Constrained Multibody Dynamics With Python: From Symbolic Equation Generation to Publication

Conference Paper · August 2013

DOI: 10.1115/DETC2013-13470

Jason Keith Moore
Delft University of Technology
44 PUBLICATIONS **986** CITATIONS
SEE PROFILE

Mont Hubbard
University of California, Davis
181 PUBLICATIONS **2,832** CITATIONS
SEE PROFILE

Proceedings of the ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference
IDETC/CIE 2013
August 4-7, 2013, Portland, Oregon, USA

DETC2013-13470

CONSTRAINED MULTIBODY DYNAMICS WITH PYTHON: FROM SYMBOLIC EQUATION GENERATION TO PUBLICATION

Gilbert Gede[*], **Dale L. Peterson, Angadh S. Nanjangud, Jason K. Moore, Mont Hubbard**
Sports Biomechanics Laboratory
Department of Mechanical and Aerospace Engineering
University of California
Davis, California 95616
Email: {ggede, dlpeterson, asnanjangud, jkmoor, mhubbard}@ucdavis.edu

ABSTRACT

Symbolic equations of motion (EOMs) for multibody systems are desirable for simulation, stability analyses, control system design, and parameter studies. Despite this, the majority of engineering software designed to analyze multibody systems are numeric in nature (or present a purely numeric user interface). To our knowledge, none of the existing software packages are 1) fully symbolic, 2) open source, and 3) implemented in a popular, general, purpose high level programming language. In response, we extended SymPy (an existing computer algebra system implemented in Python) with functionality for derivation of symbolic EOMs for constrained multibody systems with many degrees of freedom. We present the design and implementation of the software and cover the basic usage and workflow for solving and analyzing problems. The intended audience is the academic research community, graduate and advanced undergraduate students, and those in industry analyzing multibody systems. We demonstrate the software by deriving the EOMs of a N-link pendulum, show its capabilities for LaTeX output, and how it integrates with other Python scientific libraries - allowing for numerical simulation, publication quality plotting, animation, and online notebooks designed for sharing results. This software fills a unique role in dynamics and is attractive to academics and industry because of its BSD open source license which permits open source or commercial use of the code.

[*]Address all correspondence to this author

INTRODUCTION

There are many dynamic systems which can be better or more effectively studied when their EOMs are accessible in a symbolic form. For equations that may be visually inspected (i.e., of reasonable length), symbolics are generally preferable because the interrelations of the variables and constants can give clear understanding to the nature of the problem without the need for numerical simulation. Many classic problems fit this category, such as the mass-spring-damper, double pendulum, rolling disc, rattleback, and tippy-top. The benefits of symbolic equations of motion are not limited to these basic problems though. Larger, more complicated multibody systems can also be studied more effectively when the equations of motion are available symbolically. Advanced simplification routines can sometimes reduce the length of the equations such that they are human readable and the intermediate derivation steps are often short enough that symbolic checks can be used to validate the correctness. Furthermore, the symbolic form of the EOMs often evaluate much faster than their numerical counterparts, which is a significant advantage for real time computations. Problems in biomechanics, spacecraft dynamics, and single-track vehicles have all been successfully studied using symbolic EOMs.

Having the symbolic equations of motion available permits numerical simulation, but also allows for a more mathematical study of the system in question. System behavior can be studied parametrically by examining coefficients in the differential equations. This includes symbolic expressions for equilibria points and symbolic conditions for the stability of these points. The symbolic form also allows for more complicated tasks, such as analyzing how infinitesimal changes in system parameters (masses, lengths, inertias) affect the dynamics, studying lumped parameter discretization sizing, and analyzing how coordinate choices affect problem complexity or configuration singularities. It also becomes possible to share the equations of motion in a "written" form to other individuals for collaboration, validation, or comparison reasons. This allows for the EOMs to be used with other software packages, for multi-domain simulation, hardware-in-the-loop testing, or for use in optimal control/optimization problems.

