PyDy Distribution Documentation

Release 0.5.0

PyDy Authors

Dec 10, 2019

Contents

This is the central page for all PyDy's Documentation.

CHAPTER 1

PyDy

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 and eventually a physics abstraction layer which utilizes a variety of backends that can provide the user with their desired workflow, including:

- Model specification
- Equation of motion generation
- Simulation
- Visualization
- 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, and matplotlib (i.e. the SciPy stack) which provide additional code for numerical analyses, simulation, and visualization.

1.1 Installation

PyDy has hard dependencies on the following software[1]:

- 2.7 <= Python < 3.0 or Python >= 3.5
- setuptools >= 20.7.0

[1] We only test PyDy with these minimum dependencies; these module versions are provided in the Ubuntu 16.04 packages. Previous versions may work.

- NumPy >= 1.11.0
- SciPy >= 0.17.1
- SymPy >= 0.7.6.1
- PyWin32 >= 219 (Windows Only)

PyDy has optional dependencies on these packages:

- 4.0.0 <= Jupyter Notebook < 5.0.0
- 4.0.0 <= ipywidgets < 5.0.0
- Theano >= 0.8.0
- Cython >= 0.23.4

The examples may require these dependencies:

- matplotlib >= 1.5.1
- version_information

It's best to install the SciPy Stack dependencies using the instructions provided on the SciPy website first. We recommend the conda package manager and the Anaconda distribution for easy cross platform installation.

Once the dependencies are installed, the latest stable version of the package can be downloaded from PyPi[2]:

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

and extracted and installed[3]:

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

Or if you have the pip package manager installed you can simply type:

```
$ pip install pydy
```

Or if you have conda you can type:

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

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

```
$ conda install -c pydy pydy-examples
```

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:

[2] Change `X.X.X` to the latest version number.
[3] For system wide installs you may need root permissions (perhaps prepend commands with `sudo`).

```
/ / / / / / / / /
-----------------
  |     |      |    | g
  \    | |     |    V
k /    --- c   |
  |     |      | x, v
 --------     V
 |  m    | -----
 --------
    | F
    V
```

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(forces, particles)
```

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: 1.0,
                        damping: 0.2, 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)
plt.legend((str(position), str(speed)))
plt.show()
```

1.3 Documentation

The documentation is hosted at http://pydy.readthedocs.org but you can also build them from source using the following instructions.

To build the documentation you must install the dependencies:

- Sphinx
- numpydoc

To build the HTML docs, run Make from within the `docs` directory:

```
$ cd docs
$ make html
```

You can then view the documentation from your preferred web browser, for example:

```
$ firefox _build/html/index.html
```

1.4 Modules and Packages

1.4.1 Code Generation (codegen)

This package provides code generation facilities. It generates functions that can numerically evaluate the right hand side of the ordinary differential equations generated with sympy.physics.mechanics with three different backends: SymPy's lambdify, Theano, and Cython.

1.4.2 Models (models.py)

The models module provides some canned models of classic systems.

1.4.3 Systems (system.py)

The System module provides a `System` class to manage simulation of a single system.

1.4.4 Visualization (viz)

This package provides tools to create 3D animated visualizations of the systems. The visualizations utilize WebGL and run in a web browser. They can also be embedded into an IPython notebook for added interactivity.

1.5 Development Environment

The source code is managed with the Git version control system. To get the latest development version and access to the full repository, clone the repository from Github with:

```
$ git clone https://github.com/pydy/pydy.git
```

You should then install the dependencies for running the tests:

- nose: 1.3.7
- phantomjs: 1.9.0

1.5.1 Isolated Environments

It is typically advantageous to setup a virtual environment to isolate the development code from other versions on your system. There are two popular environment managers that work well with Python packages: virtualenv and conda.

The following installation assumes you have virtualenvwrapper in addition to virtualenv and all the dependencies needed to build the various packages:

```
$ mkvirtualenv pydy-dev
(pydy-dev)$ pip install numpy scipy cython nose theano sympy ipython "notebook<5.0"
↪"ipywidgets<5.0" version_information
(pydy-dev)$ pip install matplotlib # make sure to do this after numpy
(pydy-dev)$ git clone git@github.com:pydy/pydy.git
(pydy-dev)$ cd pydy
(pydy-dev)$ python setup.py develop
```

Or with conda:

```
$ conda create -c pydy -n pydy-dev setuptools numpy scipy ipython "notebook<5.0"
↪"ipywidgets<5.0" cython nose theano sympy matplotlib version_information
$ source activate pydy-dev
(pydy-dev)$ git clone git@github.com:pydy/pydy.git
(pydy-dev)$ cd pydy
(pydy-dev)$ conda develop .
```

