PyDy Distribution Documentation

Release 0.3.1

PyDy Authors

January 10, 2016

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, 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]:

- Python >= 2.7, >= 3.3
- setuptools[2]
- NumPy >= 1.8.1
- SciPy >= 0.13.3
- SymPy >= 0.7.4.1

PyDy has optional dependencies on these packages:

- IPython >= 3.0.0 (plus ipywidgets >= 4.0.0 if IPython >= 4.0.0) [3]

[1] We only test PyDy with these minimum dependencies; these module versions are provided in the Ubuntu 14.04 packages. Previous versions may work.
[2] setuptools >= 8.0 is required if development versions of SymPy are used.
[3] Note that ipywidgets will have to be installed separately until the fix for this change is included in IPython 4.0.

- Theano >= 0.7.0
- Cython >= 0.20.1

The examples may require these dependencies:

- matplotlib
- 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[4]:

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

and extracted and installed[5]:

```
$ 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 pydy 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:

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

Derive the system:

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

```
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: 1.3.0

- 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[all] ipywidgets version_informat
(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 ipython-notebook ipywidgets cython
$ source activate pydy-dev
(pydy-dev)$ git clone git@github.com:pydy/pydy.git
(pydy-dev)$ cd pydy
(pydy-dev)$ python setup.py 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 0.3.1

- 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]

1.10.2 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]

1.10.3 0.2.1

- Unbundled unnecessary files from tar ball.

1.10.4 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

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.

Examples

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)
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()
```

API

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.

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.

constants_symbols

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

coordinates

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

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

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_fun`
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.

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()
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()`).

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`.

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.

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)}
```

specifieds_symbols
A set of the dynamicsymbols you must specify.

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

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.

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. Since this attribute is not checked for compatibility, the user becomes responsible to supply it correctly.

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([
[
[
[                                                                    (-c0*v0(t) + c1*v1(t) + g*m0 - k0*x0(t) + k1*x1
[-(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) -
```

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:

```
       ----------------
        |      |      |    | g
        \      | |    |    V
```

```
    k0 /    --- c0 |
       |     |     | x0, v0
      ---------    V
      |  m0   | -----
      ---------    |
       | |   |     |
       \ v   | |   |
    k1 / f0 --- c1 |
       |     |     | x1, v1
      ---------    V
      |  m1   | -----
      ---------
         | f1
         V
```

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.

```
        |
        v   y    o
        g   ^   /
            |  /
        ----|-/--  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.

1.13 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)
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]
             );
/*