Before adequate computing technology was available, the equations of motion for multibody dynamics problems were formed by hand. There are many methodologies to obtain the correct equations of motion (Newton-Euler, Lagrange, Kane, Hamilton, etc). But all methods are tedious and error-prone when derived by hand, which limits the size and complexity of systems which can be studied. It only takes a handful of unique orientations between a small set of rigid bodies within the system to reach this point of complexity. The introduction of computer algebra systems (CAS) has reduced the difficulty involved in forming the equations of motion, but it has not completely eliminated these problems. However, the details of the symbolic algebra, differentiation, and vector calculus can be handled by a reliable CAS, eliminating the errors associated with those operations, allowing the user to think more about the implications of the dynamic equations.

The software presented herein addresses some of the limitations of hand derivations and allows for the symbolic study of complex multibody dynamics problems. There already exist software packages which similarly meet these limited criteria (e.g. Autolev/MotionGenesis, AutoSim/VehicleSim). But when developing our software, we also included these unique requirements:

1. The software should be open source with a liberal license, encourage collaborative development, ensure continued project development, not be limited by or rely on any individual or organization, and allow easy integration and use by other projects.
2. The software should be written in a popular high level programming language that balances efficient execution speed with efficient programmer development time, and has a wide selection of scientific libraries (or can conveniently interface with libraries written in other languages).
3. The software should be built on top of an existing full-featured symbolic CAS that is also open source.
4. The software should easily export the equations in formats that are publication friendly (i.e., $\LaTeX$) or are compatible with other popular computing platforms and languages (i.e., Modelica, C/C++/Fortan, MATLAB).

To meet these criteria, we selected Python as the programming language to implement our software. Python is interactive, high level, easy to learn, widely available, widely used, cross platform, open source, and has a large scientific user base.

Our software is distributed as a sub-package of SymPy [1], which is a full featured CAS written in Python. SymPy is part of the SciPy Stack [2] specification and is included with all scientific Python distributions including Enthought, Sage, Anaconda, and Python(x,y). SymPy is one of the more actively developed Python packages with a large number of maintainers, ensuring a long future. The SymPy development model allows new functionality to be easily added and allows for other users to view our code, suggest additional features, and improve upon and add to what we have already done, as well as ensure that the our code is constantly tested against any changes or updates to the base symbolic functionality offered by SymPy.

In this paper, we discuss two main topics: 1) The interface to the EOM generation sub-package sympy.physics.mechanics, and 2) the workflow for studying multibody dynamic systems (from derivation to simulation and visualization), which we call Python Dynamics (PyDy). We will explore these two topics through an explanation of the software design and by demonstrating a test problem which displays software functionality and usage, how our software is incorporated into a workflow for analyzing dynamic systems, and the results of these processes. We will then discuss a number of other features, internal constructions within our software, and verification with benchmark examples.

DEMONSTRATION PROBLEM

We now demonstrate the value of PyDy through the derivation of the N-pendulum system shown in Figure 1. The system is defined by N massless links of length l_i with particles of mass m_i fixed at one end. We selected this problem because it illustrates the power and utility of having EOM generation code available within a full-featured programming language. The angular velocity of each and every link must be found, as well as the velocity of each particle. The velocity of i-th particle is

$$\bar{v}_i = \bar{v}_{i-1} + \bar{\omega}_i \times \bar{r}^i \qquad (i = 1, \ldots, N)$$

This nesting of velocities is best addressed with a loop, something that a computer is well suited for (loop unrolling is difficult by hand when there are many steps and N is large). There are thirty-seven lines of source code required to derive the dynamic equations of motion for the N pendulum, where N is the number of links, is shown in Figure 2.

The script first imports necessary functions and classes, then declares a number of symbolic variables that represent generalized coordinates, generalized speeds, constant parameters, reference frames, points and particles. A few empty lists are created