The full Python test suite can be run with:

```
(pydy-dev)$ nosetests
```

For the JavaScript tests the Jasmine and blanket.js libraries are used. Both of these libraries are included in pydy.viz with the source. To run the JavaScript tests:

```
cd pydy/viz/static/js/tests && phantomjs run-jasmine.js SpecRunner.html && cd ../../..
↪/../../
```

1.6 Benchmark

Run the benchmark to test the n-link pendulum problem with the various backends:

```
$ python bin/benchmark_pydy_code_gen.py <max # of links> <# of time steps>
```

1.7 Related Packages

These are various related and similar Python packages:

- https://github.com/cdsousa/sympybotics
- https://pypi.python.org/pypi/Hamilton
- https://pypi.python.org/pypi/arboris
- https://pypi.python.org/pypi/PyODE
- https://pypi.python.org/pypi/odeViz
- https://pypi.python.org/pypi/ARS
- https://pypi.python.org/pypi/pymunk

1.8 Citation

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.

1.9 Questions, Bugs, Feature Requests

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

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

1.10 Release Notes

1.10.1 0.5.0

- 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]

1.10.2 0.4.0

- 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]

1.10.3 0.3.1

- 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]

1.10.4 0.3.0

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]

1.10.5 0.2.1

- Unbundled unnecessary files from tar ball.

1.10.6 0.2.0

- 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 system module

1.11.1 system

Introduction

API

1.12 models module

1.12.1 models

Introduction

The *pydy/models.py* 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_a_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.

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],
[-(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)]])
```

API

1.13 codegen package

1.13.1 codegen

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.

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 2.7.

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
==========
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 simply 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)
```

(continues on next page)

(continued from previous page)

```
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,
              <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]
```

(continues on next page)

(continued from previous page)

```
              );
/*

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()
>>> 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
```

1.13.2 codegen API

1.14 viz package

1.14.1 viz

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.

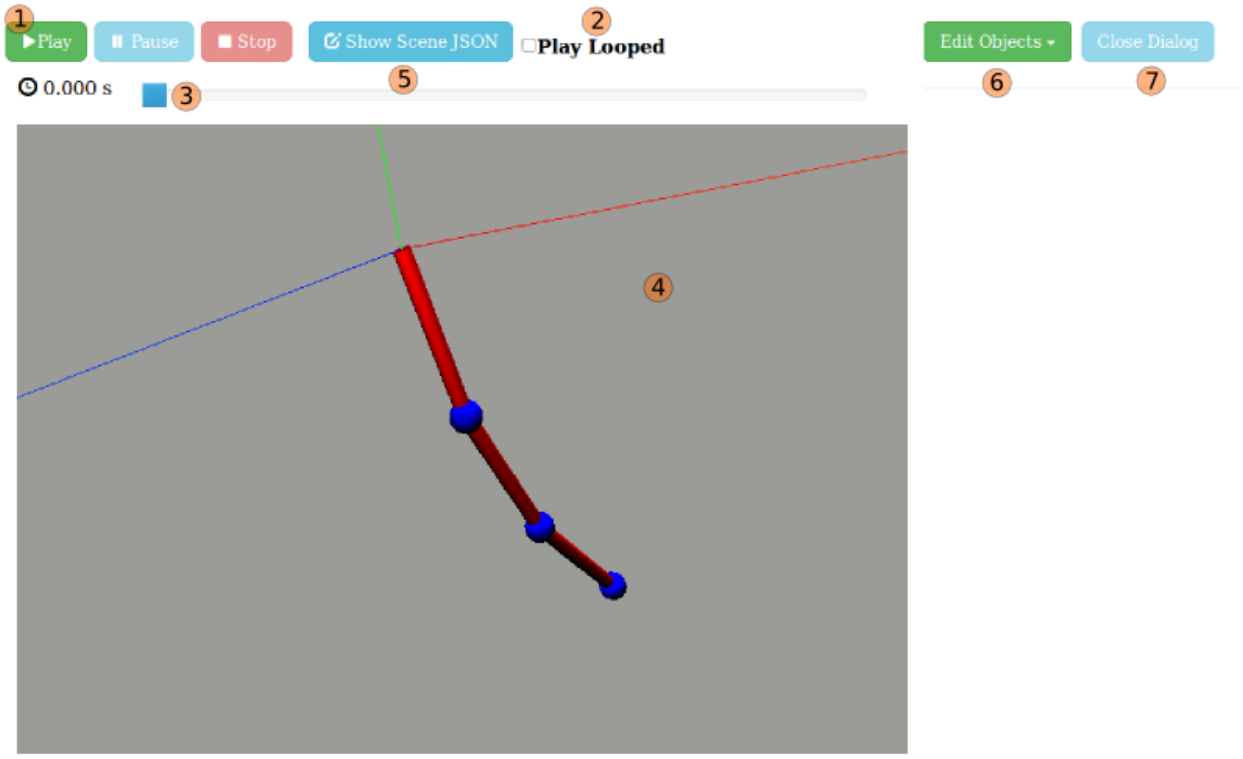
The frontend is based on three.js, a popular interface to the WebGraphics Library (WegGL). The package provides a Python wrapper for some basic functionality for Three.js i.e Geometries, Lights, Cameras etc.