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.])
```

1.13.2 codegen API

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.

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.

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

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.

Notes

The generated function still supports the pre-0.3.0 extra argument style, i.e. args = {'constants': ..., 'specified': ...}, but only if `constants_arg_type` and `specifieds_arg_type` are both set to None. This functionality is deprecated and will be removed in 0.4.0, so it's best to adjust your code to support the new argument types. See the docstring for the generated function for more info on the new style of arguments.

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

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

`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.

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.

comma_lists()

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

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`.

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.

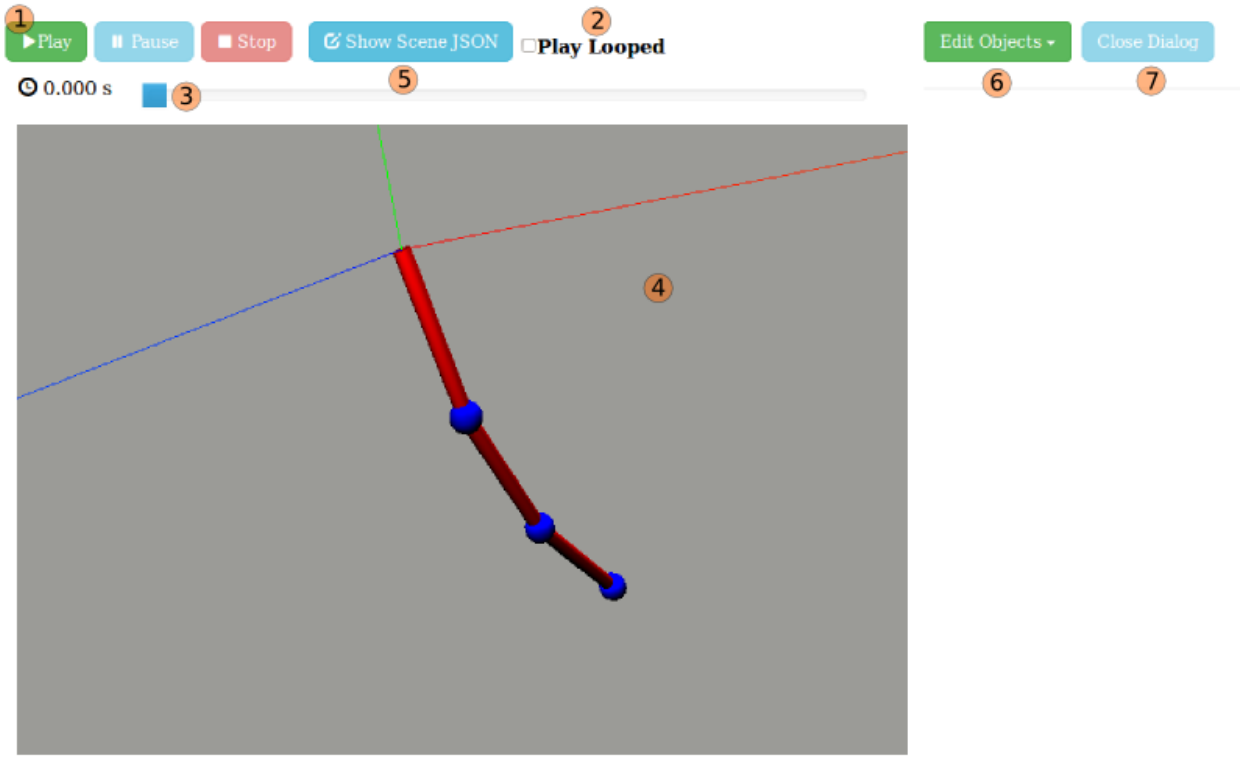
The frontend is based on three.js, a popular interface to the WebGraphics Library (WegGL). The package provides a Python wrapper for some basic functionality for Three.js i.e Geometries, Lights, Cameras etc.

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.

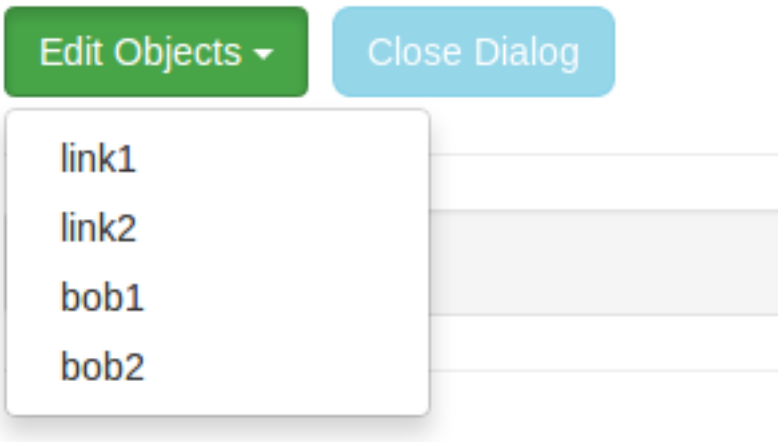
(3) Time Slider This is used to traverse to the particular frame in animation, by sliding the slider forward and backward. When the animation is running it will continue from the point where the slider is slid to.

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

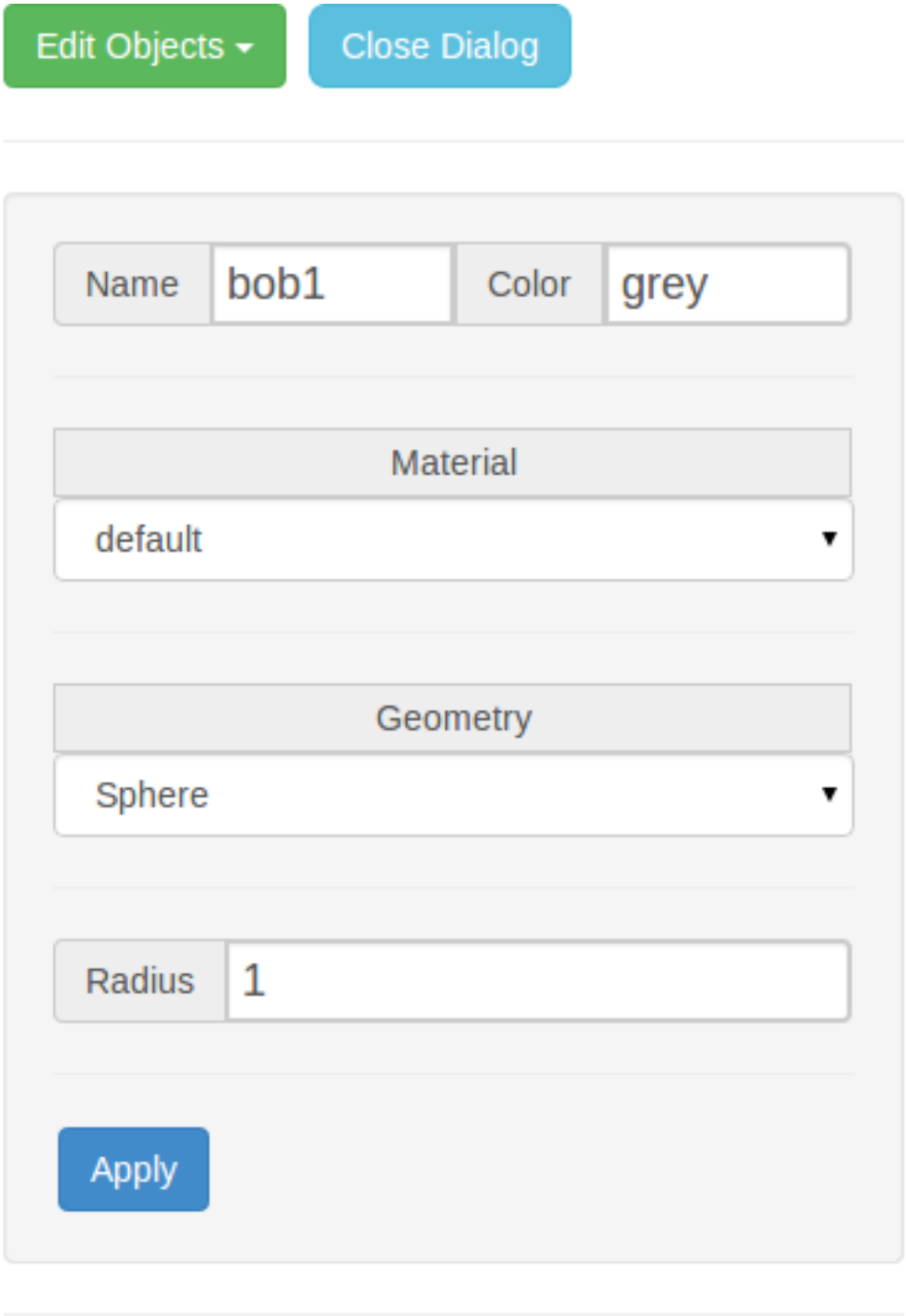
- Mouse wheel to zoom in, zoom out.
- Click and drag to rotate camera.

(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:



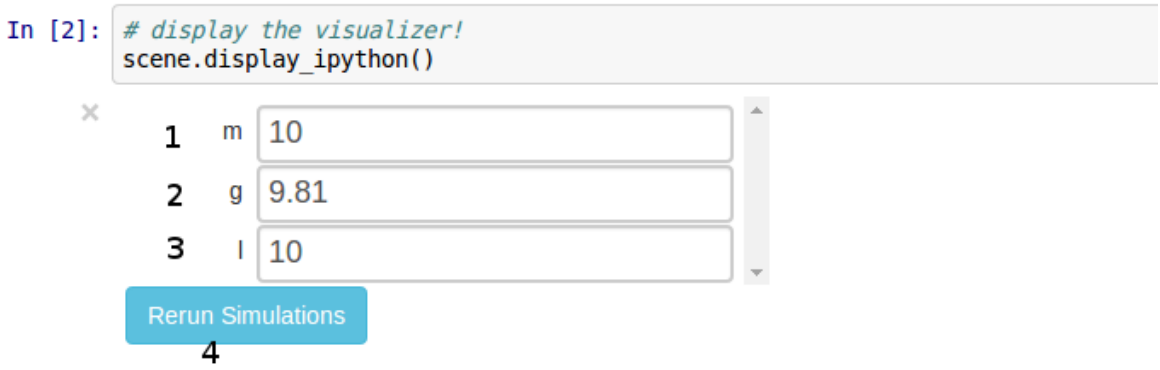
On clicking any object from the dropdown, a dialog box opens up, containing the existing info on that object. The info can be edited. After editing click the "Apply" button for the changes to be reflected in the canvas (4).



(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

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'
```

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.

material
Returns the material attribute of the shape.

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

The length of the cylinder.

radius : float or SymPy expression

The radius of the cylinder.

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
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'
>>> 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
'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

The length of the plane.

width : float or SymPy expression

The width of the plane.

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'
>>> 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
    >>> 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
    '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
    >>> 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.

name

Name of the VisualizationFrame.

origin

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

reference_frame

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

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.

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.

generate_visualization_json(*dynamic_variables*, *constant_variables*, *dynamic_values*, *constant_values*, *fps=30*, *outfile_prefix=None*)
Creates two JSON files in the current working directory. One contains the scene information and one contains the simulation data.

Parameters **dynamic_variables** : sequence of SymPy functions of time, len(m)

The variables representing the state of the system. They should be in the same order as `dynamic_values`.

constant_variables : sequence of SymPy symbols, len(p)

The variables representing the constants in the system. They should be in the same order as `constant_variables`.

dynamic_values : ndarray, shape(n, m)

The trajectories of the states.

constant_values : ndarray, shape(p,)

The numerical values of the constants.

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.

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.

Notes

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

name
Returns the name of the scene.

origin
Returns the origin point of the scene.

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.

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

p

C

D

E

G

I

L

M

N

O

P

R

S

T

V

W