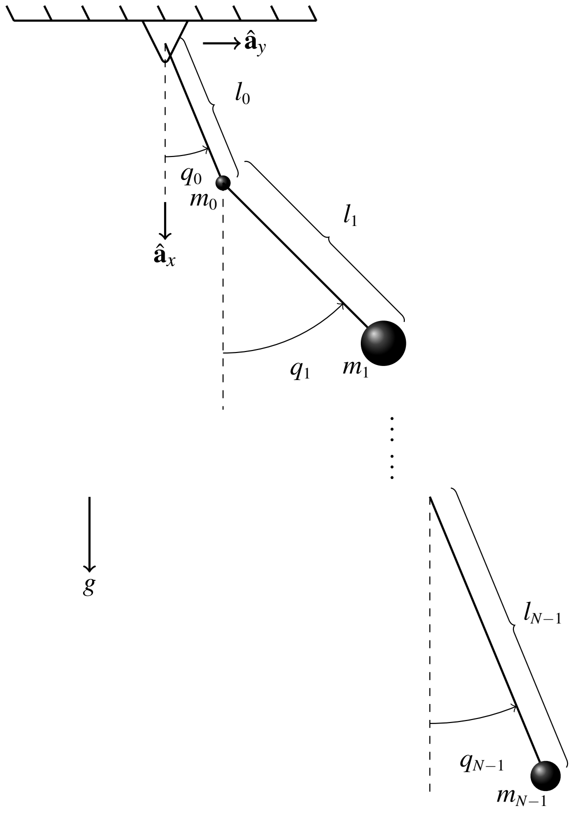

FIGURE 1. N-pendulum system, a sequence of massive links connected by revolute joints subjected to a gravitational field.

before the for loop is entered; these lists are filled by the loop. When the final iteration of the for loop is complete, the kinematics and all active forces of the problem have been completely specified. The last two lines of the script take this specification of kinematics and dynamics and generate Kane's dynamical equations of motion ($F_r + F_r^* = 0$) in symbolic form. At this point, these equations may be manipulated as fully symbolic variables for a variety of purposes. For example, the command `mlatex(fr)` generates LaTeX code for the generalized active force F_r, which renders as:

$$\begin{bmatrix} gl_0m_0\sin(q_0)+gl_0m_1\sin(q_0)+gl_0m_2\sin(q_0)+gl_0m_3\sin(q_0) \\ gl_1m_1\sin(q_1)+gl_1m_2\sin(q_1)+gl_1m_3\sin(q_1) \\ gl_2m_2\sin(q_2)+gl_2m_3\sin(q_2) \\ gl_3m_3\sin(q_3) \end{bmatrix} \tag{1}$$

with no further modification from the user.

There are other options besides LaTeX output which are useful. Within a console, commands to "pretty print" symbolic expressions can be performed with `mpprint`, which generates the output formatted just like the LaTeX equation above, but within the terminal.

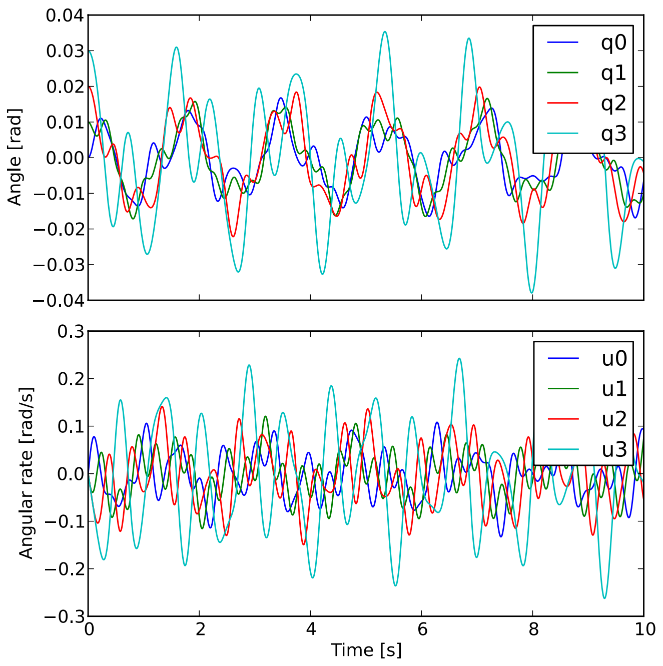

FIGURE 4. Time history of generalized coordinates and speeds for a 4 link pendulum.

