling.x)]

particles = [block]

kane = me.KanesMethod(ceiling, q_ind=[position], u_ind=[speed],
                      kd_eqs=kinematic_equations)
kane.kanes_equations(particles, loads=forces)
```

Create a system to manage integration and specify numerical values for the constants and specified quantities. Here, we specify sinusoidal forcing:

```
from numpy import array, linspace, sin
from pydy.system import System

sys = System(kane,
             constants={mass: 1.0, stiffness: 10.0,
                        damping: 0.4, gravity: 9.8},
             specifieds={force: lambda x, t: sin(t)},
             initial_conditions={position: 0.1, speed: -1.0},
             times=linspace(0.0, 10.0, 1000))
```

Integrate the equations of motion to get the state trajectories:

```
y = sys.integrate()
```

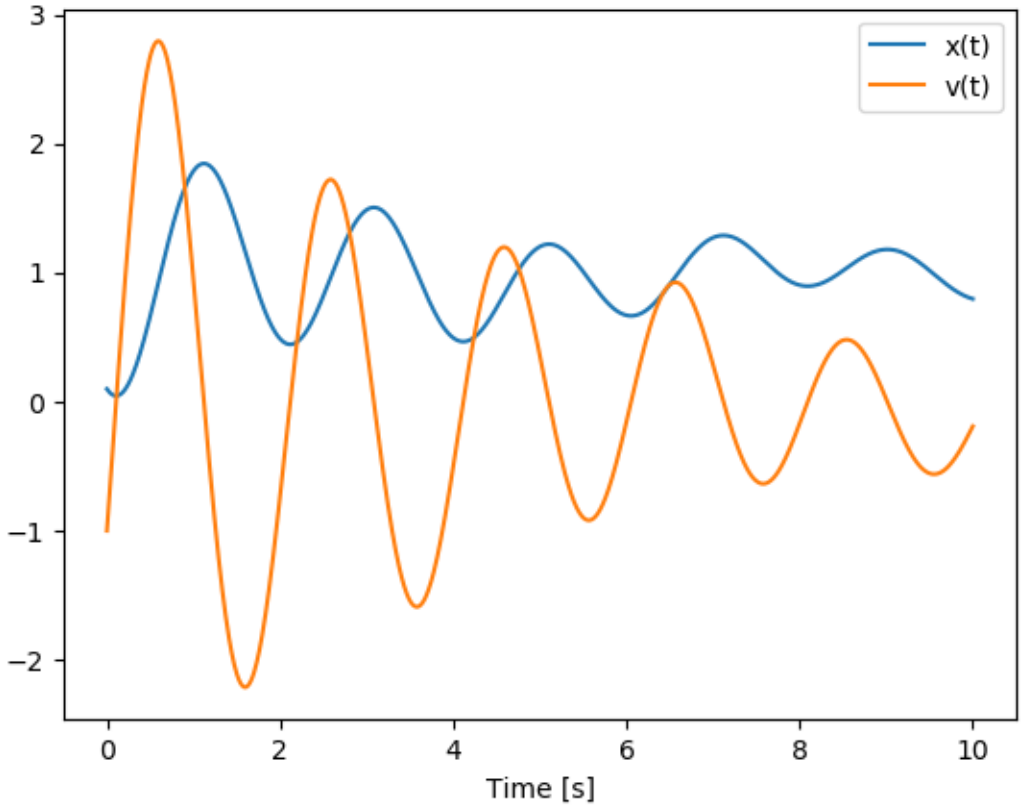
Plot the results:

```
import matplotlib.pyplot as plt

plt.plot(sys.times, y)
```

(continues on next page)

(continued from previous page)

```
plt.legend((str(position), str(speed)))
plt.xlabel('Time [s]')
plt.show()
```



1.3 Tutorials

1.3.1 Beginner

This document lists some beginner's tutorials. These tutorials are aimed at people who are starting to learn how to use PyDy. These tutorials are in the form of IPython notebooks.

Tutorials:

- *Linear Mass-Spring-Damper with Gravity*
- Inverted pendulum model of a standing human

1.3.2 Advanced

This document lists some advanced tutorials. These tutorials require sufficiently good knowledge about mechanics concepts. These tutorials are in the form of IPython notebooks.

Tutorials:

- N Pendulum example

1.4 codegen

1.4.1 Introduction

The `pydy.codegen` package contains various tools to generate numerical code from symbolic descriptions of the equations of motion of systems. It allows you to generate code using a variety of backends depending on your needs. The generated code can also be auto-wrapped for immediate use in a Python session or script. Each component of the code generators and wrappers are accessible so that you can use just the raw code or the wrapper versions.

We currently support three backends:

`lambdify`
: This generates NumPy-aware Python code which is defined in a Python `lambda` function, using the `sympy.utilities.lambdify` module and is the default generator.

`Theano`
: This generates Theano trees that are compiled into low level code, using the `sympy.printers.theano_code` module.

`Cython`
: This generates C code that can be called from Python, using SymPy's C code printer utilities and Cython.

1.4.2 On Windows

For the Cython backend to work on Windows you must install a suitable compiler. See this Cython wiki page for instructions on getting a compiler installed. The easiest solution is to use the Microsoft Visual C++ Compiler for Python.

1.4.3 Example Use

The simplest entry point to the code generation tools is through the *`System`* class.

```
>>> from pydy.models import multi_mass_spring_damper
>>> sys = multi_mass_spring_damper()
>>> type(sys)
<class 'pydy.system.System'>
>>> rhs = sys.generate_ode_function()
>>> help(rhs) # rhs is a function:
Returns the derivatives of the states, i.e. numerically evaluates the right
hand side of the first order differential equation.

x' = f(x, t, p)

Parameters
```

(continues on next page)

(continued from previous page)

```
==========
x : ndarray, shape(2,)
    The state vector is ordered as such:
        - x0(t)
        - v0(t)
t : float
    The current time.
p : dictionary len(3) or ndarray shape(3,)
    Either a dictionary that maps the constants symbols to their numerical
    values or an array with the constants in the following order:
        - m0
        - c0
        - k0

Returns
=======
dx : ndarray, shape(2,)
    The derivative of the state vector.


>>> import numpy as np
>>> rhs(np.array([1.0, 2.0]), 0.0, np.array([1.0, 2.0, 3.0]))
array([ 2., -7.])
```

You can also use the functional interface to the code generation/wrapper classes:

```
>>> from numpy import array
>>> from pydy.models import multi_mass_spring_damper
>>> from pydy.codegen.ode_function_generators import generate_ode_function
>>> sys = multi_mass_spring_damper()
>>> sym_rhs = sys.eom_method.rhs()
>>> q = sys.coordinates
>>> u = sys.speeds
>>> p = sys.constants_symbols
>>> rhs = generate_ode_function(sym_rhs, q, u, p)
>>> rhs(array([1.0, 2.0]), 0.0, array([1.0, 2.0, 3.0]))
array([ 2., -7.])
```

Other backends can be used by passing in the `generator` keyword argument, e.g.:

```
>>> rhs = generate_ode_function(sym_rhs, q, u, p, generator='cython')
>>> rhs(array([1.0, 2.0]), 0.0, array([1.0, 2.0, 3.0]))
array([ 2., -7.])
```

The backends are implemented as subclasses of *ODEFunctionGenerator*. You can make use of the `ODEFunctionGenerator` classes directly:

```
>>> from pydy.codegen.ode_function_generators import LambdifyODEFunctionGenerator
>>> g = LambdifyODEFunctionGenerator(sym_rhs, q, u, p)
>>> rhs = g.generate()
>>> rhs(array([1.0, 2.0]), 0.0, array([1.0, 2.0, 3.0]))
array([ 2., -7.])
```

Furthermore, for direct control over evaluating matrices you can use the `lamdify` and `theano_functions` in SymPy or utilize the *CythonMatrixGenerator* class in PyDy. For example, this shows you how to generate C and Cython code to evaluate matrices:

```
>>> from pydy.codegen.cython_code import CythonMatrixGenerator
>>> sys = multi_mass_spring_damper()
>>> q = sys.coordinates
>>> u = sys.speeds
>>> p = sys.constants_symbols
>>> sym_rhs = sys.eom_method.rhs()
>>> g = CythonMatrixGenerator([q, u, p], [sym_rhs])
>>> setup_py, cython_src, c_header, c_src = g.doprint()
>>> print(setup_py)
#!/usr/bin/env python

from distutils.core import setup
from distutils.extension import Extension

from Cython.Build import cythonize
import numpy

extension = Extension(name="pydy_codegen",
                      sources=["pydy_codegen.pyx",
                               "pydy_codegen_c.c"],
                      include_dirs=[numpy.get_include()])

setup(name="pydy_codegen",
      ext_modules=cythonize([extension]))

>>> print(cython_src)
import numpy as np
cimport numpy as np
cimport cython

cdef extern from "pydy_codegen_c.h":
    void evaluate(
                   double* input_0,
                   double* input_1,
                   double* input_2,
                   double* output_0
                  )

@cython.boundscheck(False)
@cython.wraparound(False)
def eval(
         np.ndarray[np.double_t, ndim=1, mode='c'] input_0,
         np.ndarray[np.double_t, ndim=1, mode='c'] input_1,
         np.ndarray[np.double_t, ndim=1, mode='c'] input_2,
         np.ndarray[np.double_t, ndim=1, mode='c'] output_0
        ):

    evaluate(
             <double*> input_0.data,
```

(continues on next page)

(continued from previous page)

```
             <double*> input_1.data,
             <double*> input_2.data,
             <double*> output_0.data
            )

    return (
            output_0
           )

>>> print(c_src)
#include <math.h>
#include "pydy_codegen_c.h"

void evaluate(
              double input_0[1],
              double input_1[1],
              double input_2[3],
              double output_0[2]
             )
{

    double pydy_0 = input_1[0];

    output_0[0] = pydy_0;
    output_0[1] = (-input_2[1]*pydy_0 - input_2[2]*input_0[0])/input_2[0];

}

>>> print(c_header)
void evaluate(
              double input_0[1],
              double input_1[1],
              double input_2[3],
              double output_0[2]
             );
/*

input_0[1] : [x0(t)]
input_1[1] : [v0(t)]
input_2[3] : [m0, c0, k0]

*/

>>> rhs = g.compile()
>>> res = array([0.0, 0.0])
>>> rhs(array([1.0]), array([2.0]), array([1.0, 2.0, 3.0]), res)
array([ 2., -7.])
```

We also support generating Octave/Matlab code as shown below:

```
>>> from pydy.codegen.octave_code import OctaveMatrixGenerator
>>> sys = multi_mass_spring_damper()
```

(continues on next page)

(continued from previous page)

```
>>> q = sys.coordinates
>>> u = sys.speeds
>>> p = sys.constants_symbols
>>> sym_rhs = sys.eom_method.rhs()
>>> g = OctaveMatrixGenerator([q + u, p], [sym_rhs])
>>> m_src = g.doprint()
>>> print(m_src)
function [output_1] = eval_mats(input_1, input_2)
% function [output_1] = eval_mats(input_1, input_2)
%
% input_1 : [x0(t), v0(t)]
% input_2 : [k0, m0, c0]

    pydy_0 = input_1(2);

    output_1 = [pydy_0; (-input_2(3).*pydy_0 - ...
    input_2(1).*input_1(1))./input_2(2)];

end
```

1.4.4 API

This module contains source code dedicated to generating C code from matrices generated from sympy.physics.mechanics.

class pydy.codegen.c_code.**CMatrixGenerator**(*arguments*, *matrices*, *cse=True*)

This class generates C source files that simultaneously numerically evaluate any number of SymPy matrices.

doprint(*prefix=None*)

Returns a string each for the header and the source files.

Parameters

prefix
[string, optional] A prefix for the name of the header file. This will cause an include statement to to be added to the source.

write(*prefix*, *path=None*)

Writes a header and source file to disk.

Parameters

prefix
[string] Two files will be generated: <prefix>.c and <prefix>.h.

class pydy.codegen.cython_code.**CythonMatrixGenerator**(*arguments*, *matrices*, *prefix='pydy_codegen'*, *cse=True*)

This class generates the Cython code for evaluating a sequence of matrices. It can compile the code and return a Python function.

__init__(*arguments*, *matrices*, *prefix='pydy_codegen'*, *cse=True*)

Parameters

arguments
[sequences of sequences of SymPy Symbol or Function.] Each of the sequences will be

converted to input arrays in the Cython function. All of the symbols/functions contained in `matrices` need to be in the sequences, but the sequences can also contain extra symbols/functions that are not contained in the matrices.

matrices
[sequence of SymPy.Matrix] A sequence of the matrices that should be evaluated in the function. The expressions should contain only sympy.Symbol or sympy.Function that are functions of me.dynamicsymbols._t.

prefix
[string, optional] The desired prefix for the generated files.

cse
[boolean] Find and replace common sub-expressions in `matrices` if True.

`compile`(*tmp_dir=None*, *verbose=False*)

Returns a function which evaluates the matrices.

Parameters

tmp_dir
[string] The path to an existing or non-existing directory where all of the generated files will be stored.

verbose
[boolean] If true the output of the completed compilation steps will be printed.

`doprint`()

Returns the text of the four source files needed to compile Cython wrapper that evaluates the provided SymPy matrices.

Returns

setup_py
[string] The text of the setup.py file used to compile the Cython extension.

cython_source
[string] The text of the Cython pyx file which includes the wrapper function `eval`.

c_header
[string] The text of the C header file that exposes the evaluate function.

c_source
[string] The text of the C source file containing the function that evaluates the matrices.

`write`(*path=None*)

Writes the four source files needed to compile the Cython function to the current working directory.

Parameters

path
[string] The absolute or relative path to an existing directory to place the files instead of the cwd.

`class` pydy.codegen.matrix_generator.`MatrixGenerator`(*arguments*, *matrices*, *cse=True*)

This abstract base class generates source files that simultaneously numerically evaluate any number of SymPy matrices.

`__init__`(*arguments*, *matrices*, *cse=True*)

Parameters

arguments
[sequence of sequences of SymPy Symbol or Function] Each of the sequences will be converted to input arrays in the generated function. All of the symbols/functions contained in `matrices` need to be in the sequences, but the sequences can also contain extra symbols/functions that are not contained in the matrices.

matrices
[sequence of SymPy.Matrix] A sequence of the matrices that should be evaluated in the function. The expressions should contain only sympy.Symbol or sympy.Function that are functions of me.dynamicsymbols._t.

cse
[boolean] Find and replace common sub-expressions in `matrices` if True.

`comma_lists()`

Returns a string output for each of the sequences of SymPy arguments.

`class pydy.codegen.octave_code.OctaveMatrixGenerator(`*arguments*, *matrices*, *cse=True*`)`

This class generates Octave/Matlab source files that simultaneously numerically evaluate any number of SymPy matrices.

`doprint(`*prefix='eval_mats'*`)`

Returns a string that implements the function.

Parameters

prefix
[string, optional] The name of the Octave/Matlab function.

`write(`*prefix='eval_mats'*, *path=None*`)`

Writes the <prefix>.m file to disc at the give path location.

`class pydy.codegen.ode_function_generators.CythonODEFunctionGenerator(`**args*, ***kwargs*`)`

`__init__(`**args*, ***kwargs*`)`

Generates a numerical function which can evaluate the right hand side of the first order ordinary differential equations from a system described by one of the following three symbolic forms:

[1] x' = F(x, t, r, p)

[2] M(x, p) x' = F(x, t, r, p)

[3] M(q, p) u' = F(q, u, t, r, p)
q' = G(q, u, t, r, p)

where

x : states, i.e. [q, u] t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds M : mass matrix (full or minimum) F : right hand side (full or minimum) G : right hand side of the kinematical differential equations

The generated function is of the form F(x, t, p) or F(x, t, r, p) depending on whether the system has specified inputs or not.

Parameters

right_hand_side
[SymPy Matrix, shape(n, 1)] A column vector containing the symbolic expressions for the right hand side of the ordinary differential equations. If the right hand side has been solved for symbolically then only F is required, see form [1]; if not then the mass matrix must also be supplied, see forms [2, 3].

coordinates
[sequence of SymPy Functions] The generalized coordinates. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

speeds
[sequence of SymPy Functions] The generalized speeds. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

constants
[sequence of SymPy Symbols, optional] All of the constants present in the equations of motion. The order does not matter.

mass_matrix
[sympy.Matrix, shape(n, n), optional] This can be either the "full" mass matrix as in [2] or the "minimal" mass matrix as in [3]. The rows and columns must be ordered to match the order of the coordinates and speeds. In the case of the full mass matrix, the speeds should always be ordered before the speeds, i.e. x = [q, u].

coordinate_derivatives
[sympy.Matrix, shape(m, 1), optional] If the "minimal" mass matrix, form [3], is supplied, then this column vector represents the right hand side of the kinematical differential equations.

specifieds
[sequence of SymPy Functions] The specified exogenous inputs to the system. These should be functions of time and the order does not matter.

linear_sys_solver
[string or function] Specify either *numpy* or *scipy* to use the linear solvers provided in each package or supply a function that solves a linear system Ax=b with the call signature x = solve(A, b). For example, if you need to use custom kwargs for the SciPy solver, pass in a lambda function that wraps the solver and sets them.

constants_arg_type
[string] The generated function accepts two different types of arguments for the numerical values of the constants: either a ndarray of the constants values in the correct order or a dictionary mapping the constants symbols to the numerical values. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you need to support choose either `array` or `dictionary`. Note that `array` is faster than `dictionary`.

specifieds_arg_type
[string] The generated function accepts three different types of arguments for the numerical values of the specifieds: either a ndarray of the specifieds values in the correct order, a function that generates the correctly ordered ndarray, or a dictionary mapping the specifieds symbols or tuples of thereof to floats, ndarrays, or functions. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you want to support choose either `array`, `function`, or `dictionary`. The speed of each, from fast to slow, are `array`, `function`, `dictionary`, None.

class pydy.codegen.ode_function_generators.**LambdifyODEFunctionGenerator**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)

Generates a numerical function which can evaluate the right hand side of the first order ordinary differential equations from a system described by one of the following three symbolic forms:

[1] x' = F(x, t, r, p)

[2] M(x, p) x' = F(x, t, r, p)

[3] M(q, p) u' = F(q, u, t, r, p)
q' = G(q, u, t, r, p)

where

x : states, i.e. [q, u] t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds M : mass matrix (full or minimum) F : right hand side (full or minimum) G : right hand side of the kinematical differential equations

The generated function is of the form F(x, t, p) or F(x, t, r, p) depending on whether the system has specified inputs or not.

Parameters

right_hand_side
[SymPy Matrix, shape(n, 1)] A column vector containing the symbolic expressions for the right hand side of the ordinary differential equations. If the right hand side has been solved for symbolically then only F is required, see form [1]; if not then the mass matrix must also be supplied, see forms [2, 3].

coordinates
[sequence of SymPy Functions] The generalized coordinates. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

speeds
[sequence of SymPy Functions] The generalized speeds. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

constants
[sequence of SymPy Symbols, optional] All of the constants present in the equations of motion. The order does not matter.

mass_matrix
[sympy.Matrix, shape(n, n), optional] This can be either the "full" mass matrix as in [2] or the "minimal" mass matrix as in [3]. The rows and columns must be ordered to match the order of the coordinates and speeds. In the case of the full mass matrix, the speeds should always be ordered before the speeds, i.e. x = [q, u].

coordinate_derivatives
[sympy.Matrix, shape(m, 1), optional] If the "minimal" mass matrix, form [3], is supplied, then this column vector represents the right hand side of the kinematical differential equations.

specifieds
[sequence of SymPy Functions] The specified exogenous inputs to the system. These should be functions of time and the order does not matter.

linear_sys_solver
[string or function] Specify either *numpy* or *scipy* to use the linear solvers provided in each package or supply a function that solves a linear system Ax=b with the call signature x = solve(A, b). For example, if you need to use custom kwargs for the SciPy solver, pass in a lambda function that wraps the solver and sets them.

constants_arg_type
[string] The generated function accepts two different types of arguments for the numerical values of the constants: either a ndarray of the constants values in the correct order or a dictionary mapping the constants symbols to the numerical values. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you need to support choose either `array` or `dictionary`. Note that `array` is faster than `dictionary`.

specifieds_arg_type
[string] The generated function accepts three different types of arguments for the numerical values of the specifieds: either a ndarray of the specifieds values in the correct order, a function that generates the correctly ordered ndarray, or a dictionary mapping the specifieds symbols or tuples of thereof to floats, ndarrays, or functions. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you want to support choose either `array`, `function`, or `dictionary`. The speed of each, from fast to slow, are `array`, `function`, `dictionary`, None.

class pydy.codegen.ode_function_generators.**ODEFunctionGenerator**(*right_hand_side*, *coordinates*, *speeds*, *constants=()*, *mass_matrix=None*, *coordinate_derivatives=None*, *specifieds=None*, *linear_sys_solver='numpy'*, *constants_arg_type=None*, *specifieds_arg_type=None*)

This is an abstract base class for all of the generators. A subclass is expected to implement the methods necessary to evaluate the arrays needed to compute xdot for the three different system specification types.

__init__(*right_hand_side*, *coordinates*, *speeds*, *constants=()*, *mass_matrix=None*, *coordinate_derivatives=None*, *specifieds=None*, *linear_sys_solver='numpy'*, *constants_arg_type=None*, *specifieds_arg_type=None*)

Generates a numerical function which can evaluate the right hand side of the first order ordinary differential equations from a system described by one of the following three symbolic forms:

[1] x' = F(x, t, r, p)

[2] M(x, p) x' = F(x, t, r, p)

[3] M(q, p) u' = F(q, u, t, r, p)
q' = G(q, u, t, r, p)

where

x : states, i.e. [q, u] t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds M : mass matrix (full or minimum) F : right hand side (full or minimum) G : right hand side of the kinematical differential equations

The generated function is of the form F(x, t, p) or F(x, t, r, p) depending on whether the system has specified inputs or not.

Parameters

right_hand_side
[SymPy Matrix, shape(n, 1)] A column vector containing the symbolic expressions for the right hand side of the ordinary differential equations. If the right hand side has been solved for symbolically then only F is required, see form [1]; if not then the mass matrix must also be supplied, see forms [2, 3].

coordinates
[sequence of SymPy Functions] The generalized coordinates. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

speeds
[sequence of SymPy Functions] The generalized speeds. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

constants
[sequence of SymPy Symbols, optional] All of the constants present in the equations of motion. The order does not matter.

mass_matrix
[sympy.Matrix, shape(n, n), optional] This can be either the "full" mass matrix as in [2] or the "minimal" mass matrix as in [3]. The rows and columns must be ordered to match the order of the coordinates and speeds. In the case of the full mass matrix, the speeds should always be ordered before the speeds, i.e. x = [q, u].

coordinate_derivatives
[sympy.Matrix, shape(m, 1), optional] If the "minimal" mass matrix, form [3], is supplied, then this column vector represents the right hand side of the kinematical differential equations.

specifieds
[sequence of SymPy Functions] The specified exogenous inputs to the system. These should be functions of time and the order does not matter.

linear_sys_solver
[string or function] Specify either *numpy* or *scipy* to use the linear solvers provided in each package or supply a function that solves a linear system Ax=b with the call signature x = solve(A, b). For example, if you need to use custom kwargs for the SciPy solver, pass in a lambda function that wraps the solver and sets them.

constants_arg_type
[string] The generated function accepts two different types of arguments for the numerical values of the constants: either a ndarray of the constants values in the correct order or a dictionary mapping the constants symbols to the numerical values. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you need to support choose either `array` or `dictionary`. Note that `array` is faster than `dictionary`.

specifieds_arg_type
[string] The generated function accepts three different types of arguments for the numerical values of the specifieds: either a ndarray of the specifieds values in the correct order, a function that generates the correctly ordered ndarray, or a dictionary mapping the specifieds symbols or tuples of thereof to floats, ndarrays, or functions. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you want to support choose either `array`, `function`, or `dictionary`. The speed of each, from fast to slow, are `array`, `function`, `dictionary`, None.

`define_inputs()`

Sets self.inputs to the list of sequences [q, u, p] or [q, u, r, p].

`generate()`

Returns a function that evaluates the right hand side of the first order ordinary differential equations in one of two forms:

x' = f(x, t, p)

or

x' = f(x, t, r, p)

See the docstring of the generated function for more details.

static list_syms(*indent*, *syms*)

Returns a string representation of a valid rst list of the symbols in the sequence syms and indents the list given the integer number of indentations.

class pydy.codegen.ode_function_generators.**TheanoODEFunctionGenerator**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)

Generates a numerical function which can evaluate the right hand side of the first order ordinary differential equations from a system described by one of the following three symbolic forms:

[1] x' = F(x, t, r, p)

[2] M(x, p) x' = F(x, t, r, p)

[3] M(q, p) u' = F(q, u, t, r, p)
q' = G(q, u, t, r, p)

where

x : states, i.e. [q, u] t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds M : mass matrix (full or minimum) F : right hand side (full or minimum) G : right hand side of the kinematical differential equations

The generated function is of the form F(x, t, p) or F(x, t, r, p) depending on whether the system has specified inputs or not.

Parameters

right_hand_side
[SymPy Matrix, shape(n, 1)] A column vector containing the symbolic expressions for the right hand side of the ordinary differential equations. If the right hand side has been solved for symbolically then only F is required, see form [1]; if not then the mass matrix must also be supplied, see forms [2, 3].

coordinates
[sequence of SymPy Functions] The generalized coordinates. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

speeds
[sequence of SymPy Functions] The generalized speeds. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

constants
[sequence of SymPy Symbols, optional] All of the constants present in the equations of motion. The order does not matter.

mass_matrix
[sympy.Matrix, shape(n, n), optional] This can be either the "full" mass matrix as in [2] or the "minimal" mass matrix as in [3]. The rows and columns must be ordered to match the order of the coordinates and speeds. In the case of the full mass matrix, the speeds should always be ordered before the speeds, i.e. x = [q, u].

coordinate_derivatives
[sympy.Matrix, shape(m, 1), optional] If the "minimal" mass matrix, form [3], is supplied, then this column vector represents the right hand side of the kinematical differential equations.

specifieds
[sequence of SymPy Functions] The specified exogenous inputs to the system. These should be functions of time and the order does not matter.

linear_sys_solver
[string or function] Specify either *numpy* or *scipy* to use the linear solvers provided in each package or supply a function that solves a linear system Ax=b with the call signature x = solve(A, b). For example, if you need to use custom kwargs for the SciPy solver, pass in a lambda function that wraps the solver and sets them.

constants_arg_type
[string] The generated function accepts two different types of arguments for the numerical values of the constants: either a ndarray of the constants values in the correct order or a dictionary mapping the constants symbols to the numerical values. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you need to support choose either `array` or `dictionary`. Note that `array` is faster than `dictionary`.

specifieds_arg_type
[string] The generated function accepts three different types of arguments for the numerical values of the specifieds: either a ndarray of the specifieds values in the correct order, a function that generates the correctly ordered ndarray, or a dictionary mapping the specifieds symbols or tuples of thereof to floats, ndarrays, or functions. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you want to support choose either `array`, `function`, or `dictionary`. The speed of each, from fast to slow, are `array`, `function`, `dictionary`, None.

`define_inputs()`

Sets self.inputs to the list of sequences [q, u, p] or [q, u, r, p].

`pydy.codegen.ode_function_generators.generate_ode_function(*args, **kwargs)`

Generates a numerical function which can evaluate the right hand side of the first order ordinary differential equations from a system described by one of the following three symbolic forms:

[1] x' = F(x, t, r, p)

[2] M(x, p) x' = F(x, t, r, p)

[3] M(q, p) u' = F(q, u, t, r, p)
q' = G(q, u, t, r, p)

where

x : states, i.e. [q, u] t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds M : mass matrix (full or minimum) F : right hand side (full or minimum) G : right hand side of the kinematical differential equations

The generated function is of the form F(x, t, p) or F(x, t, r, p) depending on whether the system has specified inputs or not.

Parameters

right_hand_side
[SymPy Matrix, shape(n, 1)] A column vector containing the symbolic expressions for the right hand side of the ordinary differential equations. If the right hand side has been solved for symbolically then only F is required, see form [1]; if not then the mass matrix must also be supplied, see forms [2, 3].

coordinates
[sequence of SymPy Functions] The generalized coordinates. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

speeds
[sequence of SymPy Functions] The generalized speeds. These must be ordered in the same order as the rows in M, F, and/or G and be functions of time.

constants
[sequence of SymPy Symbols, optional] All of the constants present in the equations of motion. The order does not matter.

mass_matrix
[sympy.Matrix, shape(n, n), optional] This can be either the "full" mass matrix as in [2] or the "minimal" mass matrix as in [3]. The rows and columns must be ordered to match the order of the coordinates and speeds. In the case of the full mass matrix, the speeds should always be ordered before the speeds, i.e. x = [q, u].

coordinate_derivatives
[sympy.Matrix, shape(m, 1), optional] If the "minimal" mass matrix, form [3], is supplied, then this column vector represents the right hand side of the kinematical differential equations.

specifieds
[sequence of SymPy Functions] The specified exogenous inputs to the system. These should be functions of time and the order does not matter.

linear_sys_solver
[string or function] Specify either *numpy* or *scipy* to use the linear solvers provided in each package or supply a function that solves a linear system Ax=b with the call signature x = solve(A, b). For example, if you need to use custom kwargs for the SciPy solver, pass in a lambda function that wraps the solver and sets them.

constants_arg_type
[string] The generated function accepts two different types of arguments for the numerical values of the constants: either a ndarray of the constants values in the correct order or a dictionary mapping the constants symbols to the numerical values. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you need to support choose either `array` or `dictionary`. Note that `array` is faster than `dictionary`.

specifieds_arg_type
[string] The generated function accepts three different types of arguments for the numerical values of the specifieds: either a ndarray of the specifieds values in the correct order, a function that generates the correctly ordered ndarray, or a dictionary mapping the specifieds symbols or tuples of thereof to floats, ndarrays, or functions. If None, this is determined inside of the generated function and can cause a significant slow down for performance critical code. If you know apriori what arg types you want to support choose either `array`, `function`, or `dictionary`. The speed of each, from fast to slow, are `array`, `function`, `dictionary`, None.

generator : string or and ODEFunctionGenerator, optional

The method used for generating the numeric right hand side. The string options are {'lambdify'|'theano'|'cython'} with 'lambdify' being the default. You can also pass in a custom subclass of ODEFunctionGenerator.

Returns

rhs
[function] A function which evaluates the derivaties of the states. See the function's docstring for more details after generation.

1.5 models

1.5.1 Introduction

The *pydy.models* file provides canned symbolic models of classical dynamic systems that are mostly for testing and example purposes. There are currently two models:

multi_mass_spring_damper()
: A one dimensional series of masses connected by linear dampers and springs that can optionally be under the influence of gravity and an arbitrary force.

n_link_pendulum_on_cart()
: This is an extension to the classic two dimensional inverted pendulum on a cart to multiple links. You can optionally apply an arbitrary lateral force to the cart and/or apply arbitrary torques between each link.

1.5.2 Example Use

A simple one degree of freedom mass spring damper system can be created with:

```
>>> from pydy.models import multi_mass_spring_damper
>>> sys = multi_mass_spring_damper()
>>> sys.constants_symbols
{m0, c0, k0}
>>> sys.coordinates
[x0(t)]
>>> sys.speeds
[v0(t)]
>>> sys.eom_method.rhs()
Matrix([
[                    v0(t)],
[(-c0*v0(t) - k0*x0(t))/m0]])
```

A two degree of freedom mass spring damper system under the influence of gravity and two external forces can be created with:

```
>>> sys = multi_mass_spring_damper(2, True, True)
>>> sys.constants_symbols
{c1, m1, k0, c0, k1, m0, g}
>>> sys.coordinates
[x0(t), x1(t)]
>>> sys.speeds
[v0(t), v1(t)]
>>> sys.specifieds_symbols
{f0(t), f1(t)}
>>> from sympy import simplify
>>> sm.simplify(sys.eom_method.rhs())
Matrix([
[                                                                              ␣
↪                     v0(t)],
[                                                                              ␣
↪                     v1(t)],
[                                                 (-c0*v0(t) + c1*v1(t) + g*m0 -␣
↪k0*x0(t) + k1*x1(t) + f0(t))/m0],
```

(continues on next page)

(continued from previous page)

```
[-(m1*(-c0*v0(t) + g*m0 + g*m1 - k0*x0(t) + f0(t) + f1(t)) + (m0 + m1)*(c1*v1(t) - g*m1␣
↪+ k1*x1(t) - f1(t)))/(m0*m1)]])
```

1.5.3 API

This module contains some sample symbolic models used for testing and examples.

pydy.models.**multi_mass_spring_damper**(*n=1*, *apply_gravity=False*, *apply_external_forces=False*)

Returns a system containing the symbolic equations of motion and associated variables for a simple mutli-degree of freedom point mass, spring, damper system with optional gravitational and external specified forces. For example, a two mass system under the influence of gravity and external forces looks like:

Parameters

n
[integer] The number of masses in the serial chain.

apply_gravity
[boolean] If true, gravity will be applied to each mass.

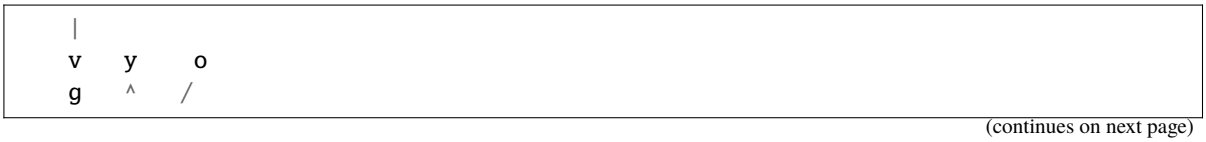
apply_external_forces
[boolean] If true, a time varying external force will be applied to each mass.

Returns

system
[pydy.system.System] A system constructed from the KanesMethod object.

pydy.models.**n_link_pendulum_on_cart**(*n=1*, *cart_force=True*, *joint_torques=False*)

Returns the system containing the symbolic first order equations of motion for a 2D n-link pendulum on a sliding cart under the influence of gravity.



(continues on next page)

(continued from previous page)

```
        |  /
    ----|-/--  theta0
    |   |/  |
F-> |   o --|---> x
    |       |
    ---------
     o     o
```

Parameters

n
[integer] The number of links in the pendulum.

cart_force
[boolean, default=True] If true an external specified lateral force is applied to the cart.

joint_torques
[boolean, default=False] If true joint torques will be added as specified inputs at each joint.

Returns

system
[pydy.system.System] The system containing the symbolics.

Notes

The degrees of freedom of the system are n + 1, i.e. one for each pendulum link and one for the lateral motion of the cart.

M x' = F, where x = [u0, ..., un+1, q0, ..., qn+1]

The joint angles are all defined relative to the ground where the x axis defines the ground line and the y axis points up. The joint torques are applied between each adjacent link and the between the cart and the lower link where a positive torque corresponds to positive angle.

1.6 system

The System class manages the simulation (integration) of a system whose equations are given by KanesMethod.

Many of the attributes are also properties, and can be directly modified.

Here is the procedure for using this class.

1. specify your options either via the constructor or via the attributes.
2. optionally, call `generate_ode_function()` if you want to customize how the ODE function is generated.
3. call `integrate()` to simulate your system.

The simplest usage of this class is as follows. First, we need a KanesMethod object on which we have already invoked `kanes_equations()`:

```
km = KanesMethod(...)
km.kanes_equations(force_list, body_list)
times = np.linspace(0, 5, 100)
```

(continues on next page)

(continued from previous page)

```
sys = System(km, times=times)
sys.integrate()
```

In this case, we use defaults for the numerical values of the constants, specified quantities, initial conditions, etc. You probably won't like these defaults. You can also specify such values via constructor keyword arguments or via the attributes:

```
sys = System(km,
             initial_conditions={dynamicsymbol('q1'): 0.5},
             times=times)
sys.constants = {symbol('m'): 5.0}
sys.integrate()
```

To double-check the constants, specifieds, states and times in your problem, look at these properties:

```
sys.constants_symbols
sys.specifieds_symbols
sys.states
sys.times
```

In this case, the System generates the numerical ode function for you behind the scenes. If you want to customize how this function is generated, you must call `generate_ode_function` on your own:

```
sys = System(KM)
sys.generate_ode_function(generator='cython')
sys.integrate()
```

class pydy.system.**System**(*eom_method*, *constants=None*, *specifieds=None*, *ode_solver=None*, *initial_conditions=None*, *times=None*)

See the class's attributes for a description of the arguments to this constructor.

The parameters to this constructor are all attributes of the System. Actually, they are properties. With the exception of `eom_method`, these attributes can be modified directly at any future point.

Parameters

eom_method
[sympy.physics.mechanics.KanesMethod] You must have called `KanesMethod.kanes_equations()` *before* constructing this `System`.

constants
[dict, optional (default: all 1.0)] This dictionary maps SymPy Symbol objects to floats.

specifieds
[dict, optional (default: all 0)] This dictionary maps SymPy Functions of time objects, or tuples of them, to floats, NumPy arrays, or functions of the state and time.

ode_solver
[function, optional (default: scipy.integrate.odeint)] This function computes the derivatives of the states.

initial_conditions
[dict, optional (default: all zero)] This dictionary maps SymPy Functions of time objects to floats.

times
[array_like, shape(n,), optional] An array_like object, which contains time values over which

equations are integrated. It has to be supplied before System.integrate() can be called.

__init__(*eom_method*, *constants=None*, *specifieds=None*, *ode_solver=None*, *initial_conditions=None*, *times=None*)

property constants

A dict that provides the numerical values for the constants in the problem (all non-dynamics symbols). Keys are the symbols for the constants, and values are floats. Constants that are not specified in this dict are given a default value of 1.0.

property constants_symbols

A set of the symbolic constants (not functions of time) in the system.

property coordinates

Returns a list of the symbolic functions of time representing the system's generalized coordinates.

property eom_method

This is a sympy.physics.mechanics.KanesMethod. The method used to generate the equations of motion. Read-only.

property evaluate_ode_function

A function generated by `generate_ode_function` that computes the state derivatives:

x' = evaluate_ode_function(x, t, args=(...))

This function is used by the `ode_solver`.

generate_ode_function(***kwargs*)

Calls `pydy.codegen.ode_function_generators.generate_ode_function` with the appropriate arguments, and sets the `evaluate_ode_function` attribute to the resulting function.

Parameters

kwargs
All other kwargs are passed onto `pydy.codegen.ode_function_generators.generate_ode_function()`. Don't specify the `specifieds` keyword argument though; the `System` class takes care of those.

Returns

evaluate_ode_function
[function] A function which evaluates the derivaties of the states.

property initial_conditions

Initial conditions for all states (coordinates and speeds). Keys are the symbols for the coordinates and speeds, and values are floats. Coordinates or speeds that are not specified in this dict are given a default value of 0.0.

integrate(***solver_kwargs*)

Integrates the equations `evaluate_ode_function()` using `ode_solver`.

It is necessary to have first generated an ode function. If you have not done so, we do so automatically by invoking `generate_ode_function()`. However, if you want to customize how this function is generated (e.g., change the generator to cython), you can call `generate_ode_function()` on your own (before calling `integrate()`).

Parameters

****solver_kwargs**
Optional arguments that are passed on to the `ode_solver`.

Returns

x_history
[np.array, shape(num_integrator_time_steps, 2)] The trajectory of states (coordinates and speeds) through the requested time interval. num_integrator_time_steps is either len(times) if len(times) > 2, or is determined by the `ode_solver`.

property ode_solver

A function that performs forward integration. It must have the same signature as scipy.integrate.odeint, which is:

```
x_history = ode_solver(f, x0, t, args=(args,))
```

where f is a function f(x, t, args), x0 are the initial conditions, x_history is the state time history, x is the state, t is the time, and args is a keyword argument takes arguments that are then passed to f. The default solver is odeint.

property specifieds

A dict that provides numerical values for the specified quantities in the problem (all dynamicsymbols that are not defined by the equations of motion). There are two possible formats. (1) is more flexible, but (2) is more efficient (by a factor of 3).

(1) Keys are the symbols for the specified quantities, or a tuple of symbols, and values are the floats, arrays of floats, or functions that generate the values. If a dictionary value is a function, it must have the same signature as `f(x, t)`, the ode right-hand-side function (see the documentation for the `ode_solver` attribute). You needn't provide values for all specified symbols. Those for which you do not give a value will default to 0.0.

(2) There are two keys: 'symbols' and 'values'. The value for 'symbols' is an iterable of *all* the specified quantities in the order that you have provided them in 'values'. Values is an ndarray, whose length is *len(sys.specifieds_symbols)*, or a function of x and t that returns an ndarray (also of length *len(sys.specifieds_symbols)*). NOTE: You must provide values for all specified symbols. In this case, we do *not* provide default values.

NOTE: If you switch formats with the same instance of System, you *must* call *generate_ode_function()* before calling *integrate()* again.

Examples

Here are examples for (1). Keys can be individual symbols, or a tuple of symbols. Length of a value must match the length of the corresponding key. Values can be functions that return iterables:

```
   sys = System(km)
   sys.specifieds = {(a, b, c): np.ones(3), d: lambda x, t: -3 * x[0]}
   sys.specifieds = {(a, b, c): lambda x, t: np.ones(3)}