1.14.2 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.

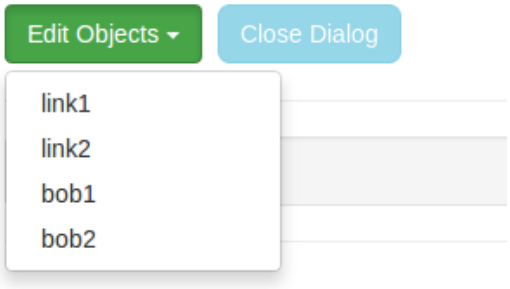
(3) Time Slider This is used to traverse to the particular frame in animation, by sliding the slider forward and backward. When the animation is running it will continue from the point where the slider is slid to.

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

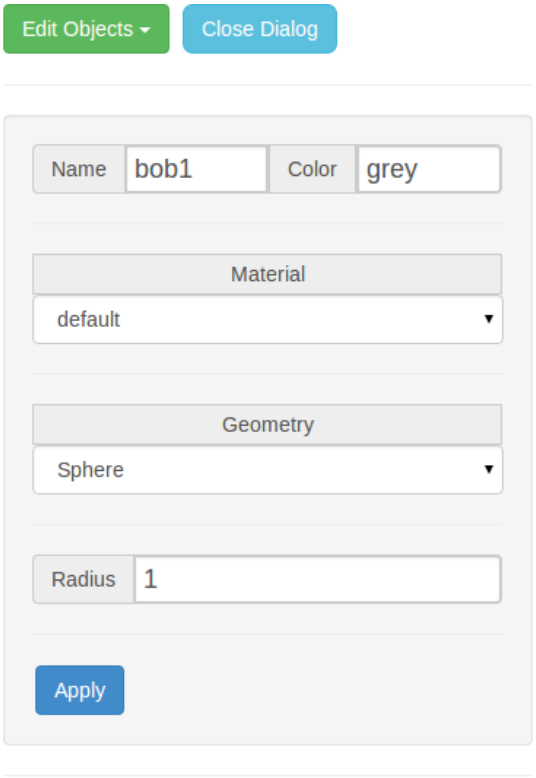
- Mouse wheel to zoom in, zoom out.
- Click and drag to rotate camera.

(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:



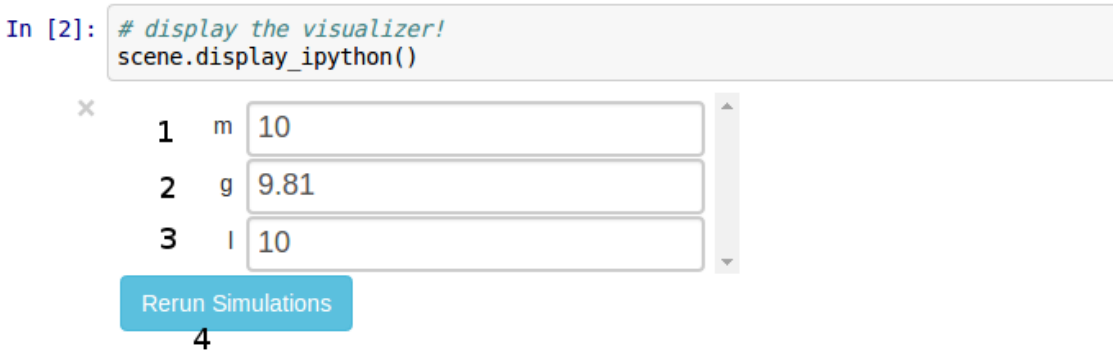
On clicking any object from the dropdown, a dialog box opens up, containing the existing info on that object. The info can be edited. After editing click the "Apply" button for the changes to be reflected in the canvas (4).



(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.

1.14.3 API

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

Python Modules Reference

Shapes

Shape

Cube

Cylinder

Cone

Sphere

Circle

Plane

Tetrahedron

Octahedron

Icosahedron

Torus

TorusKnot

Tube

VisualizationFrame

Cameras

Perspective Camera

Orthographic Camera

Lights

PointLight

Scene

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 sub-modules 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.

1.15 Tutorials

1.15.1 Tutorials(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:

- Mass Spring Damper example
- Inverted pendulum model of a standing human

1.15.2 Tutorials(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

1.16 Indices and tables

- genindex
- modindex
- search