Another part of studying dynamic systems is simulation and visualization of the results. SymPy can only analytically solve simple ODEs, so the equations of motion generated for more complex systems need to be passed to other numerical integrators. Currently, sympy.physics.mechanics can make use of existing SymPy translation functions, but more advanced options to generate compiled code are being developed and guided by user demands. The SymPy translation function `lambdify` can convert symbolic expressions to a NumPy compatible function. The code in Figure 3 (which is executed after the previously written code) shows this method.

NumPy is an integral part of the larger Scientific Python ecosystem, focusing primarily on numerical arrays and matrices and operations on these arrays and matrices. SciPy is another part of this ecosystem that provides quick and simple Python wrappers to a large library of legacy scientific code. The third necessary component of this ecosystem is matplotlib, a Python plotting library for visualization of the large datasets generated by NumPy and SciPy code. Using these three Python packages, we can numerically integrate ODEs and plot the results, which are shown in Figure 4.

Using other Python packages, such as MayaVi, 3D animations can be created. Use of human-interface-devices (with a sufficiently fast computer) allows for real-time interaction between a user and a visualized simulation.

```
from sympy import symbols                                  # Import the symbols function
from sympy.physics.mechanics import *                      # Import mechanics classes

n = 4                                                      # Number of links in N-pendulum

q = dynamicsymbols('q:' + str(n))                          # Generalized coordinates
u = dynamicsymbols('u:' + str(n))                          # Generalized speeds
m = symbols('m:' + str(n))                                 # Mass of each link
l = symbols('l:' + str(n))                                 # Length of each link
g, t = symbols('g t')                                      # gravity and time symbols

A = ReferenceFrame('A')                                    # Inertial reference frame
frames = []                                                # List to hold n link frames
P = Point('P')                                             # Hinge point of top link
P.set_vel(A, 0)                                            # Set velocity of P in A to be 0

particles = []                                             # List to hold N particles
forces = []                                                # List to hold N applied forces
kin_odes = []                                              # List to hold kinematic ODE's

for i in range(n):
    Ai = A.orientnew('A' + str(i), 'Axis', [q[i], A.z]) # Create a new frame
    Ai.set_ang_vel(A, u[i] * A.z)                          # Set angular velocity
    frames.append(Ai)                                      # Add it to Frames list

    Pi = P.locatenew('P' + str(i), l[i] * Ai.x)            # Create a new point Pi
    Pi.v2pt_theory(P, A, Ai)                               # Set velocity of Pi
    Pai = Particle('Pa' + str(i), Pi, m[i])                # Create a new particle
    particles.append(Pai)                                  # Add Pai to Particles list

    forces.append((Pi, m[i] * g * A.x))                    # Set force applied at i-th Point
    P = Pi                                                 # P is now the lowest Point

    kin_odes.append(q[i].diff(t) - u[i])                   # Kinematic ODE:  dq_i/dt-u_i=0

kane = KanesMethod(A, q_ind=q, u_ind=u, kd_eqs=kin_odes)# Generate EoM's:
fr, frstar = kane.kanes_equations(forces, particles)    # fr + frstar = 0
```