Here are examples for (2):

   sys.specifieds = {'symbols': (a, b, c, d),
                     'values': np.ones(4)}
   sys.specifieds = {'symbols': (a, b, c, d),
                     'values': lambda x, t: np.ones(4)}
```

property specifieds_symbols

A set of the dynamicsymbols you must specify.

property speeds

Returns a list of the symbolic functions of time representing the system's generalized speeds.

property states

Returns a list of the symbolic functions of time representing the system's states, i.e. generalized coordinates plus the generalized speeds. These are in the same order as used in integration (as passed into evaluate_ode_function) and match the order of the mass matrix and forcing vector.

property times

An array-like object, containing time values over which the equations of motion are integrated, numerically.

The object should be in a format which the integration module to be used can accept.

1.7 viz

1.7.1 Introduction

The viz package in pydy is designed to facilitate browser based animations for PyDy framework.

Typically the plugin is used to generate animations for multibody systems. The systems are defined with sympy.physics.mechanics, solved numerically with the codegen package and scipy, and then visualized with this package. But the required data for the animations can theorectically be generated by other methods and passed into a Scene object.

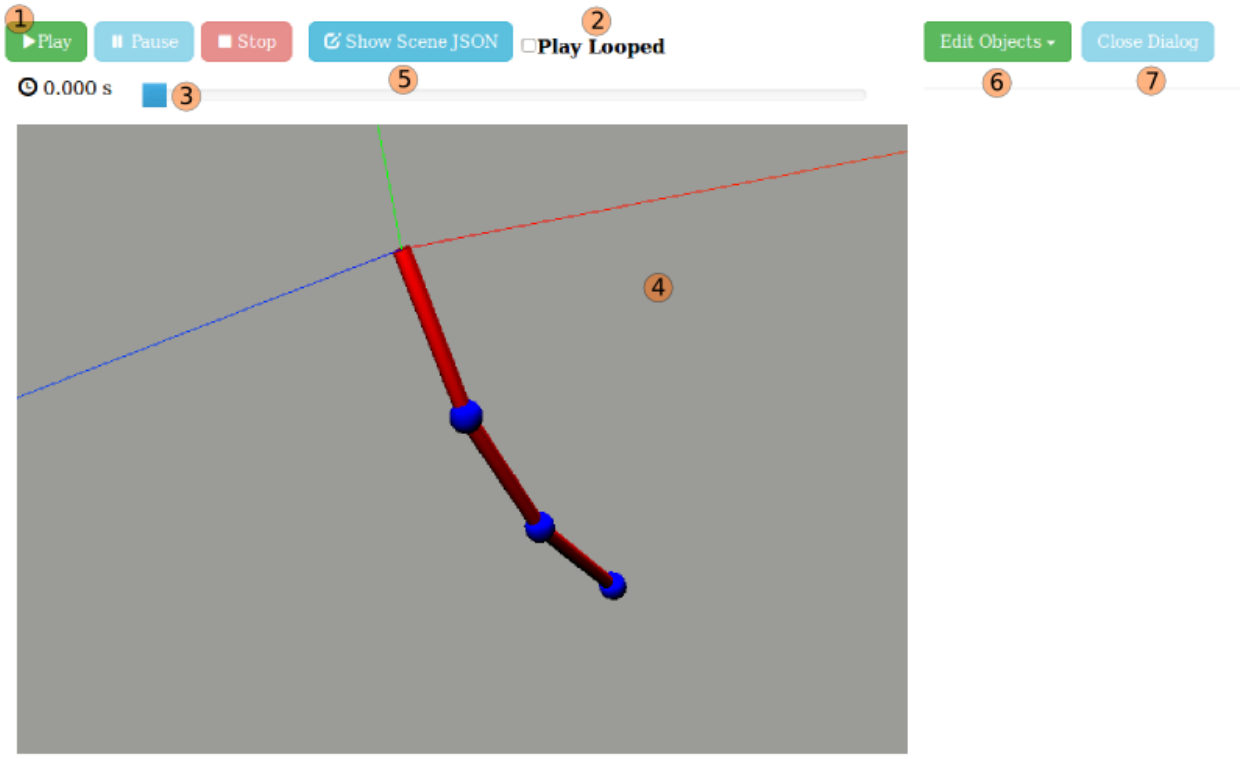
The frontend is based on three.js, a popular interface to the WebGraphics Library (WegGL). The package provides a Python wrapper for some basic functionality for Three.js i.e Geometries, Lights, Cameras etc.

1.7.2 PyDy Visualizer

The PyDy Visualizer is a browser based GUI built to render the visualizations generated by `pydy.viz`. This document provides an overview of PyDy Visualizer. It describes the various features of the visualizer and provides instructions to use it.

The visualizer can be embedded inside an IPython notebook or displayed standalone in the browser. Inside the IPython notebook, it also provides additional functionality to interactively modify the simulation parameters. The EoMs can be re-integrated using a click of a button from GUI, and can be viewed inside the same GUI in real time.

Here is a screenshot of the visualizer, when it is called from outside the notebook, i.e. from the Python interpreter:



GUI Elements

(1) Play, Pause, and Stop Buttons
Allows you to start, pause, and stop the animation.

(2) Play Looped
When checked the animation is run in a loop.

(3) Time Slider
This is used to traverse to the particular frame in animation, by sliding the slider forward and backward. When the animation is running it will continue from the point where the slider is slid to.

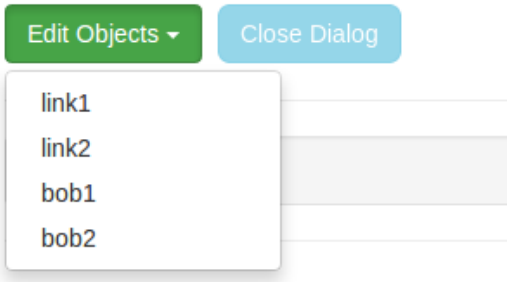
(4) Canvas
Where the animation is rendered. It supports mouse controls:

- Mouse wheel to zoom in, zoom out.
- Click and drag to rotate camera.

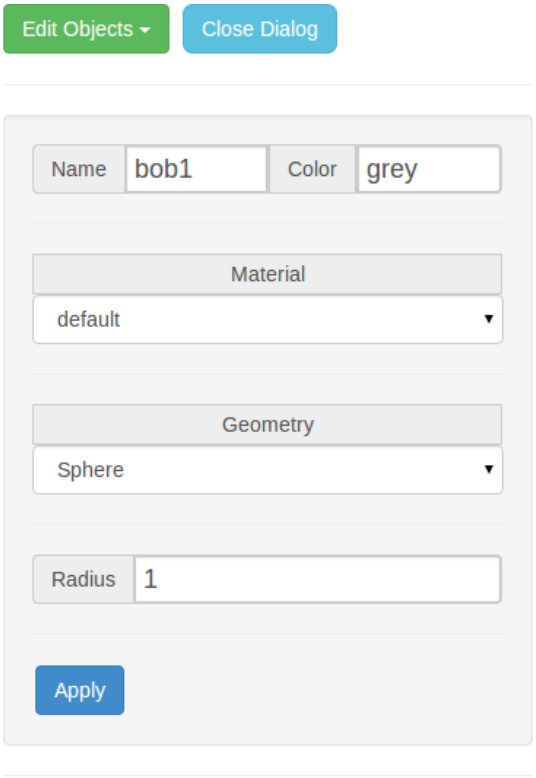
(5) Show Model
Shows the current JSON which is being rendered in visualizer. It can be copied from the text-box, as well as downloaded. On clicking "Show Model", following dialog is created:

(6) Edit Objects

On clicking this button, a dropdown opens up, showing the list of shapes which are rendered in the animation:



On clicking any object from the dropdown, a dialog box opens up, containing the existing info on that object. The info can be edited. After editing click the "Apply" button for the changes to be reflected in the canvas (4).



(7) Close Dialog

Closes/hides the "edit objects" dialog.

Additional options in IPython notebooks:

In IPython notebooks, apart from the features mentioned above, there is an additional feature to edit simulation parameters, from the GUI itself. This is how the Visualizer looks, when called from inside an IPython notebook:

Here, one can add custom values in text-boxes(1, 2, 3 etc.) and on clicking "Rerun" (4) the simulations are re-run in the background. On completing, the scene corresponding to the new data is rendered on the Canvas.

1.7.3 API

Python

class pydy.viz.camera.**OrthoGraphicCamera**(**args*, ***kwargs*)

Creates a orthographic camera for use in a scene. The camera is inherited from `VisualizationFrame`, and thus behaves similarly. It can be attached to dynamics objects, hence we can get a moving camera. All the transformation matrix generation methods are applicable to a `OrthoGraphicCameraCamera`.

__init__(**args*, ***kwargs*)

Initialises a OrthoGraphicCameraCamera object. To initialize a OrthoGraphicCameraCamera, one needs to supply a name (optional), a reference frame, a point, field of view (fov) (optional), near plane distance (optional) and far plane distance (optional).

Like `VisualizationFrame`, it can also be initialized using one of these three argument sequences:

Rigidbody
`OrthoGraphicCameraCamera(rigid_body)`

ReferenceFrame, Point
`OrthoGraphicCameraCamera(ref_frame, point)`

ReferenceFrame, Particle
`OrthoGraphicCameraCamera(ref_frame, particle)`

Note that you can also supply and optional name as the first positional argument, e.g.:

```
OrthoGraphicCameraCamera('camera_name', rigid_body)
```

Additional optional keyword arguments are below:

Parameters

near
[float] The distance of near plane of the OrthoGraphicCameraCamera. All objects closer to this distance are not displayed.

far
[int or float] The distance of far plane of the OrthoGraphicCameraCamera. All objects farther than this distance are not displayed.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import (ReferenceFrame, Point,
...                                      RigidBody, Particle,
...                                      inertia)
>>> from pydy.viz import OrthoGraphicCameraCamera
>>> I = ReferenceFrame('I')
>>> O = Point('O')
```

```
>>> # initializing with reference frame, point
>>> camera1 = OrthoGraphicCameraCamera('frame1', I, O)
```

```
>>> # Initializing with a RigidBody
>>> Ixx, Iyy, Izz, mass = symbols('Ixx Iyy Izz mass')
>>> i = inertia(I, Ixx, Iyy, Izz)
>>> rbody = RigidBody('rbody', O, I, mass, (inertia, O))
>>> camera2 = OrthoGraphicCameraCamera('frame2', rbody)
```

```
>>> # initializing with Particle, reference_frame
>>> Pa = Particle('Pa', O, mass)
>>> camera3 = OrthoGraphicCameraCamera('frame3', I, Pa)
```

generate_scene_dict(***kwargs*)

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains camera parameters followed by an init_orientation Key.

Returns

scene_dict
[dictionary] A dict with following Keys:

1. name: name for the camera
2. near: near value of the camera
3. far: far value of the camera
4. init_orientation: Initial orientation of the camera

class pydy.viz.camera.**PerspectiveCamera**(**args*, ***kwargs*)

Creates a perspective camera for use in a scene. The camera is inherited from `VisualizationFrame`, and thus behaves similarly. It can be attached to dynamics objects, hence we can get a moving camera. All the transformation matrix generation methods are applicable to a `PerspectiveCamera`.

__init__(**args*, ***kwargs*)

Initialises a PerspectiveCamera object. To initialize a PerspectiveCamera, one needs to supply a name (optional), a reference frame, a point, field of view (fov) (optional), near plane distance (optional) and far plane distance (optional).

Like `VisualizationFrame`, it can also be initialized using one of these three argument sequences:

Rigidbody `PerspectiveCamera(rigid_body)` ReferenceFrame, Point `PerspectiveCamera(ref_frame, point)` ReferenceFrame, Particle `PerspectiveCamera(ref_frame, particle)`

Note that you can also supply and optional name as the first positional argument, e.g.:

```
``PerspectiveCamera('camera_name', rigid_body)``
```

Additional optional keyword arguments are below:

Parameters

fov
[float, default=45.0] Field Of View, It determines the angle between the top and bottom of the viewable area (in degrees).

near
[float] The distance of near plane of the PerspectiveCamera. All objects closer to this distance are not displayed.

far
[int or float] The distance of far plane of the PerspectiveCamera. All objects farther than this distance are not displayed.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import (ReferenceFrame, Point,
...                                      RigidBody, Particle,
...                                      inertia)
>>> from pydy.viz import PerspectiveCamera
>>> I = ReferenceFrame('I')
>>> O = Point('O')
```

```
>>> # initializing with reference frame, point
>>> camera1 = PerspectiveCamera('frame1', I, O)
```

```
>>> # Initializing with a RigidBody
>>> Ixx, Iyy, Izz, mass = symbols('Ixx Iyy Izz mass')
>>> i = inertia(I, Ixx, Iyy, Izz)
>>> rbody = RigidBody('rbody', O, I, mass, (inertia, O))
>>> camera2 = PerspectiveCamera('frame2', rbody)
```

```
>>> # initializing with Particle, reference_frame
>>> Pa = Particle('Pa', O, mass)
>>> camera3 = PerspectiveCamera('frame3', I, Pa)
```

generate_scene_dict(***kwargs*)

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains camera parameters followed by an init_orientation key.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Returns

A dict with following Keys:

1. **name: name for the camera**
2. **fov: Field of View value of the camera**
3. **near: near value of the camera**
4. **far: far value of the camera**
5. **init_orientation: Initial orientation of the camera**

class pydy.viz.light.**PointLight**(**args*, ***kwargs*)

Creates a PointLight for the visualization The PointLight is inherited from VisualizationFrame,

It can also be attached to dynamics objects, hence we can get a moving Light. All the transformation matrix generation methods are applicable to a PointLight. Like VisualizationFrame, It can also be initialized using: 1)Rigidbody 2)ReferenceFrame, Point 3)ReferenceFrame, Particle Either one of these must be supplied during initialization

Unlike VisualizationFrame, It doesnt require a Shape argument.

__init__(**args*, ***kwargs*)

Initialises a PointLight object. To initialize a point light, we need to supply a name(optional), a reference frame, and a point.

Parameters

name
[str, optional] Name assigned to VisualizationFrame, default is unnamed

reference_frame
[ReferenceFrame] A reference_frame with respect to which all orientations of the shape takes place, during visualizations/animations.

origin
[Point] A point with respect to which all the translations of the shape takes place, during visualizations/animations.

rigidbody
[RigidBody] A rigidbody whose reference frame and mass center are to be assigned as reference_frame and origin of the VisualizationFrame.

particle
[Particle] A particle whose point is assigned as origin of the VisualizationFrame.

Examples

```
>>> from pydy.viz import PointLight
>>> from sympy.physics.mechanics import
 ␣
↪ReferenceFrame, Point, RigidBody,                                Particle,␣
↪inertia
>>> from sympy import symbols
>>> I = ReferenceFrame('I')
>>> O = Point('O')
>>> #initializing with reference frame, point
>>> light = PointLight('light', I, O)
>>> Ixx, Iyy, Izz, mass = symbols('Ixx Iyy Izz mass')
>>> i = inertia(I, Ixx, Iyy, Izz)
>>> rbody = RigidBody('rbody', O, I, mass, (inertia, O))
>>> # Initializing with a rigidbody ..
>>> light = PointLight('frame2', rbody)
>>> Pa = Particle('Pa', O, mass)
>>> #initializing with Particle, reference_frame ...
>>> light = PointLight('frame3', I, Pa)
```

property color

Color of Light.

color_in_rgb()

Returns the rgb value of the defined light color.

generate_scene_dict()

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains light parameters followed by an init_orientation Key.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error. Returns ======= A dict with following Keys:

1. name: name for the camera
2. color: Color of the light
3. init_orientation: Initial orientation of the light object

`generate_simulation_dict()`

Generates the simulation information for this Light object. It maps the simulation data information to the scene information via a unique id.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Returns

A dictionary containing list of 4x4 matrices mapped to the unique id as the key.

`class pydy.viz.scene.Scene(`*reference_frame*, *origin*, **visualization_frames*, ***kwargs*`)`

The Scene class generates all of the data required for animating a set of visualization frames.

`__init__(`*reference_frame*, *origin*, **visualization_frames*, ***kwargs*`)`

Initialize a Scene instance.

Parameters

reference_frame
[sympy.physics.mechanics.ReferenceFrame] The base reference frame for the scene. The motion of all of the visualization frames, cameras, and lights will be generated with respect to this reference frame.

origin
[sympy.physics.mechanics.Point] The base point for the scene. The motion of all of the visualization frames, cameras, and lights will be generated with respect to this point.

visualization_frames
[VisualizationFrame] One or more visualization frames which are to be displayed in the scene.

name
[string, optional, default='unnamed'] Name of Scene object.

cameras
[list of Camera instances, optional] The cameras with which to display the object. The first camera is used to display the scene initially. The default is a single PerspectiveCamera tied to the base reference frame and positioned away from the origin along the reference frame's z axis.

lights
[list of Light instances, optional] The lights used in the scene. The default is a single Light tied to the base reference frame and positioned away from the origin along the reference frame's z axis at the same point as the default camera.

system
[System, optional, default=None] A PyDy system class which is initiated such that the `integrate()` method will produce valid state trajectories.

times
[array_like, shape(n,), optional, default=None] Monotoncially increaing float values of time that correspond to the state trajectories.

constants
[dictionary, optional, default=None] A dictionary that maps SymPy symbols to floats. This should contain at least all necessary symbols to evaluate the transformation matrices of the visualization frame, cameras, and lights and to evaluate the Shapes' parameters.

states_symbols
[sequence of functions, len(m), optional, default=None] An ordered sequence of the SymPy functions that represent the states. The order must match the order of the `states_trajectories`.

states_trajectories
[array_like, shape(n, m), optional, default=None] A two dimensional array with numerical values for each state at each point in time during the animation.

Notes

The user is allowed to supply either system or times, constants, states_symbols, and states_trajectories. Providing a System allows for interactively changing the simulation parameters via the Scene GUI in the IPython notebook.

`clear_trajectories`()

Sets the 'system', 'times', 'constants', 'states_symbols', and 'states_trajectories' to None.

`create_static_html`(*overwrite=False*, *silent=False*, *prefix=None*)

Creates a directory named `pydy-visualization` in the current working directory which contains all of the HTML, CSS, Javascript, and json files required to run the vizualization application. To run the application, navigate into the `pydy-visualization` directory and start a webserver from that directory, e.g.:

```
$ python -m SimpleHTTPServer
```

Now, in a WebGL compliant browser, navigate to:

```
http://127.0.0.1:8000
```

to view and interact with the visualization.

This method can also be used to output files for embedding the visualizations in static webpages. Simply copy the contents of static directory in the relevant directory for embedding in a static website.

Parameters

overwrite
[boolean, optional, default=False] If True, the directory named `pydy-visualization` in the current working directory will be overwritten.

silent
[boolean, optional, default=False] If True, no messages will be displayed to STDOUT.

prefix
[string, optional] An optional prefix for the json data files.

`display`()

Displays the scene in the default web browser.

`display_ipython`()

Displays the scene using an IPython widget inside an IPython notebook.

Notes

IPython widgets are only supported by IPython versions >= 3.0.0.

`display_jupyter`(*window_size=(800, 600)*, *axes_arrow_length=None*)

Returns a PyThreeJS Renderer and AnimationAction for displaying and animating the scene inside a Jupyter notebook.

Parameters

window_size
[2-tuple of integers] 2-tuple containing the width and height of the renderer window in pixels.

axes_arrow_length
[float] If a positive value is supplied a red (x), green (y), and blue (z) arrows of the supplied length will be displayed as arrows for the global axes.

Returns

vbox
[widgets.VBox] A vertical box containing the action (pythreejs.AnimationAction) and renderer (pythreejs.Renderer).

`generate_visualization_json_system`(*system*, ***kwargs*)

Creates the visualization JSON files for the provided system.

Parameters

system
[pydy.system.System] A fully developed PyDy system that is prepared for the `.integrate()` method.

fps
[int, optional, default=30] Frames per second at which animation should be displayed. Please not that this should not exceed the hardware limit of the display device to be used. Default is 30fps.

outfile_prefix
[str, optional, default=None] A prefix for the JSON files. The files will be named as *outfile_prefix_scene_desc.json* and *outfile_prefix_simulation_data.json*. If not specified a timestamp shall be used as the prefix.

Notes

The optional keyword arguments are the same as those in the `generate_visualization_json` method.

`property name`

Returns the name of the scene.

`property origin`

Returns the origin point of the scene.

`property reference_frame`

Returns the base reference frame of the scene.

remove_static_html(*force=False*)

Removes the `static` directory from the current working directory.

Parameters

force
[boolean, optional, default=False] If true, no warning is issued before the removal of the directory.

class pydy.viz.server.**Server**(*scene_file*, *directory='static/'*, *port=8000*)

Parameters

port
[integer] Defines the port on which the server will run. If this port is already bind, then it increment 1 until it finds a free port.

scene_file
[name of the scene_file generated for visualization] A Valid PyDy generated scene file in 'directory' parameter.

directory
[absolute path of a directory] Absolute path to the directory which contains scene_file with all other static files.

__init__(*scene_file*, *directory='static/'*, *port=8000*)

class pydy.viz.shapes.**Box**(*width*, *height*, *depth*, ***kwargs*)

Instantiates a box of a given size.

Parameters

width
[float or SymPy expression] Width of the box along the X axis.

height
[float or SymPy expression] Height of the box along the Y axis.

depth
[float or SymPy expression] Depth of the box along the Z axis.

Examples

```
>>> from pydy.viz.shapes import Box
>>> s = Box(10.0, 5.0, 1.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.width
5.0
>>>s.height
1.0
>>>s.depth
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
```

(continues on next page)

(continued from previous page)

```
>>> s.color = 'blue'
>>> s.color
'blue'
```

__init__(*width*, *height*, *depth*, ***kwargs*)

class pydy.viz.shapes.**Circle**(*radius=10.0*, ***kwargs*)

Instantiates a circle with a given radius.

Parameters

radius
[float or SymPy Expression] The radius of the circle.

Examples

```
>>> from pydy.viz.shapes import Circle
>>> s = Circle(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Circle(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

class pydy.viz.shapes.**Cone**(*length*, *radius*, ***kwargs*)

Instantiates a cone with given length and base radius.

Parameters

length
[float or SymPy expression] The length of the cone.

radius
[float or SymPy expression] The base radius of the cone.

Examples

```
>>> from pydy.viz.shapes import Cone
>>> s = Cone(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.length
10.0
>>> s.radius
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
12.0
>>> s.radius = 6.0
>>> s.radius
6.0
>>> a = Cone(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
>>> a.radius
5.0
```

__init__(*length*, *radius*, ***kwargs*)

class pydy.viz.shapes.**Cube**(*length*, ***kwargs*)

Instantiates a cube of a given size.

Parameters

length
[float or SymPy expression] The length of the cube.

Examples

```
>>> from pydy.viz.shapes import Cube
>>> s = Cube(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.length
```

(continues on next page)

(continued from previous page)

```
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
12.0
>>> a = Cube('my-shape2', 'red', length=10)
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
```

__init__(*length*, ***kwargs*)

class pydy.viz.shapes.**Cylinder**(*length*, *radius*, ***kwargs*)

Instantiates a cylinder with given length and radius.

Parameters

length
[float or SymPy expression] Length of the cylinder along its Y axis.

radius
[float or SymPy expression] Radius of the cylinder (of the circular cross section normal to the Y axis).

Examples

```
>>> from pydy.viz.shapes import Cylinder
>>> s = Cylinder(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.length
10.0
>>> s.radius
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
```

(continues on next page)

(continued from previous page)

```
12.0
>>> s.radius = 6.0
>>> s.radius
6.0
>>> a = Cylinder(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
>>> a.radius
5.0
```

__init__(*length*, *radius*, ***kwargs*)

class pydy.viz.shapes.**Icosahedron**(*radius=10.0*, ***kwargs*)

Instantiates an icosahedron inscribed in a sphere of the given radius.

Parameters

radius
[float or a SymPy expression] Radius of the circum-scribing sphere for Icosahedron

Examples

```
>>> from pydy.viz.shapes import Icosahedron
>>> s = Icosahedron(10)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>>#These can be changed later too ..
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12
>>> a = Icosahedron(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

class pydy.viz.shapes.**Octahedron**(*radius=10.0*, ***kwargs*)

Instantiaties an Octahedron inscribed in a circle of the given radius.

Parameters

radius
[float or SymPy expression.] The radius of the circum-scribing sphere around the octahedron.

Examples

```
>>> from pydy.viz.shapes import Octahedron
>>> s = Octahedron(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Octahedron(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

class pydy.viz.shapes.**Plane**(*length=10.0*, *width=5.0*, ***kwargs*)

Instantiates a plane with a given length and width.

Parameters

length
[float or SymPy expression] Length of the plane along the Y axis.

width
[float or SymPy expression] Width of the plane along the X axis.

Examples

```
>>> from pydy.viz.shapes import Plane
>>> s = Plane(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.length
10.0
>>> s.width
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
12.0
>>> s.width = 6.0
>>> s.width
6.0
>>> a = Plane(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
>>> a.width
5.0
```

__init__(*length=10.0*, *width=5.0*, ***kwargs*)

class pydy.viz.shapes.**Sphere**(*radius=10.0*, ***kwargs*)

Instantiates a sphere with a given radius.

Parameters

radius
[float or SymPy expression] The radius of the sphere.

Examples

```
>>> from pydy.viz.shapes import Sphere
>>> s = Sphere(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
```

(continues on next page)

(continued from previous page)

```
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Sphere(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

__init__(*radius=10.0*, ***kwargs*)

class pydy.viz.shapes.**Tetrahedron**(*radius=10.0*, ***kwargs*)

Instantiates a Tetrahedron inscribed in a given radius circle.

Parameters

radius
[float or SymPy expression] The radius of the circum-scribing sphere of around the tetrahedron.

Examples

```
>>> from pydy.viz.shapes import Tetrahedron
>>> s = Tetrahedron(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Tetrahedron(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
```

(continues on next page)

(continued from previous page)

```
'red'
>>> a.radius
10.0
```

class pydy.viz.shapes.**Torus**(*radius*, *tube_radius*, ***kwargs*)

Instantiates a torus with a given radius and section radius.

Parameters

radius
[float or SymPy expression] The radius of the torus.

tube_radius
[float or SymPy expression] The radius of the torus tube.

Examples

```
>>> from pydy.viz.shapes import Torus
>>> s = Torus(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.radius
10.0
>>> s.tube_radius
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> s.tube_radius = 6.0
>>> s.tube_radius
6.0
>>> a = Torus(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
>>> a.tube_radius
5.0
```

__init__(*radius*, *tube_radius*, ***kwargs*)

class pydy.viz.shapes.**TorusKnot**(*radius*, *tube_radius*, ***kwargs*)

Instantiates a torus knot with given radius and section radius.

Parameters

radius
[float or SymPy expression] The radius of the torus knot.

tube_radius
[float or SymPy expression] The radius of the torus knot tube.

Examples

```
>>> from pydy.viz.shapes import TorusKnot
>>> s = TorusKnot(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.radius
10.0
>>> s.tube_radius
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> s.tube_radius = 6.0
>>> s.tube_radius
6.0
>>> a = TorusKnot(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
>>> a.tube_radius
5.0
```

class pydy.viz.shapes.**Tube**(*radius*, *points*, ***kwargs*)

Instantiates a tube that sweeps along a path.

Parameters

radius
[float or SymPy expression] The radius of the tube.

points
[array_like, shape(n, 3)] An array of n (x, y, z) coordinates representing points that the tube's center line should follow.

Examples

```
>>> from pydy.viz.shapes import Tube
>>> points = [[1.0, 2.0, 1.0], [2.0, 1.0, 1.0], [2.0, 3.0, 4.0]]
>>> s = Tube(10.0, points)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.points
[[1.0, 2.0, 1.0], [2.0, 1.0, 1.0], [2.0, 3.0, 4.0]]
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 14.0
>>> s.radius
14.0
>>> s.points = [[2.0, 1.0, 4.0], [1.0, 2.0, 4.0],
...             [2.0, 3.0, 1.0], [1.0, 1.0, 3.0]]
>>> s.points
[[2.0, 1.0, 4.0], [1.0, 2.0, 4.0], [2.0, 3.0, 1.0], [1.0, 1.0, 3.0]]
>>> a = Tube(12.0, points, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
12.0
>>> a.points
[[1.0, 2.0, 1.0], [2.0, 1.0, 1.0], [2.0, 3.0, 4.0]]
```

__init__(*radius*, *points*, ***kwargs*)

class pydy.viz.visualization_frame.**VisualizationFrame**(**args*)

A VisualizationFrame represents an object that you want to visualize. It allows you to easily associate a reference frame and a point with a shape.

A VisualizationFrame can be attached to only one Shape Object. It can be nested, i.e we can add/remove multiple visualization frames to one visualization frame. On adding the parent frame to the Scene object, all the children of the parent visualization frame are also added, and hence can be visualized and animated.

A VisualizationFrame needs to have a ReferenceFrame, and a Point for it to form transformation matrices for visualization and animations.

The ReferenceFrame and Point are required to be provided during initialization. They can be supplied in the form of any one of these:

1)reference_frame, point argument. 2)a RigidBody argument 3)reference_frame, particle argument.

In addition to these arguments, A shape argument is also required.

__init__(**args*)

To initialize a visualization frame a ReferenceFrame, Point, and Shape are required. These ReferenceFrame and Point can be passed provided in three ways:

1) RigidBody: the RigidBody's frame and mass center are used.
2) ReferenceFrame and a Particle: The Particle's Point is used.
3) ReferenceFrame and a Point

Parameters

name
[str, optional] Name assigned to VisualizationFrame, default is unnamed

reference_frame
[ReferenceFrame] A reference_frame with respect to which all orientations of the shape takes place, during visualizations/animations.

origin
[Point] A point with respect to which all the translations of the shape takes place, during visualizations/animations.

rigidbody
[RigidBody] A rigidbody whose reference frame and mass center are to be assigned as reference_frame and origin of the VisualizationFrame.

particle
[Particle] A particle whose point is assigned as origin of the VisualizationFrame.

shape
[Shape] A shape to be attached to the VisualizationFrame

Examples

```
>>> from pydy.viz import VisualizationFrame, Sphere
>>> from sympy.physics.mechanics import
 ↪ReferenceFrame, Point, RigidBody,                                Particle,␣
 ↪inertia
>>> from sympy import symbols
>>> I = ReferenceFrame('I')
>>> O = Point('O')
>>> shape = Sphere(5)
>>> #initializing with reference frame, point
>>> frame1 = VisualizationFrame('frame1', I, O, shape)
>>> Ixx, Iyy, Izz, mass = symbols('Ixx Iyy Izz mass')
>>> i = inertia(I, Ixx, Iyy, Izz)
>>> rbody = RigidBody('rbody', O, I, mass, (inertia, O))
>>> # Initializing with a rigidbody ..
>>> frame2 = VisualizationFrame('frame2', rbody, shape)
>>> Pa = Particle('Pa', O, mass)
>>> #initializing with Particle, reference_frame ...
>>> frame3 = VisualizationFrame('frame3', I, Pa, shape)
```

evaluate_transformation_matrix(*dynamic_values*, *constant_values*)

Returns the numerical transformation matrices for each time step.

Parameters

dynamic_values
[array_like, shape(m,) or shape(n, m)] The m state values for each n time step.

constant_values
[array_like, shape(p,)] The p constant parameter values of the system.

Returns

transform_matrix
[numpy.array, shape(n, 16)] A 4 x 4 transformation matrix for each time step.

`generate_numeric_transform_function`(*dynamic_variables*, *constant_variables*)

Returns a function which can compute the numerical values of the transformation matrix given the numerical dynamic variables (i.e. functions of time or states) and the numerical system constants.

Parameters

dynamic_variables
[list of sympy.Functions(time)] All of the dynamic symbols used in defining the orientation and position of this visualization frame.

constant_variables
[list of sympy.Symbols] All of the constants used in defining the orientation and position of this visualization frame.

Returns

numeric_transform
[list of functions] A list of functions which return the numerical transformation for each element in the transformation matrix.

`generate_scene_dict`(*constant_map={}*)

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains shape information from *Shape.generate_dict()* followed by an init_orientation Key.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Parameters

constant_map
[dictionary] Constant map is required when Shape contains sympy expressions.This dictionary maps sympy expressions/symbols to numerical values(floats)

Returns

A dictionary built with a call to *Shape.generate_dict*.
Additional keys included in the dict are following:

1. **init_orientation: Specifies the initial orientation**
   of the *VisualizationFrame*.
2. **reference_frame_name: Name(str) of the reference_frame**
   attached to this VisualizationFrame.
3. **simulation_id: an arbitrary integer to map scene description**
   with the simulation data.

`generate_simulation_dict`()

Generates the simulation information for this visualization frame. It maps the simulation data information to the scene information via a unique id.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Returns

A dictionary containing list of 4x4 matrices mapped to the unique id as the key.

`generate_transformation_matrix`(*reference_frame*, *point*)

Generates a symbolic transformation matrix, with respect to the provided reference frame and point.

Parameters

reference_frame
[ReferenceFrame] A reference_frame with respect to which transformation matrix is generated.

point
[Point] A point with respect to which transformation matrix is generated.

Returns

A 4 x 4 SymPy matrix, containing symbolic expressions describing the transformation as a function of time.

`property name`

Name of the VisualizationFrame.

`property origin`

Origin of the VisualizationFrame, with respect to which all translational transformations take place.

`property reference_frame`

reference_frame of the VisualizationFrame, with respect to which all rotational/orientational transformations take place.

`property shape`

shape in the VisualizationFrame. A shape attached to the visualization frame. NOTE: Only one shape can be attached to a visualization frame.

1.8 API

All the module specific docs have some test cases, which will prove helpful in understanding the usage of the particular module.

1.8.1 Python Modules Reference

Shapes

Shape

`class` pydy.viz.shapes.`Shape`(*name='unnamed'*, *color='grey'*, *material='default'*)

Instantiates a shape. This is primarily used as a superclass for more specific shapes like Cube, Cylinder, Sphere etc.

Shapes must be associated with a reference frame and a point using the VisualizationFrame class.

Parameters

name
[str, optional] A name assigned to the shape.

color
[str, optional] A color string from list of colors in THREE_COLORKEYWORDS

Examples

```
>>> from pydy.viz.shapes import Shape
>>> s = Shape()
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> a = Shape(name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
```

property color

Returns the color attribute of the shape.

generate_dict(*constant_map={}*)

Returns a dictionary containing all the data associated with the Shape.

Parameters

constant_map
[dictionary] If any of the shape's geometry are defined as SymPy expressions, then this dictionary should map all SymPy Symbol's found in the expressions to floats.

property material

Returns the material attribute of the shape.

property name

Returns the name attribute of the shape.

Cube

class pydy.viz.shapes.**Cube**(*length*, ***kwargs*)

Instantiates a cube of a given size.

Parameters

length
[float or SymPy expression] The length of the cube.

Examples

```
>>> from pydy.viz.shapes import Cube
>>> s = Cube(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.length
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
12.0
>>> a = Cube('my-shape2', 'red', length=10)
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
```

Cylinder

class pydy.viz.shapes.**Cylinder**(*length*, *radius*, ***kwargs*)

Instantiates a cylinder with given length and radius.

Parameters

length
[float or SymPy expression] Length of the cylinder along its Y axis.

radius
[float or SymPy expression] Radius of the cylinder (of the circular cross section normal to the Y axis).

Examples

```
>>> from pydy.viz.shapes import Cylinder
>>> s = Cylinder(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.length
10.0
>>> s.radius
```

(continues on next page)

(continued from previous page)

```
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
12.0
>>> s.radius = 6.0
>>> s.radius
6.0
>>> a = Cylinder(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
>>> a.radius
5.0
```

Cone

class pydy.viz.shapes.**Cone**(*length*, *radius*, ***kwargs*)

Instantiates a cone with given length and base radius.

Parameters

length
[float or SymPy expression] The length of the cone.

radius
[float or SymPy expression] The base radius of the cone.

Examples

```
>>> from pydy.viz.shapes import Cone
>>> s = Cone(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.length
10.0
>>> s.radius
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
```

(continues on next page)

(continued from previous page)

```
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
12.0
>>> s.radius = 6.0
>>> s.radius
6.0
>>> a = Cone(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
>>> a.radius
5.0
```

Sphere

class pydy.viz.shapes.**Sphere**(*radius=10.0*, ***kwargs*)

Instantiates a sphere with a given radius.

Parameters

radius
[float or SymPy expression] The radius of the sphere.

Examples

```
>>> from pydy.viz.shapes import Sphere
>>> s = Sphere(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Sphere(10.0, name='my-shape2', color='red')
>>> a.name
```

(continues on next page)

(continued from previous page)

```
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

Circle

class pydy.viz.shapes.**Circle**(*radius=10.0*, ***kwargs*)

Instantiates a circle with a given radius.

Parameters

radius
[float or SymPy Expression] The radius of the circle.

Examples

```
>>> from pydy.viz.shapes import Circle
>>> s = Circle(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Circle(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

Plane

class pydy.viz.shapes.**Plane**(*length=10.0*, *width=5.0*, ***kwargs*)

Instantiates a plane with a given length and width.

Parameters

length
[float or SymPy expression] Length of the plane along the Y axis.

width
[float or SymPy expression] Width of the plane along the X axis.

Examples

```
>>> from pydy.viz.shapes import Plane
>>> s = Plane(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.length
10.0
>>> s.width
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.length = 12.0
>>> s.length
12.0
>>> s.width = 6.0
>>> s.width
6.0
>>> a = Plane(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.length
10.0
>>> a.width
5.0
```

Tetrahedron

class pydy.viz.shapes.**Tetrahedron**(*radius=10.0*, ***kwargs*)

Instantiates a Tetrahedron inscribed in a given radius circle.

Parameters

radius
[float or SymPy expression] The radius of the circum-scribing sphere of around the tetrahedron.

Examples

```
>>> from pydy.viz.shapes import Tetrahedron
>>> s = Tetrahedron(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Tetrahedron(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

Octahedron

class pydy.viz.shapes.**Tetrahedron**(*radius=10.0*, ***kwargs*)

Instantiates a Tetrahedron inscribed in a given radius circle.

Parameters

radius
[float or SymPy expression] The radius of the circum-scribing sphere of around the tetrahedron.

Examples

```
>>> from pydy.viz.shapes import Tetrahedron
>>> s = Tetrahedron(10.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> a = Tetrahedron(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

Icosahedron

class pydy.viz.shapes.**Icosahedron**(*radius=10.0*, ***kwargs*)

Instantiates an icosahedron inscribed in a sphere of the given radius.

Parameters

radius
[float or a SymPy expression] Radius of the circum-scribing sphere for Icosahedron

Examples

```
>>> from pydy.viz.shapes import Icosahedron
>>> s = Icosahedron(10)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>>s.radius
10.0
>>>#These can be changed later too ..
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
```

(continues on next page)

(continued from previous page)

```
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12
>>> a = Icosahedron(10.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
```

Torus

class pydy.viz.shapes.**Torus**(*radius*, *tube_radius*, ***kwargs*)

Instantiates a torus with a given radius and section radius.

Parameters

radius
[float or SymPy expression] The radius of the torus.

tube_radius
[float or SymPy expression] The radius of the torus tube.

Examples

```
>>> from pydy.viz.shapes import Torus
>>> s = Torus(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.radius
10.0
>>> s.tube_radius
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> s.tube_radius = 6.0
>>> s.tube_radius
6.0
```

(continues on next page)

(continued from previous page)

```
>>> a = Torus(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
10.0
>>> a.tube_radius
5.0
```

TorusKnot

class pydy.viz.shapes.**TorusKnot**(*radius*, *tube_radius*, ***kwargs*)

Instantiates a torus knot with given radius and section radius.

Parameters

radius
[float or SymPy expression] The radius of the torus knot.

tube_radius
[float or SymPy expression] The radius of the torus knot tube.

Examples

```
>>> from pydy.viz.shapes import TorusKnot
>>> s = TorusKnot(10.0, 5.0)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.radius
10.0
>>> s.tube_radius
5.0
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 12.0
>>> s.radius
12.0
>>> s.tube_radius = 6.0
>>> s.tube_radius
6.0
>>> a = TorusKnot(10.0, 5.0, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
```

(continues on next page)

(continued from previous page)

```
'red'
>>> a.radius
10.0
>>> a.tube_radius
5.0
```

Tube

class pydy.viz.shapes.**Tube**(*radius*, *points*, ***kwargs*)

Instantiates a tube that sweeps along a path.

Parameters

radius
[float or SymPy expression] The radius of the tube.

points
[array_like, shape(n, 3)] An array of n (x, y, z) coordinates representing points that the tube's center line should follow.

Examples

```
>>> from pydy.viz.shapes import Tube
>>> points = [[1.0, 2.0, 1.0], [2.0, 1.0, 1.0], [2.0, 3.0, 4.0]]
>>> s = Tube(10.0, points)
>>> s.name
'unnamed'
>>> s.color
'grey'
>>> s.points
[[1.0, 2.0, 1.0], [2.0, 1.0, 1.0], [2.0, 3.0, 4.0]]
>>> s.name = 'my-shape1'
>>> s.name
'my-shape1'
>>> s.color = 'blue'
>>> s.color
'blue'
>>> s.radius = 14.0
>>> s.radius
14.0
>>> s.points = [[2.0, 1.0, 4.0], [1.0, 2.0, 4.0],
...             [2.0, 3.0, 1.0], [1.0, 1.0, 3.0]]
>>> s.points
[[2.0, 1.0, 4.0], [1.0, 2.0, 4.0], [2.0, 3.0, 1.0], [1.0, 1.0, 3.0]]
>>> a = Tube(12.0, points, name='my-shape2', color='red')
>>> a.name
'my-shape2'
>>> a.color
'red'
>>> a.radius
12.0
```

(continues on next page)

(continued from previous page)

```
>>> a.points
[[1.0, 2.0, 1.0], [2.0, 1.0, 1.0], [2.0, 3.0, 4.0]]
```

VisualizationFrame

class pydy.viz.**VisualizationFrame**(**args*)

A VisualizationFrame represents an object that you want to visualize. It allows you to easily associate a reference frame and a point with a shape.

A VisualizationFrame can be attached to only one Shape Object. It can be nested, i.e we can add/remove multiple visualization frames to one visualization frame. On adding the parent frame to the Scene object, all the children of the parent visualization frame are also added, and hence can be visualized and animated.

A VisualizationFrame needs to have a ReferenceFrame, and a Point for it to form transformation matrices for visualization and animations.

The ReferenceFrame and Point are required to be provided during initialization. They can be supplied in the form of any one of these:

1)reference_frame, point argument. 2)a RigidBody argument 3)reference_frame, particle argument.

In addition to these arguments, A shape argument is also required.

evaluate_transformation_matrix(*dynamic_values*, *constant_values*)

Returns the numerical transformation matrices for each time step.

Parameters

dynamic_values
[array_like, shape(m,) or shape(n, m)] The m state values for each n time step.

constant_values
[array_like, shape(p,)] The p constant parameter values of the system.

Returns

transform_matrix
[numpy.array, shape(n, 16)] A 4 x 4 transformation matrix for each time step.

generate_numeric_transform_function(*dynamic_variables*, *constant_variables*)

Returns a function which can compute the numerical values of the transformation matrix given the numerical dynamic variables (i.e. functions of time or states) and the numerical system constants.

Parameters

dynamic_variables
[list of sympy.Functions(time)] All of the dynamic symbols used in defining the orientation and position of this visualization frame.

constant_variables
[list of sympy.Symbols] All of the constants used in defining the orientation and position of this visualization frame.

Returns

numeric_transform
[list of functions] A list of functions which return the numerical transformation for each element in the transformation matrix.

`generate_scene_dict`(*constant_map={}*)

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains shape information from *Shape.generate_dict()* followed by an init_orientation Key.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Parameters

constant_map
[dictionary] Constant map is required when Shape contains sympy expressions.This dictionary maps sympy expressions/symbols to numerical values(floats)

Returns

A dictionary built with a call to *Shape.generate_dict*.
Additional keys included in the dict are following:

1. **init_orientation: Specifies the initial orientation**
   of the *VisualizationFrame*.
2. **reference_frame_name: Name(str) of the reference_frame**
   attached to this VisualizationFrame.
3. **simulation_id: an arbitrary integer to map scene description**
   with the simulation data.

`generate_simulation_dict`()

Generates the simulation information for this visualization frame. It maps the simulation data information to the scene information via a unique id.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Returns

A dictionary containing list of 4x4 matrices mapped to
the unique id as the key.

`generate_transformation_matrix`(*reference_frame*, *point*)

Generates a symbolic transformation matrix, with respect to the provided reference frame and point.

Parameters

reference_frame
[ReferenceFrame] A reference_frame with respect to which transformation matrix is generated.

point
[Point] A point with respect to which transformation matrix is generated.

Returns

A 4 x 4 SymPy matrix, containing symbolic expressions describing the
transformation as a function of time.

`property name`

Name of the VisualizationFrame.

`property origin`

Origin of the VisualizationFrame, with respect to which all translational transformations take place.

property reference_frame

reference_frame of the VisualizationFrame, with respect to which all rotational/orientational transformations take place.

property shape

shape in the VisualizationFrame. A shape attached to the visualization frame. NOTE: Only one shape can be attached to a visualization frame.

Cameras

Perspective Camera

class pydy.viz.camera.**PerspectiveCamera**(**args*, ***kwargs*)

Creates a perspective camera for use in a scene. The camera is inherited from `VisualizationFrame`, and thus behaves similarly. It can be attached to dynamics objects, hence we can get a moving camera. All the transformation matrix generation methods are applicable to a `PerspectiveCamera`.

generate_scene_dict(***kwargs*)

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains camera parameters followed by an init_orientation key.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Returns

A dict with following Keys:

1. **name: name for the camera**
2. **fov: Field of View value of the camera**
3. **near: near value of the camera**
4. **far: far value of the camera**
5. **init_orientation: Initial orientation of the camera**

Orthographic Camera

class pydy.viz.camera.**OrthoGraphicCamera**(**args*, ***kwargs*)

Creates a orthographic camera for use in a scene. The camera is inherited from `VisualizationFrame`, and thus behaves similarly. It can be attached to dynamics objects, hence we can get a moving camera. All the transformation matrix generation methods are applicable to a `OrthoGraphicCameraCamera`.

generate_scene_dict(***kwargs*)

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains camera parameters followed by an init_orientation Key.

Returns

scene_dict
[dictionary] A dict with following Keys:

1. name: name for the camera
2. near: near value of the camera

3. far: far value of the camera
4. init_orientation: Initial orientation of the camera

Lights

PointLight

class pydy.viz.light.**PointLight**(**args*, ***kwargs*)

Creates a PointLight for the visualization The PointLight is inherited from VisualizationFrame,

It can also be attached to dynamics objects, hence we can get a moving Light. All the transformation matrix generation methods are applicable to a PointLight. Like VisualizationFrame, It can also be initialized using: 1)Rigidbody 2)ReferenceFrame, Point 3)ReferenceFrame, Particle Either one of these must be supplied during initialization

Unlike VisualizationFrame, It doesnt require a Shape argument.

property color

Color of Light.

color_in_rgb()

Returns the rgb value of the defined light color.

generate_scene_dict()

This method generates information for a static visualization in the initial conditions, in the form of dictionary. This contains light parameters followed by an init_orientation Key.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error. Returns ======= A dict with following Keys:

1. name: name for the camera
2. color: Color of the light
3. init_orientation: Initial orientation of the light object

generate_simulation_dict()

Generates the simulation information for this Light object. It maps the simulation data information to the scene information via a unique id.

Before calling this method, all the transformation matrix generation methods should be called, or it will give an error.

Returns

A dictionary containing list of 4x4 matrices mapped to the unique id as the key.

Scene

class pydy.viz.scene.**Scene**(*reference_frame*, *origin*, **visualization_frames*, ***kwargs*)

The Scene class generates all of the data required for animating a set of visualization frames.

clear_trajectories()

Sets the 'system', 'times', 'constants', 'states_symbols', and 'states_trajectories' to None.

create_static_html(*overwrite=False*, *silent=False*, *prefix=None*)

Creates a directory named `pydy-visualization` in the current working directory which contains all of the HTML, CSS, Javascript, and json files required to run the vizualization application. To run the application, navigate into the `pydy-visualization` directory and start a webserver from that directory, e.g.:

```
$ python -m SimpleHTTPServer
```

Now, in a WebGL compliant browser, navigate to:

```
http://127.0.0.1:8000
```

to view and interact with the visualization.

This method can also be used to output files for embedding the visualizations in static webpages. Simply copy the contents of static directory in the relevant directory for embedding in a static website.

Parameters

overwrite
[boolean, optional, default=False] If True, the directory named `pydy-visualization` in the current working directory will be overwritten.

silent
[boolean, optional, default=False] If True, no messages will be displayed to STDOUT.

prefix
[string, optional] An optional prefix for the json data files.

display()

Displays the scene in the default web browser.

display_ipython()

Displays the scene using an IPython widget inside an IPython notebook cell.

Notes

IPython widgets are only supported by IPython versions >= 3.0.0.

display_jupyter(*window_size=(800, 600)*, *axes_arrow_length=None*)

Returns a PyThreeJS Renderer and AnimationAction for displaying and animating the scene inside a Jupyter notebook.

Parameters

window_size
[2-tuple of integers] 2-tuple containing the width and height of the renderer window in pixels.

axes_arrow_length
[float] If a positive value is supplied a red (x), green (y), and blue (z) arrows of the supplied length will be displayed as arrows for the global axes.

Returns

vbox
[widgets.VBox] A vertical box containing the action (pythreejs.AnimationAction) and renderer (pythreejs.Renderer).

`generate_visualization_json_system`(*system*, ***kwargs*)

Creates the visualization JSON files for the provided system.

Parameters

system
[pydy.system.System] A fully developed PyDy system that is prepared for the `.integrate()` method.

fps
[int, optional, default=30] Frames per second at which animation should be displayed. Please not that this should not exceed the hardware limit of the display device to be used. Default is 30fps.

outfile_prefix
[str, optional, default=None] A prefix for the JSON files. The files will be named as *outfile_prefix_scene_desc.json* and *outfile_prefix_simulation_data.json*. If not specified a timestamp shall be used as the prefix.

Notes

The optional keyword arguments are the same as those in the `generate_visualization_json` method.

`property name`

Returns the name of the scene.

`property origin`

Returns the origin point of the scene.

`property reference_frame`

Returns the base reference frame of the scene.

`remove_static_html`(*force=False*)

Removes the `static` directory from the current working directory.

Parameters

force
[boolean, optional, default=False] If true, no warning is issued before the removal of the directory.

1.8.2 JavaScript Classes Reference

Note: The Javascript docs are meant for the developers, who are interested in developing the js part of *pydy.viz*. If you simply intend to use the software then *Python Modules Reference* is what you should be looking into.

DynamicsVisualizer

DynamicsVisualizer is the main class for Dynamics Visualizer. It contains methods to set up a default UI, and maps buttons' *onClick* to functions.

_initialize

args: None

Checks whether the browser supports webGLs, and initializes the DynamicVisualizer object.

isWebGLCompatible

args: None

Checks whether the browser used is compatible for handling webGL based animations. Requires external script: Modernizr.js

activateUIControls

args: None

This method adds functions to the UI buttons It should be **strictly** called after the other DynamicsVisualizer submodules are loaded in the browser, else certain functionality will be(not might be!) hindered.

loadUIElements

args: None

This method loads UI elements which can be loaded only **after** scene JSON is loaded onto canvas.

getBasePath

args: None

Returns the base path of the loaded Scene file.

getFileExtenstion

args: None

Returns the extension of the uploaded Scene file.

getQueryString

args: key

Returns the GET Parameter from url corresponding to *key*

DynamicVisualizer.Parser

loadScene

args: None

This method calls an ajax request on the JSON file and reads the scene info from the JSON file, and saves it as an object at self.model.

loadSimulation

args: None

This method loads the simulation data from the simulation JSON file. The data is saved in the form of 4x4 matrices mapped to the simulation object id, at a particular time.

createTimeArray

args: None

Creates a time array from the information inferred from simulation data.

DynamicsVisualizer.Scene

create

args: None

This method creates the scene from the self.model and renders it onto the canvas.

_createRenderer

args: None

Creates a webGL Renderer
with a default background color.

_addDefaultLightsandCameras

args: None

This method adds a default light and a Perspective camera to the initial visualization

_addAxes

args: None

Adds a default system of axes to the initial visualization.

_addTrackBallControls

args: None

Adds Mouse controls to the initial visualization using Three's TrackballControls library.

_resetControls

args: None

Resets the scene camera to the initial values(zoom, displacement etc.)

addObjects

args: None

Adds the geometries loaded from the JSON file onto the scene. The file is saved as an object in self.model and then rendered to canvas with this function.

addCameras

args: None

Adds the cameras loaded from the JSON file onto the scene. The cameras can be switched during animation from the *switch cameras* UI button.

addLights

args: None

Adds the Lights loaded from the JSON file onto the scene.

_addIndividualObject

args: JS object, { object }

Adds a single geometry object which is taken as an argument to this function.

_addIndividualCamera

args: JS object, { object }

Adds a single camera object which is taken as an argument to this function.

_addIndividualLight

args: JS object, { object }

Adds a single light object which is taken as an argument to this function.

runAnimation

args: None

This function iterates over the the simulation data to render them on the canvas.

setAnimationTime

args: time, (float)

Takes a time value as the argument and renders the simulation data corresponding to that time value.

stopAnimation

args: None

Stops the animation, and sets the current time value to initial.

_removeAll

args: None

Removes all the geometry elements added to the scene from the loaded scene JSON file. Keeps the default elements, i.e. default axis, camera and light.

_blink

args: id, (int) Blinks the geometry element. takes the element simulation_id as the argument and blinks it until some event is triggered(UI button press)

DynamicsVisualizer.ParamEditor

openDialog

args: id, (str)

This function takes object's id as the argument, and populates the edit objects dialog box.

applySceneInfo

args: id, (str)

This object applies the changes made in the edit objects dialog box to self.model and then renders the model onto canvas. It takes the id of the object as its argument.

_addGeometryFor

args: JS object,{ object }

Adds geometry info for a particular object onto the edit objects dialog box. Takes the object as the argument.

showModel

args: None

Updates the codemirror instance with the updated model, and shows it in the UI.

1.9 utils

exception pydy.utils.**PyDyDeprecationWarning**

exception pydy.utils.**PyDyFutureWarning**

exception pydy.utils.**PyDyImportWarning**

exception pydy.utils.**PyDyUserWarning**

pydy.utils.**find_dynamicsymbols**(*expression*, *exclude=None*)

Find all dynamicsymbols in expression.

```
>>> from sympy.physics.mechanics import dynamicsymbols, find_dynamicsymbols
>>> x, y = dynamicsymbols('x, y')
>>> expr = x + x.diff()*y
>>> find_dynamicsymbols(expr)
set([x(t), y(t), Derivative(x(t), t)])
```

If the optional `exclude` kwarg is used, only dynamicsymbols not in the iterable `exclude` are returned.

```
>>> find_dynamicsymbols(expr, [x, y])
set([Derivative(x(t), t)])
```

pydy.utils.**sympy_equal_to_or_newer_than**(*version*, *installed_version=None*)

Returns true if the installed version of SymPy is equal to or newer than the provided version string.

pydy.utils.**sympy_newer_than**(*version*)

Returns true if the installed version of SymPy is newer than the provided version string.

pydy.utils.**wrap_and_indent**(*lines*, *indentation=4*, *width=79*, *continuation=None*, *comment=None*)

Returns a single string in which the lines have been indented and wrapped into a block of text.

Parameters

indentation
[integer] The number of characters to indent.

width
[integer] The maximum line width.

continuation
[string] The continuation characters.

comment
[string] The character that designates a comment line.

1.10 Release Notes

1.10.1 0.7.1 (March 4, 2023)

- Reduced sdist size by moving the MANIFEST.in prune command last.

1.10.2 0.7.0 (March 4, 2023)

- Support Python 3.10 and 3.11. [PR #488]
- Fixed the Carvallo-Whipple bicycle model to match Basu-Mandal benchmark numbers. [PR #486]
- Added Box geometry to the javascript GUI [PR #484]
- Updated the three link conical pendulum example to use new kanes_equations() syntax. [PR #481]
- Added example of a 3D multilink pendulum with colliding bobs. [PR #467]
- `LambdifyODEFunctionGenerator` now accepts a `cse=True/False` kwarg and if SymPy >=1.9 is installed, then the underlying generated code by `lambdify` will be simplified. It is `True` by default. [PR #464]
- Visualization supports durations that don't start at 0.

1.10.3 0.6.0 (February 4, 2022)

- Dropped support for Python 2.7 and 3.6. [PR #459]
- Moved chaos pendulum example to Sphinx docs.
- Added Astrobee example [PR #453]
- Added the ability to pass optional arguments to the ODE solver in System. [PR #447]
- Cylinders, Spheres, and Circles loaded via PyThreeJS will appear more round. [PR #440]
- Added a Carvallo-Whipple bicycle example to the documentation [PR #442]
- Oldest supported dependencies for Python 3 are aligned with Ubuntu 20.04 LTS. For Python 2, the oldest necessary dependencies are used if the ones for Ubuntu 20.04 LTS are too new. [PR #432]
- Dropped support for Python 3.5 [PR #429]
- Improved the README and documentation integration. [PR #424]
- Moved some examples to Sphinx [PRs #421, #423]
- jupyter-sphinx enabled for examples in the documentation [PR #419]
- Added an example with no constraints that uses `display_jupyter()` for animation. [PR #418]
- Added an example that has both configuration and motion constraints. [PR #417]
- `display_jupyter()` method added to `Scene` that utilizes pythreejs for animating a system. [PR #416]
- Remove support for required dependencies prior to those in Ubuntu 18.04 LTS. [PR #415]
- Recommend installing from Conda Forge [PR #411]

1.10.4 0.5.0 (January 9, 2019)

- SymPy introduced a backward incompatibility to differentiation Matrices in SymPy 1.2, which remained in SymPy 1.3, see: https://github.com/sympy/sympy/issues/14958. This breaks PyDy's System class, see: https://github.com/pydy/pydy/issues/395. A fix is introduced to handle all support versions of SymPy. [PR #408]
- Added a new example for anthropomorphic arm. [PR #406]
- Fixed errors in the differential drive example. [PR #405]
- Added a new example for a scara arm. [PR #402]
- Fixed errors due to backwards incompatible changes with various dependencies. [PR #397]
- ODEFunctionGenerator now works with no constants symbols. [PR #391]

1.10.5 0.4.0 (May 30, 2017)

- Bumped minimum Jupyter notebook to 4.0 and restricted to < 5.0. [PR #381]
- Removed several deprecated functions. [PR #375]
- Bumped minimum required hard dependencies to Ubuntu 16.04 LTS package versions. [PR #372]
- Implemented ThreeJS Tube Geometry. [PR #368]
- Improved circle rendering. [PR #357]
- kwargs can be passed from System.generate_ode_function to the matrix generator. [PR #356]

- Lagrangian simple pendulum example added. [PR #351]
- Derivatives can now be used as specifies in System. [PR #340]
- The initial conditions can now be adjusted in the notebook GUI. [PR #333]
- The width of the viz canvas is now properly bounded in the notebook. [PR #332]
- Planes now render both sides in the visualization GUI. [PR #330]
- Adds in more type checks for System.times. [PR #322]
- Added an OctaveMatrixGenerator for basic Octave/Matlab printing. [PR #323]
- Simplified the right hand side evaluation code in the ODEFunctionGenerator. Note that this change comes with some performance hits. [PR #301]

1.10.6 0.3.1 (January 6, 2016)

- Removed the general deprecation warning from System. [PR #262]
- Don't assume user enters input in server shutdown. [PR #264]
- Use vectorized operations to compute transformations. [PR #266]
- Speedup theano generators. [PR #267]
- Correct time is displayed on the animation slider. [PR #272]
- Test optional dependencies only if installed. [PR #276]
- Require benchmark to run in Travis. [PR #277]
- Fix dependency minimum versions in setup.py [PR #279]
- Make CSE optional in CMatrixGenerator. [PR #284]
- Fix codegen line break. [PR #292]
- Don't assume Scene always has a System. [PR #295]
- Python 3.5 support and testing against Python 3.5 on Travis. [PR #305]
- Set minimum dependency versions to match Ubuntu Trusty 14.04 LTS. [PR #306]
- Replace sympy.phyics.mechanics deprecated methods. [PR #309]
- Updated installation details to work with IPython/Jupyter 4.0. [PR #311]
- Avoid the IPython widget deprecation warning if possible. [PR #311]
- Updated the mass-spring-damper example to IPy4 and added version_information. [PR #312]
- The Cython backend now compiles on Windows. [PR #313]
- CI testing is now run on appveyor with Windows VMs. [PR #315]
- Added a verbose option to the Cython compilation. [PR #315]
- Fixed the RHS autogeneration. [PR #318]
- Improved the camera code through inheritance [PR #319]

1.10.7 0.3.0 (January 19, 2015)

User Facing

- Introduced conda builds and binstar support. [PR #219]
- Dropped support for IPython < 3.0. [PR #237]
- Added support Python 3.3 and 3.4. [PR #229]
- Bumped up the minimum dependencies for NumPy, SciPy, and Cython [PR #233].
- Removed the partial implementation of the Mesh shape. [PR #172]
- Overhauled the code generation package to make the generators more easily extensible and to improve simulation speed. [PR #113]
- The visualizer has been overhauled as part of Tarun Gaba's 2014 GSoC internship [PR #82]. Here are some of the changes:
  - The JavaScript is now handled by AJAX and requires a simple server.
  - The JavaScript has been overhauled and now uses prototype.js for object oriented design.
  - The visualizer can now be loaded in an IPython notebook via IPython's widgets using `Scene.display_ipython()`.
  - A slider was added to manually control the frame playback.
  - The visualization shapes' attributes can be manipulated via the GUI.
  - The scene json file can be edited and downloaded from the GUI.
  - pydy.viz generates two JSONs now (instead of one in earlier versions). The JSON generated from earlier versions will **not** work in the new version.
  - Shapes can now have a material attribute.
  - Model constants can be modified and the simulations can be rerun all via the GUI.
  - Switched from socket based server to python's core SimpleHTTPServer.
  - The server has a proper shutdown response [PR #241]
- Added a new experimental System class and module to more seamlessly manage integrating the equations of motion. [PR #81]

Development

- Switched to a conda based Travis testing setup. [PR #231]
- When using older SymPy development versions with non-PEP440 compliant version identifiers, setuptools < 8 is required. [PR #166]
- Development version numbers are now PEP 440 compliant. [PR #141]
- Introduced pull request checklists and CONTRIBUTING file. [PR #146]
- Introduced light code linting into Travis. [PR #148]

1.10.8 0.2.1 (June 19, 2014)

- Unbundled unnecessary files from tar ball.

1.10.9 0.2.0 (June 19, 2014)

- Merged pydy_viz, pydy_code_gen, and pydy_examples into the source tree.
- Added a method to output "static" visualizations from a Scene object.
- Dropped the matplotlib dependency and now only three.js colors are valid.
- Added joint torques to the n_pendulum model.
- Added basic examples for codegen and viz.
- Graceful fail if theano or cython are not present.
- Shapes can now use sympy symbols for geometric dimensions.

1.11 Astrobee: A Holonomic Free-Flying Space Robot

Note: You can download this example as a Python script: `astrobee.py` or Jupyter notebook: `astrobee.ipynb`.

Astrobee is a new generation of free-flying robots aboard the International Space Station (ISS). It is a cubic robot with sides measuring about 30 cm each. The robot is propelled by two fans located on the sides of the robot and servo-actuated louvred vent nozzles, which allow for full six degree-of-freedom holonomic control [Smith2016]. Here, the nonlinear dynamics of Astrobee are modeled using Kane's method and the holonomic behavior of the system is demonstrated. After derivation of the nonlinear equations of motion, the system is linearized about a chosen operating point to obtain an explicit first order state-space representation, which can be used for control design.

```
import sympy as sm
import sympy.physics.mechanics as me
from pydy.system import System
import numpy as np
import matplotlib.pyplot as plt
from pydy.codegen.ode_function_generators import generate_ode_function
from scipy.integrate import odeint
import scipy.io as sio
me.init_vprinting()
```

1.11.1 Reference Frames

```
ISS = me.ReferenceFrame('N') # ISS RF
B = me.ReferenceFrame('B') # body RF

q1, q2, q3 = me.dynamicsymbols('q1:4') # attitude coordinates (Euler angles)

B.orient(ISS, 'Body', (q1, q2, q3), 'xyz') # body RF

t = me.dynamicsymbols._t
```

1.11.2 Significant Points

```
O = me.Point('O') # fixed point in the ISS
O.set_vel(ISS, 0)

x, y, z = me.dynamicsymbols('x, y, z') # translation coordinates (position of the mass-
↪center of Astrobee relative to 'O')
l = sm.symbols('l') # length of Astrobee (side of cube)

C = O.locatenew('C', x * ISS.x + y * ISS.y + z * ISS.z) # Astrobee CM
```

1.11.3 Kinematical Differential Equations

```
ux = me.dynamicsymbols('u_x')
uy = me.dynamicsymbols('u_y')
uz = me.dynamicsymbols('u_z')
u1 = me.dynamicsymbols('u_1')
u2 = me.dynamicsymbols('u_2')
u3 = me.dynamicsymbols('u_3')

z1 = sm.Eq(ux, x.diff())
z2 = sm.Eq(uy, y.diff())
z3 = sm.Eq(uz, z.diff())
z4 = sm.Eq(u1, q1.diff())
z5 = sm.Eq(u2, q2.diff())
z6 = sm.Eq(u3, q3.diff())
u = sm.solve([z1, z2, z3, z4, z5, z6], x.diff(), y.diff(), z.diff(), q1.diff(), q2.
↪diff(), q3.diff())
u
```

$$\left\{\dot{q}_1 : u_1,\ \dot{q}_2 : u_2,\ \dot{q}_3 : u_3,\ \dot{x} : u_x,\ \dot{y} : u_y,\ \dot{z} : u_z\right\}$$

1.11.4 Translational Motion

Velocity

```
C.set_vel(ISS, C.pos_from(O).dt(ISS).subs(u))
V_B_ISS_ISS = C.vel(ISS)
V_B_ISS_ISS # "velocity of Astrobee CM w.r.t ISS RF expressed in ISS RF"
```

$$u_x \hat{\mathbf{n}}_\mathbf{x} + u_y \hat{\mathbf{n}}_\mathbf{y} + u_z \hat{\mathbf{n}}_\mathbf{z}$$

Acceleration

```
A_B_ISS_ISS = C.acc(ISS).subs(u) #.subs(ud)
A_B_ISS_ISS # "acceleration of Astrobee CM w.r.t ISS RF expressed in ISS RF"
```

$$\dot{u}_x \hat{\mathbf{n}}_\mathbf{x} + \dot{u}_y \hat{\mathbf{n}}_\mathbf{y} + \dot{u}_z \hat{\mathbf{n}}_\mathbf{z}$$

1.11.5 Angular Motion

Angular Velocity

```
B.set_ang_vel(ISS, B.ang_vel_in(ISS).subs(u))
Omega_B_ISS_B = B.ang_vel_in(ISS)
Omega_B_ISS_B # "angular velocity of body RF w.r.t ISS RF expressed in body RF"
```

$$(u_1 \cos(q_2)\cos(q_3) + u_2 \sin(q_3))\hat{\mathbf{b}}_\mathbf{x} + (-u_1 \sin(q_3)\cos(q_2) + u_2 \cos(q_3))\hat{\mathbf{b}}_\mathbf{y} + (u_1 \sin(q_2) + u_3)\hat{\mathbf{b}}_\mathbf{z}$$

Angular Acceleration

```
Alpha_B_ISS_B = B.ang_acc_in(ISS).subs(u) #.subs(ud)
Alpha_B_ISS_B # "angular acceleration of body RF w.r.t ISS RF expressed in body RF"
```

$$(-u_1 u_2 \sin(q_2)\cos(q_3) - u_1 u_3 \sin(q_3)\cos(q_2) + u_2 u_3 \cos(q_3) + \sin(q_3)\dot{u}_2 + \cos(q_2)\cos(q_3)\dot{u}_1)\hat{\mathbf{b}}_\mathbf{x} + (u_1 u_2 \sin(q_2)\sin(q_3) - u_1 u_3 \cos(q_2)\cos(q_3) - u_2 u_3 \sin(q_3) - \sin(q_3)\cos(q_2)\dot{u}_1 + \cos(q_3)\dot{u}_2)\hat{\mathbf{b}}_\mathbf{y} + (u_1 u_2 \cos(q_2) + \sin(q_2)\dot{u}_1 + \dot{u}_3)\hat{\mathbf{b}}_\mathbf{z}$$

1.11.6 Mass and Inertia

```
m = sm.symbols('m') # Astrobee mass

Ix, Iy, Iz = sm.symbols('I_x, I_y, I_z') # principal moments of inertia

I = me.inertia(B, Ix, Iy, Iz) # inertia dyadic
I
```

$$I_x \hat{\mathbf{b}}_\mathbf{x} \otimes \hat{\mathbf{b}}_\mathbf{x} + I_y \hat{\mathbf{b}}_\mathbf{y} \otimes \hat{\mathbf{b}}_\mathbf{y} + I_z \hat{\mathbf{b}}_\mathbf{z} \otimes \hat{\mathbf{b}}_\mathbf{z}$$

1.11.7 Loads

Forces

```
Fx_mag, Fy_mag, Fz_mag = me.dynamicsymbols('Fmag_x, Fmag_y, Fmag_z')

Fx = Fx_mag * ISS.x
Fy = Fy_mag * ISS.y
Fz = Fz_mag * ISS.z

Fx, Fy, Fz
```

$$\left(|F|_x \, \hat{\mathbf{n}}_\mathbf{x}, \ |F|_y \, \hat{\mathbf{n}}_\mathbf{y}, \ |F|_z \, \hat{\mathbf{n}}_\mathbf{z} \right)$$

Torques

```
T1_mag, T2_mag, T3_mag = me.dynamicsymbols('Tmag_1, Tmag_2, Tmag_3')

T1 = T1_mag * B.x
T2 = T2_mag * B.y
T3 = T3_mag * B.z

T1, T2, T3
```

$$\left(|T|_1 \, \hat{\mathbf{b}}_\mathbf{x}, \ |T|_2 \, \hat{\mathbf{b}}_\mathbf{y}, \ |T|_3 \, \hat{\mathbf{b}}_\mathbf{z} \right)$$

1.11.8 Kane's Method

```
kdes = [z1.rhs - z1.lhs,
        z2.rhs - z2.lhs,
        z3.rhs - z3.lhs,
        z4.rhs - z4.lhs,
        z5.rhs - z5.lhs,
        z6.rhs - z6.lhs]

body = me.RigidBody('body', C, B, m, (I, C))
bodies = [body]

loads = [
         (C, Fx),
         (C, Fy),
         (C, Fz),
         (B, T1),
         (B, T2),
         (B, T3)
        ]

kane = me.KanesMethod(ISS, (x, y, z, q1, q2, q3), (ux, uy, uz, u1, u2, u3), kd_eqs=kdes)
```

(continues on next page)

(continued from previous page)

```
fr, frstar = kane.kanes_equations(bodies, loads=loads)
```

1.11.9 Simulation

```
sys = System(kane)

sys.constants_symbols

sys.constants = {
                 Ix: 0.1083,
                 Iy: 0.1083,
                 Iz: 0.1083,
                 m: 7
                }

sys.constants
```

$$\{I_x : 0.1083,\ I_y : 0.1083,\ I_z : 0.1083,\ m : 7\}$$

```
sys.times = np.linspace(0.0, 50.0, num=1000)

sys.coordinates
```

$$[x,\ y,\ z,\ q_1,\ q_2,\ q_3]$$

```
sys.speeds
```

$$[u_x,\ u_y,\ u_z,\ u_1,\ u_2,\ u_3]$$

```
sys.states
```

$$[x,\ y,\ z,\ q_1,\ q_2,\ q_3,\ u_x,\ u_y,\ u_z,\ u_1,\ u_2,\ u_3]$$

```
sys.initial_conditions = {
                          x: 0.0,
                          y: 0.0,
                          z: 0.0,
                          q1: 0.0,
                          q2: 0.0,
                          q3: 0.0,
                          ux: 0.2,
                          uy: 0.0,
                          uz: 0.0,
                          u1: 0.0,
                          u2: 0.0,
                          u3: 0.5
                         }
```

```
sys.specifieds_symbols
```

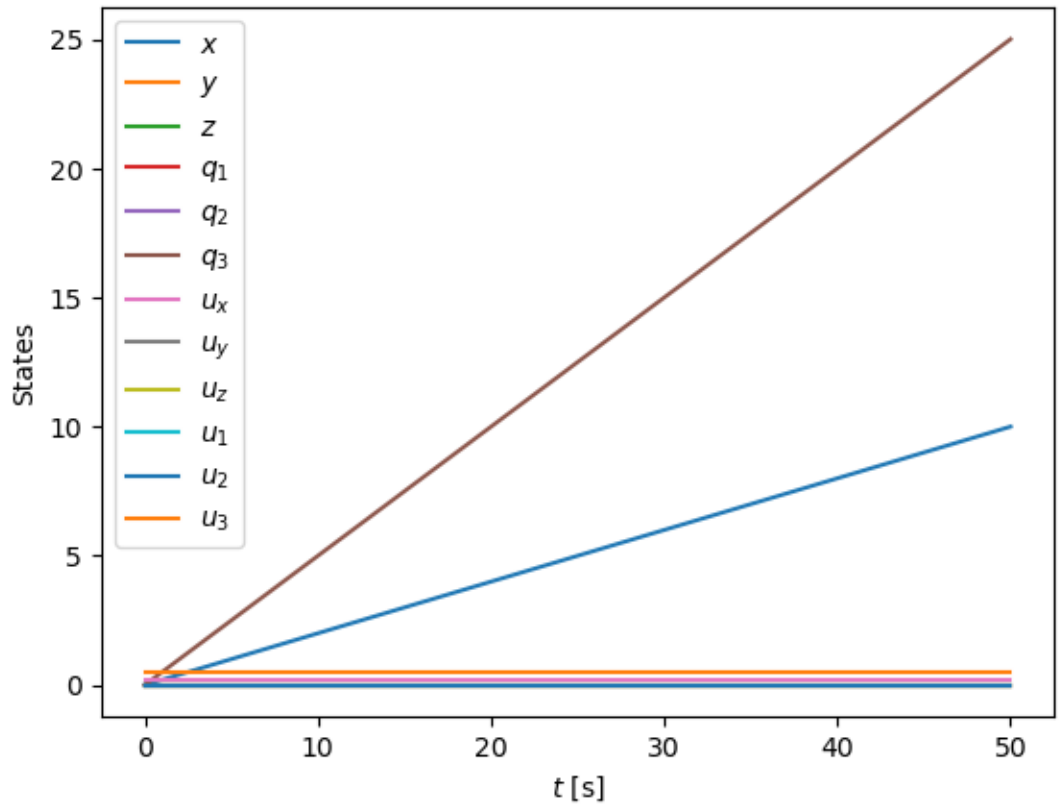
$$\left\{ |F|_x , |F|_y , |F|_z , |T|_1 , |T|_2 , |T|_3 \right\}$$

```
sys.specifieds = {
                  Fx_mag: 0.0,
                  Fy_mag: 0.0,
                  Fz_mag: 0.0,
                  T1_mag: 0.0,
                  T2_mag: 0.0,
                  T3_mag: 0.0
                 }
```

```
states = sys.integrate()
```

```
import matplotlib.pyplot as plt
```

```
fig, ax = plt.subplots()
ax.plot(sys.times, states)
ax.set_xlabel('{} [s]'.format(sm.latex(t, mode='inline')));
ax.set_ylabel('States');
ax.legend(['$x$', '$y$', '$z$', '$q_1$', '$q_2$', '$q_3$', '$u_x$', '$u_y$', '$u_z$', '
↪$u_1$', '$u_2$', '$u_3$'], fontsize=10)
plt.show()
```



1.11.10 3D Visualization

```
from pydy.viz import Box, Cube, Sphere, Cylinder, VisualizationFrame, Scene
```

```
l = 0.32

body_m_shape = Box(l, (1/2) * l, (2/3) * l, color='black', name='body_m_shape')
body_l_shape = Box(l, (1/4) * l, l, color='green', name='body_l_shape')
body_r_shape = Box(l, (1/4) * l, l, color='green', name='body_r_shape')

v1 = VisualizationFrame('Body_m',
                        B,
                        C.locatenew('C_m', (1/6) * l * B.z),
                        body_m_shape)

v2 = VisualizationFrame('Body_l',
                        B,
                        C.locatenew('C_l', (3/8) * l * -B.y),
                        body_l_shape)

v3 = VisualizationFrame('Body_r',
                        B,
                        C.locatenew('C_r', (3/8) * l * B.y),
                        body_r_shape)

scene = Scene(ISS, O, system=sys)

scene.visualization_frames = [v1, v2, v3]
```

```
scene.display_jupyter(axes_arrow_length=1.0)
```

```
VBox(children=(AnimationAction(clip=AnimationClip(duration=50.0,␣
↪tracks=(VectorKeyframeTrack(name='scene/body_...
```

1.11.11 Linearization

```
f = fr + frstar
f
```

```
V = {
      x: 0.0,
      y: 0.0,
      z: 0.0,
      q1: 0.0,
      q2: 0.0,
      q3: 0.0,
      ux: 0.0,
      uy: 0.0,
      uz: 0.0,
```

(continues on next page)

(continued from previous page)

```
      u1: 0.0,
      u2: 0.0,
      u3: 0.0,
      Fx_mag: 0.0,
      Fy_mag: 0.0,
      Fz_mag: 0.0,
      T1_mag: 0.0,
      T2_mag: 0.0,
      T3_mag: 0.0
}

V_keys = sm.Matrix([ v for v in V.keys() ])
V_values = sm.Matrix([ v for v in V.values() ])
```

```
us = sm.Matrix([ux, uy, uz, u1, u2, u3])
us_diff = sm.Matrix([ux.diff(), uy.diff(), uz.diff(), u1.diff(), u2.diff(), u3.diff()])
qs = sm.Matrix([x, y, z, q1, q2, q3])
rs = sm.Matrix([Fx_mag, Fy_mag, Fz_mag, T1_mag, T2_mag, T3_mag])
```

```
Ml = f.jacobian(us_diff).subs(sys.constants).subs(V)
Ml
```

$$\begin{bmatrix} -7 & 0 & 0 & 0 & 0 & 0 \\ 0 & -7 & 0 & 0 & 0 & 0 \\ 0 & 0 & -7 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.1083 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.1083 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.1083 \end{bmatrix}$$

```
Cl = f.jacobian(us).subs(V)
Cl.subs(sys.constants)
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
Kl = f.jacobian(qs).subs(V)
sm.simplify(Kl.subs(sys.constants))
```

$$\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

```
Hl = -f.jacobian(rs).subs(V)
sm.simplify(Hl.subs(sys.constants))
```

$$\begin{bmatrix}-1 & 0 & 0 & 0 & 0 & 0\\0 & -1 & 0 & 0 & 0 & 0\\0 & 0 & -1 & 0 & 0 & 0\\0 & 0 & 0 & -1 & 0 & 0\\0 & 0 & 0 & 0 & -1 & 0\\0 & 0 & 0 & 0 & 0 & -1\end{bmatrix}$$

```
A = sm.Matrix([[(-Ml.inv()*Cl), (-Ml.inv()*Kl)], [(sm.eye(6)), sm.zeros(6, 6)]])
sm.simplify(A.subs(sys.constants))
```

$$\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

```
B = sm.Matrix([[Ml.inv() * Hl], [sm.zeros(6, 6)]])
sm.nsimplify(B.subs(sys.constants))
```

$$\begin{bmatrix} \frac{1}{7} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{7} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{7} & 0 & 0 & 0 \\ 0 & 0 & 0 & 9.23361034164358 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9.23361034164358 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9.23361034164358 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

1.11.12 References

1.12 Carvallo-Whipple Bicycle Model

Note: You can download this example as a Python script: `carvallo-whipple.py` or Jupyter notebook: `carvallo-whipple.ipynb`.

This example creates a nonlinear model and simulation of the Carvallo-Whipple Bicycle Model ([Carvallo1899], [Whipple1899]). This formulation uses the conventions described in [Moore2012] which are equivalent to the models described in [Meijaard2007] and [Basu-Mandal2007].

Import the necessary libraries, classes, and functions:

```
import numpy as np
from scipy.optimize import fsolve
import sympy as sm
import sympy.physics.mechanics as mec
from pydy.system import System
from pydy.viz import Sphere, Cylinder, VisualizationFrame, Scene
```

1.12.1 System Diagrams

1.12.2 Reference Frames

Use a reference frame that mimics the notation in [Moore2012].

```
class ReferenceFrame(mec.ReferenceFrame):
    """Subclass that enforces the desired unit vector indice style."""

    def __init__(self, *args, **kwargs):
```

(continues on next page)

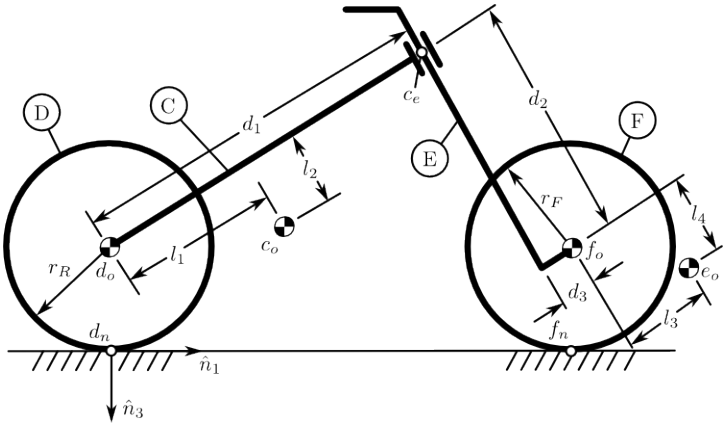


Fig. 1: Geometric variable definitions.

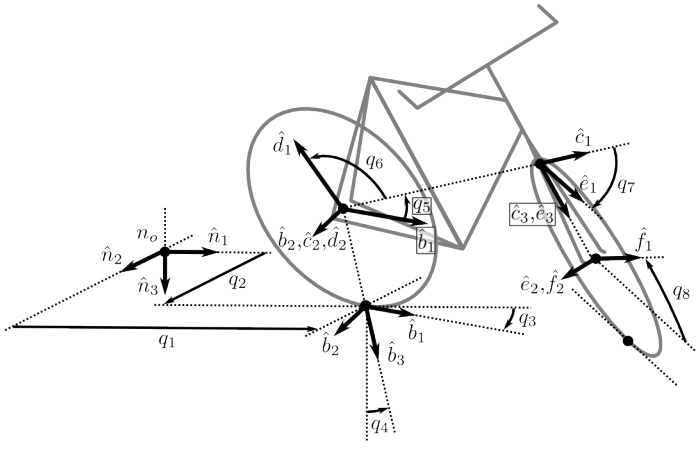


Fig. 2: Configuration coordinate definitions.

(continued from previous page)

```
        kwargs.pop('indices', None)
        kwargs.pop('latexs', None)

        lab = args[0].lower()
        tex = r'\hat{{{}}}_{}'

        super(ReferenceFrame, self).__init__(*args,
                                             indices=('1', '2', '3'),
                                             latexs=(tex.format(lab, '1'),
                                                     tex.format(lab, '2'),
                                                     tex.format(lab, '3')),
                                             **kwargs)
```

Define some useful references frames:

- N: Newtonian Frame
- A: Yaw Frame, auxiliary frame
- B: Roll Frame, axillary frame
- C: Rear Frame
- D: Rear Wheel Frame
- E: Front Frame
- F: Front Wheel Frame

```
N = ReferenceFrame('N')
A = ReferenceFrame('A')
B = ReferenceFrame('B')
C = ReferenceFrame('C')
D = ReferenceFrame('D')
E = ReferenceFrame('E')
F = ReferenceFrame('F')
```

1.12.3 Generalized Coordinates and Speeds

All of the following variables are a functions of time, t.

- q_1: perpendicular distance from the $\hat{n}_2$ axis to the rear contact point in the ground plane
- q_2: perpendicular distance from the $\hat{n}_1$ axis to the rear contact point in the ground plane
- q_3: frame yaw angle
- q_4: frame roll angle
- q_5: frame pitch angle
- q_6: front wheel rotation angle
- q_7: steering rotation angle
- q_8: rear wheel rotation angle
- q_9: perpendicular distance from the $\hat{n}_2$ axis to the front contact point in the ground plane

- q_{10}: perpendicular distance from the $\hat{n}_1$ axis to the front contact point in the ground plane

```
q1, q2, q3, q4 = mec.dynamicsymbols('q1 q2 q3 q4')
q5, q6, q7, q8 = mec.dynamicsymbols('q5 q6 q7 q8')

u1, u2, u3, u4 = mec.dynamicsymbols('u1 u2 u3 u4')
u5, u6, u7, u8 = mec.dynamicsymbols('u5 u6 u7 u8')
```

1.12.4 Orientation of Reference Frames

Declare the orientation of each frame to define the yaw, roll, and pitch of the rear frame relative to the Newtonian frame. The define steer of the front frame relative to the rear frame.

```
# rear frame yaw
A.orient(N, 'Axis', (q3, N['3']))
# rear frame roll
B.orient(A, 'Axis', (q4, A['1']))
# rear frame pitch
C.orient(B, 'Axis', (q5, B['2']))
# front frame steer
E.orient(C, 'Axis', (q7, C['3']))
```

1.12.5 Constants

Declare variables that are constant with respect to time for the model's physical parameters.

- r_f: radius of front wheel
- r_r: radius of rear wheel
- d_1: the perpendicular distance from the steer axis to the center of the rear wheel (rear offset)
- d_2: the distance between wheels along the steer axis
- d_3: the perpendicular distance from the steer axis to the center of the front wheel (fork offset)
- l_1: the distance in the $\hat{c}_1$ direction from the center of the rear wheel to the frame center of mass
- l_2: the distance in the $\hat{c}_3$ direction from the center of the rear wheel to the frame center of mass
- l_3: the distance in the $\hat{e}_1$ direction from the front wheel center to the center of mass of the fork
- l_4: the distance in the $\hat{e}_3$ direction from the front wheel center to the center of mass of the fork

```
rf, rr = sm.symbols('rf rr')
d1, d2, d3 = sm.symbols('d1 d2 d3')
l1, l2, l3, l4 = sm.symbols('l1 l2 l3 l4')

# acceleration due to gravity
g = sm.symbols('g')

# mass
mc, md, me, mf = sm.symbols('mc md me mf')

# inertia
```

(continues on next page)

(continued from previous page)

```
ic11, ic22, ic33, ic31 = sm.symbols('ic11 ic22 ic33 ic31')
id11, id22 = sm.symbols('id11 id22')
ie11, ie22, ie33, ie31 = sm.symbols('ie11 ie22 ie33 ie31')
if11, if22 = sm.symbols('if11 if22')
```

1.12.6 Specified

Declare three specified torques that are functions of time.

- T_4 : roll torque, between Newtonian frame and rear frame
- T_6 : rear wheel torque, between rear wheel and rear frame
- T_7 : steer torque, between rear frame and front frame

```
T4, T6, T7 = mec.dynamicsymbols('T4 T6 T7')
```

1.12.7 Position Vectors

```
# rear wheel contact point
dn = mec.Point('dn')

# rear wheel contact point to rear wheel center
do = mec.Point('do')
do.set_pos(dn, -rr * B['3'])

# rear wheel center to bicycle frame center
co = mec.Point('co')
co.set_pos(do, l1 * C['1'] + l2 * C['3'])

# rear wheel center to steer axis point
ce = mec.Point('ce')
ce.set_pos(do, d1 * C['1'])

# steer axis point to the front wheel center
fo = mec.Point('fo')
fo.set_pos(ce, d2 * E['3'] + d3 * E['1'])

# front wheel center to front frame center
eo = mec.Point('eo')
eo.set_pos(fo, l3 * E['1'] + l4 * E['3'])

# locate the point fixed on the wheel which instantaneously touches the
# ground
fn = mec.Point('fn')
fn.set_pos(fo, rf * E['2'].cross(A['3']).cross(E['2']).normalize())
```

1.12.8 Holonomic Constraint

The front contact point f_n and the rear contact point d_n must both reside in the ground plane.

```
holonomic = fn.pos_from(dn).dot(A['3'])
```

This expression defines a configuration constraint among q_4, q_5, and q_7.

```
mec.find_dynamicsymbols(holonomic)
```

```
{q4(t), q5(t), q7(t)}
```

1.12.9 Kinematical Differential Equations

Define the generalized speeds all as $u = \dot{q}$.

```
t = mec.dynamicsymbols._t

kinematical = [q3.diff(t) - u3,  # yaw
               q4.diff(t) - u4,  # roll
               q5.diff(t) - u5,  # pitch
               q7.diff(t) - u7]  # steer
```

1.12.10 Angular Velocities

```
A.set_ang_vel(N, u3 * N['3'])  # yaw rate
B.set_ang_vel(A, u4 * A['1'])  # roll rate
C.set_ang_vel(B, u5 * B['2'])  # pitch rate
D.set_ang_vel(C, u6 * C['2'])  # rear wheel rate
E.set_ang_vel(C, u7 * C['3'])  # steer rate
F.set_ang_vel(E, u8 * E['2'])  # front wheel rate
```

1.12.11 Linear Velocities

```
# rear wheel contact stays in ground plane and does not slip
dn.set_vel(N, 0.0 * N['1'])

# mass centers
do.v2pt_theory(dn, N, D)
co.v2pt_theory(do, N, C)
ce.v2pt_theory(do, N, C)
fo.v2pt_theory(ce, N, E)
eo.v2pt_theory(fo, N, E)

# wheel contact velocities
fn.v2pt_theory(fo, N, F);  # supress output
```

1.12.12 Motion Constraints

Enforce the no slip condition at the front wheel contact point. Note that the no-slip condition is already enforced with the velocity of n_o set to 0. Also include an extra motion constraint not allowing vertical motion of the contact point. Note that this is an integrable constraint, i.e. the derivative of `nonholonomic` above. It is not a nonholonomic constraint, but we include it because we can't easy eliminate a dependent generalized coordinate with `nonholonmic`.

Warning: The floating point numerical stability of the solution is affected by the order of the nonholonomic constraint expressions in the following list. If ordered `A['1'],A['2'],A['3']` stability degrades.

```
nonholonomic = [
    fn.vel(N).dot(A['1']),
    fn.vel(N).dot(A['3']),
    fn.vel(N).dot(A['2']),
]
```

1.12.13 Inertia

The inertia dyadics are defined with respect to the rear and front frames.

```
Ic = mec.inertia(C, ic11, ic22, ic33, 0.0, 0.0, ic31)
Id = mec.inertia(C, id11, id22, id11, 0.0, 0.0, 0.0)
Ie = mec.inertia(E, ie11, ie22, ie33, 0.0, 0.0, ie31)
If = mec.inertia(E, if11, if22, if11, 0.0, 0.0, 0.0)
```

1.12.14 Rigid Bodies

```
rear_frame = mec.RigidBody('Rear Frame', co, C, mc, (Ic, co))
rear_wheel = mec.RigidBody('Rear Wheel', do, D, md, (Id, do))
front_frame = mec.RigidBody('Front Frame', eo, E, me, (Ie, eo))
front_wheel = mec.RigidBody('Front Wheel', fo, F, mf, (If, fo))

bodies = [rear_frame, rear_wheel, front_frame, front_wheel]
```

1.12.15 Loads

```
# gravity
Fco = (co, mc*g*A['3'])
Fdo = (do, md*g*A['3'])
Feo = (eo, me*g*A['3'])
Ffo = (fo, mf*g*A['3'])

# input torques
Tc = (C, T4*A['1'] - T6*B['2'] - T7*C['3'])
Td = (D, T6*C['2'])
Te = (E, T7*C['3'])

loads = [Fco, Fdo, Feo, Ffo, Tc, Td, Te]
```

1.12.16 Kane's Method

```
kane = mec.KanesMethod(N,
                       [q3, q4, q7],  # yaw, roll, steer
                       [u4, u6, u7],  # roll rate, rear wheel rate, steer rate
                       kd_eqs=kinematical,
                       q_dependent=[q5],  # pitch angle
                       configuration_constraints=[holonomic],
                       u_dependent=[u3, u5, u8],  # yaw rate, pitch rate, front wheel␣
↪rate
                       velocity_constraints=nonholonomic)

fr, frstar = kane.kanes_equations(bodies, loads)
```

1.12.17 Simulating the system

PyDy's `System` is a wrapper that holds the `KanesMethod` object to integrate the equations of motion using numerical values of constants.

```
from pydy.system import System
sys = System(kane)
```

Now, we specify the numerical values of the constants and the initial values of states in the form of a dict. The are the benchmark values used in [Meijaard2007] converted to the [Moore2012] formulation.

```
sys.constants = {
   rf: 0.35,
   rr: 0.3,
   d1: 0.9534570696121849,
   d3: 0.03207142672761929,
   d2: 0.2676445084476887,
   l1: 0.4707271515135145,
   l2: -0.47792881146460797,
   l4: -0.3699518200282974,
   l3: -0.00597083392418685,
   mc: 85.0,
   md: 2.0,
   me: 4.0,
   mf: 3.0,
   id11: 0.0603,
   id22: 0.12,
   if11: 0.1405,
   if22: 0.28,
   ic11: 7.178169776497895,
   ic22: 11.0,
   ic31: 3.8225535938357873,
   ic33: 4.821830223502103,
   ie11: 0.05841337700152972,
   ie22: 0.06,
   ie31: 0.009119225261946298,
   ie33: 0.007586622998470264,
```

(continues on next page)

(continued from previous page)

```
   g: 9.81
}
```

Setup the initial conditions such that the bicycle is traveling at some forward speeds and has an initial positive roll rate.

```
initial_speed = 4.6  # m/s
initial_roll_rate = 0.5  # rad/s
```

The initial configuration will be the upright equilibrium position. The holonomic constraint requires that either the roll, pitch, or steer angle need be dependent. Below, the pitch angle is taken as dependent and solved for using *fsolve()*. Note that it is equivalent to the steer axis tilt [Meijaard2007].

```
eval_holonomic = sm.lambdify((q5, q4, q7, d1, d2, d3, rf, rr), holonomic)
initial_pitch_angle = float(fsolve(eval_holonomic, 0.0,
                                   args=(0.0,  # q4
                                         1e-8,  # q7
                                         sys.constants[d1],
                                         sys.constants[d2],
                                         sys.constants[d3],
                                         sys.constants[rf],
                                         sys.constants[rr])))
np.rad2deg(initial_pitch_angle)
```

```
18.000000000000007
```

Set all of the initial conditions.

Warning: A divide-by-zero will occur if the steer angle is set to zero. Thus the *1e-8* values. The integration is also sensitive to the size of this value. This shouldn't be the case and may point to some errors in the derivation and implementation. More careful attention to the integration tolerances may help too.

```
sys.initial_conditions = {q3: 0.0,
                          q4: 0.0,
                          q5: initial_pitch_angle,
                          q7: 1e-8,
                          u3: 0.0,
                          u4: initial_roll_rate,
                          u5: 0.0,
                          u6: -initial_speed/sys.constants[rr],
                          u7: 0.0,
                          u8: -initial_speed/sys.constants[rf]}
```

Generate a time vector over which the integration will be carried out.

```
fps = 30  # frames per second
duration = 6.0  # seconds
sys.times = np.linspace(0.0, duration, num=int(duration*fps))
```

The trajectory of the states over time can be found by calling the `.integrate()` method. But due to the complexity of the equations of motion it is helpful to use the *cython* generator for faster numerical evaluation.

Warning: The holonomic constraint equation is not explicitly enforced, as PyDy does not yet support integration of differential algebraic equations (DAEs) yet. The solution will drift from the true solution over time with magnitudes dependent on the intiial conditions and constants values.

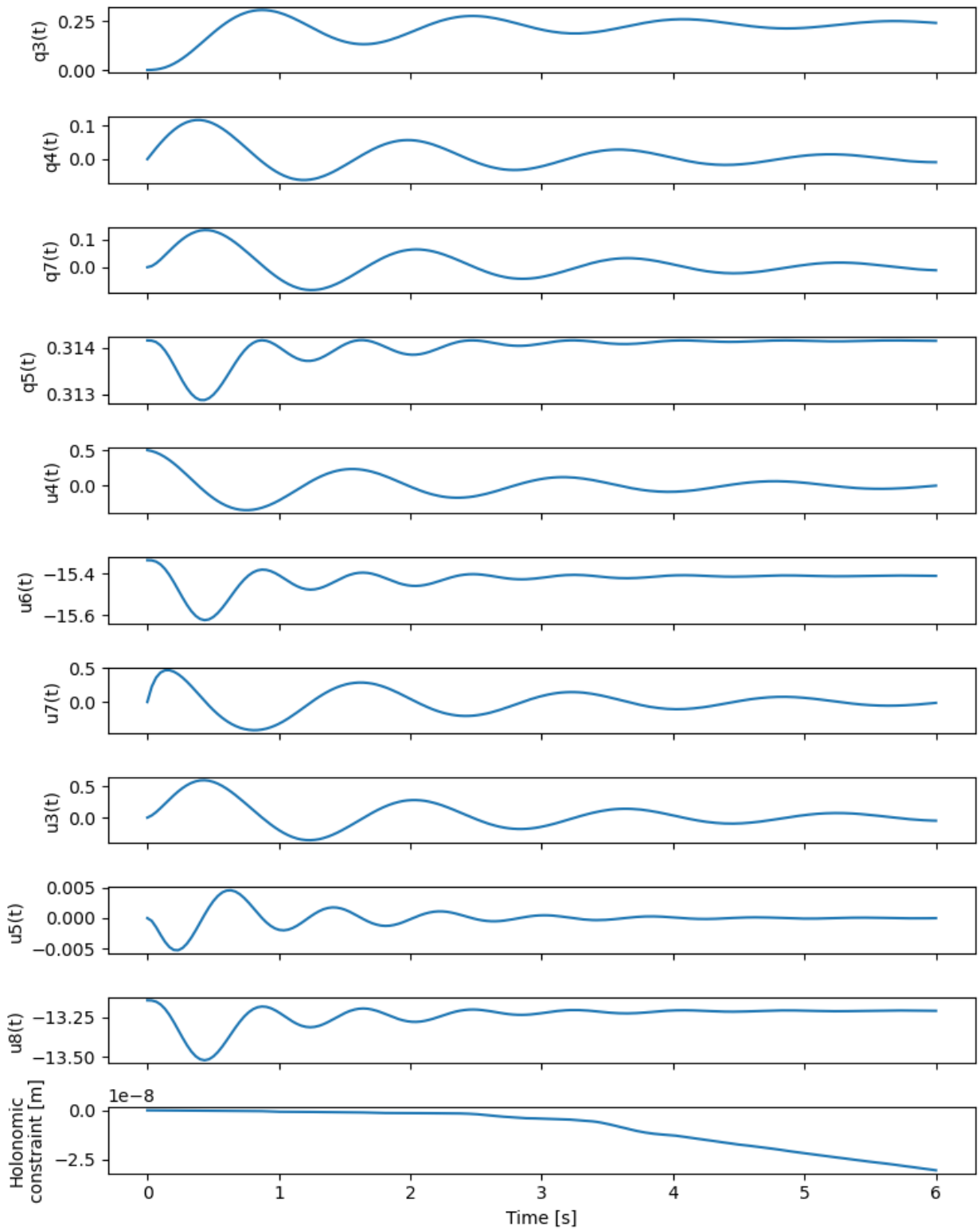
```
sys.generate_ode_function(generator='cython')

x_trajectory = sys.integrate()
```

Evaluate the holonomic constraint across the simulation.

```
holonomic_vs_time  = eval_holonomic(x_trajectory[:, 3],  # q5
                                    x_trajectory[:, 1],  # q4
                                    x_trajectory[:, 2],  # q7
                                    sys.constants[d1],
                                    sys.constants[d2],
                                    sys.constants[d3],
                                    sys.constants[rf],
                                    sys.constants[rr])
```

1.12.18 Plot the State Trajectories

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(len(sys.states) + 1, 1, sharex=True)
fig.set_size_inches(8, 10)
for ax, traj, s in zip(axes, x_trajectory.T, sys.states):
    ax.plot(sys.times, traj)
    ax.set_ylabel(s)
axes[-1].plot(sys.times, np.squeeze(holonomic_vs_time))
axes[-1].set_ylabel('Holonomic\nconstraint [m]')
axes[-1].set_xlabel('Time [s]')
plt.tight_layout()
```


q3(t)
0.25
0.00
q4(t)
0.1
0.0
q7(t)
0.1
0.0
q5(t)
0.314
0.313
u4(t)
0.5
0.0
u6(t)
−15.4
−15.6
u7(t)
0.5
0.0
u3(t)
0.5
0.0
u5(t)
0.005
0.000
−0.005
u8(t)
−13.25
−13.50
Holonomic constraint [m]
1e−8
0.0
−2.5
0
1
2
3
4
5
6
Time [s]

1.12.19 Visualizing the System Motion

Create two cylinders to represent the front and rear wheels.

```
rear_wheel_circle = Cylinder(radius=sys.constants[rr], length=0.01,
                             color="green", name='rear wheel')
front_wheel_circle = Cylinder(radius=sys.constants[rf], length=0.01,
                              color="green", name='front wheel')
rear_wheel_vframe = VisualizationFrame(B, do, rear_wheel_circle)
front_wheel_vframe = VisualizationFrame(E, fo, front_wheel_circle)
```

Create some cylinders to represent the front and rear frames.

```
d1_cylinder = Cylinder(radius=0.02, length=sys.constants[d1],
                       color='black', name='rear frame d1')
d2_cylinder = Cylinder(radius=0.02, length=sys.constants[d2],
                       color='black', name='front frame d2')
d3_cylinder = Cylinder(radius=0.02, length=sys.constants[d3],
                       color='black', name='front frame d3')

d1_frame = VisualizationFrame(C.orientnew('C_r', 'Axis', (sm.pi/2, C.z)),
                              do.locatenew('d1_half', d1/2*C.x), d1_cylinder)
d2_frame = VisualizationFrame(E.orientnew('E_r', 'Axis', (-sm.pi/2, E.x)),
                              fo.locatenew('d2_half', -d3*E.x - d2/2*E.z), d2_cylinder)
d3_frame = VisualizationFrame(E.orientnew('E_r', 'Axis', (sm.pi/2, E.z)),
                              fo.locatenew('d3_half', -d3/2*E.x), d3_cylinder)
```

Create some spheres to represent the mass centers of the front and rear frames.

```
co_sphere = Sphere(radius=0.05, color='blue', name='rear frame co')
eo_sphere = Sphere(radius=0.05, color='blue', name='rear frame eo')
co_frame = VisualizationFrame(C, co, co_sphere)
eo_frame = VisualizationFrame(E, eo, eo_sphere)
```

Create the scene and add the visualization frames.

```
scene = Scene(N, dn, system=sys)
scene.visualization_frames = [front_wheel_vframe, rear_wheel_vframe,
                              d1_frame, d2_frame, d3_frame,
                              co_frame, eo_frame]
```

Now, call the display method.

```
scene.display_jupyter(axes_arrow_length=5.0)
```

```
VBox(children=(AnimationAction(clip=AnimationClip(duration=6.0,␣
↪tracks=(VectorKeyframeTrack(name='scene/front ...
```

1.12.20 References

1.13 Chaos Pendulum

Note: You can download this example as a Python script: `chaos-pendulum.py` or Jupyter notebook: `chaos-pendulum.ipynb`.

This example gives a simple demostration of chaotic behavior in a simple two body system. The system is made up of a slender rod that is connected to the ceiling at one end with a revolute joint that rotates about the $\hat{\mathbf{n}}_y$ unit vector. At the other end of the rod a flat plate is attached via a second revolute joint allowing the plate to rotate about the rod's axis with aligns with the $\hat{\mathbf{a}_z}$ unit vector.

1.13.1 Setup

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sm
import sympy.physics.mechanics as me
from pydy.system import System
from pydy.viz import Cylinder, Plane, VisualizationFrame, Scene
```

```
%matplotlib inline
```

```
me.init_vprinting(use_latex='mathjax')
```

1.13.2 Define Variables

First define the system constants:

- m_A: Mass of the slender rod.
- m_B: Mass of the plate.
- l_B: Distance from N_o to B_o along the slender rod's axis.
- w: The width of the plate.
- h: The height of the plate.
- g: The acceleratoin due to gravity.

```
mA, mB, lB, w, h, g = sm.symbols('m_A, m_B, L_B, w, h, g')
```

There are two time varying generalized coordinates:

- $\theta(t)$: The angle of the slender rod with respect to the ceiling.
- $\phi(t)$: The angle of the plate with respect to the slender rod.

The two generalized speeds will then be defined as:

- $\omega(t) = \dot{\theta}$: The angular rate of the slender rod with respect to the ceiling.

- $\alpha(t) = \dot{\phi}$: The angluer rate of the plate with respect to the slender rod.

```
theta, phi = me.dynamicsymbols('theta, phi')
omega, alpha = me.dynamicsymbols('omega, alpha')
```

The kinematical differential equations are defined in this fashion for the `KanesMethod` class:

$$0 = \omega - \dot{\theta}$$
$$0 = \alpha - \dot{\phi}$$

```
kin_diff = (omega - theta.diff(), alpha - phi.diff())
kin_diff
```

$$\left(\omega - \dot{\theta},\ \alpha - \dot{\phi}\right)$$

1.13.3 Define Orientations

There are three reference frames. These are defined as such:

```
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
```

The frames are oriented with respect to each other by simple revolute rotations. The following lines set the orientations:

```
A.orient(N, 'Axis', (theta, N.y))
B.orient(A, 'Axis', (phi, A.z))
```

1.13.4 Define Positions

Three points are necessary to define the problem:

- N_o: The fixed point which the slender rod rotates about.
- A_o: The center of mass of the slender rod.
- B_o: The center of mass of the plate.

```
No = me.Point('No')
Ao = me.Point('Ao')
Bo = me.Point('Bo')
```

The two centers of mass positions can be set relative to the fixed point, N_o.

```
lA = (lB - h / 2) / 2
Ao.set_pos(No, lA * A.z)
Bo.set_pos(No, lB * A.z)
```

1.13.5 Specify the Velocities

The generalized speeds should be used in the definition of the linear and angular velocities when using Kane's method. For simple rotations and the defined kinematical differential equations the angular rates are:

```
A.set_ang_vel(N, omega * N.y)
B.set_ang_vel(A, alpha * A.z)
```

Once the angular velocities are specified the linear velocities can be computed using the two point velocity thereom, starting with the origin point having a velocity of zero.

```
No.set_vel(N, 0)
```

```
Ao.v2pt_theory(No, N, A)
```

$$\left(\frac{L_B}{2} - \frac{h}{4}\right) \omega \hat{\mathbf{a}}_\mathbf{x}$$

```
Bo.v2pt_theory(No, N, A)
```

$$L_B \omega \hat{\mathbf{a}}_\mathbf{x}$$

1.13.6 Inertia

The central inertia of the symmetric slender rod with respect to its reference frame is a function of its length and its mass.

```
IAxx = sm.S(1) / 12 * mA * (2 * lA)**2
IAyy = IAxx
IAzz = 0

IA = (me.inertia(A, IAxx, IAyy, IAzz), Ao)
```

This gives the inertia tensor:

```
IA[0].to_matrix(A)
```

$$\begin{bmatrix} \frac{m_A\left(L_B - \frac{h}{2}\right)^2}{12} & 0 & 0 \\ 0 & \frac{m_A\left(L_B - \frac{h}{2}\right)^2}{12} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The central inerita of the symmetric plate with respect to its reference frame is a function of its width and height.

```
IBxx = sm.S(1)/12 * mB * h**2
IByy = sm.S(1)/12 * mB * (w**2 + h**2)
IBzz = sm.S(1)/12 * mB * w**2

IB = (me.inertia(B, IBxx, IByy, IBzz), Bo)
```

```
IB[0].to_matrix(B)
```

$$\begin{bmatrix} \frac{h^2 m_B}{12} & 0 & 0 \\ 0 & \frac{m_B\left(h^2+w^2\right)}{12} & 0 \\ 0 & 0 & \frac{m_B w^2}{12} \end{bmatrix}$$

All of the information to define the two rigid bodies are now available. This information is used to create an object for the rod and the plate.

```
rod = me.RigidBody('rod', Ao, A, mA, IA)
```

```
plate = me.RigidBody('plate', Bo, B, mB, IB)
```

1.13.7 Loads

The only loads in this problem is the force due to gravity that acts on the center of mass of each body. These forces are specified with a tuple containing the point of application and the force vector.

```
rod_gravity = (Ao, mA * g * N.z)
plate_gravity = (Bo, mB * g * N.z)
```

1.13.8 Equations of motion

Now that the kinematics, kinetics, and inertia have all been defined the `KanesMethod` class can be used to generate the equations of motion of the system. In this case the independent generalized speeds, independent generalized speeds, the kinematical differential equations, and the inertial reference frame are used to initialize the class.

```
kane = me.KanesMethod(N, q_ind=(theta, phi), u_ind=(omega, alpha), kd_eqs=kin_diff)
```

The equations of motion are then generated by passing in all of the loads and bodies to the `kanes_equations` method. This produces f_r and f_r^*.

```
bodies = (rod, plate)
loads = (rod_gravity, plate_gravity)

fr, frstar = kane.kanes_equations(bodies, loads)
```

```
sm.trigsimp(fr)
```

$$\begin{bmatrix} g\left(-\frac{L_B m_A}{2} - L_B m_B + \frac{h m_A}{4}\right)\sin\left(\theta\right) \\ 0 \end{bmatrix}$$

```
sm.trigsimp(frstar)
```

$$\begin{bmatrix} \frac{m_B w^2 \alpha \omega \sin(2\phi)}{12} - \left(\frac{L_B^2 m_A}{3} + L_B^2 m_B - \frac{L_B h m_A}{3} + \frac{h^2 m_A}{12} + \frac{h^2 m_B}{12} + \frac{m_B w^2 \cos^2(\phi)}{12} \right) \dot{\omega} \\ - \frac{m_B w^2 \left(\omega^2 \sin(2\phi) + 2\dot{\alpha} \right)}{24} \end{bmatrix}$$

1.13.9 Simulation

The equations of motion can now be simulated numerically. Values for the constants, initial conditions, and time are provided to the `System` class along with the symbolic `KanesMethod` object.

```
sys = System(kane)
```

```
sys.constants = {lB: 0.2, # meters
                 h: 0.1, # meters
                 w: 0.2, # meters
                 mA: 0.01, # kilograms
                 mB: 0.1, # kilograms
                 g: 9.81} # meters per second squared
```

```
sys.initial_conditions = {theta: np.deg2rad(45),
                          phi: np.deg2rad(0.5),
                          omega: 0,
                          alpha: 0}
```

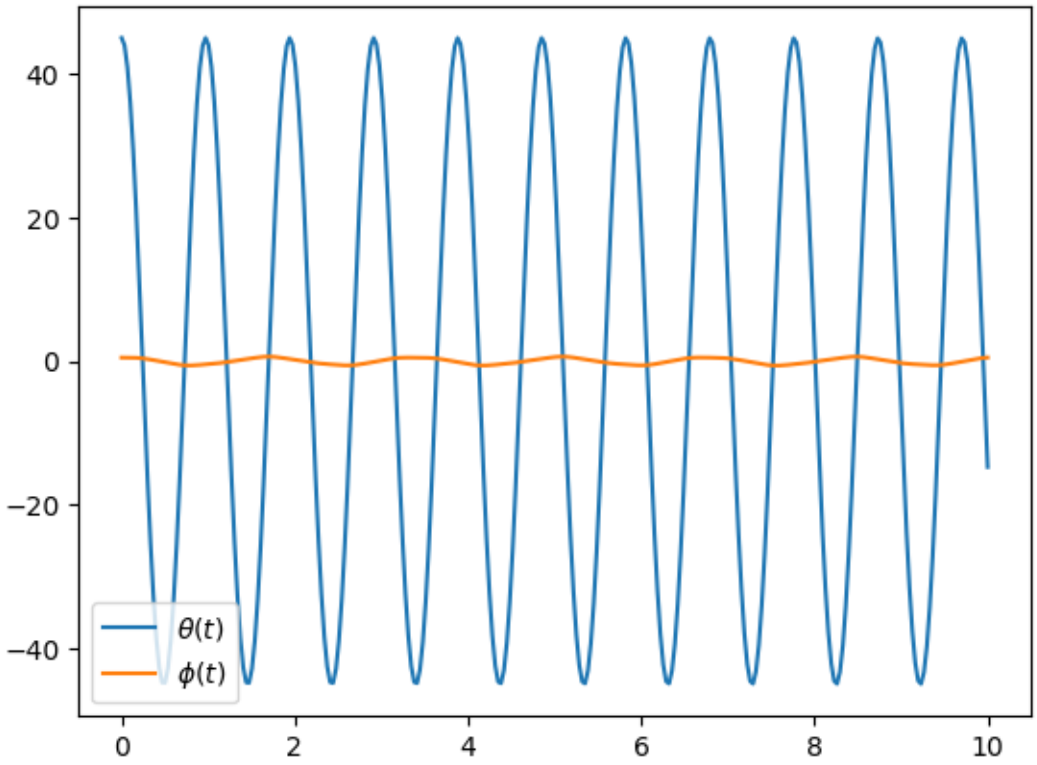
```
sys.times = np.linspace(0.0, 10.0, num=300)
```

The trajectories of the states are found with the `integrate` method.

```
x = sys.integrate()
```

The angles can be plotted to see how they change with respect to time given the initial conditions.

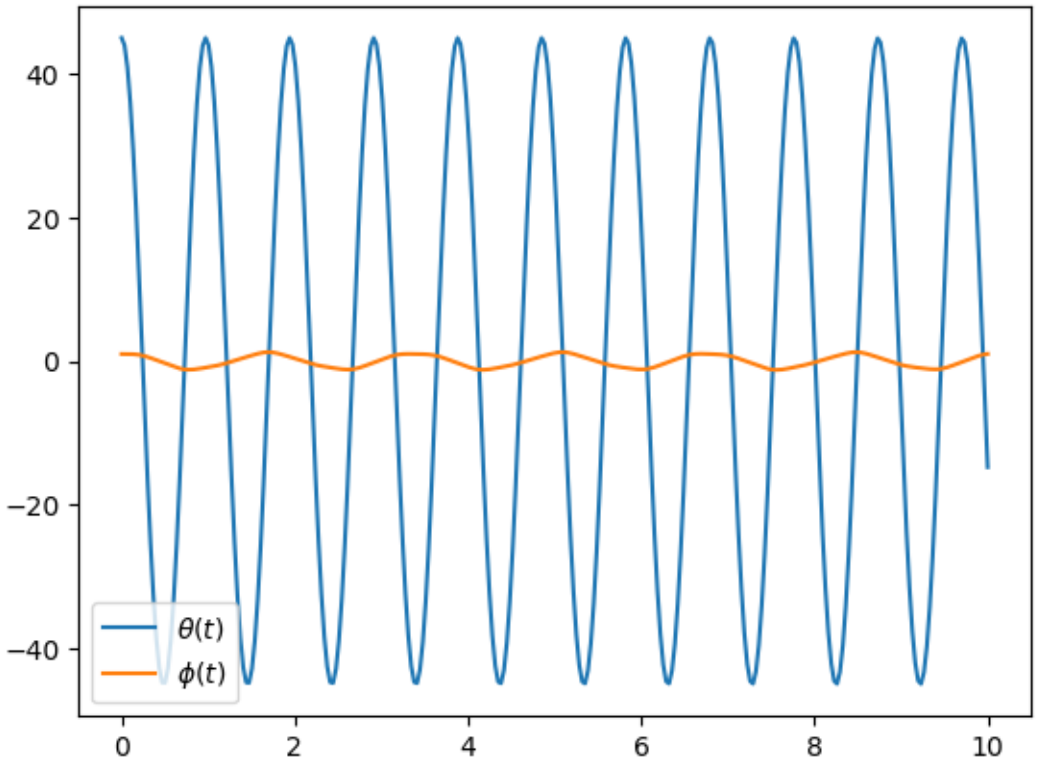
```
def plot():
    plt.figure()
    plt.plot(sys.times, np.rad2deg(x[:, :2]))
    plt.legend([sm.latex(s, mode='inline') for s in sys.coordinates])

plot()
```



1.13.10 Chaotic Behavior

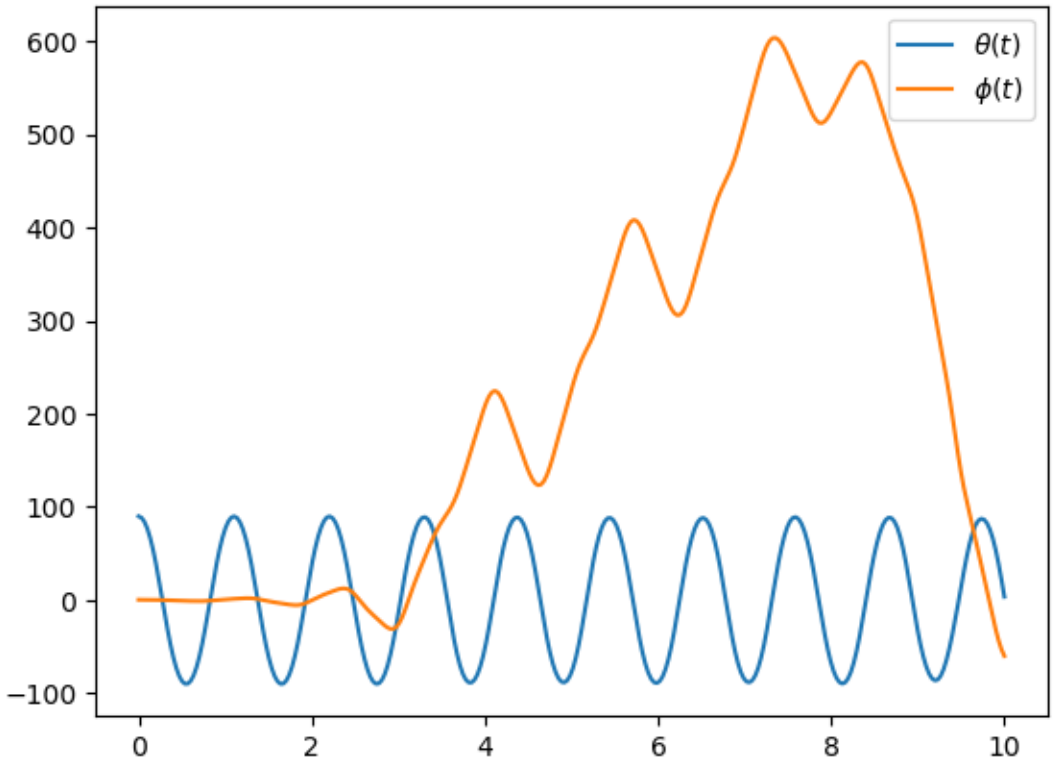
Now change the intial condition of the plat angle just slighty to see if the behvior of the system is similar.

```
sys.initial_conditions[phi] = np.deg2rad(1.0)
x = sys.integrate()
plot()
```



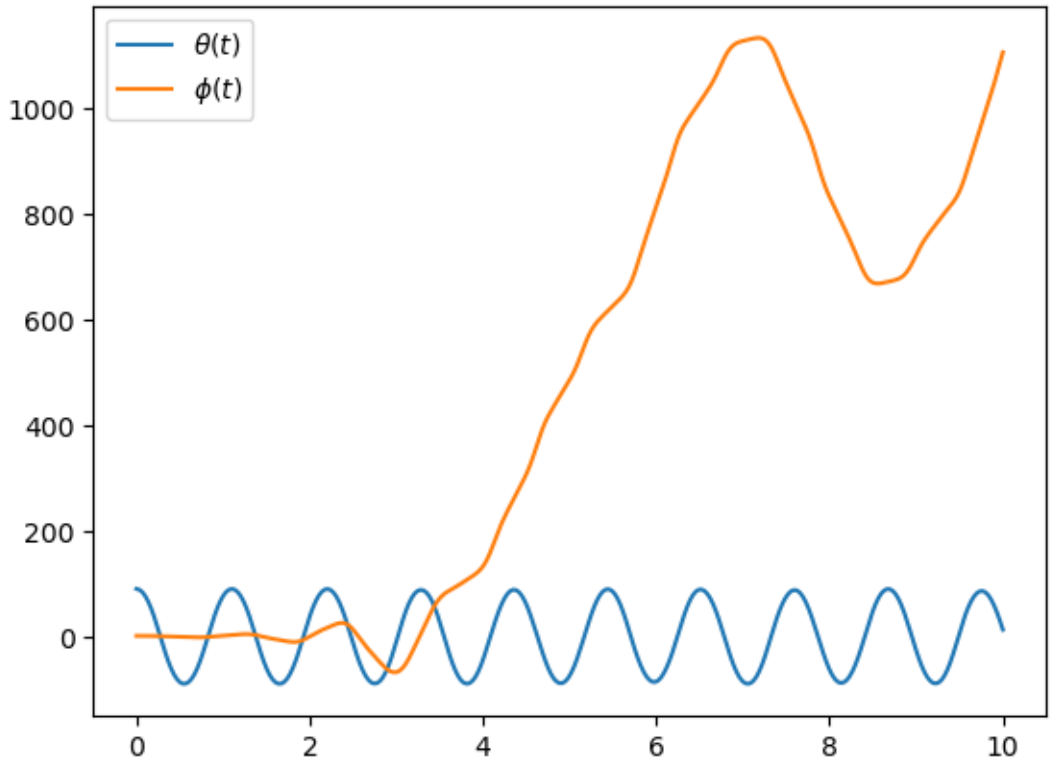
Seems all good, very similar behavior. But now set the rod angle to $90°$ and try the same slight change in plate angle.

```
sys.initial_conditions[theta] = np.deg2rad(90)
sys.initial_conditions[phi] = np.deg2rad(0.5)
x = sys.integrate()
plot()
```



First note that the plate behaves wildly. What happens when the initial plate angle is altered slightly.

```
sys.initial_conditions[phi] = np.deg2rad(1.0)
x = sys.integrate()
plot()
```



The behavior does not look similar to the previous simulation. This is an example of chaotic behavior. The plate angle can not be reliably predicted because slight changes in the initial conditions cause the behavior of the system to vary widely.

1.13.11 Visualization

Finally, the system can be animated by attached a cylinder and a plane shape to the rigid bodies. To properly align the coordinate axes of the shapes with the bodies, simple rotations are used.

```
rod_shape = Cylinder(2 * lA, 0.005, color='red', name='rod')
plate_shape = Plane(w, h, color='blue', name='plate')

v1 = VisualizationFrame('rod',
                        A.orientnew('rod', 'Axis', (sm.pi / 2, A.x)),
                        Ao,
                        rod_shape)

v2 = VisualizationFrame('plate',
                        B.orientnew('plate', 'Body', (sm.pi / 2, sm.pi / 2, 0), 'XZX'),
                        Bo,
                        plate_shape)

scene = Scene(N, No, v1, v2, system=sys)
```

The following method opens up a simple gui that shows a 3D animatoin of the system.

```
scene.display_jupyter(axes_arrow_length=1.0)
```

```
VBox(children=(AnimationAction(clip=AnimationClip(duration=10.0,␣
↪tracks=(VectorKeyframeTrack(name='scene/rod.m...
```

1.14 Exercises from Chapter 2 in Kane and Levinson 1985

1.14.1 Exercise 2.7

Note: You can download this example as a Python script: `ex2-7.py` or Jupyter notebook: `ex2-7.ipynb`.

```
from sympy.physics.mechanics import ReferenceFrame, dynamicsymbols, mprint
from sympy import solve, pi, Eq

q1, q2, q3, q4, q5 = dynamicsymbols('q1 q2 q3 q4 q5')
q1d, q2d, q3d, q4d, q5d = dynamicsymbols('q1 q2 q3 q4 q5', level=1)

ux, uy, uz = dynamicsymbols('ux uy uz')
u1, u2, u3 = dynamicsymbols('u1 u2 u3')

A = ReferenceFrame('A')
B_prime = A.orientnew('B_prime', 'Axis', [q1, A.z])
B = B_prime.orientnew('B', 'Axis', [pi/2 - q2, B_prime.x])
C = B.orientnew('C', 'Axis', [q3, B.z])

# Angular velocity based on coordinate time derivatives
w_C_in_A_qd = C.ang_vel_in(A)

# First definition of Angular velocity
w_C_in_A_uxuyuz = ux * A.x + uy * A.y + uz * A.z
print("Using w_C_A as")
print(w_C_in_A_uxuyuz)
```

```
Using w_C_A as
ux(t)*A.x + uy(t)*A.y + uz(t)*A.z
```

```
kinematic_eqs = [(w_C_in_A_qd - w_C_in_A_uxuyuz) & uv for uv in A]
print("The kinematic equations are:")
soln = solve(kinematic_eqs, [q1d, q2d, q3d])
for qd in [q1d, q2d, q3d]:
    mprint(Eq(qd, soln[qd]))
```

```
The kinematic equations are:
Eq(q1', -ux*sin(q1)*sin(q2)/(sin(q1)**2*cos(q2) + cos(q1)**2*cos(q2)) +␣
↪uy*sin(q2)*cos(q1)/(sin(q1)**2*cos(q2) + cos(q1)**2*cos(q2)) + uz*sin(q1)**2*cos(q2)/
↪(sin(q1)**2*cos(q2) + cos(q1)**2*cos(q2)) + uz*cos(q1)**2*cos(q2)/(sin(q1)**2*cos(q2)␣
↪+ cos(q1)**2*cos(q2)))
Eq(q2', -ux*cos(q1)/(sin(q1)**2 + cos(q1)**2) - uy*sin(q1)/(sin(q1)**2 + cos(q1)**2))
Eq(q3', ux*sin(q1)/(sin(q1)**2*cos(q2) + cos(q1)**2*cos(q2)) - uy*cos(q1)/
↪(sin(q1)**2*cos(q2) + cos(q1)**2*cos(q2)))
```

```
# Second definition of Angular velocity
w_C_in_A_u1u2u3 = u1 * B.x + u2 * B.y + u3 * B.z
print("Using w_C_A as")
print(w_C_in_A_u1u2u3)
```

```
Using w_C_A as
u1(t)*B.x + u2(t)*B.y + u3(t)*B.z
```

```
kinematic_eqs = [(w_C_in_A_qd - w_C_in_A_u1u2u3) & uv for uv in A]
print("The kinematic equations are:")
soln = solve(kinematic_eqs, [q1d, q2d, q3d])
for qd in [q1d, q2d, q3d]:
    mprint(Eq(qd, soln[qd]))
```

```
The kinematic equations are:
```

```
Eq(q1', u2*sin(q2)**2/cos(q2) + u2*cos(q2))
Eq(q2', -u1)
Eq(q3', -u2*sin(q2)/cos(q2) + u3)
```

1.14.2 Exercise 3.10

Note: You can download this example as a Python script: `ex3-10.py` or Jupyter notebook: `ex3-10.ipynb`.

```
from sympy import cancel, collect, expand_trig, solve, symbols, trigsimp
from sympy import sin, cos
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import dot, dynamicsymbols, msprint

q1, q2, q3, q4, q5, q6, q7 = q = dynamicsymbols('q1:8')
u1, u2, u3, u4, u5, u6, u7 = u = dynamicsymbols('q1:8', level=1)

r, theta, b = symbols('r  b', real=True, positive=True)

# define reference frames
R = ReferenceFrame('R') # fixed race rf, let R.z point upwards
A = R.orientnew('A', 'axis', [q7, R.z]) # rf that rotates with S* about R.z
# B.x, B.z are parallel with face of cone, B.y is perpendicular
B = A.orientnew('B', 'axis', [-theta, A.x])
S = ReferenceFrame('S')
S.set_ang_vel(A, u1*A.x + u2*A.y + u3*A.z)
C = ReferenceFrame('C')
C.set_ang_vel(A, u4*B.x + u5*B.y + u6*B.z)

# define points
pO = Point('O')
pS_star = pO.locatenew('S*', b*A.y)
pS_hat = pS_star.locatenew('S^', -r*B.y) # S^ touches the cone
```

(continues on next page)

(continued from previous page)

```
pS1 = pS_star.locatenew('S1', -r*A.z) # S1 touches horizontal wall of the race
pS2 = pS_star.locatenew('S2', r*A.y) # S2 touches vertical wall of the race

pO.set_vel(R, 0)
pS_star.v2pt_theory(pO, R, A)
pS1.v2pt_theory(pS_star, R, S)
pS2.v2pt_theory(pS_star, R, S)

# Since S is rolling against R, v_S1_R = 0, v_S2_R = 0.
vc = [dot(p.vel(R), basis) for p in [pS1, pS2] for basis in R]

pO.set_vel(C, 0)
pS_star.v2pt_theory(pO, C, A)
pS_hat.v2pt_theory(pS_star, C, S)

# Since S is rolling against C, v_S^_C = 0.
# Cone has only angular velocity in R.z direction.
vc += [dot(pS_hat.vel(C), basis) for basis in A]
vc += [dot(C.ang_vel_in(R), basis) for basis in [R.x, R.y]]
vc_map = solve(vc, u)

# Pure rolling between S and C, dot(_C_S, B.y) = 0.
b_val = solve([dot(C.ang_vel_in(S), B.y).subs(vc_map).simplify()], b)[0][0]
print('b = {0}'.format(msprint(collect(cancel(expand_trig(b_val)), r))))
```

```
b = r*(sin() + 1)/(-sin() + cos())
```

```
b_expected = r*(1 + sin(theta))/(cos(theta) - sin(theta))
assert trigsimp(b_val - b_expected) == 0
```

1.14.3 Exercise 3.15

Note: You can download this example as a Python script: `ex3-15.py` or Jupyter notebook: `ex3-15.ipynb`.

A robot arm, composed of Rigid Bodies 'A', 'B', 'C', operates in Reference Frame E. 'A*', 'B*', 'C*' are Points marking the centers of mass for the Rigid Bodies 'A', 'B', 'C'.

Rigid Body 'D' also is lies in Reference Frame 'E'. The center of mass of 'D' is marked as Point 'D*'. 'D' is fixed relative to 'C'.

Each Reference Frame has a set of mutually perpendicular vectors x, y, z. 'A' is rotated by 'q0' relative to 'E' about an axis parallel to A.x. 'B' is rotated by 'q1' relative to 'A' about an axis parallel to A.y. Point 'P' is fixed in both 'A' and 'B'. A.x is parallel to E.x. A.y is parallel to B.y.

Point 'O' is a point fixed in both 'E' and 'A'. 'A*' is separated from 'O' by 'LA' * A.z. 'P' is separated from 'O' by 'LP' * A.z. 'B*' is separated from 'P' by 'LB' * B.z. 'C*' is separated from 'B*' by 'q2' * B.z. 'D*' is separated from 'C*' by `p1*B.x + p2*B.y + p3*B.z`.

We define:'q0d' = 'u1', 'q1d' = 'u2', 'q2d' = 'u3'. 'LA', 'LB', 'LP', 'p1', 'p2', 'p3' are constants.