FIGURE 2. User Python script to derive equations of motion for N-pendulum

SOFTWARE VALIDATION

Anytime new software is developed to generate equations of motion, the validity and accuracy of the software comes into question. We have addressed these concerns in three ways to ensure that our code does generate correct equations of motion for arbitrarily complex systems. Firstly, the code is open source and well documented. This allows anyone to review the code and check for bugs. Linus's Law "given enough eyeballs, all bugs are shallow" [3] applies, if true. Secondly, the code functionality is thoroughly checked with unit testing; each piece of independent functionality in the code has a test (known input/output) in place that guarantees correct functioning of each unit. This ensures that not only the current code works as expected, but also that future versions are automatically checked against the same expected behavior. Thirdly, there are built in tests for well benchmarked problems in mutlibody dynamics. There are many problems, both simple and complex, that have well known symbolic solutions. We have chosen several benchmark problems that include sequential rotations, configuration and motion constraints, and examination of noncontributing forces. These problems are

```
from pylab import *
from sympy import Dummy, lambdify
from scipy.integrate import odeint

parameters = [g]                                               # Parameter Definitions
parameter_vals = [9.81]                                        # First we define gravity
for i in range(n):
    parameters += [l[i], m[i]]                                 # Then each mass
    parameter_vals += [1. / n, 0.01 / n]                       # and length

dummy_symbols = [Dummy() for i in q + u]                       # Necessary to translate
dummy_dict = dict(zip(q + u, dummy_symbols))                   # out of functions of time
kds = kane.kindiffdict()                                       # Need to eliminate qdots
mm_sym = kane.mass_matrix_full.subs(kds).subs(dummy_dict)      # Substituting away qdots
fo_sym = kane.forcing_full.subs(kds).subs(dummy_dict)          # and in dummy symbols
mm = lambdify(dummy_symbols + parameters, mm_sym)              # The actual call that gets
fo = lambdify(dummy_symbols + parameters, fo_sym)              # us to a NumPy function

def rhs(y, t, args):                                           # Creating the rhs function
    vals = hstack((y, args))                                   # States and parameters
    sol = linalg.solve(mm_sym(*vals), fo_sym(*vals))           # Solving for the udots
    return array(sol).T[0]

y0 = hstack((arange(n) * 0.01, arange(n) * 0))                 # Initial conditions, q & u
t = linspace(0, 10, 1000)                                      # Time vector
y = odeint(rhs, y0, t, args=(parameter_vals,))                 # Actual integration
```

FIGURE 3. User input Python script to perform numerical integration

built into the test suite for the package.

Built in tests which validate the equations of motion that are generated include: varying degrees of freedom spring-mass-damper systems, various pendulums, multiple rolling disc examples testing auxiliary and dependent speeds, and an inverted-pendulum cart. These are all compared to results found by hand (checked multiple times). The other parts of sympy.physics.mechanics include hundreds of tests on other pieces of the overall functionality.

There are also some external (non-automated) tests which are more advanced. Currently, the most complicated test validates the formulation of the equations of motion for a bicycle (a system with configuration and velocity constraints), symbolically linearizes it, and compares it (successfully) to benchmark values presented in [4].

USAGE

This section deals with obtaining the software demonstrated in this paper and the available resources for learning to use it. All of the software demonstrated in this paper can be readily obtained for free and installed on virtually any platform. Also, all of the packages are liberally licensed with a BSD or compatible license and either stable releases or development branches can be downloaded.

To obtain and use the core `mechanics` package for symbolic equation of motion generation, one must simply download and install Python version 2.5+ and SymPy version 0.7.2+[1]. Download and installation instructions for each can be found on the softwares' respective web sites, www.python.org and www.sympy.org.

To run a full PyDy example from equation of motion generation to simulation and visualization, Python and at least the Scientific Python stack (SciPy) [2] must be obtained. SciPy provides more comprehensive installation instructions than space permits

[1]One of the major requirements for code added to SymPy is that every function and object has to be thoroughly tested and documented. SymPy has over 88% of its code tested by the automated unit-tests. This ensures that no update causes regressions or breaks current functionality. Generally, the SymPy development branch is more stable and contains fewer bugs than the numbered releases.

here, but we make use of these packages in the SciPy stack in the previous demo problem:

SymPy: http://sympy.org/
NumPy: http://www.numpy.org/
SciPy: http://www.scipy.org/
matplotlib: http://matplotlib.org/

Download and installation instructions for various platforms are available on their respective web sites. But it is worth noting that these packages are part of most of the widely used scientific Python distributions. Downloading a distribution binary is generally the easiest method of installing *all* of the needed software for a user unfamiliar with Python. Popular scientific distributions that provide the needed software are listed below:

Enthought: http://www.enthought.com/
Sage: http://www.sagemath.org/
Anaconda: http://continuum.io/
Python(x,y): http://www.pythonxy.com

The first step in doing multibody dynamics with Python is to get familiar using `mechanics`[2]. The first stop is the SymPy documentation which is available online [5]. The documentation provides detailed instructions on installation, has introductory tutorials, lists common mistakes, details development procedures and the internal architecture, and contains detailed documentation for all of the subpackages and modules. In particular, the `mechanics` documentation contains over 60 pages that aim to provide an overview of using the package and a brief introduction to the fundamentals dynamics. Several example classic dynamics problems are included in the documentation and there are even more in the `mechanics.tests` package. Working through the material in the `mechanics` documentation will provide the basics of generating the symbolic equations of motion of multibody systems.