```
from sympy.physics.mechanics import dynamicsymbols, msprint
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy import solve, symbols

# Define generalized coordinates, speeds, and constants:
q0, q1, q2 = dynamicsymbols('q0 q1 q2')
q0d, q1d, q2d = dynamicsymbols('q0 q1 q2', level=1)
u1, u2, u3 = dynamicsymbols('u1 u2 u3')
LA, LB, LP = symbols('LA LB LP')
p1, p2, p3 = symbols('p1 p2 p3')

E = ReferenceFrame('E')
# A.x of Rigid Body A is fixed in Reference Frame E and is rotated by q0.
A = E.orientnew('A', 'Axis', [q0, E.x])
# B.y of Rigid Body B is fixed in Reference Frame A and is rotated by q1.
B = A.orientnew('B', 'Axis', [q1, A.y])
# Reference Frame C has no rotation relative to Reference Frame B.
C = B.orientnew('C', 'Axis', [0, B.x])
# Reference Frame D has no rotation relative to Reference Frame C.
D = C.orientnew('D', 'Axis', [0, C.x])

pO = Point('O')
# The vector from Point O to Point A*, the center of mass of A, is LA * A.z.
pAs = pO.locatenew('A*', LA * A.z)
# The vector from Point O to Point P, which lies on the axis where
# B rotates about A, is LP * A.z.
pP = pO.locatenew('P', LP * A.z)
# The vector from Point P to Point B*, the center of mass of B, is LB * B.z.
pBs = pP.locatenew('B*', LB * B.z)
# The vector from Point B* to Point C*, the center of mass of C, is q2 * B.z.
pCs = pBs.locatenew('C*', q2 * B.z)
# The vector from Point C* to Point D*, the center of mass of D,
# is p1 * B.x + p2 * B.y + p3 * B.z.
pDs = pCs.locatenew('D*', p1 * B.x + p2 * B.y + p3 * B.z)

# Define generalized speeds as:
# u1 = q0d
# u2 = q1d
# u3 = q2d
A.set_ang_vel(E, u1 * A.x) # A.x = E.x
B.set_ang_vel(A, u2 * B.y) # B.y = A.y
pCs.set_vel(B, u3 * B.z)

pO.set_vel(E, 0) # Point O is fixed in Reference Frame E
pAs.v2pt_theory(pO, E, A) # Point A* is fixed in Reference Frame A
pP.v2pt_theory(pO, E, A) # Point P is fixed in Reference Frame A
pBs.v2pt_theory(pP, E, B) # Point B* is fixed in Reference Frame B
pCs.v1pt_theory(pBs, E, B) # Point C* is moving in Reference Frame B
pDs.set_vel(B, pCs.vel(B)) # Point D* is fixed relative to Point C* in B
pDs.v1pt_theory(pBs, E, B) # Point D* is moving in Reference Frame B

# Write generalized speeds as kinematic equations:
kinematic_eqs = []
```

(continues on next page)

(continued from previous page)

```
kinematic_eqs.append(u1 - q0d)
kinematic_eqs.append(u2 - q1d)
kinematic_eqs.append(u3 - q2d)
soln = solve(kinematic_eqs, [q0d, q1d, q2d])
print("kinematic equations:")
for qd in [q0d, q1d, q2d]:
   print("{0} = {1}".format(msprint(qd), msprint(soln[qd])))
```

```
kinematic equations:
q0' = u1
q1' = u2
q2' = u3
```

```
ang_vels = ["\nangular velocities:"]
ang_accs = ["\nangular accelerations:"]
for rf in [A, B, C, D]:
   ang_v = getattr(rf, 'ang_vel_in')(E)
   ang_a = getattr(rf, 'ang_acc_in')(E)
   express_rf = B
   if rf == A:
      express_rf = A
   ang_vels.append("ang vel {0} wrt {1} = {2}".format(
            rf, E, ang_v.express(express_rf)))
   ang_accs.append("ang acc {0} wrt {1} = {2}".format(
            rf, E, ang_a.express(express_rf)))

vels = ["\nvelocities:"]
accs = ["\naccelerations:"]
for point in [pAs, pBs, pCs, pDs]:
   v = getattr(point, 'vel')(E)
   a = getattr(point, 'acc')(E)
   express_rf = B
   if point == pAs:
      express_rf = A
   vels.append("vel {0} wrt {1} = {2}".format(
            point, E, v.express(express_rf)))
   accs.append("acc {0} wrt {1} = {2}".format(
            point, E, a.express(express_rf)))

for results in ang_vels + ang_accs + vels + accs:
   print(results)
```

```

angular velocities:
ang vel A wrt E = u1(t)*A.x
ang vel B wrt E = u1(t)*cos(q1(t))*B.x + u2(t)*B.y + u1(t)*sin(q1(t))*B.z
ang vel C wrt E = u1(t)*cos(q1(t))*B.x + u2(t)*B.y + u1(t)*sin(q1(t))*B.z
ang vel D wrt E = u1(t)*cos(q1(t))*B.x + u2(t)*B.y + u1(t)*sin(q1(t))*B.z

angular accelerations:
ang acc A wrt E = Derivative(u1(t), t)*A.x
```

(continues on next page)

(continued from previous page)

```
ang acc B wrt E = (-u1(t)*u2(t)*sin(q1(t)) + cos(q1(t))*Derivative(u1(t), t))*B.x +␣
↪Derivative(u2(t), t)*B.y + (u1(t)*u2(t)*cos(q1(t)) + sin(q1(t))*Derivative(u1(t),␣
↪t))*B.z
ang acc C wrt E = (-u1(t)*u2(t)*sin(q1(t)) + cos(q1(t))*Derivative(u1(t), t))*B.x +␣
↪Derivative(u2(t), t)*B.y + (u1(t)*u2(t)*cos(q1(t)) + sin(q1(t))*Derivative(u1(t),␣
↪t))*B.z
ang acc D wrt E = (-u1(t)*u2(t)*sin(q1(t)) + cos(q1(t))*Derivative(u1(t), t))*B.x +␣
↪Derivative(u2(t), t)*B.y + (u1(t)*u2(t)*cos(q1(t)) + sin(q1(t))*Derivative(u1(t),␣
↪t))*B.z

velocities:
vel A* wrt E = - LA*u1(t)*A.y
vel B* wrt E = LB*u2(t)*B.x + (-LB*u1(t)*cos(q1(t)) - LP*u1(t))*B.y
vel C* wrt E = (LB*u2(t) + q2(t)*u2(t))*B.x + (-LB*u1(t)*cos(q1(t)) - LP*u1(t) -␣
↪q2(t)*u1(t)*cos(q1(t)))*B.y + u3(t)*B.z
vel D* wrt E = (LB*u2(t) - p2*u1(t)*sin(q1(t)) + (p3 + q2(t))*u2(t))*B.x + (-
↪LB*u1(t)*cos(q1(t)) - LP*u1(t) + p1*u1(t)*sin(q1(t)) - (p3 +␣
↪q2(t))*u1(t)*cos(q1(t)))*B.y + (-p1*u2(t) + p2*u1(t)*cos(q1(t)) + u3(t))*B.z

accelerations:
acc A* wrt E = - LA*Derivative(u1(t), t)*A.y - LA*u1(t)**2*A.z
acc B* wrt E = (LB*u1(t)**2*sin(q1(t))*cos(q1(t)) + LB*Derivative(u2(t), t) +␣
↪LP*u1(t)**2*sin(q1(t)))*B.x + (LB*u1(t)*u2(t)*sin(q1(t)) +␣
↪LB*u1(t)*sin(q1(t))*Derivative(q1(t), t) - LB*cos(q1(t))*Derivative(u1(t), t) -␣
↪LP*Derivative(u1(t), t))*B.y + (-LB*u1(t)**2*cos(q1(t))**2 - LB*u2(t)**2 -␣
↪LP*u1(t)**2*cos(q1(t)))*B.z
acc C* wrt E = (LB*Derivative(u2(t), t) + LP*u1(t)**2*sin(q1(t)) - (-LB*u1(t)*cos(q1(t))␣
↪- q2(t)*u1(t)*cos(q1(t)))*u1(t)*sin(q1(t)) + q2(t)*Derivative(u2(t), t) + u2(t)*u3(t)␣
↪+ u2(t)*Derivative(q2(t), t))*B.x + (LB*u1(t)*sin(q1(t))*Derivative(q1(t), t) -␣
↪LB*cos(q1(t))*Derivative(u1(t), t) - LP*Derivative(u1(t), t) + (LB*u2(t) +␣
↪q2(t)*u2(t))*u1(t)*sin(q1(t)) + q2(t)*u1(t)*sin(q1(t))*Derivative(q1(t), t) -␣
↪q2(t)*cos(q1(t))*Derivative(u1(t), t) - u1(t)*u3(t)*cos(q1(t)) -␣
↪u1(t)*cos(q1(t))*Derivative(q2(t), t))*B.y + (-LP*u1(t)**2*cos(q1(t)) - (LB*u2(t) +␣
↪q2(t)*u2(t))*u2(t) + (-LB*u1(t)*cos(q1(t)) - q2(t)*u1(t)*cos(q1(t)))*u1(t)*cos(q1(t))␣
↪+ Derivative(u3(t), t))*B.z
acc D* wrt E = (LB*Derivative(u2(t), t) + LP*u1(t)**2*sin(q1(t)) -␣
↪p2*u1(t)*cos(q1(t))*Derivative(q1(t), t) - p2*sin(q1(t))*Derivative(u1(t), t) + (p3 +␣
↪q2(t))*Derivative(u2(t), t) + (-p1*u2(t) + p2*u1(t)*cos(q1(t)) + u3(t))*u2(t) - (-
↪LB*u1(t)*cos(q1(t)) + p1*u1(t)*sin(q1(t)) - (p3 +␣
↪q2(t))*u1(t)*cos(q1(t)))*u1(t)*sin(q1(t)) + u2(t)*Derivative(q2(t), t))*B.x +␣
↪(LB*u1(t)*sin(q1(t))*Derivative(q1(t), t) - LB*cos(q1(t))*Derivative(u1(t), t) -␣
↪LP*Derivative(u1(t), t) + p1*u1(t)*cos(q1(t))*Derivative(q1(t), t) +␣
↪p1*sin(q1(t))*Derivative(u1(t), t) + (p3 + q2(t))*u1(t)*sin(q1(t))*Derivative(q1(t),␣
↪t) - (p3 + q2(t))*cos(q1(t))*Derivative(u1(t), t) + (LB*u2(t) - p2*u1(t)*sin(q1(t)) +␣
↪(p3 + q2(t))*u2(t))*u1(t)*sin(q1(t)) - (-p1*u2(t) + p2*u1(t)*cos(q1(t)) +␣
↪u3(t))*u1(t)*cos(q1(t)) - u1(t)*cos(q1(t))*Derivative(q2(t), t))*B.y + (-
↪LP*u1(t)**2*cos(q1(t)) - p1*Derivative(u2(t), t) -␣
↪p2*u1(t)*sin(q1(t))*Derivative(q1(t), t) + p2*cos(q1(t))*Derivative(u1(t), t) -␣
↪(LB*u2(t) - p2*u1(t)*sin(q1(t)) + (p3 + q2(t))*u2(t))*u2(t) + (-LB*u1(t)*cos(q1(t)) +␣
↪p1*u1(t)*sin(q1(t)) - (p3 + q2(t))*u1(t)*cos(q1(t)))*u1(t)*cos(q1(t)) +␣
↪Derivative(u3(t), t))*B.z
```

1.14.4 Exercise 4.1

Note: You can download this example as a Python script: `ex4-1.py` or Jupyter notebook: `ex4-1.ipynb`.

```
from sympy.physics.mechanics import dot, msprint
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy import symbols, sin, cos
from sympy.simplify.simplify import trigsimp

theta = symbols('theta:3')
x = symbols('x:3')
q = symbols('q')

A = ReferenceFrame('A')
B = A.orientnew('B', 'SPACE', theta, 'xyz')

O = Point('O')
P = O.locatenew('P', x[0] * A.x + x[1] * A.y + x[2] * A.z)
p = P.pos_from(O)

# From problem, point P is on L (span(B.x)) when:
constraint_eqs = {x[0] : q*cos(theta[1])*cos(theta[2]),
                  x[1] : q*cos(theta[1])*sin(theta[2]),
                  x[2] : -q*sin(theta[1])}

# If point P is on line L then r^{P/O} will have no components in the B.y or
# B.z directions since point O is also on line L and B.x is parallel to L.
assert(trigsimp(dot(P.pos_from(O), B.y).subs(constraint_eqs)) == 0)
assert(trigsimp(dot(P.pos_from(O), B.z).subs(constraint_eqs)) == 0)
```

1.14.5 Exercise 4.18

Note: You can download this example as a Python script: `ex4-18.py` or Jupyter notebook: `ex4-18.ipynb`.

```
from sympy.physics.mechanics import dynamicsymbols, msprint
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy import solve, symbols, pi

# Define generalized coordinates, speeds, and constants:
qi = dynamicsymbols('q0 q1 q2 q3 q4 q5')
qid = dynamicsymbols('q0 q1 q2 q3 q4 q5', level=1)
ui = dynamicsymbols('u0 u1 u2 u3 u4 u5')
R = symbols('R')

A = ReferenceFrame('A')
A_1 = A.orientnew("A'", 'Axis', [qi[1], A.z])
B = A_1.orientnew('B', 'Axis', [pi/2 - qi[2], A_1.x])
C = B.orientnew('C', 'Axis', [qi[3], B.z])
```

(continues on next page)

(continued from previous page)

```
pO = Point('O')
pCs = pO.locatenew('C*', qi[4] * A.x + qi[5] * A.y + R * B.y)

pO.set_vel(A, 0) # Point O is fixed in Reference Frame A
pCs.v2pt_theory(pO, A, B) # Point C* is fixed in Reference Frame B

# Set ui = qid
kinematic_eqs = []
for u, qd in zip(ui, qid):
   kinematic_eqs.append(u - qd)
soln = solve(kinematic_eqs, qid)
print("kinematic equations:")
for qd in qid:
   print("{0} = {1}".format(msprint(qd), msprint(soln[qd])))

print("\nposition of C* from O = {0}".format(msprint(pCs.pos_from(pO))))
print("\nvelocity of C* wrt A = {0}".format(msprint(pCs.vel(A).express(B))))
```

```
kinematic equations:
q0' = u0
q1' = u1
q2' = u2
q3' = u3
q4' = u4
q5' = u5

position of C* from O = q4*A.x + q5*A.y + R*B.y

velocity of C* wrt A = (-R*sin(q2)*q1' + q4*sin(q1)*q1' - q5*cos(q1)*q1')*B.x +␣
↪((q4*sin(q1)*q2' - q5*cos(q1)*q2')*cos(q2) + q4*sin(q2)*cos(q1)*q1' +␣
↪q5*sin(q1)*sin(q2)*q1')*B.y + (-R*q2' + (q4*sin(q1)*q2' - q5*cos(q1)*q2')*sin(q2) -␣
↪q4*cos(q1)*cos(q2)*q1' - q5*sin(q1)*cos(q2)*q1')*B.z
```

1.15 Exercises from Chapter 3 in Kane and Levinson 1985

1.15.1 Exercise 5.1

Note: You can download this example as a Python script: `ex5-1.py` or Jupyter notebook: `ex5-1.ipynb`.

```
from sympy import symbols, integrate
from sympy import pi, acos
from sympy import Matrix, S, simplify
from sympy.physics.mechanics import dot, ReferenceFrame
import scipy.integrate
```

(continues on next page)

(continued from previous page)

```
def eval_v(v, N):
    return sum(dot(v, n).evalf() * n for n in N)


def integrate_v(integrand, rf, bounds):
    """Return the integral for a Vector integrand."""
    # integration problems if option meijerg=True is not used
    return sum(simplify(integrate(dot(integrand, n),
                                  bounds,
                                  meijerg=True)) * n for n in rf)

m, R = symbols('m R', real=True, nonnegative=True)
theta, r, x, y, z = symbols('theta r x y z', real=True)
```

Regarding Fig. P5.1 as showing two views of a body B formed by matter distributed uniformly (a) over a surface having no planar portions and (b) throughout a solid, determine (by integration) the coordinates x*, y*, z* of the mass center of B.

```
A = ReferenceFrame('A')

# part a
print("a) mass center of surface area")
SA = (2 * pi * R) * 2
rho_a = m / SA

B = A.orientnew('B', 'Axis', [theta, A.x])
p = x * A.x  + r * B.y
y_ = dot(p, A.y)
z_ = dot(p, A.z)
J = Matrix([x, y_, z_]).jacobian([x, r, theta])
dJ = simplify(J.det())
print("dJ = {0}".format(dJ))

# calculate center of mass for the cut cylinder
# ranges from x = [1, 3], y = [-1, 1], z = [-1, 1] in Fig. P5.1
mass_cc_a = rho_a * 2*pi*R
cm_cc_a = (integrate_v(integrate_v((rho_a * p * dJ).subs(r, R),
                                   A, (theta, acos(1-x),
                                   2*pi - acos(1-x))),
                                   A, (x, 0, 2)) / mass_cc_a + A.x)
print("cm = {0}".format(cm_cc_a))
```

```
a) mass center of surface area
dJ = r
```

```
cm = 7/4*A.x - R/2*A.y
```

```
mass_cyl_a = rho_a * 2*pi*R
cm_cyl_a = A.x/S(2)

cm_a = (mass_cyl_a*cm_cyl_a + mass_cc_a*cm_cc_a) / m
```

(continues on next page)

(continued from previous page)

```
print("center of mass = {0}".format(cm_a.subs(R, 1)))
# part b
print("b) mass center of volume")
V = (pi*R**2) * 2
rho_b = m / V
mass_cc_b = rho_b * pi*R**2

# calculate center of mass for the cut cylinder
# ranges from x = [1, 3], y = [-1, 1], z = [-1, 1] in Fig. P5.1
# compute the value using scipy due to issues with sympy
R_ = 1
def pos(z, y, x, coord):
   if coord == 0:
      return x
   elif coord == 1:
      return y
   elif coord == 2:
      return z
   else:
      raise ValueError
# y bounds
def ybu(x):
   return R_*(1 - x)
def ybl(x):
   return -R_
# z bounds
def zbu(x, y):
   return (R_**2 - y**2)**0.5
def zbl(x, y):
   return -1 * zbu(x, y)

cm_cc_b = 0
for i, b in enumerate(A):
   p_i = lambda z, y, x: pos(z, y, x, i)
   cm_cc_b += scipy.integrate.tplquad(p_i, 0, 2, ybl, ybu, zbl, zbu)[0] * b
cm_cc_b *= (rho_b / mass_cc_b).subs(R, R_)
cm_cc_b += A.x

#integrand = rho_b * (x*A.x + y*A.y + z*A.z)
#cm_cc_b = (integrate_v(integrate_v(integrate_v(integrand,
#                                               A, (z,
#                                                   -sqrt(R**2 - y**2),
#                                                   sqrt(R**2 - y**2))),
#                                    A, (y, -R, R*(1 - x))),
#                        A, (x, 0, 2)) / mass_cc_b +
#           A.x)
print("cm = {0}".format(cm_cc_b))

mass_cyl_b = rho_b * pi*R**2
cm_cyl_b = A.x/S(2)

cm_b = (mass_cyl_b*cm_cyl_b + mass_cc_b*cm_cc_b) / m
```

(continues on next page)

(continued from previous page)

```
print("center of mass = {0}".format(eval_v(cm_b.subs(R, 1), A)))
```

```
center of mass = 9/8*A.x - 1/4*A.y
b) mass center of volume
```

```
cm = (1.96349540850478/pi + 1)*A.x - 0.785398163387119/pi*A.y
center of mass = 1.06250000000178*A.x - 0.124999999998356*A.y
```

1.15.2 Exercise 5.4

Note: You can download this example as a Python script: `ex5-4.py` or Jupyter notebook: `ex5-4.ipynb`.

```
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import dot
from sympy import symbols, integrate
from sympy import sin, cos, pi

a, b, R = symbols('a b R', real=True, nonnegative=True)
theta, x, y, z = symbols('theta x y z', real=True)
ab_val = {a: 0.3, b: 0.3}

# R is the radius of the circle, theta is the half angle.
centroid_sector = 2*R*sin(theta) / (3 * theta)

# common R, theta values
theta_pi_4 = {theta: pi/4, R: a}
R_theta_val = {theta: pi/4 * (1 - z/a), R: a}

N = ReferenceFrame('N')
def eval_vec(v):
    vs = v.subs(ab_val)
    return sum(dot(vs, n).evalf() * n for n in N)

# For each part A, B, C, D, define an origin for that part such that the
# centers of mass of each part of the component have positive N.x, N.y,
# and N.z values.
## FIND CENTER OF MASS OF A
vol_A_1 = pi * a**2 * b / 4
vol_A_2 = pi * a**2 * a / 4 / 2
vol_A = vol_A_1 + vol_A_2
pA_O = Point('A_O')
pAs_1 = pA_O.locatenew(
      'A*_1', (b/2 * N.z +
               centroid_sector.subs(theta_pi_4) * sin(pi/4) * (N.x + N.y)))
pAs_2 = pA_O.locatenew(
      'A*_2', (b * N.z +
               N.z * integrate((theta*R**2*(z)).subs(R_theta_val),
                               (z, 0, a)) / vol_A_2 +
```

(continues on next page)

(continued from previous page)

```
                N.x * integrate((theta*R**2 * cos(theta) *
                                 centroid_sector).subs(R_theta_val),
                                 (z, 0, a)) / vol_A_2 +
                N.y * integrate((2*R**3/3 * 4*a/pi *
                                 sin(theta)**2).subs(R, a),
                                 (theta, 0, pi/4)) / vol_A_2))
pAs = pA_0.locatenew('A*', ((pAs_1.pos_from(pA_0) * vol_A_1 +
                             pAs_2.pos_from(pA_0) * vol_A_2) /
                             vol_A))
print('A* = {0}'.format(pAs.pos_from(pA_0)))
print('A* = {0}'.format(eval_vec(pAs.pos_from(pA_0))))

## FIND CENTER OF MASS OF B
vol_B_1 = pi*a**2/2
vol_B_2 = a**2 / 2
vol_B  = vol_B_1 + vol_B_2
pB_0 = Point('B_0')
pBs_1 = pB_0.locatenew(
      'B*_1', (a*(N.x + N.z) + a/2*N.y +
                (-N.x + N.z) * (R*sin(theta)/theta *
                                 sin(pi/4)).subs(theta_pi_4)))
pBs_2 = pB_0.locatenew('B*_2', (a*N.y + a*N.z -
                                (a/3 * N.y + a/3 * N.z)))
pBs = pB_0.locatenew('B*', ((pBs_1.pos_from(pB_0) * vol_B_1 +
                             pBs_2.pos_from(pB_0) * vol_B_2) /
                             vol_B))
print('\nB* = {0}'.format(pBs.pos_from(pB_0)))
print('B* = {0}'.format(eval_vec(pBs.pos_from(pB_0))))

## FIND CENTER OF MASS OF C
vol_C_1 = 2 * a**2 * b
vol_C_2 = a**3 / 2
vol_C_3 = a**3
vol_C_4 = -pi*a**3/4
vol_C = vol_C_1 + vol_C_2 + vol_C_3 + vol_C_4
pC_0 = Point('C_0')
pCs_1 = pC_0.locatenew('C*_1', (a*N.x + a/2*N.y + b/2*N.z))
pCs_2 = pC_0.locatenew('C*_2', (a*N.x + b*N.z +
                                (a/3*N.x + a/2*N.y + a/3*N.z)))
pCs_3 = pC_0.locatenew('C*_3', (b*N.z + a/2*(N.x + N.y + N.z)))
pCs_4 = pC_0.locatenew(
      'C*_4', ((a + b)*N.z + a/2*N.y +
                (N.x - N.z)*(centroid_sector.subs(
                        theta_pi_4)*sin(pi/4))))
pCs = pC_0.locatenew('C*', ((pCs_1.pos_from(pC_0)*vol_C_1 +
                             pCs_2.pos_from(pC_0)*vol_C_2 +
                             pCs_3.pos_from(pC_0)*vol_C_3 +
                             pCs_4.pos_from(pC_0)*vol_C_4) /
                             vol_C))
print('\nC* = {0}'.format(pCs.pos_from(pC_0)))
print('C* = {0}'.format(eval_vec(pCs.pos_from(pC_0))))
```

(continues on next page)

(continued from previous page)

```
## FIND CENTER OF MASS OF D
vol_D = pi*a**3/4
pD_O = Point('D_O')
pDs = pD_O.locatenew('D*', (a*N.z + a/2*N.y +
                            (N.x - N.z)*(centroid_sector.subs(
                                      theta_pi_4) * sin(pi/4))))
print('\nD* = {0}'.format(pDs.pos_from(pD_O)))
print('D* = {0}'.format(eval_vec(pDs.pos_from(pD_O))))

## FIND CENTER OF MASS OF ASSEMBLY
pO = Point('O')
pA_O.set_pos(pO, 2*a*N.x - (a+b)*N.z)
pB_O.set_pos(pO, 2*a*N.x - a*N.z)
pC_O.set_pos(pO, -(a+b)*N.z)
pD_O.set_pos(pO, -a*N.z)

density_A = 7800
density_B = 17.00
density_C = 2700
density_D = 8400
mass_A = vol_A * density_A
mass_B = vol_B * density_B
mass_C = vol_C * density_C
mass_D = vol_D * density_D

pms = pO.locatenew('m*', ((pAs.pos_from(pO)*mass_A + pBs.pos_from(pO)*mass_B +
                           pCs.pos_from(pO)*mass_C + pDs.pos_from(pO)*mass_D) /
                          (mass_A + mass_B + mass_C + mass_D)))
print('\nm* = {0}'.format(eval_vec(pms.pos_from(pO))))
```

```
A* = (2*a**4/(3*pi) + a**3*b/3)/(pi*a**3/8 + pi*a**2*b/4)*N.x + (8*a**4*(-1/4 + pi/8)/
↪(3*pi) + a**3*b/3)/(pi*a**3/8 + pi*a**2*b/4)*N.y + (pi*a**3*(a/3 + b)/8 + pi*a**2*b**2/
↪8)/(pi*a**3/8 + pi*a**2*b/4)*N.z
A* = 0.138920600924924*N.x + 0.115727308022108*N.y + 0.233333333333333*N.z

B* = pi*a**2*(-2*a/pi + a)/(2*(a**2/2 + pi*a**2/2))*N.x + (a**3/3 + pi*a**3/4)/(a**2/2 +␣
↪pi*a**2/2)*N.y + (a**3/3 + pi*a**2*(2*a/pi + a)/2)/(a**2/2 + pi*a**2/2)*N.z
B* = 0.0826922936952985*N.x + 0.162072650350261*N.y + 0.420726503502612*N.z

C* = (5*a**4/6 + 2*a**3*b)/(-pi*a**3/4 + 3*a**3/2 + 2*a**2*b)*N.x + (-pi*a**4/8 + 3*a**4/
↪4 + a**3*b)/(-pi*a**3/4 + 3*a**3/2 + 2*a**2*b)*N.y + (a**3*(a/3 + b)/2 + a**3*(a/2 +␣
↪b) - pi*a**3*(-4*a/(3*pi) + a + b)/4 + a**2*b**2)/(-pi*a**3/4 + 3*a**3/2 + 2*a**2*b)*N.
↪z
C* = 0.313121426700209*N.x + 0.15*N.y + 0.213202943487976*N.z

D* = 4*a/(3*pi)*N.x + a/2*N.y + (-4*a/(3*pi) + a)*N.z
D* = 0.127323954473516*N.x + 0.15*N.y + 0.172676045526484*N.z

m* = 0.430639672235866*N.x + 0.136505537542152*N.y - 0.302591520505184*N.z
```

1.15.3 Exercise 5.8

Note: You can download this example as a Python script: `ex5-8.py` or Jupyter notebook: `ex5-8.ipynb`.

```
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import cross, dot
from sympy import integrate, simplify, symbols, integrate
from sympy import sin, cos, pi


def calc_inertia_vec(rho, p, n_a, N, iv):
    integrand = rho * cross(p, cross(n_a, p))
    return sum(simplify(integrate(dot(integrand, n), iv)) * n
               for n in N)


a, b, L, l, m, h = symbols('a b L l m h', real=True, nonnegative=True)
theta = symbols('theta', real=True)
h_theta_val = {h:b*l/L, theta:2*pi*l/L}

density = m/L
N = ReferenceFrame('N')
pO = Point('O')
pP = pO.locatenew('P', h*N.x + a*cos(theta)*N.y + a*sin(theta)*N.z)

I_1 = calc_inertia_vec(density, pP.pos_from(pO).subs(h_theta_val),
                       N.x, N, (l, 0, L))
I_2 = calc_inertia_vec(density, pP.pos_from(pO).subs(h_theta_val),
                       N.y, N, (l, 0, L))
print('I_1 = {0}'.format(I_1))
print('I_2 = {0}'.format(I_2))
```

```
I_1 = a**2*m*N.x + a*b*m/(2*pi)*N.z
I_2 = m*(3*a**2 + 2*b**2)/6*N.y
```

1.15.4 Exercise 5.12

Note: You can download this example as a Python script: `ex5-12.py` or Jupyter notebook: `ex5-12.ipynb`.

```
import numpy as np

# n_a = 3/5*n_1 - 4/5*n_3. Substituting n_i for e_i results in
# n_a = 4/5*e_1 + 3/5*n_2.
a = np.matrix([4/5, 3/5, 0])
Iij = np.matrix([[169, 144, -96],
                 [144, 260, 72],
                 [-96, 72, 325]])
```

(continues on next page)

(continued from previous page)

```
print("Moment of inertia of B with respect to a line that is parallel to")
print("line PQ and passes through point O.")
print("{0} kg m**2".format((a * Iij * a.T).item(0)))
```

```
Moment of inertia of B with respect to a line that is parallel to
line PQ and passes through point O.
340.0 kg m**2
```

1.15.5 Exercise 6.6

Note: You can download this example as a Python script: `ex6-6.py` or Jupyter notebook: `ex6-6.ipynb`.

```
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import inertia, inertia_of_point_mass
from sympy.physics.mechanics import dot
from sympy import symbols
from sympy import S

m = symbols('m')
m_val = 12

N = ReferenceFrame('N')
pO = Point('O')
pBs = pO.locatenew('B*', -3*N.x + 2*N.y - 4*N.z)

I_B_O = inertia(N, 260*m/m_val, 325*m/m_val, 169*m/m_val,
                72*m/m_val, 96*m/m_val, -144*m/m_val)
print("I_B_rel_O = {0}".format(I_B_O))

I_Bs_O = inertia_of_point_mass(m, pBs.pos_from(pO), N)
print("\nI_B*_rel_O = {0}".format(I_Bs_O))

I_B_Bs = I_B_O - I_Bs_O
print("\nI_B_rel_B* = {0}".format(I_B_Bs))

pQ = pO.locatenew('Q', -4*N.z)
I_Bs_Q = inertia_of_point_mass(m, pBs.pos_from(pQ), N)
print("\nI_B*_rel_Q = {0}".format(I_Bs_Q))

I_B_Q = I_B_Bs + I_Bs_Q
print("\nI_B_rel_Q = {0}".format(I_B_Q))

# n_a is a vector parallel to line PQ
n_a = S(3)/5 * N.x - S(4)/5 * N.z
I_a_a_B_Q = dot(dot(n_a, I_B_Q), n_a)
print("\nn_a = {0}".format(n_a))
print("\nI_a_a_B_Q = {0} = {1}".format(I_a_a_B_Q, I_a_a_B_Q.subs(m, m_val)))
```

```
I_B_rel_O = 65*m/3*(N.x|N.x) + 6*m*(N.x|N.y) - 12*m*(N.x|N.z) + 6*m*(N.y|N.x) + 325*m/
↪12*(N.y|N.y) + 8*m*(N.y|N.z) - 12*m*(N.z|N.x) + 8*m*(N.z|N.y) + 169*m/12*(N.z|N.z)

I_B*_rel_O = 20*m*(N.x|N.x) + 25*m*(N.y|N.y) + 13*m*(N.z|N.z) + 6*m*(N.x|N.y) - 12*m*(N.
↪x|N.z) + 6*m*(N.y|N.x) + 8*m*(N.y|N.z) - 12*m*(N.z|N.x) + 8*m*(N.z|N.y)

I_B_rel_B* = 5*m/3*(N.x|N.x) + 25*m/12*(N.y|N.y) + 13*m/12*(N.z|N.z)

I_B*_rel_Q = 4*m*(N.x|N.x) + 9*m*(N.y|N.y) + 13*m*(N.z|N.z) + 6*m*(N.x|N.y) + 6*m*(N.y|N.
↪x)

I_B_rel_Q = 17*m/3*(N.x|N.x) + 133*m/12*(N.y|N.y) + 169*m/12*(N.z|N.z) + 6*m*(N.x|N.y) +␣
↪6*m*(N.y|N.x)

n_a = 3/5*N.x - 4/5*N.z

I_a_a_B_Q = 829*m/75 = 3316/25
```

1.15.6 Exercise 6.7

Note: You can download this example as a Python script: `ex6-7.py` or Jupyter notebook: `ex6-7.ipynb`.

```
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import inertia, inertia_of_point_mass
from sympy.physics.mechanics import dot
from sympy import solve, sqrt, symbols, diff
from sympy import sin, cos, pi, Matrix
from sympy import simplify

b, m, k_a = symbols('b m k_a', real=True, nonnegative=True)
theta = symbols('theta', real=True)

N = ReferenceFrame('N')
N2 = N.orientnew('N2', 'Axis', [theta, N.z])
pO = Point('O')
pP1s = pO.locatenew('P1*', b/2 * (N.x + N.y))
pP2s = pO.locatenew('P2*', b/2 * (2*N.x + N.y + N.z))
pP3s = pO.locatenew('P3*', b/2 * ((2 + sin(theta))*N.x +
                                  (2 - cos(theta))*N.y +
                                  N.z))

I_1s_O = inertia_of_point_mass(m, pP1s.pos_from(pO), N)
I_2s_O = inertia_of_point_mass(m, pP2s.pos_from(pO), N)
I_3s_O = inertia_of_point_mass(m, pP3s.pos_from(pO), N)
print("\nI_1s_rel_O = {0}".format(I_1s_O))
print("\nI_2s_rel_O = {0}".format(I_2s_O))
print("\nI_3s_rel_O = {0}".format(I_3s_O))
```

(continues on next page)

(continued from previous page)

```
I_1_1s = inertia(N, m*b**2/12, m*b**2/12, 2*m*b**2/12)
I_2_2s = inertia(N, 2*m*b**2/12, m*b**2/12, m*b**2/12)

I_3_3s_N2 = Matrix([[2, 0, 0],
                    [0, 1, 0],
                    [0, 0, 1]])
I_3_3s_N = m*b**2/12 * simplify(N.dcm(N2) * I_3_3s_N2 * N2.dcm(N))
I_3_3s = inertia(N, I_3_3s_N[0, 0], I_3_3s_N[1, 1], I_3_3s_N[2, 2],
                 I_3_3s_N[0, 1], I_3_3s_N[1, 2], I_3_3s_N[2, 0])

I_1_0 = I_1_1s + I_1s_0
I_2_0 = I_2_2s + I_2s_0
I_3_0 = I_3_3s + I_3s_0
print("\nI_1_rel_0 = {0}".format(I_1_0))
print("\nI_2_rel_0 = {0}".format(I_2_0))
print("\nI_3_rel_0 = {0}".format(I_3_0))

# assembly inertia tensor is the sum of the inertia tensor of each component
I_B_0 = I_1_0 + I_2_0 + I_3_0
print("\nI_B_rel_0 = {0}".format(I_B_0))

# n_a is parallel to line L
n_a = sqrt(2) / 2 * (N.x + N.z)
print("\nn_a = {0}".format(n_a))

# calculate moment of inertia of for point 0 of assembly about line L
I_a_a_B_0 = simplify(dot(n_a, dot(I_B_0, n_a)))
print("\nI_a_a_B_rel_0 = {0}".format(I_a_a_B_0))

# use the second value since k_a is non-negative
k_a_val = solve(I_a_a_B_0 - 3 * m * k_a**2, k_a)[1]
print("\nk_a = {0}".format(k_a_val))

dk_a_dtheta = diff(k_a_val, theta)
print("\ndk_a/dtheta = {0}".format(dk_a_dtheta))

# solve for theta = 0 using a simplified expression or
# else no solution will be found
soln = solve(3*cos(theta) + 12*sin(theta) - 4*sin(theta)*cos(theta), theta)
# ignore complex values of theta
theta_vals = [s for s in soln if s.is_real]

print("")
for th in theta_vals:
    print("k_a({0}) = {1}".format((th * 180 / pi).evalf(),
                                  k_a_val.subs(theta, th).evalf()))
```

```
I_1s_rel_0 = b**2*m/4*(N.x|N.x) + b**2*m/4*(N.y|N.y) + b**2*m/2*(N.z|N.z) - b**2*m/4*(N.
↪x|N.y) - b**2*m/4*(N.y|N.x)

I_2s_rel_0 = b**2*m/2*(N.x|N.x) + 5*b**2*m/4*(N.y|N.y) + 5*b**2*m/4*(N.z|N.z) - b**2*m/
```

(continues on next page)

(continued from previous page)

```
↪2*(N.x|N.y) - b**2*m/2*(N.x|N.z) - b**2*m/2*(N.y|N.x) - b**2*m/4*(N.y|N.z) - b**2*m/
↪2*(N.z|N.x) - b**2*m/4*(N.z|N.y)

I_3s_rel_0 = m*(b**2*(2 - cos(theta))**2/4 + b**2/4)*(N.x|N.x) + m*(b**2*(sin(theta) +␣
↪2)**2/4 + b**2/4)*(N.y|N.y) + m*(b**2*(2 - cos(theta))**2/4 + b**2*(sin(theta) + 2)**2/
↪4)*(N.z|N.z) - b**2*m*(2 - cos(theta))*(sin(theta) + 2)/4*(N.x|N.y) -␣
↪b**2*m*(sin(theta) + 2)/4*(N.x|N.z) - b**2*m*(2 - cos(theta))*(sin(theta) + 2)/4*(N.
↪y|N.x) - b**2*m*(2 - cos(theta))/4*(N.y|N.z) - b**2*m*(sin(theta) + 2)/4*(N.z|N.x) -␣
↪b**2*m*(2 - cos(theta))/4*(N.z|N.y)

I_1_rel_0 = b**2*m/3*(N.x|N.x) + b**2*m/3*(N.y|N.y) + 2*b**2*m/3*(N.z|N.z) - b**2*m/4*(N.
↪x|N.y) - b**2*m/4*(N.y|N.x)

I_2_rel_0 = 2*b**2*m/3*(N.x|N.x) + 4*b**2*m/3*(N.y|N.y) + 4*b**2*m/3*(N.z|N.z) - b**2*m/
↪2*(N.x|N.y) - b**2*m/2*(N.x|N.z) - b**2*m/2*(N.y|N.x) - b**2*m/4*(N.y|N.z) - b**2*m/
↪2*(N.z|N.x) - b**2*m/4*(N.z|N.y)

I_3_rel_0 = (b**2*m*(cos(theta)**2 + 1)/12 + m*(b**2*(2 - cos(theta))**2/4 + b**2/4))*(N.
↪x|N.x) + (-b**2*m*(2 - cos(theta))*(sin(theta) + 2)/4 + b**2*m*sin(2*theta)/24)*(N.x|N.
↪y) + (-b**2*m*(2 - cos(theta))*(sin(theta) + 2)/4 + b**2*m*sin(2*theta)/24)*(N.y|N.x)␣
↪+ (b**2*m*(sin(theta)**2 + 1)/12 + m*(b**2*(sin(theta) + 2)**2/4 + b**2/4))*(N.y|N.y)␣
↪+ (b**2*m/12 + m*(b**2*(2 - cos(theta))**2/4 + b**2*(sin(theta) + 2)**2/4))*(N.z|N.z) -
↪ b**2*m*(sin(theta) + 2)/4*(N.x|N.z) - b**2*m*(2 - cos(theta))/4*(N.y|N.z) -␣
↪b**2*m*(sin(theta) + 2)/4*(N.z|N.x) - b**2*m*(2 - cos(theta))/4*(N.z|N.y)

I_B_rel_0 = (b**2*m*(cos(theta)**2 + 1)/12 + b**2*m + m*(b**2*(2 - cos(theta))**2/4 +␣
↪b**2/4))*(N.x|N.x) + (b**2*m*(sin(theta)**2 + 1)/12 + 5*b**2*m/3 + m*(b**2*(sin(theta)␣
↪+ 2)**2/4 + b**2/4))*(N.y|N.y) + (25*b**2*m/12 + m*(b**2*(2 - cos(theta))**2/4 +␣
↪b**2*(sin(theta) + 2)**2/4))*(N.z|N.z) + (-b**2*m*(2 - cos(theta))*(sin(theta) + 2)/4␣
↪+ b**2*m*sin(2*theta)/24 - 3*b**2*m/4)*(N.x|N.y) + (-b**2*m*(2 -␣
↪cos(theta))*(sin(theta) + 2)/4 + b**2*m*sin(2*theta)/24 - 3*b**2*m/4)*(N.y|N.x) + (-
↪b**2*m*(sin(theta) + 2)/4 - b**2*m/2)*(N.x|N.z) + (-b**2*m*(2 - cos(theta))/4 - b**2*m/
↪4)*(N.y|N.z) + (-b**2*m*(sin(theta) + 2)/4 - b**2*m/2)*(N.z|N.x) + (-b**2*m*(2 -␣
↪cos(theta))/4 - b**2*m/4)*(N.z|N.y)

n_a = sqrt(2)/2*N.x + sqrt(2)/2*N.z
```

```
I_a_a_B_rel_0 = b**2*m*(-4*sin(theta)**2 + 6*sin(theta) - 24*cos(theta) + 60)/24
```

```
k_a = b*sqrt(3*sin(theta) - 12*cos(theta) + cos(2*theta) + 29)/6

dk_a/dtheta = b*(6*sin(theta) - sin(2*theta) + 3*cos(theta)/2)/(6*sqrt(3*sin(theta) -␣
↪12*cos(theta) + cos(2*theta) + 29))
```

```
k_a(169.335342108210) = 1.08371034822267*b
k_a(-20.0025512645583) = 0.696492652799896*b
```

1.15.7 Exercise 6.10

Note: You can download this example as a Python script: `ex6-10.py` or Jupyter notebook: `ex6-10.ipynb`.

```
from sympy import Matrix
from sympy import pi, acos
from sympy import simplify, symbols
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import inertia, inertia_of_point_mass
from sympy.physics.mechanics import dot


def inertia_matrix(dyadic, rf):
    """Return the inertia matrix of a given dyadic for a specified
    reference frame.
    """
    return Matrix([[dot(dot(dyadic, i), j) for j in rf] for i in rf])


def angle_between_vectors(a, b):
    """Return the minimum angle between two vectors. The angle returned for
    vectors a and -a is 0.
    """
    angle = (acos(dot(a, b)/(a.magnitude() * b.magnitude())) *
             180 / pi).evalf()
    return min(angle, 180 - angle)


m, m_R, m_C, rho, r = symbols('m m_R m_C rho r', real=True, nonnegative=True)

N = ReferenceFrame('N')
pA = Point('A')
pPs = pA.locatenew('P*', 3*r*N.x - 2*r*N.y)

m_R = rho * 24 * r**2
m_C = rho * pi * r**2
m = m_R - m_C

I_Cs_A = inertia_of_point_mass(m, pPs.pos_from(pA), N)
I_C_Cs = inertia(N, m_R*(4*r)**2/12 - m_C*r**2/4,
                    m_R*(6*r)**2/12 - m_C*r**2/4,
                    m_R*((4*r)**2+(6*r)**2)/12 - m_C*r**2/2)

I_C_A = I_C_Cs + I_Cs_A
print("\nI_C_rel_A = {0}".format(I_C_A))

# Eigenvectors of I_C_A are the parallel to the principal axis for point A
# of Body C.
evecs_m = [triple[2]
           for triple in inertia_matrix(I_C_A, N).eigenvects()]
```

(continues on next page)

(continued from previous page)

```
# Convert eigenvectors from Matrix type to Vector type.
evecs = [sum(simplify(v[0][i]).evalf() * n for i, n in enumerate(N))
         for v in evecs_m]

# N.x is parallel to line AB
print("\nVectors parallel to the principal axis for point A of Body C and the" +
      "\ncorresponding angle between the principal axis and line AB (degrees):")
for v in evecs:
    print("{0}\t{1}".format(v, angle_between_vectors(N.x, v)))
```

```
I_C_rel_A = (-pi*r**4*rho/4 + 32*r**4*rho + 4*r**2*(-pi*r**2*rho + 24*r**2*rho))*(N.x|N.
↪x) + (-pi*r**4*rho/4 + 72*r**4*rho + 9*r**2*(-pi*r**2*rho + 24*r**2*rho))*(N.y|N.y) +␣
↪(-pi*r**4*rho/2 + 104*r**4*rho + 13*r**2*(-pi*r**2*rho + 24*r**2*rho))*(N.z|N.z) +␣
↪6*r**2*(-pi*r**2*rho + 24*r**2*rho)*(N.x|N.y) + 6*r**2*(-pi*r**2*rho + 24*r**2*rho)*(N.
↪y|N.x)
```

```
Vectors parallel to the principal axis for point A of Body C and the
corresponding angle between the principal axis and line AB (degrees):
N.z         90.0000000000000
- 1.73073714765796*N.x + N.y        30.0188275011498
0.57778848819025*N.x + N.y        59.9811724988502
```

1.15.8 Exercise 6.13

Note: You can download this example as a Python script: `ex6-13.py` or Jupyter notebook: `ex6-13.ipynb`.

```
from sympy import S, Matrix
from sympy import pi, acos
from sympy import simplify, sqrt, symbols
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import inertia_of_point_mass
from sympy.physics.mechanics import dot
import numpy as np


def inertia_matrix(dyadic, rf):
    """Return the inertia matrix of a given dyadic for a specified
    reference frame.
    """
    return Matrix([[dot(dot(dyadic, i), j) for j in rf] for i in rf])


def convert_eigenvectors_matrix_vector(eigenvectors, rf):
    """Return a list of Vectors converted from a list of Matrices.
    rf is the implicit ReferenceFrame for the Matrix representation of the
    eigenvectors.
```

(continues on next page)

(continued from previous page)

```
    """
    return [sum(simplify(v[0][i]).evalf() * n for i, n in enumerate(N))
            for v in eigenvectors]


def angle_between_vectors(a, b):
    """Return the minimum angle between two vectors. The angle returned for
    vectors a and -a is 0.
    """
    angle = (acos(dot(a, b) / (a.magnitude() * b.magnitude())) *
             180 / pi).evalf()
    return min(angle, 180 - angle)


m = symbols('m', real=True, nonnegative=True)
m_val = 1
N = ReferenceFrame('N')
pO = Point('O')
pP = pO.locatenew('P', -3 * N.y)
pQ = pO.locatenew('Q', -4 * N.z)
pR = pO.locatenew('R', 2 * N.x)
points = [pO, pP, pQ, pR]

# center of mass of assembly
pCs = pO.locatenew('C*', sum(p.pos_from(pO) for p in points) / S(len(points)))
print(pCs.pos_from(pO))

I_C_Cs = (inertia_of_point_mass(m, points[0].pos_from(pCs), N) +
          inertia_of_point_mass(m, points[1].pos_from(pCs), N) +
          inertia_of_point_mass(m, points[2].pos_from(pCs), N) +
          inertia_of_point_mass(m, points[3].pos_from(pCs), N))
print("I_C_Cs = {0}".format(I_C_Cs))

# calculate the principal moments of inertia and the principal axes
M = inertia_matrix(I_C_Cs, N)

# use numpy to find eigenvalues/eigenvectors since sympy failed
# note that the eigenvlaues/eigenvectors are the
# prinicpal moments of inertia/principal axes
eigvals, eigvecs_np = np.linalg.eigh(np.matrix(M.subs(m, m_val).n().tolist(),␣
↪dtype=float))
eigvecs = [sum(eigvecs_np[i, j] * n for i, n in enumerate(N))
           for j in range(3)]

# get the minimum moment of inertia and its associated principal axis
e, v = min(zip(eigvals, eigvecs))

# I = m * k**2, where I is the moment of inertia,
# m is the mass of the body, k is the radius of gyration
k = sqrt(e / (4 * m_val))
print("\nradius of gyration, k = {0} m".format(k))
```

(continues on next page)

(continued from previous page)

```
# calculate the angle between the associated principal axis and the line OP
# line OP is parallel to N.y
theta = angle_between_vectors(N.y, v)
print("\nangle between associated principal axis and line OP = {0}°".format(theta))
```

```
1/2*N.x - 3/4*N.y - N.z
I_C_Cs = 75*m/4*(N.x|N.x) + 15*m*(N.y|N.y) + 39*m/4*(N.z|N.z) - 3*m/2*(N.x|N.y) - 2*m*(N.
↪x|N.z) - 3*m/2*(N.y|N.x) + 3*m*(N.y|N.z) - 2*m*(N.z|N.x) + 3*m*(N.z|N.y)

radius of gyration, k = 1.43554075942754 m

angle between associated principal axis and line OP = 67.6396101383384°
```

1.15.9 Exercise 6.14

Note: You can download this example as a Python script: `ex6-14.py` or Jupyter notebook: `ex6-14.ipynb`.

```
from sympy import Matrix, S
from sympy import integrate, pi, simplify, solve, symbols
from sympy.physics.mechanics import ReferenceFrame, Point
from sympy.physics.mechanics import cross, dot
from sympy.physics.mechanics import inertia, inertia_of_point_mass


def inertia_matrix(dyadic, rf):
    """Return the inertia matrix of a given dyadic for a specified
    reference frame.
    """
    return Matrix([[dot(dot(dyadic, i), j) for j in rf] for i in rf])


def integrate_v(integrand, rf, bounds):
    """Return the integral for a Vector integrand."""
    return sum(simplify(integrate(dot(integrand, n), bounds)) * n for n in rf)


def index_min(values):
    return min(range(len(values)), key=values.__getitem__)


m, a, b, c = symbols('m a b c', real=True, nonnegative=True)
x, y, r = symbols('x y r', real=True)
k, n = symbols('k n', real=True, positive=True)
N = ReferenceFrame('N')

# calculate right triangle density
V = b * c / 2
rho = m / V
```

(continues on next page)

(continued from previous page)

```
# Kane 1985 lists scalars of inertia I_11, I_12, I_22 for a right triangular
# plate, but not I_3x.
pO = Point('O')
pCs = pO.locatenew('C*', b/3*N.x + c/3*N.y)
pP = pO.locatenew('P', x*N.x + y*N.y)
p = pP.pos_from(pCs)

I_3 = rho * integrate_v(integrate_v(cross(p, cross(N.z, p)),
                                    N, (x, 0, b*(1 - y/c))),
                        N, (y, 0, c))
print("I_3 = {0}".format(I_3))

# inertia for a right triangle given ReferenceFrame, height b, base c, mass
inertia_rt = lambda rf, b_, c_, m_: inertia(rf,
                m_*c_**2/18,
                m_*b_**2/18,
                dot(I_3, N.z),
                m_*b_*c_/36,
                dot(I_3, N.y),
                dot(I_3, N.x)).subs({m:m_, b:b_, c:c_})

theta = (30 + 90) * pi / 180
N2 = N.orientnew('N2', 'Axis', [theta, N.x])

# calculate the mass center of the assembly
# Point O is located at the right angle corner of triangle 1.
pCs_1 = pO.locatenew('C*_1', 1/S(3) * (a*N.y + b*N.x))
pCs_2 = pO.locatenew('C*_2', 1/S(3) * (a*N2.y + b*N2.x))
pBs = pO.locatenew('B*', 1/(2*m) * m * (pCs_1.pos_from(pO) +
                                        pCs_2.pos_from(pO)))
print("\nB* = {0}".format(pBs.pos_from(pO).express(N)))

# central inertia of each triangle
I1_C_Cs = inertia_rt(N, b, a, m)
I2_C_Cs = inertia_rt(N2, b, a, m)

# inertia of mass center of each triangle about Point B*
I1_Cs_Bs = inertia_of_point_mass(m, pCs_1.pos_from(pBs), N)
I2_Cs_Bs = inertia_of_point_mass(m, pCs_2.pos_from(pBs), N)

I_B_Bs = I1_C_Cs + I1_Cs_Bs + I2_C_Cs + I2_Cs_Bs
print("\nI_B_Bs = {0}".format(I_B_Bs.express(N)))

# central principal moments of inertia
evals = inertia_matrix(I_B_Bs, N).eigenvals()

print("\neigenvalues:")
for e in evals.keys():
   print(e)

print("\nuse a/b = r")
```

(continues on next page)

(continued from previous page)

```
evals_sub_a = [simplify(e.subs(a, r*b)) for e in evals.keys()]
for e in evals_sub_a:
    print(e)

for r_val in [2, 1/S(2)]:
    print("\nfor r = {0}".format(r_val))
    evals_sub_r = [e.subs(r, r_val) for e in evals_sub_a]
    print("eigenvalues:")
    for e in evals_sub_r:
        print("{0} = {1}".format(e, e.n()))

    # substitute dummy values for m, b so that min will actually work
    min_index = index_min([e.subs({m : 1, b : 1}) for e in evals_sub_r])
    min_e = evals_sub_r[min_index]
    print("min: {0}".format(min_e))

    k_val = solve(min_e - 2*m*k**2, k)
    assert(len(k_val) == 1)
    print("k = {0}".format(k_val[0]))
    n_val = solve(k_val[0] - n*b, n)
    assert(len(n_val) == 1)
    print("n = {0}".format(n_val[0]))

print("\nResults in text: n = 1/3, sqrt(35 - sqrt(241))/24")
```

```
I_3 = m*(b**2 + c**2)/18*N.z

B* = b/3*N.x + a/12*N.y + sqrt(3)*a/12*N.z
```

```
I_B_Bs = (a**2*m/9 + b**2*m/18 + 2*m*(a**2/12 - b**2/36))*(N.x|N.x) + (a**2*m/24 +␣
↪5*b**2*m/72 + m*(a**2 + b**2)/24)*(N.y|N.y) + (a**2*m/8 + b**2*m/24 + 5*m*(a**2 +␣
↪b**2)/72)*(N.z|N.z) + a*b*m/72*(N.x|N.y) + sqrt(3)*a*b*m/72*(N.x|N.z) + a*b*m/72*(N.
↪y|N.x) + (sqrt(3)*a**2*m/36 + sqrt(3)*m*(a**2 + b**2)/72 - sqrt(3)*(-a**2*m/18 +␣
↪b**2*m/18)/4)*(N.y|N.z) + sqrt(3)*a*b*m/72*(N.z|N.x) + (sqrt(3)*a**2*m/36 +␣
↪sqrt(3)*m*(a**2 + b**2)/72 - sqrt(3)*(-a**2*m/18 + b**2*m/18)/4)*(N.z|N.y)
```

```
eigenvalues:
a**2*m/36 + b**2*m/9
19*a**2*m/72 + b**2*m/18 - m*sqrt(a**4 - 4*a**2*b**2 + 16*b**4)/72
19*a**2*m/72 + b**2*m/18 + m*sqrt(a**4 - 4*a**2*b**2 + 16*b**4)/72

use a/b = r
b**2*m*(r**2 + 4)/36
b**2*m*(19*r**2 - sqrt(r**4 - 4*r**2 + 16) + 4)/72
b**2*m*(19*r**2 + sqrt(r**4 - 4*r**2 + 16) + 4)/72

for r = 2
eigenvalues:
2*b**2*m/9 = 0.222222222222222*b**2*m
```

(continues on next page)

(continued from previous page)

```
19*b**2*m/18 = 1.05555555555556*b**2*m
7*b**2*m/6 = 1.16666666666667*b**2*m
min: 2*b**2*m/9
k = b/3
n = 1/3

for r = 1/2
eigenvalues:
17*b**2*m/144 = 0.118055555555556*b**2*m
b**2*m*(35/4 - sqrt(241)/4)/72 = 0.0676243934157638*b**2*m
b**2*m*(sqrt(241)/4 + 35/4)/72 = 0.175431162139792*b**2*m
min: b**2*m*(35/4 - sqrt(241)/4)/72
k = b*sqrt(35 - sqrt(241))/24
n = sqrt(35 - sqrt(241))/24

Results in text: n = 1/3, sqrt(35 - sqrt(241))/24
```

1.16 Linear Mass-Spring-Damper with Gravity

Note: You can download this example as a Python script: `mass-spring-damper.py` or Jupyter notebook: `mass-spring-damper.ipynb`.

1.16.1 Defining the Problem

Here we will derive the equations of motion for the classic mass-spring-damper system under the influence of gravity. The following figure gives a pictorial description of the problem.

Start by loading in the core functionality of both SymPy and Mechanics.

```
import sympy as sm
import sympy.physics.mechanics as me
```

We can make use of the pretty printing of our results by loading SymPy's printing extension, in particular we will use the vector printing which is nice for mechanics objects.

```
me.init_vprinting()
```

We'll start by defining the variables we will need for this problem:

- $x(t)$: distance of the particle from the ceiling
- $v(t)$: speed of the particle
- m: mass of the particle
- c: damping coefficient of the damper
- k: stiffness of the spring
- g: acceleration due to gravity

- t: time

```
x, v = me.dynamicsymbols('x v')
m, c, k, g, t = sm.symbols('m c k g t')
```

Now, we define a Newtonian reference frame that represents the ceiling which the particle is attached to, C.

```
ceiling = me.ReferenceFrame('C')
```

We will need two points, one to represent the original position of the particle which stays fixed in the ceiling frame, O, and the second one, P which is aligned with the particle as it moves.

```
O = me.Point('O')
P = me.Point('P')
```

The velocity of point O in the ceiling is zero.

```
O.set_vel(ceiling, 0)
```

Point P can move downward in the y direction and its velocity is specified as v in the downward direction.

```
P.set_pos(O, x * ceiling.x)
P.set_vel(ceiling, v * ceiling.x)
P.vel(ceiling)
```

$$v\hat{\mathbf{c}}_\mathbf{x}$$

There are three forces acting on the particle. Those due to the acceleration of gravity, the damper, and the spring.

```
damping = -c * P.vel(ceiling)
stiffness = -k * P.pos_from(O)
gravity = m * g * ceiling.x
forces = damping + stiffness + gravity
forces
```

$$(-cv + gm - kx)\hat{\mathbf{c}}_\mathbf{x}$$

Now we can use Newton's second law, $0 = F - ma$, to form the equation of motion of the system.

```
zero = me.dot(forces - m * P.acc(ceiling), ceiling.x)
zero
```

$$-cv + gm - kx - m\dot{v}$$

We can then form the first order equations of motion by solving for $\frac{dv}{dt}$ and introducing the kinematical differential equation, $v = \frac{dx}{dt}$.

```
dv_by_dt = sm.solve(zero, v.diff(t))[0]
dx_by_dt = v
dv_by_dt, dx_by_dt
```

$$\left(\frac{-cv + gm - kx}{m}, \ v\right)$$

Forming the equations of motion can also be done with the automated methods available in the Mechanics package: `LagrangesMethod` and `KanesMethod`. Here we will make use of Kane's method to find the same equations of motion that we found manually above. First, define a particle that represents the mass attached to the damper and spring.

```
mass = me.Particle('mass', P, m)
```

Now we can construct a `KanesMethod` object by passing in the generalized coordinate, x, the generalized speed, v, and the kinematical differential equation which relates the two, $0 = v - \frac{dx}{dt}$.

```
kane = me.KanesMethod(ceiling, q_ind=[x], u_ind=[v], kd_eqs=[v - x.diff(t)])
```

Now Kane's equations can be computed, and we can obtain F_r and F_r^*.

```
fr, frstar = kane.kanes_equations([mass], loads=[(P, forces)])
fr, frstar
```

$$\left(\left[-cv + gm - kx\right],\ \left[-m\dot{v}\right]\right)$$

The equations are also available in the form $M\frac{d}{dt}[q, u]^T = f(q, u)$ and we can extract the mass matrix, M, and the forcing functions, f.

```
M = kane.mass_matrix_full
f = kane.forcing_full
M, f
```

$$\left(\begin{bmatrix}1 & 0\\ 0 & m\end{bmatrix},\ \begin{bmatrix}v\\ -cv + gm - kx\end{bmatrix}\right)$$

Finally, we can form the first order differential equations of motion $\frac{d}{dt}[q, u]^T = M^{-1}f(\dot{u}, u, q)$, which is the same as previously found.

```
M.inv() * f
```

$$\begin{bmatrix}v\\ \frac{-cv + gm - kx}{m}\end{bmatrix}$$

1.16.2 Simulating the system

Now that we have defined the mass-spring-damper system, we are going to simulate it.

PyDy's `System` is a wrapper that holds the Kanes object to integrate the equations of motion using numerical values of constants.

```
from pydy.system import System
sys = System(kane)
```

Now, we specify the numerical values of the constants and the initial values of states in the form of a dict.

```
sys.constants = {m:10.0, g:9.8, c:5.0, k:10.0}
sys.initial_conditions = {x:0.0, v:0.0}
```

We must generate a time vector over which the integration will be carried out. NumPy's `linspace` is often useful for this.

```
from numpy import linspace
fps = 60
duration = 10.0
sys.times = linspace(0.0, duration, num=int(duration*fps))
```

The trajectory of the states over time can be found by calling the `.integrate()` method.

```
x_trajectory = sys.integrate()
```

1.16.3 Visualizing the System

PyDy has a native module `pydy.viz` which is used to visualize a System in an interactive 3D GUI.

```
from pydy.viz import *
```

For visualizing the system, we need to create shapes for the objects we wish to visualize, and map each of them to a `VisualizationFrame`, which holds the position and orientation of the object. First create a sphere to represent the bob and attach it to the point P and the ceiling reference frame (the sphere does not rotate with respect to the ceiling).

```
bob = Sphere(2.0, color="red", name='bob')
bob_vframe = VisualizationFrame(ceiling, P, bob)
```

Now create a circular disc that represents the ceiling and fix it to the ceiling reference frame. The circle's default axis is aligned with its local z axis, so we need to attach it to a rotated ceiling reference frame if we want the circle's axis to align with the $\hat{c}_x$ unit vector.

```
ceiling_circle = Circle(radius=10, color="white", name='ceiling')
rotated = ceiling.orientnew("C_R", 'Axis', [sm.pi/2, ceiling.y])
ceiling_vframe = VisualizationFrame(rotated, O, ceiling_circle)
```

Now we initialize a Scene. A Scene contains all the information required to visualize a `System` onto a canvas. It takes a ReferenceFrame and Point as arguments.

```
scene = Scene(ceiling, O, system=sys)
```

We provide the VisualizationFrames, which we want to visualize as a list to scene.

```
scene.visualization_frames = [bob_vframe, ceiling_vframe]
```

Now, we call the display method.

```
scene.display_jupyter(axes_arrow_length=5.0)
```

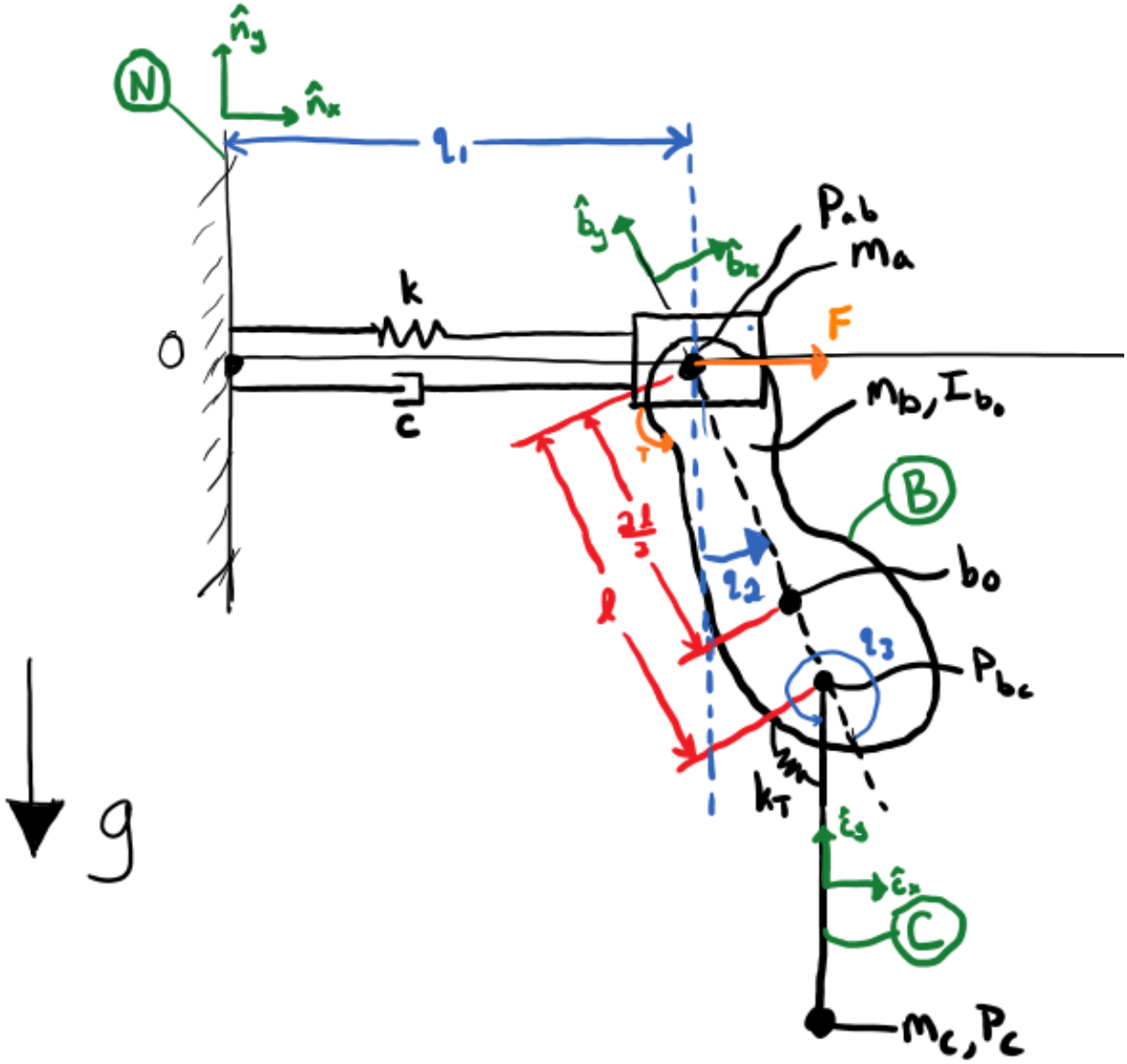
```
VBox(children=(AnimationAction(clip=AnimationClip(duration=10.0,␣
↪tracks=(VectorKeyframeTrack(name='scene/bob.m...
```

1.17 Multi Degree of Freedom Holonomic System

Note: You can download this example as a Python script: `multidof-holonomic.py` or Jupyter notebook: `multidof-holonomic.ipynb`.

1.17.1 Problem Description

This is an example of a holonomic system that includes both particles and rigid bodies with contributing forces and torques, some of which are specified forces and torques.



Generalized coordinates:

- q_1: Lateral distance of block from wall.
- q_2: Angle of the compound pendulum from vertical.
- q_3: Angle of the simple pendulum from the compound pendulum.

Generalized speeds:

- $u_1 = \dot{q}_1$: Lateral speed of block.
- $u_2 = {}^N\bar{v}^{Bo} \cdot \hat{b}_x$
- $u_3 = \dot{q}_3$: Angular speed of C relative to B.

The block and the bob of the simple pendulum are modeled as particles and B is a rigid body.

Loads:

- gravity acts on B and P_c.
- a linear spring and damper act on P_{ab}
- a rotational linear spring acts on C relative to B
- specified torque T acts on B relative to N
- specified force F acts on P_{ab}

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting()
```

1.17.2 Generalized coordinates

```
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
```

1.17.3 Generalized speeds

```
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')
```

1.17.4 Specified Inputs

```
F, T = me.dynamicsymbols('F, T')
```

1.17.5 Constants

```
k, c, ma, mb, mc, IB_bo, l, kT, g = sm.symbols('k, c, m_a, m_b, m_c, I_{B_bo}, l, k_T, g
↪')
k, c, ma, mb, mc, IB_bo, l, kT, g
```

$$(k,\ c,\ m_a,\ m_b,\ m_c,\ I_{B_bo},\ l,\ k_T,\ g)$$

1.17.6 Reference Frames

```
N = me.ReferenceFrame('N')
B = N.orientnew('B', 'Axis', (q2, N.z))
C = B.orientnew('C', 'Axis', (q3, N.z))
```

1.17.7 Points

```
O = me.Point('O')
Pab = O.locatenew('P_{ab}', q1 * N.x)
Bo = Pab.locatenew('B_o', - 2 * l / 3 * B.y)
Pbc = Pab.locatenew('P_{bc}', -l * B.y)
Pc = Pbc.locatenew('P_c', -l * C.y)
Pc.pos_from(O)
```

$$-l\hat{\mathbf{c}}_\mathbf{y} - l\hat{\mathbf{b}}_\mathbf{y} + q_1\hat{\mathbf{n}}_\mathbf{x}$$

1.17.8 Linear Velocities

```
Pab.set_vel(N, Pab.pos_from(O).dt(N))
Pab.vel(N)
```

$$\dot{q}_1\hat{\mathbf{n}}_\mathbf{x}$$

```
Bo.v2pt_theory(Pab, N, B)
```

$$\dot{q}_1\hat{\mathbf{n}}_\mathbf{x} + \frac{2l\dot{q}_2}{3}\hat{\mathbf{b}}_\mathbf{x}$$

```
Pbc.v2pt_theory(Pab, N, B)
```

$$\dot{q}_1\hat{\mathbf{n}}_\mathbf{x} + l\dot{q}_2\hat{\mathbf{b}}_\mathbf{x}$$

```
Pc.v2pt_theory(Pbc, N, C)
```

$$\dot{q}_1\hat{\mathbf{n}}_\mathbf{x} + l\dot{q}_2\hat{\mathbf{b}}_\mathbf{x} + l\left(\dot{q}_2 + \dot{q}_3\right)\hat{\mathbf{c}}_\mathbf{x}$$

1.17.9 Kinematic Differential Equations

One non-trivial generalized speed definition is used.

```
u1_eq = sm.Eq(u1, Pab.vel(N).dot(N.x))
u2_eq = sm.Eq(u2, Bo.vel(N).dot(B.x))
u3_eq = sm.Eq(u3, C.ang_vel_in(B).dot(B.z))
qdots = sm.solve([u1_eq, u2_eq, u3_eq], q1.diff(), q2.diff(), q3.diff())
qdots
```

$$\left\{\dot{q}_1 : u_1,\ \dot{q}_2 : -\frac{3u_1 \cos\left(q_2\right)}{2l} + \frac{3u_2}{2l},\ \dot{q}_3 : u_3\right\}$$

Substitute expressions for the $\dot{q}$'s.

```
Pab.set_vel(N, Pab.vel(N).subs(qdots).simplify())
Pab.vel(N)
```

$$u_1 \hat{\mathbf{n}}_\mathbf{x}$$

```
Bo.set_vel(N, Bo.vel(N).subs(qdots).express(B).simplify())
Bo.vel(N)
```

$$u_2 \hat{\mathbf{b}}_\mathbf{x} - u_1 \sin\left(q_2\right) \hat{\mathbf{b}}_\mathbf{y}$$

```
Pc.set_vel(N, Pc.vel(N).subs(qdots).simplify())
Pc.vel(N)
```

$$u_1 \hat{\mathbf{n}}_\mathbf{x} + (-\frac{3u_1 \cos\left(q_2\right)}{2} + \frac{3u_2}{2}) \hat{\mathbf{b}}_\mathbf{x} + (lu_3 - \frac{3u_1 \cos\left(q_2\right)}{2} + \frac{3u_2}{2}) \hat{\mathbf{c}}_\mathbf{x}$$

1.17.10 Angular Velocities

```
B.set_ang_vel(N, B.ang_vel_in(N).subs(qdots).simplify())
B.ang_vel_in(N)
```

$$\frac{3\left(-u_1 \cos\left(q_2\right) + u_2\right)}{2l} \hat{\mathbf{n}}_\mathbf{z}$$

```
C.set_ang_vel(B, u3 * N.z)
C.ang_vel_in(N)
```

$$(u_3 + \frac{3\left(-u_1 \cos\left(q_2\right) + u_2\right)}{2l}) \hat{\mathbf{n}}_\mathbf{z}$$

1.17.11 Mass and Inertia

```
ma, mc
```

$$\left(m_a,\ m_c\right)$$

```
IB = me.inertia(B, 0, 0, IB_bo)
IB
```

$$I_{B_bo} \hat{\mathbf{b}}_\mathbf{z} \otimes \hat{\mathbf{b}}_\mathbf{z}$$

1.17.12 Loads (forces and torques)

Make sure these are defined in terms of the q's and u's.

```
Rab = (F - k*q1 - c*qdots[q1.diff()]) * N.x
Rab
```

$$(-cu_1 - kq_1 + F)\hat{\mathbf{n}}_\mathbf{x}$$

```
Rbo = -(mb*g)*N.y
Rbo
```

$$-gm_b\hat{\mathbf{n}}_\mathbf{y}$$

```
Rc = -(mc*g)*N.y
Rc
```

$$-gm_c\hat{\mathbf{n}}_\mathbf{y}$$

```
TB = (T + kT*q3)*N.z
TB
```

$$(k_T q_3 + T)\hat{\mathbf{n}}_\mathbf{z}$$

Equal and opposite torque on C from B.

```
TC = -kT*q3*N.z
TC
```

$$-k_T q_3\hat{\mathbf{n}}_\mathbf{z}$$

1.17.13 Kane's Equations

```
kdes = [u1_eq.rhs - u1_eq.lhs,
        u2_eq.rhs - u2_eq.lhs,
        u3_eq.rhs - u3_eq.lhs]
kdes
```

$$\left[-u_1 + \dot{q}_1,\ \frac{2l\dot{q}_2}{3} - u_2 + \cos\left(q_2\right)\dot{q}_1,\ -u_3 + \dot{q}_3\right]$$

```
block = me.Particle('block', Pab, ma)
pendulum = me.RigidBody('pendulum', Bo, B, mb, (IB, Bo))
bob = me.Particle('bob', Pc, mc)

bodies = [block, pendulum, bob]
```

Loads are a list of (force, point) or reference (frame, torque) 2-tuples:

```
loads = [(Pab, Rab),
         (Bo, Rbo),
         (Pc, Rc),
         (B, TB),
         (C, TC)]
```

```
kane = me.KanesMethod(N, (q1, q2, q3), (u1, u2, u3), kd_eqs=kdes)
fr, frstar = kane.kanes_equations(bodies, loads=loads)
```

1.17.14 Simulation

```
from pydy.system import System
import numpy as np  # provides basic array types and some linear algebra
import matplotlib.pyplot as plt  # used for plots
```

```
sys = System(kane)
```

Define numerical values for each constant using a dictionary. Make sure units are compatible!

```
sys.constants = {ma: 1.0,  # kg
                 mb: 2.0,  # kg
                 mc: 1.0,  # kg
                 g: 9.81,  # m/s/s
                 l: 2.0,  # m
                 IB_bo: 2.0,  # kg*m**2
                 c: 10.0,  # kg/s
                 k: 60.0,  # N/m
                 kT: 10.0}  # N*m/rad
```

Provide an array of monotonic values of time that you'd like the state values reported at.

```
sys.times = np.linspace(0.0, 20.0, num=500)
```

Set the initial conditions for each state.

```
sys.states
```

$$[q_1,\ q_2,\ q_3,\ u_1,\ u_2,\ u_3]$$

```
sys.initial_conditions = {q1: 1.0,  # m
                          q2: 0.0,  # rad
                          q3: 0.0,  # rad
                          u1: 0.0,  # m/s
                          u2: 0.0,  # rad/s
                          u3: 0.0}  # rad/s
```

There are several ways that the specified force and torque can be provided to the system. Here are three options, the last one is actually used.

1) A single value can be provided to set the force and torque to be constant.

```
specifieds = {F: 0.0,  # N
              T: 1.0}  # N*m
```

2) The same thing as 1) can be done using an array.

```
specifieds = {(F, T): np.array([0.0, 1.0])}
```

3) A numerical function can be defined to calculate the input at a specific time.

```
def sin_f(x, t):
    """Returns the force F given the state vector x and time value t.

    Parameters
    ==========
    x : ndarray, shape(n,)
        The states in the order specified in System.states.
    t : float
        The value of time.

    Returns
    =======
    float
        The value of the force at time t.

    """
    return 1.0 * np.sin(2 * np.pi * t)

specifieds = {F: sin_f,  # N
              T: 1.0}  # N*m
```

4) A single numerical function can also be used for both.

```
def sin_f_t(x, t):
    return  np.array([5.0 * np.sin(2.0 * np.pi * t),
                      10.0 * np.cos(2.0 * np.pi * t)])

specifieds = {(F, T): sin_f_t}
```

```
sys.specifieds = specifieds
```

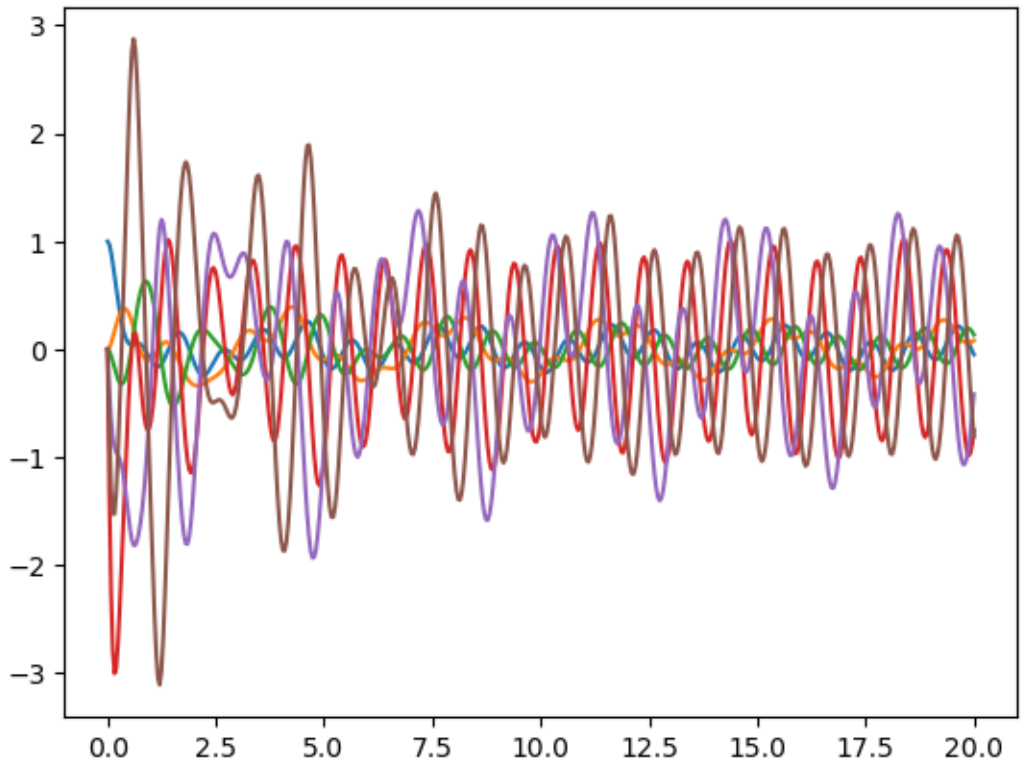
Integrate the equations of motion and get the state trajectories `x`:

```
x = sys.integrate()
x.shape
```

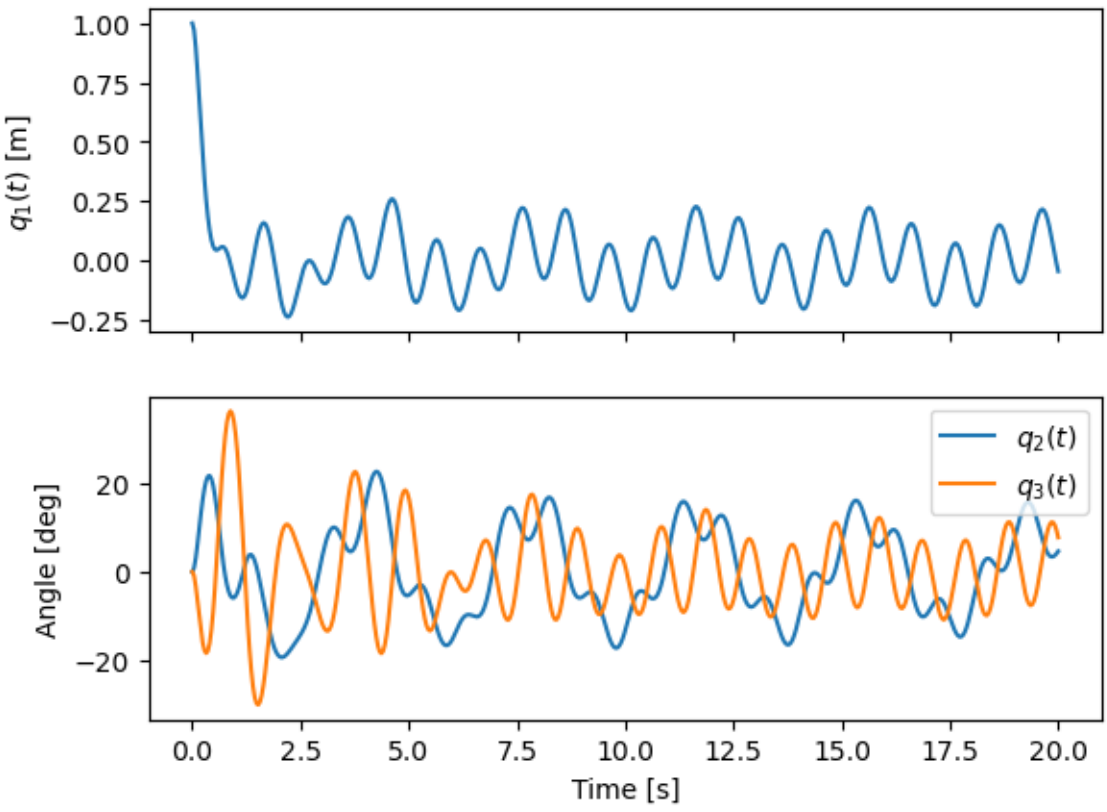
$$(500,\ 6)$$

1.17.15 Plot Results

```
plt.plot(sys.times, x);
```




```
fig, axes = plt.subplots(2, 1, sharex=True)
axes[0].plot(sys.times, x[:, 0])
axes[0].set_ylabel('{} [m]'.format(sm.latex(q1, mode='inline')))
axes[1].plot(sys.times, np.rad2deg(x[:, 1:3]))
axes[1].legend([sm.latex(q, mode='inline') for q in (q2, q3)])
axes[1].set_xlabel('Time [s]')
axes[1].set_ylabel('Angle [deg]');
```



1.17.16 Animate with PyDy and pythreejs

```
from pydy.viz.shapes import Cube, Cylinder, Sphere, Plane
from pydy.viz.visualization_frame import VisualizationFrame
from pydy.viz import Scene
```

Define some PyDy shapes for each moving object you want visible in the scene. Each shape needs a unique name with no spaces.

```
block_shape = Cube(0.25, color='azure', name='block')
cpendulum_shape = Plane(l, l/4, color='mediumpurple', name='cpendulum')
spendulum_shape = Cylinder(l, 0.02, color='azure', name='spendulum')
bob_shape = Sphere(0.2, color='mediumpurple', name='bob')
```

Create a visualization frame that attaches a shape to a reference frame and point. Note that the center of the plane and cylinder for the two pendulums is at its geometric center, so two new points are created so that the position of those points are calculated instead of the mass centers, which are not at the geometric centers.

```
v1 = VisualizationFrame('block', N, Pab, block_shape)

v2 = VisualizationFrame('cpendulum',
                        B,
                        Pab.locatenew('Bc', -l/2*B.y),
                        cpendulum_shape)
```

(continues on next page)

(continued from previous page)

```
v3 = VisualizationFrame('spendulum',
                        C,
                        Pbc.locatenew('Cc', -l/2*C.y),
                        spendulum_shape)

v4 = VisualizationFrame('bob', C, Pc, bob_shape)
```

Create a scene with the origin point O and base reference frame N and the fully defined `System`.

```
scene = Scene(N, O, v1, v2, v3, v4, system=sys)
```

Make sure pythreejs is installed and then call `display_jupyter` for a 3D animation of the system.

```
scene.display_jupyter(axes_arrow_length=1.0)
```

```
VBox(children=(AnimationAction(clip=AnimationClip(duration=20.0,␣
↪tracks=(VectorKeyframeTrack(name='scene/block...
```

It is then fairly simple to change constants, initial conditions, simulation time, or specified inputs and visualize the effects. Below the lateral spring is stretched more initially and when `display_jupyter()` is called the system equations are integrated with the new initial condition.

```
sys.initial_conditions[q1] = 5.0  # m
```

```
scene.display_jupyter(axes_arrow_length=1.0)
```

```
VBox(children=(AnimationAction(clip=AnimationClip(duration=20.0,␣
↪tracks=(VectorKeyframeTrack(name='scene/block...
```

1.18 Modeling of a Variable-Mass Nonholonomic Gyrostatic Rocket Car Using Extended Kane's Equations

Note: You can download this example as a Python script: `rocket-car.py` or Jupyter notebook: `rocket-car.ipynb`.

This is an implementation of the nonholonomic rocket-engine-powered jet racing car example from [Ge1982]. This example provides insight into analytically modeling variable-mass systems using Kane's method and the extended Kane's equations for variable-mass systems. It also demonstrates the efficacy of Kane's method in the modeling of complex dynamical systems.

An idealized model of a jet racing car that is propelled by a rocket engine at point P is considered. The rocket engine is treated as a variable-mass particle at P.