Additionally, a wiki for PyDy is maintained at www.pydy.org. This is a user-editable guide to solving dynamics problems in Python, and contains numerous introductory examples that demonstrate the process from system definition to simulation and visualization, examples of interfacing `mechanics` with various compiled languages, and more advanced use cases to demonstrate capabilities. As the PyDy workflow is integrated into more university courses, the goal is to allow professors, teaching assistants, and students to utilize, refine, and expand the wiki.

Finally, each of the software packages listed above maintains email lists and IRC channels for "live" help and the communities supporting the software are quite amicable to beginners. The PyDy Google Group is a good place to start if you have questions [6].

[2]Some may prefer to learn Python and the SciPy Stack first, but the authors do not think this is such a necessity. It boils down to preference and learning style.

SOFTWARE DESIGN

The software design for `mechanics` is influenced by Python's object oriented nature, the underlying SymPy data types, Kane's method for generating equations of motion [7] for multibody systems, and the proprietary (now defunct) software, Autolev [8], which also implemented Kane's method in a symbolic fashion.

Kane's method powers many of the dynamic system software packages available [9, 10] due to its detailed bookkeeping design and ease of mapping to programming languages. Kane's method has influenced the design of `mechanics` a great deal, but the code is structured so that any method for deriving the equations can be used. This is possible because we separated the kinematic code from the equations of motion generation code. We have recently included Lagrange's method to demonstrate that unique ability.

SymPy's symbolic manipulation library provides the core functionality for the mechanics package. The majority of objects that are created when using SymPy are expression objects, `Expr`. Other symbolic objects in SymPy, such as symbols, adds and muls, or exponentials are inherited from the `Expr` class. The `mechanics` package is dependent on these objects, but not via inheritance. `mechanics` is made up of a number of classes and functions spread out over several modules which are explained in the following list:

essential.py contains the basic building blocks for working with vector calculus and dynamic systems and includes the following classes and functions:

- **ReferenceFrame** is a class that represent a rotational reference frame in a dynamic system. `ReferenceFrame` has three orthonormal basis unit vectors and manages information about the frame's orientation, angular velocity, and angular acceleration relative to other reference frames through a direction cosine matrix and velocity and acceleration vectors. It is closely interlinked to the `Vector` class, i.e. `Vectors` have associated `ReferenceFrames` and `ReferenceFramess` have associated `Vectors`.
- **Vector** is a class that represents a generic three dimensional vector built from components in multiple frames of reference. Vectors support all of the common operations as one would expect such as addition, subtraction, multiplication, division, dot products, cross products, outer products, re-expression into different reference frames, and frame dependent differentiation.
- **Dyadic** is a class that represents a generic Dyadic. Dyadics are used in this software as a basis independent method of defining an inertia tensor. Dyadics support all of the common operations, similar to the Vector class.
- **dynamicsymbols** is a function that is used to facilitate the generation of time dependent quantities and their

derivatives. These variables are created as SymPy `Functions` of time, where the default time is the symbols `t`.

point.py contains the `Point` class. This class manages and tracks the location, velocity, and acceleration of a point in space relative to another point. It also provides some convenience methods to set velocities and accelerations via the one point and two point theories [7].

functions.py contains an assortment of convenience functions. For example there is a function for generating inertia dyadics from tensor notation, a function that outputs kinematical differential equations for various body or spaced fixed coordinates and speeds, functions to generate linear and angular momenta or kinetic and potential energies for a general system of particles and bodies, among others.

particle.py contains the `Particle` class which functions as a container class for a point and an associated mass. It also has methods for computing momentum and energy.

rigidbody.py contains the `RigidBody` is analogous to the `Particle` class but for rigid bodies and contains mass, mass center, inertia, and a reference frame associated with the rigid body. It also has methods for momentum and energy.