Here, $\{a_x : g_3, a_y : g_1, a_z : g_2\}$

```
import sympy as sm
import sympy.physics.mechanics as me
from pydy.system import System
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from sympy.simplify.fu import TR2
import matplotlib.pyplot as plt
from scipy.integrate import odeint
me.init_vprinting()

from IPython.display import Latex, display
def print_answer(x, x_res):
    for xi, xri in zip(x, x_res):
        display(Latex(sm.latex(sm.Eq(xi, xri), mode='inline')))
```

1.18.1 Reference Frames, Generalized Coordinates, and Generalized Speeds

```
N = me.ReferenceFrame('N')

q1, q2, q3, q4, q5, q6, q7, q8 = me.dynamicsymbols('q1:9')

A2 = N.orientnew('A_2', 'Axis', [q3, N.z])
A1 = A2.orientnew('A_1', 'Axis', [q4, A2.z])

B1 = A1.orientnew('B_1', 'Axis', [q5, A1.y])
B2 = A1.orientnew('B_2', 'Axis', [q6, A1.y])
B3 = A2.orientnew('B_3', 'Axis', [q7, A2.y])
B4 = A2.orientnew('B_4', 'Axis', [q8, A2.y])

t = me.dynamicsymbols._t
```

```
O = me.Point('O') # fixed point in the inertial reference frame
O.set_vel(N, 0)
```

```
L, l , a, b, r1, r2 = sm.symbols('L, l , a, b, r_1, r_2')
```

```
Q = O.locatenew('Q', q1 * N.x + q2 * N.y)
```

```
P = Q.locatenew('P', L * -A2.x)
```

```
C = P.locatenew('C', l * A2.x)
```

```
Q.set_vel(N, Q.pos_from(O).dt(N))
Q.vel(N)
```

$$\dot{q}_1 \hat{\mathbf{n}}_\mathbf{x} + \dot{q}_2 \hat{\mathbf{n}}_\mathbf{y}$$

```
P.v2pt_theory(Q, N, A2)
P.vel(N)
```

$$\dot{q}_1 \hat{\mathbf{n}}_\mathbf{x} + \dot{q}_2 \hat{\mathbf{n}}_\mathbf{y} - L\dot{q}_3 \hat{\mathbf{a}}_{\mathbf{2y}}$$

```
C.v2pt_theory(P, N, A2)
# C.vel(N)
```

$$\dot{q}_1 \hat{\mathbf{n}}_\mathbf{x} + \dot{q}_2 \hat{\mathbf{n}}_\mathbf{y} + (-L\dot{q}_3 + l\dot{q}_3)\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
A1.ang_vel_in(A2).express(A1)
```

$$\dot{q}_4 \hat{\mathbf{a}}_{\mathbf{1z}}$$

```
u1, u2 = me.dynamicsymbols('u_1:3')
```

```
z1 = sm.Eq(u1, A1.ang_vel_in(A2).dot(A1.z))
z2 = sm.Eq(u2, Q.vel(N).dot(A1.x))
```

```
u = sm.trigsimp(sm.solve([z1, z2], A1.ang_vel_in(A2).dot(A1.z), Q.vel(N).dot(A1.x)))
u
```

$$\{\sin(q_3 + q_4)\dot{q}_2 + \cos(q_3 + q_4)\dot{q}_1 : u_2,\ \dot{q}_4 : u_1\}$$

1.18.2 Formulation of the Constraint Equations

Nonholonomic Constraints: B_1

```
B1_center = Q.locatenew('B_1_center', a * A1.y)
B1_center.pos_from(Q)
```

$$a\hat{\mathbf{a}}_{\mathbf{1y}}$$

```
B1_center.v2pt_theory(Q, N, A1)
B1_center.vel(N).express(A1).simplify()
```

$$(-a(\dot{q}_3 + \dot{q}_4) + \sin(q_3 + q_4)\dot{q}_2 + \cos(q_3 + q_4)\dot{q}_1)\hat{\mathbf{a}}_{\mathbf{1x}} + (-\sin(q_3 + q_4)\dot{q}_1 + \cos(q_3 + q_4)\dot{q}_2)\hat{\mathbf{a}}_{\mathbf{1y}}$$

```
B1_ground = B1_center.locatenew('B_1_ground', r1 * -A1.z)
B1_ground.pos_from(B1_center)
```

$$-r_1\hat{\mathbf{a}}_{\mathbf{1z}}$$

```
B1_ground.v2pt_theory(B1_center, N, B1)
B1_ground.vel(N).simplify()
```

$$\dot{q}_1 \hat{\mathbf{n}}_\mathbf{x} + \dot{q}_2 \hat{\mathbf{n}}_\mathbf{y} + (-a(\dot{q}_3 + \dot{q}_4) - r_1\dot{q}_5)\hat{\mathbf{a}}_{\mathbf{1x}}$$

```
B1_cons = [me.dot(B1_ground.vel(N).simplify(), uv) for uv in A1]
for i in range(len(B1_cons)):
    display(sm.trigsimp(B1_cons[i]))
```

$$-a\left(\dot{q}_3 + \dot{q}_4\right) - r_1\dot{q}_5 + \sin\left(q_3 + q_4\right)\dot{q}_2 + \cos\left(q_3 + q_4\right)\dot{q}_1$$
$$- \sin\left(q_3 + q_4\right)\dot{q}_1 + \cos\left(q_3 + q_4\right)\dot{q}_2$$
$$0$$

```
eq1 = sm.Eq(B1_cons[0].simplify().subs(u), 0)
eq1
```

$$-a\left(u_1 + \dot{q}_3\right) - r_1\dot{q}_5 + u_2 = 0$$

```
eq2 = sm.Eq(B1_cons[1].simplify().subs(u), 0)
eq2
```

$$- \sin\left(q_3 + q_4\right)\dot{q}_1 + \cos\left(q_3 + q_4\right)\dot{q}_2 = 0$$

Nonholonomic Constraints: B_2

```
B2_center = Q.locatenew('B_1_center', a * -A1.y)
B2_center.pos_from(Q)
```

$$-a\hat{\mathbf{a}}_{\mathbf{1y}}$$

```
B2_center.v2pt_theory(Q, N, A1)
B2_center.vel(N).express(A1).simplify()
```

$$\left(a\left(\dot{q}_3 + \dot{q}_4\right) + \sin\left(q_3 + q_4\right)\dot{q}_2 + \cos\left(q_3 + q_4\right)\dot{q}_1\right)\hat{\mathbf{a}}_{\mathbf{1x}} + \left(- \sin\left(q_3 + q_4\right)\dot{q}_1 + \cos\left(q_3 + q_4\right)\dot{q}_2\right)\hat{\mathbf{a}}_{\mathbf{1y}}$$

```
B2_ground = B2_center.locatenew('B_2_ground', r1 * -A1.z)
B2_ground.pos_from(B2_center)
```

$$-r_1\hat{\mathbf{a}}_{\mathbf{1z}}$$

```
B2_ground.v2pt_theory(B2_center, N, B2)
B2_ground.vel(N).simplify()
```

$$\dot{q}_1\hat{\mathbf{n}}_{\mathbf{x}} + \dot{q}_2\hat{\mathbf{n}}_{\mathbf{y}} + \left(a\left(\dot{q}_3 + \dot{q}_4\right) - r_1\dot{q}_6\right)\hat{\mathbf{a}}_{\mathbf{1x}}$$

```
B2_cons = [me.dot(B2_ground.vel(N).simplify(), uv) for uv in A1]
for i in range(len(B2_cons)):
    display(sm.trigsimp(B2_cons[i]))
```

$$a\left(\dot{q}_3 + \dot{q}_4\right) - r_1\dot{q}_6 + \sin\left(q_3 + q_4\right)\dot{q}_2 + \cos\left(q_3 + q_4\right)\dot{q}_1$$
$$- \sin\left(q_3 + q_4\right)\dot{q}_1 + \cos\left(q_3 + q_4\right)\dot{q}_2$$
$$0$$

```
eq3 = sm.Eq(B2_cons[0].simplify().subs(u), 0)
eq3
```

$$a\left(u_1 + \dot{q}_3\right) - r_1\dot{q}_6 + u_2 = 0$$

```
eq4 = sm.Eq(B2_cons[1].simplify().subs(u), 0)
eq4
```

$$-\sin\left(q_3 + q_4\right)\dot{q}_1 + \cos\left(q_3 + q_4\right)\dot{q}_2 = 0$$

Nonholonomic Constraints: B_3

```
B3_center = P.locatenew('B_3_center', b * A2.y)
B3_center.pos_from(P)
```

$$b\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
B3_center.v2pt_theory(P, N, A2)
B3_center.vel(N).express(A2).simplify()
```

$$\left(-b\dot{q}_3 + \sin\left(q_3\right)\dot{q}_2 + \cos\left(q_3\right)\dot{q}_1\right)\hat{\mathbf{a}}_{\mathbf{2x}} + \left(-L\dot{q}_3 - \sin\left(q_3\right)\dot{q}_1 + \cos\left(q_3\right)\dot{q}_2\right)\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
B3_ground = B3_center.locatenew('B_3_ground', r2 * -A2.z)
B3_ground.pos_from(B3_center)
```

$$-r_2\hat{\mathbf{a}}_{\mathbf{2z}}$$

```
B3_ground.v2pt_theory(B3_center, N, B3)
B3_ground.vel(N).simplify()
```

$$\dot{q}_1\hat{\mathbf{n}}_{\mathbf{x}} + \dot{q}_2\hat{\mathbf{n}}_{\mathbf{y}} + \left(-b\dot{q}_3 - r_2\dot{q}_7\right)\hat{\mathbf{a}}_{\mathbf{2x}} - L\dot{q}_3\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
B3_cons = [me.dot(B3_ground.vel(N).simplify(), uv) for uv in A2]
for i in range(len(B3_cons)):
    display(sm.trigsimp(B3_cons[i]))
```

$$-b\dot{q}_3 - r_2\dot{q}_7 + \sin\left(q_3\right)\dot{q}_2 + \cos\left(q_3\right)\dot{q}_1$$

$$-L\dot{q}_3 - \sin\left(q_3\right)\dot{q}_1 + \cos\left(q_3\right)\dot{q}_2$$

$$0$$

```
eq5 = sm.Eq(B3_cons[0].simplify().subs(u), 0)
eq5
```

$$-b\dot{q}_3 - r_2\dot{q}_7 + \sin\left(q_3\right)\dot{q}_2 + \cos\left(q_3\right)\dot{q}_1 = 0$$

```
eq6 = sm.Eq(B3_cons[1].simplify().subs(u), 0)
eq6
```

$$-L\dot{q}_3 - \sin\left(q_3\right)\dot{q}_1 + \cos\left(q_3\right)\dot{q}_2 = 0$$

Nonholonomic Constraints: B_4

```
B4_center = P.locatenew('B_4_center', b * -A2.y)
B4_center.pos_from(P)
```

$$-b\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
B4_center.v2pt_theory(P, N, A2)
B4_center.vel(N).express(A2).simplify()
```

$$(b\dot{q}_3 + \sin(q_3)\dot{q}_2 + \cos(q_3)\dot{q}_1)\hat{\mathbf{a}}_{\mathbf{2x}} + (-L\dot{q}_3 - \sin(q_3)\dot{q}_1 + \cos(q_3)\dot{q}_2)\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
B4_ground = B4_center.locatenew('B_4_ground', r2 * -A2.z)
B4_ground.pos_from(B4_center)
```

$$-r_2\hat{\mathbf{a}}_{\mathbf{2z}}$$

```
B4_ground.v2pt_theory(B4_center, N, B4)
B4_ground.vel(N).simplify()
```

$$\dot{q}_1\hat{\mathbf{n}}_{\mathbf{x}} + \dot{q}_2\hat{\mathbf{n}}_{\mathbf{y}} + (b\dot{q}_3 - r_2\dot{q}_8)\hat{\mathbf{a}}_{\mathbf{2x}} - L\dot{q}_3\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
B4_cons = [me.dot(B4_ground.vel(N).simplify(), uv) for uv in A2]
for i in range(len(B4_cons)):
    display(sm.trigsimp(B4_cons[i]))
```

$$b\dot{q}_3 - r_2\dot{q}_8 + \sin(q_3)\dot{q}_2 + \cos(q_3)\dot{q}_1$$

$$-L\dot{q}_3 - \sin(q_3)\dot{q}_1 + \cos(q_3)\dot{q}_2$$

$$0$$

```
eq7 = sm.Eq(B4_cons[0].simplify().subs(u), 0)
eq7
```

$$b\dot{q}_3 - r_2\dot{q}_8 + \sin(q_3)\dot{q}_2 + \cos(q_3)\dot{q}_1 = 0$$

```
eq8 = sm.Eq(B4_cons[1].simplify().subs(u), 0)
eq8
```

$$-L\dot{q}_3 - \sin(q_3)\dot{q}_1 + \cos(q_3)\dot{q}_2 = 0$$

LHS $\Longleftrightarrow$ RHS in $z_1, z_2 \rightarrow$ Equations $9, 10$

```
eq9 = sm.Eq(A1.ang_vel_in(A2).dot(A1.z), u1)
eq9
```

$$\dot{q}_4 = u_1$$

```
eq10 = sm.Eq(Q.vel(N).dot(A1.x), u2)
eq10
```

$$(-\sin(q_3)\sin(q_4) + \cos(q_3)\cos(q_4))\,\dot{q}_1 + (\sin(q_3)\cos(q_4) + \sin(q_4)\cos(q_3))\,\dot{q}_2 = u_2$$

1.18.3 Solving the System of Linear Equations

The system of equations is linear in $\dot{q}_1, \dot{q}_2, \ldots$

Note: `eq4` $\equiv$ `eq2`; `eq8` $\equiv$ `eq6`

```
solution = sm.linsolve([eq1, eq2, eq3, eq5, eq6, eq7, eq9, eq10], q1.diff(), q2.diff(),␣
↪q3.diff(),  q4.diff(), q5.diff(), q6.diff(), q7.diff(), q8.diff())
```

```
sollist_keys = [q1.diff(), q2.diff(), q3.diff(),  q4.diff(), q5.diff(), q6.diff(), q7.
↪diff(), q8.diff()]
sollist_keys
```

$$[\dot{q}_1,\ \dot{q}_2,\ \dot{q}_3,\ \dot{q}_4,\ \dot{q}_5,\ \dot{q}_6,\ \dot{q}_7,\ \dot{q}_8]$$

```
sollist_values = list(solution.args[0])
```

```
sollist_values_simple = []
for i in range(len(sollist_values)):
    sollist_values_simple.append(sm.factor(TR2(sollist_values[i]).simplify()))
```

```
soldict = dict(zip(sollist_keys, sollist_values_simple))
print_answer(sollist_keys, sollist_values_simple)
```

$$\frac{d}{dt}q_1(t) = u_2(t)\cos(q_3(t) + q_4(t))$$

$$\frac{d}{dt}q_2(t) = u_2(t)\sin(q_3(t) + q_4(t))$$

$$\frac{d}{dt}q_3(t) = u_2(t)\sin(q_4(t))/L$$

$$\frac{d}{dt}q_4(t) = u_1(t)$$

$$\frac{d}{dt}q_5(t) = -\left(Lau_1(t) - Lu_2(t) + au_2(t)\sin(q_4(t))\right)/Lr_1$$

$$\frac{d}{dt}q_6(t) = \left(Lau_1(t) + Lu_2(t) + au_2(t)\sin(q_4(t))\right)/Lr_1$$

$$\frac{d}{dt}q_7(t) = -\left(-L\cos(q_4(t)) + b\sin(q_4(t))\right)u_2(t)/Lr_2$$

$$\frac{d}{dt}q_8(t) = \left(L\cos(q_4(t)) + b\sin(q_4(t))\right)u_2(t)/Lr_2$$

1.18.4 Reformulated Velocity and Angular Velocity Expressions

```
N_v_Q = Q.vel(N).subs(soldict).express(A1).simplify()
N_v_Q
```

$$u_2 \hat{\mathbf{a}}_{\mathbf{1x}}$$

```
N_v_P = P.vel(N).subs(soldict).express(A2).simplify()
N_v_P
```

$$u_2 \cos\left(q_4\right) \hat{\mathbf{a}}_{\mathbf{2x}}$$

```
N_v_C = C.vel(N).subs(soldict).express(A2).simplify()
N_v_C
```

$$u_2 \cos\left(q_4\right) \hat{\mathbf{a}}_{\mathbf{2x}} + \frac{l u_2 \sin\left(q_4\right)}{L} \hat{\mathbf{a}}_{\mathbf{2y}}$$

```
N_w_A1 = A1.ang_vel_in(N).subs(soldict).express(A1).simplify()
N_w_A1
```

$$(u_1 + \frac{u_2 \sin\left(q_4\right)}{L}) \hat{\mathbf{a}}_{\mathbf{1z}}$$

```
N_w_A2 = A2.ang_vel_in(N).subs(soldict).express(A2).simplify()
N_w_A2
```

$$\frac{u_2 \sin\left(q_4\right)}{L} \hat{\mathbf{a}}_{\mathbf{2z}}$$

1.18.5 Partial Velocities and Partial Angular Velocities

```
V_1_Q = N_v_Q.diff(u1, N)
V_1_Q
```

$$0$$

```
V_2_Q = N_v_Q.diff(u2, N)
V_2_Q
```

$$\hat{\mathbf{a}}_{\mathbf{1x}}$$

```
V_1_C = N_v_C.diff(u1, N)
V_1_C
```

$$0$$

```
V_2_C = N_v_C.diff(u2, N)
V_2_C
```

$$\cos\left(q_4\right)\hat{\mathbf{a}}_{\mathbf{2x}} + \frac{l\sin\left(q_4\right)}{L}\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
V_1_P = N_v_P.diff(u1, N)
V_1_P
```

$$0$$

```
V_2_P = N_v_P.diff(u2, N)
V_2_P
```

$$\cos\left(q_4\right)\hat{\mathbf{a}}_{\mathbf{2x}}$$

```
w_1_A1 = N_w_A1.diff(u1, N)
w_1_A1
```

$$\hat{\mathbf{a}}_{\mathbf{1z}}$$

```
w_2_A1 = N_w_A1.diff(u2, N)
w_2_A1
```

$$\frac{\sin\left(q_4\right)}{L}\hat{\mathbf{a}}_{\mathbf{1z}}$$

```
w_1_A2 = N_w_A2.diff(u1, N)
w_1_A2
```

$$0$$

```
w_2_A2 = N_w_A2.diff(u2, N)
w_2_A2
```

$$\frac{\sin\left(q_4\right)}{L}\hat{\mathbf{a}}_{\mathbf{2z}}$$

1.18.6 Accelerations and Angular Accelerations

```
a_1__P, a_2__P, a_3__P, a_1__C, a_2__C, a_3__C, a__Q, alpha__A1, alpha__A2 = sm.symbols(
↪'a_1__P, a_2__P, a_3__P, a_1__C, a_2__C, a_3__C, a__Q, alpha__A1, alpha__A2')
```

```
N_a_P = N_v_P.dt(N).subs(soldict)
N_a_P
```

$$\left(-u_1u_2\sin\left(q_4\right) + \cos\left(q_4\right)\dot{u}_2\right)\hat{\mathbf{a}}_{\mathbf{2x}} + \frac{u_2^2\sin\left(q_4\right)\cos\left(q_4\right)}{L}\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
N_a_C = N_v_C.dt(N).subs(soldict)
N_a_C
```

$$(-u_1 u_2 \sin(q_4) + \cos(q_4)\dot{u}_2 - \frac{l u_2^2 \sin^2(q_4)}{L^2})\hat{\mathbf{a}}_{\mathbf{2x}} + (\frac{l u_1 u_2 \cos(q_4)}{L} + \frac{l \sin(q_4)\dot{u}_2}{L} + \frac{u_2^2 \sin(q_4)\cos(q_4)}{L})\hat{\mathbf{a}}_{\mathbf{2y}}$$

```
N_a_Q = N_v_Q.dt(N).subs(soldict)
N_a_Q
```

$$\dot{u}_2 \hat{\mathbf{a}}_{\mathbf{1x}} + \left(u_1 + \frac{u_2 \sin(q_4)}{L}\right) u_2 \hat{\mathbf{a}}_{\mathbf{1y}}$$

```
N_aa_A1 = N_w_A1.dt(N).subs(soldict)
N_aa_A1
```

$$(\dot{u}_1 + \frac{u_1 u_2 \cos(q_4)}{L} + \frac{\sin(q_4)\dot{u}_2}{L})\hat{\mathbf{a}}_{\mathbf{1z}}$$

```
N_aa_A2 = N_w_A2.dt(N).subs(soldict)
N_aa_A2
```

$$(\frac{u_1 u_2 \cos(q_4)}{L} + \frac{\sin(q_4)\dot{u}_2}{L})\hat{\mathbf{a}}_{\mathbf{2z}}$$

1.18.7 Forces and Torques

$(F_r^*)_G = (F_r^*)_{GR} + (F_r^*)_{GI}$

where,

$(F_r^*)_{GR} = {V_r}^G \cdot {F_G}^* + \omega_r^A \cdot {T_G}^*$

$F_G^* = -m_G a^{G^*}$

$T_G^* \triangleq -[\alpha_A \cdot I_G + \omega_r^A \times (I_G \cdot \omega_r^A)]$

$({F_r}^*)_{GI} = -J\{\omega_r^A[\ddot{q_k} g_1 + \dot{q_k}(\omega_3^A g_2 - \omega_2^A g_3)] + C_{kr}(\dot{\omega_1^A} + \ddot{q_k})\}$

[Kane1978]

Naming Convention:

$({F_r}^*)_{G_n R}$ (rigid)

$({F_r}^*)_{G_n I}$ (internal)

1.18.8 Masses and Moments of Inertia

```
M1, M2 = sm.symbols('M_1, M_2')
m = me.dynamicsymbols('m')
```

```
I1x, I1y, I1z = sm.symbols('I_{1_x}, I_{1_y}, I_{1_z}')
I2x, I2y, I2z = sm.symbols('I_{2_x}, I_{2_y}, I_{2_z}')
J1, J2 = sm.symbols('J_1, J_2')
```

```
I1 = me.inertia(A1, I1x, I1y, I1z)
I1
```

$$I_{1_x}\hat{\mathbf{a}}_{\mathbf{1x}} \otimes \hat{\mathbf{a}}_{\mathbf{1x}} + I_{1_y}\hat{\mathbf{a}}_{\mathbf{1y}} \otimes \hat{\mathbf{a}}_{\mathbf{1y}} + I_{1_z}\hat{\mathbf{a}}_{\mathbf{1z}} \otimes \hat{\mathbf{a}}_{\mathbf{1z}}$$

```
I2 = me.inertia(A2, I2x, I2y, I2z)
I2
```

$$I_{2_x}\hat{\mathbf{a}}_{\mathbf{2x}} \otimes \hat{\mathbf{a}}_{\mathbf{2x}} + I_{2_y}\hat{\mathbf{a}}_{\mathbf{2y}} \otimes \hat{\mathbf{a}}_{\mathbf{2y}} + I_{2_z}\hat{\mathbf{a}}_{\mathbf{2z}} \otimes \hat{\mathbf{a}}_{\mathbf{2z}}$$

1.18.9 Gyrostat G_1

$\rightarrow F_G{}^* = -m_G a^{G*}$

```
Fstar_G1 = -M1 * N_a_Q
Fstar_G1
```

$$-M_1\dot{u}_2\hat{\mathbf{a}}_{\mathbf{1x}} - M_1\left(u_1 + \frac{u_2\sin\left(q_4\right)}{L}\right)u_2\hat{\mathbf{a}}_{\mathbf{1y}}$$

$\rightarrow T_G{}^* \triangleq -[\alpha_A \cdot I_G + \omega_r{}^A \times (I_G \cdot \omega_r{}^A)]$

```
Tstar_G1 = -(N_aa_A1.dot(I1) + me.cross(N_w_A1, I1.dot(N_w_A1)))
Tstar_G1
```

$$-I_{1_z}\left(\dot{u}_1 + \frac{u_1u_2\cos\left(q_4\right)}{L} + \frac{\sin\left(q_4\right)\dot{u}_2}{L}\right)\hat{\mathbf{a}}_{\mathbf{1z}}$$

$\rightarrow (F_r{}^*)_{GR} = V_r{}^G \cdot F_G{}^* + \omega_r{}^A \cdot T_G{}^*$

```
Fstar_1_G1_R = V_1_Q.dot(Fstar_G1) + w_1_A1.dot(Tstar_G1).subs(soldict)
Fstar_1_G1_R.subs({N_w_A1.dt(N).subs(soldict).dot(A1.z): alpha__A1})
```

$$-I_{1_z}\alpha^{A1}$$

```
Fstar_2_G1_R = V_2_Q.dot(Fstar_G1) + w_2_A1.dot(Tstar_G1).subs(soldict)
Fstar_2_G1_R.subs({N_w_A1.dt(N).subs(soldict).dot(A1.z): alpha__A1})
```

$$-\frac{I_{1_z}\alpha^{A1}\sin\left(q_4\right)}{L} - M_1\dot{u}_2$$

$\rightarrow (F_r^*)_{GI} = -J\{\omega_r^A \cdot [\ddot{q_k} g_1 + \dot{q_k}(\omega_3^A g_2 - \omega_2^A g_3)] + C_{kr}(\dot{\omega}_1^A + \ddot{q_k})\} \qquad (r = 1, ..., n-m)$

Here, $\{\omega_1^A : \omega_2^A,\ \omega_2^A : \omega_3^A,\ \omega_3^A : \omega_1^A\}$

$\rightarrow \dot{q_k} = \sum_{s=1}^{n-m} C_{ks} u_s + D_k$ (Generalized Speeds)

$\omega_i^A \triangleq \omega^A \cdot \hat{g}_i \quad (i = 1, 2, 3)$

```
# C_kr
C51, C61 = sm.symbols('C_51, C_61')
C_51 = soldict[q5.diff()].diff(u1)
C_61 = soldict[q6.diff()].diff(u1)
Fstar_1_G1_I = -J1 * (N_w_A1.dot(q5.diff().diff() * A1.y + q5.diff()*(N_w_A1.dot(A1.
↪x)*A1.z - N_w_A1.dot(A1.z)*A1.x)) + C_51 * (N_w_A1.dot(A1.y).diff() + q5.diff().
↪diff())) \
               -J1 * (N_w_A1.dot(q6.diff().diff() * A1.y + q6.diff()*(N_w_A1.dot(A1.
↪x)*A1.z - N_w_A1.dot(A1.z)*A1.x)) + C_61 * (N_w_A1.dot(A1.y).diff() + q6.diff().
↪diff()))   # B1 \ B2

Fstar_1_G1_I, C_51, C_61, Fstar_1_G1_I.subs({-C_51: -C51, -C_61: -C61}).simplify()
```

$$\left(\frac{J_1 a \ddot{q_5}}{r_1} - \frac{J_1 a \ddot{q_6}}{r_1},\ -\frac{a}{r_1},\ \frac{a}{r_1},\ J_1 \left(-C_{51}\ddot{q_5} - C_{61}\ddot{q_6}\right) \right)$$

```
# C_kr
C52, C62 = sm.symbols('C_52, C_62')
C_52 = soldict[q5.diff()].diff(u2)
C_62 = soldict[q6.diff()].diff(u2)
Fstar_2_G1_I = -J1 * (N_w_A1.dot(q5.diff().diff() * A1.y + q5.diff()*(N_w_A1.dot(A1.
↪x)*A1.z - N_w_A1.dot(A1.z)*A1.x)) + C_52 * (N_w_A1.dot(A1.y).diff() + q5.diff().
↪diff())) \
               -J1 * (N_w_A1.dot(q6.diff().diff() * A1.y + q6.diff()*(N_w_A1.dot(A1.
↪x)*A1.z - N_w_A1.dot(A1.z)*A1.x)) + C_62 * (N_w_A1.dot(A1.y).diff() + q6.diff().
↪diff()))   # B1 \ B2

display(Fstar_2_G1_I),
display(C_52)
display(C_62)
display(Fstar_2_G1_I.subs({-C_52: -C52, -C_62: -C62}).simplify())
```

$$\frac{J_1 \left(-L + a\sin\left(q_4\right)\right) \ddot{q_5}}{L r_1} - \frac{J_1 \left(L + a\sin\left(q_4\right)\right) \ddot{q_6}}{L r_1}$$

$$-\frac{-L + a\sin\left(q_4\right)}{L r_1}$$

$$\frac{L + a\sin\left(q_4\right)}{L r_1}$$

$$J_1 \left(-C_{52}\ddot{q_5} - C_{62}\ddot{q_6}\right)$$

$\rightarrow (F_r^*)_G = (F_r^*)_{GR} + (F_r^*)_{GI}$

```
Fstar_1_G1 = Fstar_1_G1_R + Fstar_1_G1_I
Fstar_1_G1.subs({N_w_A1.dt(N).subs(soldict).dot(A1.z): alpha__A1}).subs({-C_51: -C51, -C_
↪61: -C61}).simplify()
```

$$-C_{51} J_1 \ddot{q}_5 - C_{61} J_1 \ddot{q}_6 - I_{1_z} \alpha^{A1}$$

```
Fstar_2_G1 = Fstar_2_G1_R + Fstar_2_G1_I
Fstar_2_G1.subs({N_w_A1.dt(N).subs(soldict).dot(A1.z): alpha__A1}).subs({-C_52: -C52, -C_
↪62: -C62}).simplify()
```

$$-C_{52} J_1 \ddot{q}_5 - C_{62} J_1 \ddot{q}_6 - \frac{I_{1_z} \alpha^{A1} \sin\left(q_4\right)}{L} - M_1 \dot{u}_2$$

1.18.10 Gyrostat G_2

$\rightarrow {F_G}^* = -m_G {a^G}^*$

```
Fstar_G2 = -M2 * N_a_C
Fstar_G2
```

$$-M_2 \left(-u_1 u_2 \sin\left(q_4\right) + \cos\left(q_4\right)\dot{u}_2 - \frac{l u_2^2 \sin^2\left(q_4\right)}{L^2}\right) \hat{\mathbf{a}}_{\mathbf{2x}} - M_2 \left(\frac{l u_1 u_2 \cos\left(q_4\right)}{L} + \frac{l \sin\left(q_4\right)\dot{u}_2}{L} + \frac{u_2^2 \sin\left(q_4\right)\cos\left(q_4\right)}{L}\right) \hat{\mathbf{a}}_{\mathbf{2y}}$$

$\rightarrow {T_G}^* \triangleq -[\alpha_A \cdot I_G + {\omega_r}^A \times (I_G \cdot {\omega_r}^A)]$

```
Tstar_G2 = -(N_aa_A2.dot(I2) + me.cross(N_w_A2, I2.dot(N_w_A2)))
Tstar_G2
```

$$-I_{2_z} \left(\frac{u_1 u_2 \cos\left(q_4\right)}{L} + \frac{\sin\left(q_4\right)\dot{u}_2}{L}\right) \hat{\mathbf{a}}_{\mathbf{2z}}$$

$\rightarrow ({F_r}^*)_{GR} = {V_r}^G \cdot {F_G}^* + {\omega_r}^A \cdot {T_G}^*$

```
Fstar_1_G2_R = V_1_C.dot(Fstar_G2) + w_1_A2.dot(Tstar_G2).subs(soldict)
Fstar_1_G2_R.subs({N_w_A2.dt(N).subs(soldict).dot(A2.z): alpha__A2})
```

$$0$$

```
Fstar_2_G2_R = V_2_C.dot(Fstar_G2) + w_2_A1.dot(Tstar_G2).subs(soldict)
Fstar_2_G2_R.subs({N_w_A2.dt(N).subs(soldict).dot(A2.z): alpha__A2})
```

$$-\frac{I_{2_z}\alpha^{A2}\sin(q_4)}{L} - M_2\left(-u_1u_2\sin(q_4)+\cos(q_4)\dot{u}_2-\frac{lu_2^2\sin^2(q_4)}{L^2}\right)\cos(q_4) - \frac{M_2l\left(\frac{lu_1u_2\cos(q_4)}{L}+\frac{l\sin(q_4)\dot{u}_2}{L}+\frac{u_2^2\sin(q_4)\cos(q_4)}{L}\right)\sin(q_4)}{L}$$

$\rightarrow (F_r^*)_{GI} = -J\{\omega_r^A \cdot [\ddot{q_k}g_1 + \dot{q_k}(\omega_3^A g_2 - \omega_2^A g_3)] + C_{kr}(\dot{\omega}_1^A + \ddot{q_k})\} \qquad (r = 1, ..., n-m)$

Here, $\{\omega_1^A : \omega_2^A,\ \omega_2^A : \omega_3^A,\ \omega_3^A : \omega_1^A\}$

$\rightarrow \dot{q_k} = \sum_{s=1}^{n-m} C_{ks}u_s + D_k$ (Generalized Speeds)

$\omega_i^A \triangleq \omega^A \cdot \hat{g}_i \quad (i = 1, 2, 3)$

```
# C_kr
C71, C81 = sm.symbols('C_71, C_81')
C_71 = soldict[q7.diff()].diff(u1)
C_81 = soldict[q8.diff()].diff(u1)
Fstar_1_G2_I = -J2 * (N_w_A2.dot(q7.diff().diff() * A2.y + q7.diff()*(N_w_A2.dot(A2.
↪x)*A2.z - N_w_A2.dot(A2.z)*A2.x)) + C_71 * (N_w_A2.dot(A2.y).diff() + q7.diff().
↪diff())) \
              -J2 * (N_w_A2.dot(q8.diff().diff() * A2.y + q8.diff()*(N_w_A2.dot(A2.
↪x)*A2.z - N_w_A2.dot(A2.z)*A2.x)) + C_81 * (N_w_A2.dot(A2.y).diff() + q8.diff().
↪diff()))   # B1 \ B2

Fstar_1_G2_I, C_71, C_81, # Fstar_1_G2_I.subs({-C_71: -C71, -C_81: -C81}).simplify()
```

$$(0,\ 0,\ 0)$$

```
# C_kr
C72, C82 = sm.symbols('C_72, C_82')
C_72 = soldict[q7.diff()].diff(u2)
C_82 = soldict[q8.diff()].diff(u2)
Fstar_2_G2_I = -J2 * (N_w_A2.dot(q7.diff().diff() * A2.y + q7.diff()*(N_w_A2.dot(A2.
↪x)*A2.z - N_w_A2.dot(A2.z)*A2.x)) + C_72 * (N_w_A2.dot(A2.y).diff() + q7.diff().
↪diff())) \
              -J2 * (N_w_A2.dot(q8.diff().diff() * A2.y + q8.diff()*(N_w_A2.dot(A2.
↪x)*A2.z - N_w_A2.dot(A2.z)*A2.x)) + C_82 * (N_w_A2.dot(A2.y).diff() + q8.diff().
↪diff()))   # B1 \ B2

display(Fstar_2_G2_I)
display(C_72)
display(C_82)
display(Fstar_2_G2_I.subs({-C_72: -C72, -C_82: -C82}).simplify())
```

$$\frac{J_2\left(-L\cos(q_4)+b\sin(q_4)\right)\ddot{q}_7}{Lr_2} - \frac{J_2\left(L\cos(q_4)+b\sin(q_4)\right)\ddot{q}_8}{Lr_2}$$
$$-\frac{-L\cos(q_4)+b\sin(q_4)}{Lr_2}$$

$$\frac{L\cos\left(q_4\right) + b\sin\left(q_4\right)}{Lr_2}$$

$$J_2\left(-C_{72}\ddot{q}_7 - C_{82}\ddot{q}_8\right)$$

$\rightarrow (F_r^*)_G = (F_r^*)_{GR} + (F_r^*)_{GI}$

```
Fstar_1_G2 = Fstar_1_G2_R + Fstar_1_G2_I
# Fstar_1_G2.subs({N_w_A2.dt(N).subs(soldict).dot(A2.z): alpha__A2}) # .subs({-C_71: -
↪C71, -C_81: -C81}).simplify()
Fstar_1_G2 = 0
```

Here, $\{a_1^C : a_2^C,\ a_2^C : a_3^C,\ a_3^C : a_1^C\}$

```
Fstar_2_G2 = Fstar_2_G2_R + Fstar_2_G2_I
Fstar_2_G2.subs({N_w_A2.dt(N).subs(soldict).dot(A2.z): alpha__A2}).subs({N_v_C.dt(N).
↪subs(soldict).dot(A2.x): a_3__C}).subs({N_v_C.dt(N).subs(soldict).dot(A2.y): a_1__C}).
↪subs({-C_72: -C72, -C_82: -C82}).simplify()
```

$$-C_{72}J_2\ddot{q}_7 - C_{82}J_2\ddot{q}_8 - \frac{I_{2_z}\alpha^{A2}\sin\left(q_4\right)}{L} - M_2a_3^C\cos\left(q_4\right) - \frac{M_2a_1^Cl\sin\left(q_4\right)}{L}$$

1.18.11 Variable-Mass Particle, P

$\rightarrow {F_G}^* = -m_Ga^{G^*}$

```
Fstar_P = -m * N_a_P
Fstar_P
```

$$-\left(-u_1u_2\sin\left(q_4\right) + \cos\left(q_4\right)\dot{u}_2\right)m\hat{\mathbf{a}}_{\mathbf{2x}} - \frac{mu_2^2\sin\left(q_4\right)\cos\left(q_4\right)}{L}\hat{\mathbf{a}}_{\mathbf{2y}}$$

$\rightarrow ({F_r}^*)_{GR} = {V_r}^G \cdot {F_G}^*$

```
Fstar_1_P_R = V_1_P.dot(Fstar_P)
Fstar_1_P_R
```

$$0$$

```
Fstar_2_P_R = V_2_P.dot(Fstar_P)
Fstar_2_P_R
```

$$-\left(-u_1u_2\sin\left(q_4\right) + \cos\left(q_4\right)\dot{u}_2\right)m\cos\left(q_4\right)$$

$\rightarrow (F_r^*)_G = (F_r^*)_{GR}$

```
Fstar_1_P = Fstar_1_P_R
Fstar_1_P
```

$$0$$

Here, $\{a_1^P : a_2^P,\ a_2^P : a_3^P,\ a_3^P : a_1^P\}$

```
Fstar_2_P = Fstar_2_P_R
Fstar_2_P.subs({N_v_P.dt(N).subs(soldict).dot(A2.x): a_3__P}).subs({N_v_P.dt(N).
↪subs(soldict).dot(A2.y): a_1__P}).simplify()
```

$$-a_3^P m \cos\left(q_4\right)$$

1.18.12 Generalized Inertia Forces

$\rightarrow F_r^* = (F_r^*)_{G_1} + (F_r^*)_{G_2} + (F_r^*)_P \quad (r = 1, 2)$

```
Fstar_1 = Fstar_1_G1 + Fstar_1_G2 + Fstar_1_P
Fstar_1.subs(soldict).simplify()
```

$$\frac{-I_{1_z} r_1 \left(L\dot{u}_1 + u_1 u_2 \cos\left(q_4\right) + \sin\left(q_4\right)\dot{u}_2\right) + J_1 La \left(\frac{-a\dot{u}_1 + \dot{u}_2 - \frac{a u_2 \cos\left(q_4\right) \dot{q}_4}{L} - \frac{a \sin\left(q_4\right) \dot{u}_2}{L}}{r_1} - \frac{a\dot{u}_1 + \dot{u}_2 + \frac{a u_2 \cos\left(q_4\right) \dot{q}_4}{L} + \frac{a \sin\left(q_4\right) \dot{u}_2}{L}}{r_1}\right)}{Lr_1}$$

```
Fstar_2 = Fstar_2_G1 + Fstar_2_G2 + Fstar_2_P
Fstar_2.subs(soldict).simplify()
```

$$\frac{Ja\left(\frac{-a\dot{u}_1 + \dot{u}_2 - \frac{a u_2 \cos\left(q_4\right) \dot{q}_4}{L} - \frac{a \sin\left(q_4\right) \dot{u}_2}{L}}{r_1} - \frac{a\dot{u}_1 + \dot{u}_2 + \frac{a u_2 \cos\left(q_4\right) \dot{q}_4}{L} + \frac{a \sin\left(q_4\right) \dot{u}_2}{L}}{r_1}\right) - Ja\left(\frac{-a\dot{u}_1 + \dot{u}_2 - \frac{a u_2 \cos\left(q_4\right) \dot{q}_4}{L} - \frac{a \sin\left(q_4\right) \dot{u}_2}{L}}{r_1} - \frac{a\dot{u}_1 + \dot{u}_2 + \frac{a u_2 \cos\left(q_4\right) \dot{q}_4}{L} + \frac{a \sin\left(q_4\right) \dot{u}_2}{L}}{r_1}\right) + \cdots}{Lr_1}$$

Velocity of material ejected at P relative to $A_2 \rightarrow -C(t)g_3^{'}$

$C(t) \rightarrow$ positive

```
C = me.dynamicsymbols('C')
C_t = -C*A2.x
C_t
```

$$-C\hat{\mathbf{a}}_{2\mathbf{x}}$$

1.18.13 Generalized Thrust

$\rightarrow F_r' \triangleq \sum_{i=1}^{N} \mathbf{V}_r^{Pi} \cdot \mathbf{C}^{Pi} \dot{m}_i \quad (r = 1, \ldots, k)$

```
Fprime_1 = V_1_P.dot(C_t)*m.diff()
Fprime_1
```

$$0$$

```
Fprime_2 = V_2_P.dot(C_t)*m.diff()
Fprime_2
```

$$-C\cos\left(q_4\right)\dot{m}$$

1.18.14 Extended Kane's Equations for Variable-Mass Systems

$\rightarrow F_r + F_r^* + F_r' = 0 \quad (r = 1, ..., k)$

Here, $F_r = 0 \rightarrow$ no forces contributing to generalized active forces

```
kane_1 = Fstar_1.simplify() + Fprime_1.simplify()
kane_1.subs(soldict).simplify()
```

$$\frac{-I_{1_z} r_1 \left(L\dot{u}_1 + u_1 u_2 \cos(q_4) + \sin(q_4)\dot{u}_2\right) + J_1 L a \left(\frac{-a\dot{u}_1 + \dot{u}_2 - \frac{a u_2 \cos(q_4)\dot{q}_4}{L} - \frac{a \sin(q_4)\dot{u}_2}{L}}{r_1} - \frac{a\dot{u}_1 + \dot{u}_2 + \frac{a u_2 \cos(q_4)\dot{q}_4}{L} + \frac{a \sin(q_4)\dot{u}_2}{L}}{r_1}\right)}{L r_1}$$

```
kane_2 = Fstar_2 + Fprime_2
kane_2.subs(soldict).simplify()
```

$$\frac{-J_1 L r_2 \left(\frac{(L - a\sin(q_4))\left(-a\dot{u}_1 + \dot{u}_2 - \frac{a u_2 \cos(q_4)\dot{q}_4}{L} - \frac{a\sin(q_4)\dot{u}_2}{L}\right)}{r_1} + \frac{(L + a\sin(q_4))\left(a\dot{u}_1 + \dot{u}_2 + \frac{a u_2 \cos(q_4)\dot{q}_4}{L} + \frac{a\sin(q_4)\dot{u}_2}{L}\right)}{r_1}\right) - J_2 L r_1 \left((L\cos(q_4) - b\sin(q_4))\left(\frac{\left(-\sin(q_4)\dot{q}_4 - \frac{b\cos(q_4)\dot{q}_4}{L}\right)u_2}{r_2} + \frac{\left(\cos(q_4) - \frac{b\sin(q_4)}{L}\right)\dot{u}_2}{r_2}\right) + (L\cos(q_4) + b\sin(q_4))\left(\frac{\left(-\sin(q_4)\dot{q}_4 + \frac{b\cos(q_4)\dot{q}_4}{L}\right)u_2}{r_2} + \frac{\left(\cos(q_4) + \frac{b\sin(q_4)}{L}\right)\dot{u}_2}{r_2}\right)\right) + \frac{L^2 r_1 r_2 \left(-2M_1\dot{u}_2 + (u_1 u_2 \sin(2q_4) - \cos(2q_4)\dot{u}_2 - \dot{u}_2)m - 2C\cos(q_4)\dot{m}\right)}{2} + \frac{r_1 r_2 \left(-2I_{1_z} L\sin(q_4)\dot{u}_1 - I_{1_z} u_1 u_2 \sin(2q_4) + I_{1_z}\cos(2q_4)\dot{u}_2 - I_{1_z}\dot{u}_2 - I_{2_z} u_1 u_2 \sin(2q_4) + I_{2_z}\cos(2q_4)\dot{u}_2 - I_{2_z}\dot{u}_2 + L^2 M_2 u_1 u_2 \sin(2q_4) - L^2 M_2 \cos(2q_4)\dot{u}_2 - L^2 M_2 \dot{u}_2 - M_2 l^2 u_1 u_2 \sin(2q_4) + M_2 l^2 \cos(2q_4)\dot{u}_2 - M_2 l^2 \dot{u}_2\right)}{2}}{L^2 r_1 r_2}$$

```
kane_1_eq = sm.Eq(kane_1.simplify().subs(soldict).simplify().subs(u).simplify(), 0)
kane_1_eq
```

$$-\frac{\left(I_{1_z} r_1^2 + 2J_1 a^2\right)\left(L\dot{u}_1 + u_1 u_2 \cos(q_4) + \sin(q_4)\dot{u}_2\right)}{L r_1^2} = 0$$

```
kane_2_eq = sm.Eq(kane_2.simplify().subs(soldict).simplify().subs(u).simplify(), 0)
kane_2_eq
```

$$\frac{-J_1 r_2^2 \cdot \left(2L^2\dot{u}_2 + 2La^2\sin(q_4)\dot{u}_1 + a^2 u_1 u_2 \sin(2q_4) - a^2\cos(2q_4)\dot{u}_2 + a^2\dot{u}_2\right) - J_2 r_1^2 \left(-L^2 u_1 u_2 \sin(2q_4) + L^2\cos(2q_4)\dot{u}_2 + L^2\dot{u}_2 + b^2 u_1 u_2 \sin(2q_4) - b^2\cos(2q_4)\dot{u}_2 + b^2\dot{u}_2\right) + \frac{r_1^2 r_2^2 \left(-2I_{1_z} L\sin(q_4)\dot{u}_1 - I_{1_z} u_1 u_2 \sin(2q_4) + I_{1_z}\cos(2q_4)\dot{u}_2 - I_{1_z}\dot{u}_2 - I_{2_z} u_1 u_2 \sin(2q_4) + I_{2_z}\cos(2q_4)\dot{u}_2 - I_{2_z}\dot{u}_2 + L^2 M_2 u_1 u_2 \sin(2q_4) - L^2 M_2\cos(2q_4)\dot{u}_2 - L^2 M_2\dot{u}_2 - L^2\cdot\left(2M_1\dot{u}_2 + (-u_1 u_2 \sin(2q_4) + \cos(2q_4)\dot{u}_2 + \dot{u}_2)m + 2C\cos(q_4)\dot{m}\right) - M_2 l^2 u_1 u_2 \sin(2q_4) + M_2 l^2\cos(2q_4)\dot{u}_2 - M_2 l^2\dot{u}_2\right)}{2}}{L^2 r_1^2 r_2^2} = 0$$

1.18.15 References

1.19 Three Link Conical Pendulum

Note: You can download this example as a Python script: `three-link-conical-pendulum.py` or Jupyter notebook: `three-link-conical-pendulum.ipynb`.

This example shows how to simulate a three link conical compound pendulum made up of cylindrical bars connecting particles at the joints. The is an example of a 3D holonomic system. Wikipedia provides a basic description of a conical pendulum.

1.19.1 Derive the Equations of Motion

```
from sympy import symbols
import sympy.physics.mechanics as me

print("Defining the problem.")

# The conical pendulum will have three links and three bobs.
n = 3
```

(continues on next page)

(continued from previous page)

```
# Each link's orientation is described by two spaced fixed angles: alpha and
# beta.

# Generalized coordinates
alpha = me.dynamicsymbols('alpha:{}'.format(n))
beta = me.dynamicsymbols('beta:{}'.format(n))

# Generalized speeds
omega = me.dynamicsymbols('omega:{}'.format(n))
delta = me.dynamicsymbols('delta:{}'.format(n))

# At each joint there are point masses (i.e. the bobs).
m_bob = symbols('m:{}'.format(n))

# Each link is modeled as a cylinder so it will have a length, mass, and a
# symmetric inertia tensor.
l = symbols('l:{}'.format(n))
m_link = symbols('M:{}'.format(n))
Ixx = symbols('Ixx:{}'.format(n))
Iyy = symbols('Iyy:{}'.format(n))
Izz = symbols('Izz:{}'.format(n))

# Acceleration due to gravity will be used when prescribing the forces
# acting on the links and bobs.
g = symbols('g')

# Now defining an inertial reference frame for the system to live in. The Y
# axis of the frame will be aligned with, but opposite to, the gravity
# vector.

I = me.ReferenceFrame('I')

# Three reference frames will track the orientation of the three links.

A = me.ReferenceFrame('A')
A.orient(I, 'Space', [alpha[0], beta[0], 0], 'ZXY')

B = me.ReferenceFrame('B')
B.orient(A, 'Space', [alpha[1], beta[1], 0], 'ZXY')

C = me.ReferenceFrame('C')
C.orient(B, 'Space', [alpha[2], beta[2], 0], 'ZXY')

# Define the kinematical differential equations such that the generalized
# speeds equal the time derivative of the generalized coordinates.
kinematic_differentials = []
for i in range(n):
   kinematic_differentials.append(omega[i] - alpha[i].diff())
   kinematic_differentials.append(delta[i] - beta[i].diff())

# The angular velocities of the three frames can then be set.
```

(continues on next page)

(continued from previous page)

```
A.set_ang_vel(I, omega[0] * I.z + delta[0] * I.x)
B.set_ang_vel(I, omega[1] * I.z + delta[1] * I.x)
C.set_ang_vel(I, omega[2] * I.z + delta[2] * I.x)

# The base of the pendulum will be located at a point O which is stationary
# in the inertial reference frame.
O = me.Point('O')
O.set_vel(I, 0)

# The location of the bobs (at the joints between the links) are created by
# specifiying the vectors between the points.
P1 = O.locatenew('P1', -l[0] * A.y)
P2 = P1.locatenew('P2', -l[1] * B.y)
P3 = P2.locatenew('P3', -l[2] * C.y)

# The velocities of the points can be computed by taking advantage that
# pairs of points are fixed on the referene frames.
P1.v2pt_theory(O, I, A)
P2.v2pt_theory(P1, I, B)
P3.v2pt_theory(P2, I, C)
points = [P1, P2, P3]

# Now create a particle to represent each bob.
Pa1 = me.Particle('Pa1', points[0], m_bob[0])
Pa2 = me.Particle('Pa2', points[1], m_bob[1])
Pa3 = me.Particle('Pa3', points[2], m_bob[2])
particles = [Pa1, Pa2, Pa3]

# The mass centers of each link need to be specified and, assuming a
# constant density cylinder, it is equidistance from each joint.
P_link1 = O.locatenew('P_link1', -l[0] / 2 * A.y)
P_link2 = P1.locatenew('P_link2', -l[1] / 2 * B.y)
P_link3 = P2.locatenew('P_link3', -l[2] / 2 * C.y)

# The linear velocities can be specified the same way as the bob points.
P_link1.v2pt_theory(O, I, A)
P_link2.v2pt_theory(P1, I, B)
P_link3.v2pt_theory(P2, I, C)

points_rigid_body = [P_link1, P_link2, P_link3]

# The inertia tensors for the links are defined with respect to the mass
# center of the link and the link's reference frame.
inertia_link1 = (me.inertia(A, Ixx[0], Iyy[0], Izz[0]), P_link1)
inertia_link2 = (me.inertia(B, Ixx[1], Iyy[1], Izz[1]), P_link2)
inertia_link3 = (me.inertia(C, Ixx[2], Iyy[2], Izz[2]), P_link3)

# Now rigid bodies can be created for each link.
link1 = me.RigidBody('link1', P_link1, A, m_link[0], inertia_link1)
link2 = me.RigidBody('link2', P_link2, B, m_link[1], inertia_link2)
link3 = me.RigidBody('link3', P_link3, C, m_link[2], inertia_link3)
links = [link1, link2, link3]
```

(continues on next page)

(continued from previous page)

```
# The only contributing forces to the system is the force due to gravity
# acting on each particle and body.
forces = []

for particle in particles:
   mass = particle.mass
   point = particle.point
   forces.append((point, -mass * g * I.y))

for link in links:
   mass = link.mass
   point = link.masscenter
   forces.append((point, -mass * g * I.y))

# Make a list of all the particles and bodies in the system.
total_system = links + particles

# Lists of all generalized coordinates and speeds.
q = alpha + beta
u = omega + delta

# Now the equations of motion of the system can be formed.
print("Generating equations of motion.")
kane = me.KanesMethod(I, q_ind=q, u_ind=u, kd_eqs=kinematic_differentials)
fr, frstar = kane.kanes_equations(total_system, loads=forces)
print("Derivation complete.")
```

```
Defining the problem.
```

```
Generating equations of motion.
```

```
Derivation complete.
```

1.19.2 Simulate the System

```
# external
from numpy import radians, linspace, hstack, zeros, ones
from scipy.integrate import odeint
from pydy.codegen.ode_function_generators import generate_ode_function

param_syms = []
for par_seq in [l, m_bob, m_link, Ixx, Iyy, Izz, (g,)]:
   param_syms += list(par_seq)

# All of the links and bobs will have the same numerical values for the
# parameters.

link_length = 10.0  # meters
```

(continues on next page)

(continued from previous page)

```
link_mass = 10.0  # kg
link_radius = 0.5  # meters
link_ixx = 1.0 / 12.0 * link_mass * (3.0 * link_radius**2 + link_length**2)
link_iyy = link_mass * link_radius**2
link_izz = link_ixx

particle_mass = 5.0  # kg
particle_radius = 1.0  # meters

# Create a list of the numerical values which have the same order as the
# list of symbolic parameters.
param_vals = [link_length for x in l] + \
             [particle_mass for x in m_bob] + \
             [link_mass for x in m_link] + \
             [link_ixx for x in list(Ixx)] + \
             [link_iyy for x in list(Iyy)] + \
             [link_izz for x in list(Izz)] + \
             [9.8]

# A function that evaluates the right hand side of the set of first order
# ODEs can be generated.
print("Generating numeric right hand side.")
right_hand_side = generate_ode_function(kane.forcing_full, q, u, param_syms,
                                        mass_matrix=kane.mass_matrix_full,
                                        generator='cython')

# To simulate the system, a time vector and initial conditions for the
# system's states is required.
duration = 10.0
fps = 60.0
t = linspace(0.0, duration, num=int(duration*fps))
x0 = hstack((ones(6) * radians(10.0), zeros(6)))

print("Integrating equations of motion.")
state_trajectories = odeint(right_hand_side, x0, t, args=(dict(zip(param_syms,
                                                                   param_vals)),
                                                          ))
print("Integration done.")
```

```
Generating numeric right hand side.
```

```
Integrating equations of motion.
Integration done.
```

1.19.3 Visualize the System

```
# external
from pydy.viz.shapes import Cylinder, Sphere
from pydy.viz.scene import Scene
from pydy.viz.visualization_frame import VisualizationFrame

# A cylinder will be attached to each link and a sphere to each bob for the
# visualization.

viz_frames = []

for i, (link, particle) in enumerate(zip(links, particles)):

   link_shape = Cylinder(name='cylinder{}'.format(i),
                         radius=link_radius,
                         length=link_length,
                         color='red')

   viz_frames.append(VisualizationFrame('link_frame{}'.format(i), link,
                                        link_shape))

   particle_shape = Sphere(name='sphere{}'.format(i),
                           radius=particle_radius,
                           color='blue')

   viz_frames.append(VisualizationFrame('particle_frame{}'.format(i),
                                        link.frame,
                                        particle,
                                        particle_shape))

# Now the visualization frames can be passed in to create a scene.
scene = Scene(I, O, *viz_frames)

# Provide the data to compute the trajectories of the visualization frames.
scene.times = t
scene.constants = dict(zip(param_syms, param_vals))
scene.states_symbols = q + u
scene.states_trajectories = state_trajectories

scene.display_jupyter()
```

```
VBox(children=(AnimationAction(clip=AnimationClip(duration=10.0,␣
↪tracks=(VectorKeyframeTrack(name='scene/cylin...
```

1.20 3D N-Body Pendulum

Note: You can download this example as a Python script: `3d-n-body-pendulum.py` or Jupyter notebook: `3d-n-body-pendulum.ipynb`.

```
import sympy as sm
import sympy.physics.mechanics as me
import time
import numpy as np
from scipy.integrate import odeint, solve_ivp
from scipy.optimize import fsolve, minimize
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline

import pythreejs as p3js
```

A pendulum consisting on n massless rods of length l. A ball of mass m and radius r is attached to each rod, such that the rod goes through the center of each ball. the center of the ball is fixed at the middle of the rod. the ball rotates around the rod. If two balls collide, they are ideally elastic, with 'spring constant' k. The collision is modeled using the sm.Heaviside(..) function. The balls are ideally slick, so collisions will not affect their rotation.

The total energy does not always remain constant. As method = 'Radau' in solve_ivp gives much 'better' results than no method (e.g. 15% deviation vs. 0.3% in some cases), I assume that this is due to numerical errors in the integration. - of course I do not know for sure.

```
start = time.time()
#=============================================================================
n = 3                    # number of pendulum bodies, labelled 0, 1, .., n-1, must be two␣
↪ore more.
#=============================================================================


m, m1, g, r, l, reibung, k, t = sm.symbols('m, m1, g, r, l, reibung, k, t')
iXX, iYY, iZZ = sm.symbols('iXX, iYY, iZZ')

q = []     #holds the generalized coordinates of each rod
u = []     # generalized angular speeds


A = []     # frames of each rod

Dmc = []   # geometric center of each body
P = []     # points at end of each rod. Dmc_i is between P_i and P_i+1
punkt = [] # marks a red dot on each ball, just used for animation

for i in range(n):
    for j in ('x', 'y', 'z'):
        q.append(me.dynamicsymbols('q' + j + str(i)))
        u.append(me.dynamicsymbols('u' + j + str(i)))

    A.append(me.ReferenceFrame('A' + str(i)))
```

(continues on next page)

(continued from previous page)

```
    Dmc.append(me.Point('Dmc' + str(i)))
    P.append(me.Point('P' + str(i)))
    punkt.append(me.Point('punkt' + str(i)))


N = me.ReferenceFrame('N')        # inertial frame
P0 = me.Point('P0')

# set up the relevant frames, one for each body
rot = []      # for kinetatic equations
rot1 = []     # dto

A[0].orient_body_fixed(N, (q[0], q[1], q[2]), '123')
rot.append(A[0].ang_vel_in(N))
# it is VERY important, that the angular speed be expressed in therms of the 'child frame
↪', otherwise
# the equations of motion become very large!
A[0].set_ang_vel(N, u[0]*A[0].x + u[1]*A[0].y + u[2]*A[0].z )
rot1.append(A[0].ang_vel_in(N))


for i in range(1, n):
    A[i].orient_body_fixed(A[i-1], (q[3*i], q[3*i+1],q[3*i+2]), '123')
    rot.append(A[i].ang_vel_in(N))             # needed for the kinematiic equations␣
↪below
# it is VERY important, that the angular speed be expressed in therms of the 'child frame
↪', otherwise
# the equations of motion become very large!
    A[i].set_ang_vel(N, u[3*i]*A[i].x + u[3*i+1]*A[i].y + u[3*i+2]*A[i].z)
    rot1.append(A[i].ang_vel_in(N))            #         dto.


# locate the various points, and define their speeds
P[0].set_pos(P0, 0.)
P[0].set_vel(N, 0.)             # fixed point
Dmc[0].set_pos(P[0], l/2. * A[0].y)
Dmc[0].v2pt_theory(P[0], N, A[0])
punkt[0].set_pos(Dmc[0], r*A[0].z)             # only for the red dot in the animation
punkt[0].v2pt_theory(Dmc[0], N, A[0])

for i in range(1, n):
    P[i].set_pos(P[i-1], l * A[i-1].y)
    P[i].v2pt_theory(P[i-1], N, A[i-1])
    Dmc[i].set_pos(P[i], l/sm.S(2.) * A[i].y)
    Dmc[i].v2pt_theory(P[i], N, A[i])
    punkt[i].set_pos(Dmc[i], r*A[i].z)
    punkt[i].v2pt_theory(Dmc[i], N, A[i])


# make the list of the bodies
BODY = []
for i in range(n):
```

(continues on next page)

(continued from previous page)

```
    I = me.inertia(A[i], iXX, iYY, iZZ)
    BODY.append(me.RigidBody('body' + str(i), Dmc[i], A[i], m, (I, Dmc[i])))
    BODY.append(me.Particle('punct' + str(I), punkt[i], m1))  # the red dot may have a␣
↪mass


# set up the forces
# weights
FG = [(Dmc[i], -m*g*N.y) for i in range(n)] + [(punkt[i], -m1*g*N.y) for i in range(n)]

# when the balls collide, they are ideally elastic, with 'spring constant' k. They are␣
↪also completely
# slick, so collisions will not affect their rotational speeds
FB = []
for i in range(n):
    for j in range(i+1, n):
        aa = Dmc[j].pos_from(Dmc[i])
        bb = aa.magnitude()
        aa = aa.normalize()
        forceij = (Dmc[j],  k * (2*r - bb) * aa * sm.Heaviside(2.*r - bb))
        FB.append(forceij)
        forceji = (Dmc[i], -k * (2*r - bb) * aa * sm.Heaviside(2.*r - bb))
        FB.append(forceji)

FL = FG + FB       # list of forces

# kinematic equations
kd = []
for i in range(n):
# It is very important that below the frames A[i] be used, not N. Otherwise the␣
↪equations of motion become very large.
    for uv in A[i]:
        kd.append(me.dot(rot[i] - rot1[i], uv))

# Kanes's equations
q1 = q
u1 = u

KM = me.KanesMethod(N, q_ind=q1, u_ind=u1, kd_eqs=kd)
(fr, frstar) = KM.kanes_equations(BODY, FL)

MM = KM.mass_matrix_full
print('MM DS', me.find_dynamicsymbols(MM))
print('MM free symbols', MM.free_symbols)
print('MM contains {} operations'.format(sum([MM[i, j].count_ops(visual=False)
        for i in range(MM.shape[0]) for j in range(MM.shape[1])])), '\n')

force = KM.forcing_full
print('force DS', me.find_dynamicsymbols(force))
print('force free symbols', force.free_symbols)
print('force contains {} operations'.format(sum([force[i].count_ops(visual=False)
        for i in range(force.shape[0])])), '\n')
```

(continues on next page)

(continued from previous page)

```
# set up the energie equations. Absent ary friction the total energie should be cnstant
pot_energie = sum([m*g*me.dot(Dmc[i].pos_from(P[0]), N.y) for i in range(n)]) +␣
↪sum([m1*g*me.dot(punkt[i]
                        .pos_from(P[0]), N.y) for i in range(n)])
kin_energie = sum([BODY[i].kinetic_energy(N) for i in range(2*n)])
spring_energie = sm.S(0.)
for i in range(n):
    for j in range(i+1, n):
        aa = Dmc[j].pos_from(Dmc[i])
        bb = aa.magnitude()
        aa = aa.normalize()
        spring_energie  += 0.5 * k * (2*r - bb)**2 * sm.Heaviside(2.*r - bb)


# position of the centers of the balls and the red dots on the ball. Needed for the␣
↪animation
Dmc_loc = []
punkt_loc = []
for i in range(n):
    Dmc_loc.append([me.dot(Dmc[i].pos_from(P[0]), uv) for uv in N])
    punkt_loc.append([me.dot(punkt[i].pos_from(P[0]), uv) for uv in N])

# Lambdification
qL = q1 + u1
pL = [m, m1, g, r, l, iXX, iYY, iZZ, reibung, k]

MM_lam = sm.lambdify(qL + pL, MM, cse=True)
force_lam = sm.lambdify(qL + pL, force, cse=True)

pot_lam = sm.lambdify(qL + pL, pot_energie, cse=True)
kin_lam = sm.lambdify(qL + pL, kin_energie, cse=True)
spring_lam = sm.lambdify(qL + pL, spring_energie, cse=True)

Dmc_loc_lam = sm.lambdify(qL + pL, Dmc_loc, cse=True)
punkt_loc_lam = sm.lambdify(qL + pL, punkt_loc, cse=True)

print('it took {:.3f} sec to set up Kanes equations'.format(time.time() - start))
```

```
MM DS {qx2(t), qz2(t), qx1(t), qy1(t), qy2(t), qz1(t)}
MM free symbols {l, iYY, m, t, iZZ, iXX, r, m1}
MM contains 2067 operations

force DS {qx2(t), qx0(t), uz1(t), qy2(t), ux2(t), uy1(t), ux0(t), qz0(t), uy2(t), qy1(t),
↪ uy0(t), ux1(t), uz0(t), qz2(t), qx1(t), qy0(t), uz2(t), qz1(t)}
force free symbols {iYY, l, k, t, r, g, m1, m, iZZ, iXX}
```

```
force contains 12061 operations
```

```
it took 5.966 sec to set up Kanes equations
```

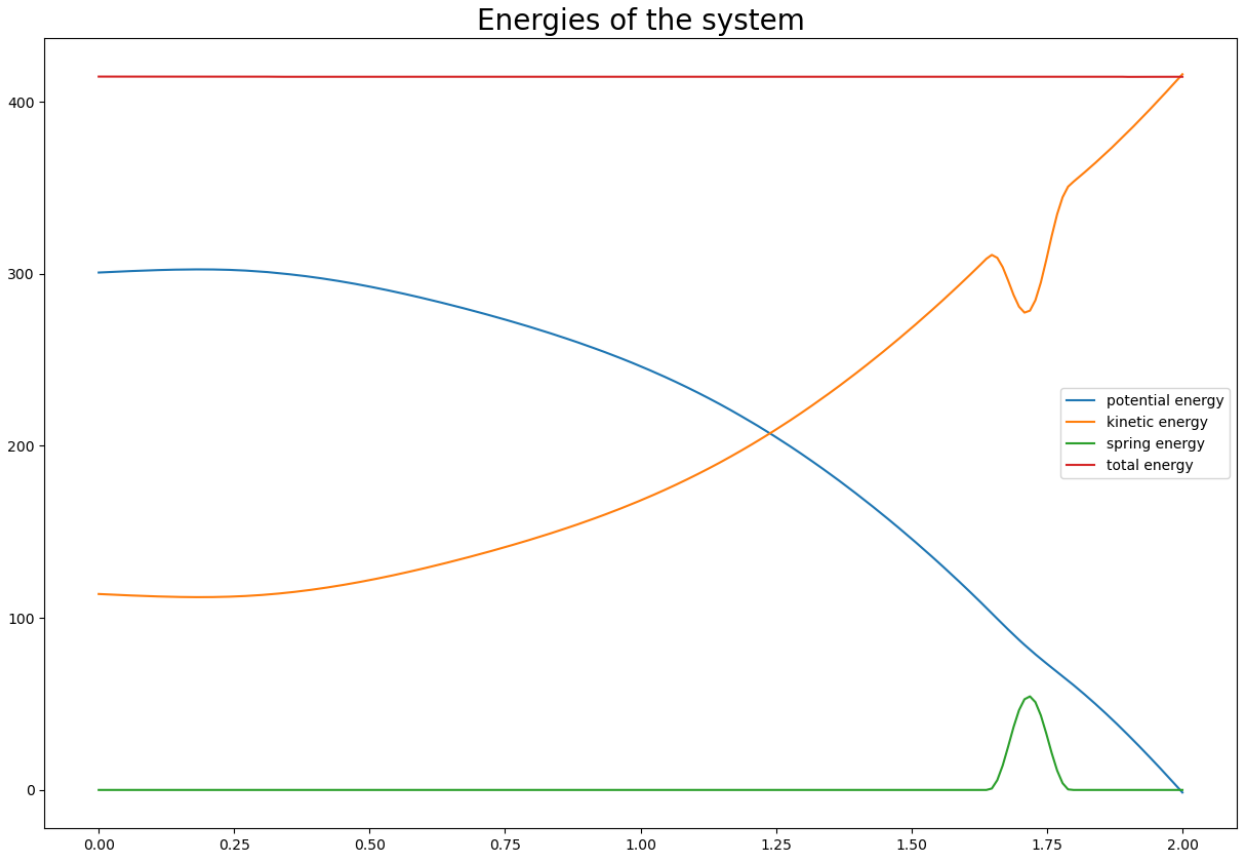
```
# numerical integration
start = time.time()

# Input values
#======================================================================
r1 = 1.5                                # radius of the ball
m1 = 1.                                 # mass of the ball
m11 = m1 / 5.                           # mass of the red dot
l1 = 6.                                 # length of the massless rod of the pendulum
k1 = 1000.                              # 'spring constant' of the balls
reibung1 = 0.                           # friction of the ball against the rod

q1x, q1y, q1z = 0.2, 0.2, 0.2           # initial deflection of the first rod
                                        # for simplicity, I assume that the pendulum is␣
↪straight initially

omega1 = 7.5                            # initial rotation speed of ball_i around A[i].y
u1x, u1y, u1z = 0., omega1, 0.          # initial rotational speed of the ball

intervall = 2.
#======================================================================
schritte = 100 * int(intervall)
times = np.linspace(0., intervall, schritte)
iXX1 = 2./5. * m1 * r1**2               # from the internet
iYY1 = iXX1
iZZ1 = iXX1

#pL = [m, g, r, l, iXX, iYY, iZZ, reibung, k]
pL_vals = [m1, m11, 9.8, r1, l1, iXX1, iYY1, iZZ1, reibung1, k1]

y0 = [q1x, q1y, q1z] + [0., 0., 0.] * (n-1) + [u1x, u1y, u1z] + [0. ,u1y, 0.] * (n-1)
print('Starting values: ', y0)

t_span = (0., intervall)

def gradient(t, y, args):
    sol = np.linalg.solve(MM_lam(*y, *args), force_lam(*y, *args))
    return np.array(sol).T[0]

resultat1 = solve_ivp(gradient, t_span, y0, t_eval = times, args=(pL_vals,), method=
↪'Radau')

resultat = resultat1.y.T
print('shape of resultat', resultat.shape)
event_dict = {-1: 'Integration failed', 0: 'Integration finished successfully', 1: 'some␣
↪termination event'}
print(event_dict[resultat1.status])
print("To numerically integrate an intervall of {:.3f} sec the routine cycled {} times␣
↪and it took {:.3f} sec"
      .format(intervall, resultat1.nfev, time.time() - start))
```

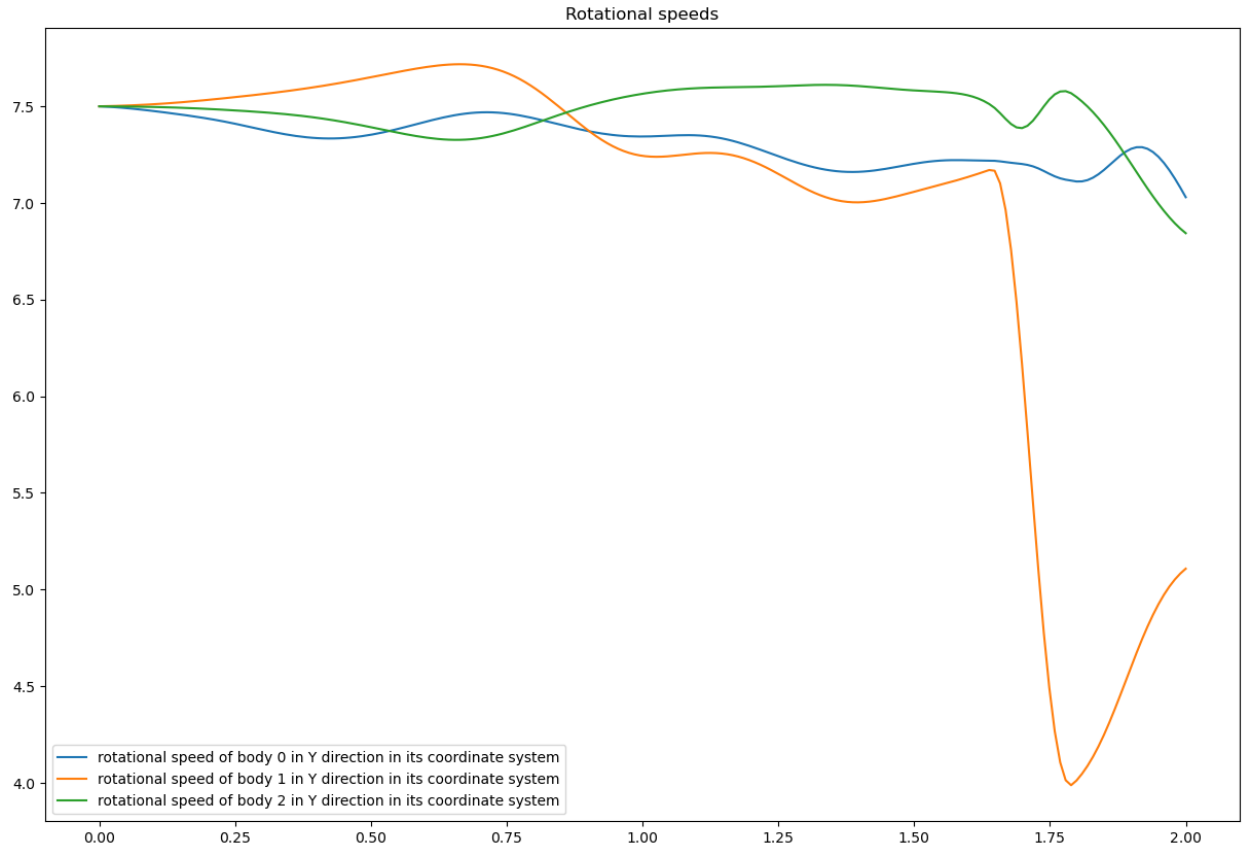
```
Starting values:  [0.2, 0.2, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 7.5, 0.0, 0.0, 7.5,␣
↪0.0, 0.0, 7.5, 0.0]
```

```
shape of resultat (200, 18)
Integration finished successfully
To numerically integrate an intervall of 2.000 sec the routine cycled 826 times and it␣
↪took 1.061 sec
```

```
# plot the energies

pot_np = np.empty(schritte)
kin_np = np.empty(schritte)
spring_np = np.empty(schritte)
total_np = np.empty(schritte)

for i in range(schritte):
    zeit = times[i]
    pot_np[i] = pot_lam(*[resultat[i, j] for j in range(resultat.shape[1])], *pL_vals)
    kin_np[i] = kin_lam(*[resultat[i, j] for j in range(resultat.shape[1])], *pL_vals)
    spring_np[i] = spring_lam(*[resultat[i, j] for j in range(resultat.shape[1])], *pL_
↪vals)
    total_np[i] = pot_np[i] + kin_np[i] + spring_np[i]

if reibung1 == 0.:
    total_max = np.max(total_np)
    total_min = np.min(total_np)
    print('deviation of total energy from being constant is {:.5f} % of max. total energy
↪'
          .format((total_max - total_min)/total_max*100) )

fig, ax = plt.subplots(figsize=(15, 10))
ax.plot(times, pot_np, label='potential energy')
ax.plot(times, kin_np, label='kinetic energy')
ax.plot(times, spring_np, label='spring energy')
ax.plot(times, total_np, label='total energy')
ax.set_title('Energies of the system', fontsize=20)
ax.legend();

#plot the main rotational speeds, uy_r
fig, ax = plt.subplots(figsize=(15, 10))
for i in range(n, 2*n):
    ax.plot(times, resultat[:, 3*i+1],
        label='rotational speed of body {} in Y direction in its coordinate system'.
↪format(i-n))
ax.set_title('Rotational speeds')
ax.legend();
```

```
deviation of total energy from being constant is 0.03864 % of max. total energy
```


Energies of the system
400
300
200
100
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
potential energy
kinetic energy
spring energy
total energy



Animation using pythreejs. This is basically copied from a program by Jason Moore, just adapted to my needs here.

NOTE: the 'reference frame' for pythreejs seems to be: X - axis downwards, color red Y - axis to the right, color green (hence:) Z - axis pointing to the observer, color blue

Rotation is used to transform my coordinate system used above to set up the equations of motion to the one prescribed by pythreejs.

If you know from the beginning, that you want to use pythreejs it is probably better to use its orientation of coordinates, when setting up Kane's equations. Saves the trouble of guessing, which rotation is correct. I am not sure my rotation is fully correct, just played around until it 'looked' reasonable.

```
winkel = sm.symbols('winkel')
Rotation1 = sm.Matrix([[sm.cos(winkel), -sm.sin(winkel), 0], [sm.sin(winkel), sm.
↪cos(winkel), 0], [0., 0., 1]])
Rot_lam = sm.lambdify(winkel, Rotation1.T, cse=True)
Rotation = Rot_lam(np.pi/2.)

TC_store = []
TR_store = []
TP_store = []
body_mesh_store = []
track_store = []
farben = ['orange', 'blue', 'green', 'yellow', 'red']
for i in range(n):
#for its mass center
    TC = sm.eye(4)
```

(continues on next page)

(continued from previous page)

```
    TC[:3, :3] = (A[i].dcm(N)) * Rotation
    TC = TC.reshape(16, 1)
    TC_lam = sm.lambdify(qL + pL, TC, cse=True)

    TR = sm.eye(4)
    TR[:3, :3] = (A[i].dcm(N)) * Rotation
    TR = TR.reshape(16, 1)
    TR_lam = sm.lambdify(qL + pL, TR, cse=True)

    TP = sm.eye(4)
    TP[:3, :3] = (A[i].dcm(N)) * Rotation
    TP = TP.reshape(16, 1)
    TP_lam = sm.lambdify(qL + pL, TP, cse=True)


# store the information about the body, expressed in TAc for every time step.
    TCs = []   # for the ball
    TRs = []   # for the rod
    TPs = []   # for the red dot

# Create the TAs, containing 'one TA' for each time step
# resultat contains the results of the numeric integration.
# where the numeric integration was evaluated
# scala is the factor by which the position of the body is changed, to keep it on the␣
↪screen.
    scala = 1.
    for k in range(resultat.shape[0]):
        zeit = times[i]
        TCi = TC_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_vals)  #␣
↪the balls
        TRi = TR_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_vals)  #␣
↪the rod
        TPi = TP_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_vals)  #␣
↪the dot

# TAi[12], TAi[13], TAi[14] hold the location of A2 w.r.t. N.
# As the axis chosen for solving the equations of motion, and the axis given by␣
↪pythreejs do not
# coincide, the values for TAi[..] must be given accordingly.
# of course here different locations for center of ball and center of mass.
        TRi[12] = -Dmc_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
↪vals)[i][1]
        TRi[13] = Dmc_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
↪vals)[i][0] / scala
        TRi[14] = Dmc_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
↪vals)[i][2] / scala

        TCi[12] = -Dmc_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
↪vals)[i][1]
        TCi[13] = Dmc_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
↪vals)[i][0] / scala
        TCi[14] = Dmc_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
```

(continues on next page)

(continued from previous page)

```
↪vals)[i][2] / scala

        TPi[12] = -punkt_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])],␣
↪*pL_vals)[i][1]
        TPi[13] = punkt_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
↪vals)[i][0] / scala
        TPi[14] = punkt_loc_lam(*[resultat[k, l] for l in range(resultat.shape[1])], *pL_
↪vals)[i][2] / scala

        TRs.append(TRi.squeeze().tolist())
        TCs.append(TCi.squeeze().tolist())
        TPs.append(TPi.squeeze().tolist())

    TC_store.append(TCs)
    TR_store.append(TRs)
    TP_store.append(TPs)

# Create the objects, which will move
# 1. The ball
    body_geom_C = p3js.SphereGeometry(r1, 12, 12)
    body_material_C = p3js.MeshStandardMaterial(color=farben[i], wireframe=False)
    body_mesh_C = p3js.Mesh(geometry=body_geom_C, material=body_material_C, name='ball_'␣
↪+ str(i))

# 2. Rod
    body_geom_R = p3js.CylinderGeometry(radiusTop=0.05, radiusBottom=0.05, height=l1,
                    radialSegments=6, heightSegments=10, openEnded=False)
    body_material_R = p3js.MeshStandardMaterial(color='black', wireframe=False)
    body_mesh_R = p3js.Mesh(geometry=body_geom_R, material=body_material_R, name='rod_'␣
↪+ str(i))

# 3. the dot
    body_geom_P = p3js.SphereGeometry(0.25, 12, 12)
    body_material_P = p3js.MeshStandardMaterial(color='red', wireframe=False)
    body_mesh_P = p3js.Mesh(geometry=body_geom_P, material=body_material_P, name='punkt_
↪' + str(i))

# locate the body in 3D space and add the coordinate system of the body
    body_mesh_R.matrixAutoUpdate = False
    body_mesh_R.add(p3js.AxesHelper(0.1))  # length of the axis of the ball system A2
    body_mesh_R.matrix = TR_store[i][0]             # starting point of the animation

    body_mesh_C.matrixAutoUpdate = False
    body_mesh_C.add(p3js.AxesHelper(0.01))    # length of the axis of the center of mass␣
↪system A2
    body_mesh_C.matrix = TC_store[i][0]          # starting point of the animation

    body_mesh_P.matrixAutoUpdate = False
    body_mesh_P.add(p3js.AxesHelper(0.01))    # length of the axis of the center of mass␣
↪system A2
    body_mesh_P.matrix = TP_store[i][0]          # starting point of the animation
```

(continues on next page)

(continued from previous page)

```
    body_mesh_store.append(body_mesh_C)
    body_mesh_store.append(body_mesh_R)
    body_mesh_store.append(body_mesh_P)


# Create the 'picture'.
# all the 'paramters' are taken by trial and error.
view_width = 1200
view_height = 400

# Values just found by trial an error.
if n == 3:
    p1, p2 = 7, 7
    p3 = 35
elif n == 4:
    p1, p2 = 5, 5
    p3 = 50
elif n == 5:
    p1, p2 = 5, 5
    p3 = 65
else:
    p1, p2 = 5, 5
    p3 = 25
camera = p3js.PerspectiveCamera(position=[p1, p2, p3],
                                up=[-1.0, 0.0, 0.0],
                                aspect=view_width/view_height)

key_light = p3js.DirectionalLight(position=[0, 0, 10])
ambient_light = p3js.AmbientLight()

axes = p3js.AxesHelper(20)
print(p1, p2, p3)
children = []
for i in range(3*n):
    children = children + [body_mesh_store[i], axes, camera, key_light, ambient_light]

scene = p3js.Scene(children=children)
controller = p3js.OrbitControls(controlling=camera)
renderer = p3js.Renderer(camera=camera, scene=scene, controls=[controller],
                         width=view_width, height=view_height)

# Create the action, simply copied from JM's lecture.

for i in range(n):
    eigenname = 'ball_'+str(i)
    track_C = p3js.VectorKeyframeTrack(
        name="scene/" + eigenname + ".matrix",
        times=times,
        values=TC_store[i])

    eigenname = 'rod_' + str(i)
```

(continues on next page)

(continued from previous page)

```
    track_R = p3js.VectorKeyframeTrack(
        name="scene/" + eigenname + ".matrix",
        times=times,
        values=TR_store[i])

    eigenname = 'punkt_' + str(i)
    track_P = p3js.VectorKeyframeTrack(
        name="scene/" + eigenname + ".matrix",
        times=times,
        values=TP_store[i])

    track_store += [track_C] + [track_R] + [track_P]

duration = times[-1] - times[0]
clip = p3js.AnimationClip(tracks=track_store, duration=duration)
action = p3js.AnimationAction(p3js.AnimationMixer(scene), clip, scene)
renderer
```

```
7 7 35
```

```
Renderer(camera=PerspectiveCamera(aspect=3.0, position=(7.0, 7.0, 35.0),␣
↪projectionMatrix=(1.0, 0.0, 0.0, 0.0,...
```

```
action
```

```
AnimationAction(clip=AnimationClip(duration=2.0, tracks=(VectorKeyframeTrack(name='scene/
↪ball_0.matrix', times...
```

1.21 Examples

Fig. 3: Linear mass-spring-damper system with gravity.

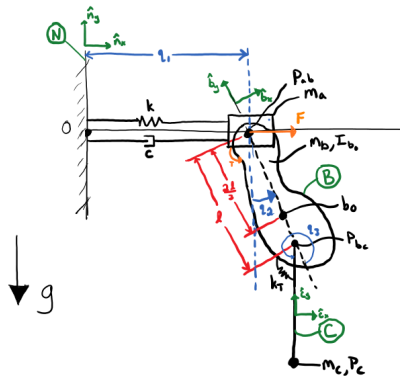


Fig. 4: A double compound and simple pendulum.

Fig. 5: Three link conical compound pendulum.

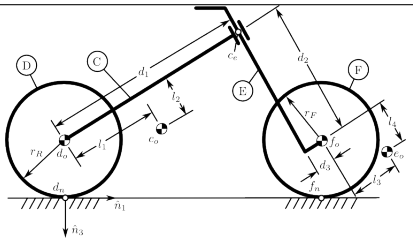


Fig. 6: Carvallo-Whipple bicycle model.

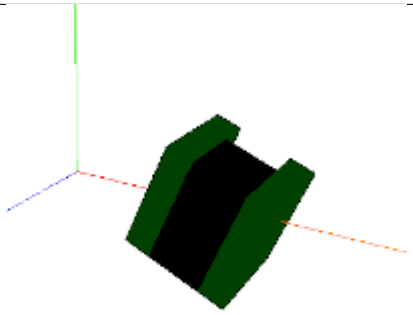


Fig. 7: Astrobee free-flying ISS robot.

Fig. 8: 3D perpendicular axis double pendulum that exhibits chaos.

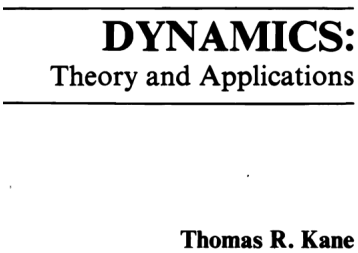


Fig. 9: Exercises from Chapter 2 of Kane & Levinson 1985.

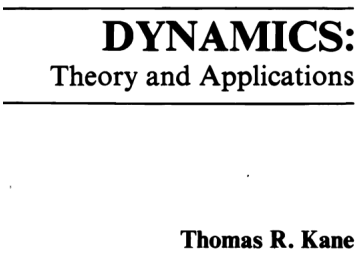


Fig. 10: Exercises from Chapter 3 of Kane & Levinson 1985.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Smith2016] Smith, T., Barlow, J., Bualat, M., Fong, T., Provencher, C., Sanchez, H., & Smith, E. (2016). Astrobee: A new platform for free-flying robotics on the international space station.

[Whipple1899] Whipple, Francis J. W. "The Stability of the Motion of a Bicycle." Quarterly Journal of Pure and Applied Mathematics 30 (1899): 312–48.

[Carvallo1899] Carvallo, E. Théorie Du Mouvement Du Monocycle et de La Bicyclette. Paris, France: Gauthier-Villars, 1899.

[Moore2012] Moore, Jason K. "Human Control of a Bicycle." Doctor of Philosophy, University of California, 2012. http://moorepants.github.io/dissertation.

[Meijaard2007] Meijaard, J. P., Jim M. Papadopoulos, Andy Ruina, and A. L. Schwab. "Linearized Dynamics Equations for the Balance and Steer of a Bicycle: A Benchmark and Review." Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences 463, no. 2084 (August 8, 2007): 1955–82.

[Basu-Mandal2007] Basu-Mandal, Pradipta, Anindya Chatterjee, and J.M Papadopoulos. "Hands-Free Circular Motions of a Benchmark Bicycle." Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences 463, no. 2084 (August 8, 2007): 1983–2003. https://doi.org/10.1098/rspa.2007.1849.

[Ge1982] Ge, Z., and Cheng, Y. (June 1, 1982). "Extended Kane's Equations for Nonholonomic Variable Mass System." ASME. J. Appl. Mech. June 1982; 49(2): 429–431. https://doi.org/10.1115/1.3162105

[Kane1978] Kane, T.R., 1978. Nonholonomic multibody systems containing gyrostats. In Dynamics of Multibody Systems (pp. 97-107). Springer, Berlin, Heidelberg.

PYTHON MODULE INDEX

p

INDEX

Symbols

B

C

D

E