kane.py contains the `KanesMethod` class which automates the generation of a system's non-linear equations of motion using Kane's method [7] and linear equations of motion using the method presented in [11]. A `KanesMethod` object is initialized with an inertial reference frame, set of particles and/or rigid bodies, a set of forces and torques acting on the system, the desired kinematical differential equations, the desired independent and dependent generalized speeds, any configuration constraints, any velocity constraints, and any auxiliary speeds needed to compute non-contributing forces.

lagrange.py contains the `LagrangesMethod` class which automates the generation of a system's non-linear equations of motion via the methods of Lagrangian mechanics [12]. To derive the equations of motion of a constrained system using Lagrange's equations, one requires the configuration and velocity constraint equations, if any, before proceeding to derive a complete set of dynamical equations that describe said system. The `LagrangesMethod` object is initialized with a Lagrangian and the desired generalized coordinates and optionally forces acting on the system, velocity constraints, and an inertial reference frame. If velocity constraints are supplied, Lagrange multipliers are generated to account for them and non-conservative forces are appropriately handled if they affect the system.

ESSENTIAL

As previously mentioned, the submodule essential.py (which is the core of sympy.physics.mechanics) includes `ReferenceFrame`, `Vector`, `Dyadic`, and `dynamicsymbols`, as well as some output functions. These classes will be explained in some detail, to allow for a greater understanding of how our software was written and how it functions.

Firstly, `dynamicsymbols` will be explained - it is merely a shortcut to producing quantities which are functions of time. It is included for two reasons: specifics relating to Python importing requirements, and because `dynamicsymbols` defines the symbol used to represent time (default "`t`"). All of the vector calculus methods assume that functions of time define rotations between `ReferenceFrame`s; if basic symbols or expressions which are not functions of time are supplied, they will be treated as constants when differentiating with respect to time.

The relationship between `ReferenceFrame` and `Vector` is very complex. Upon initialization, a `ReferenceFrame` has three vectors representing the orthonormal basis vectors of that reference frame. A `Vector` is made up of a list of segments per associated frame. For each unique `ReferenceFrame` involved in the definition of a vector, one element in this list exists. Each element in the list contains the associated `ReferenceFrame` and a 3×1 SymPy matrix representing the measure numbers for each frame. The basis vectors, which are created upon `ReferenceFrame` initialization, have a list of length 1 and are made up of the `ReferenceFrame` they are created in and a matrix that defines one of the standard basis vectors (e.g. $[1,0,0]$).

As objects, both these basis vectors and other general vectors are instances of the same `Vector` class. More complicated `Vector`s just consist of longer lists and matrices. More frames can be introduced by adding vectors together or performing more advanced vector and vector calculus operations.

The final two parts of `Vector`'s design are in the users interaction with these objects, i.e. creating and displaying them. Direct initialization of a `Vector` object by a user should never take place; more complicated vectors will be formed out of basis vectors from the `ReferenceFrame`s and operations between `Vector`s. When displaying a `Vector` object to the user, the complex implementation details of each instance are masked. Instead, the user only sees something along the following lines:

```
3*A.x + 5*A.y + sin(theta)*C.z
```

Each measure number and accompanying basis vector are printed out in a way that the output can be copied and reentered to form the `Vector` again. This leaves our `Vector` class as a symbolic object (although, not a subclass of the main SymPy `Expr` class) which users will interact with on a symbolic/mathematical level. The complex initialization should never have to be done by hand, thus making use of this class is easy for people not intimately familiar with Python and SymPy.

In order to allow interactions between vectors defined in different reference frames, information is stored which relates each reference frame to others. The user defines how one frame is oriented to another, using a simple rotation around an arbitrary axis, standard body- or space-fixed angles, quaternions, or Euler parameters. Once this is defined, each of the involved `ReferenceFrame`s sets its internal dictionary, where a pointer to the other frame and the generated direction cosine matrix are stored together (transposing happens for one of the frames). When the orientation matrix between two frames is requested, if it has not already been defined, the code will search through all possible branches to find the other frame, and then multiply all of the orientation matrices in order to generate the direction cosine matrix between the two frames. After doing so, it will be saved so future searching does not have to happen again.

Angular velocities between frames are set in a similar way. When the user orients one frame relative to another, the angular velocity is generated by taking the time derivative of the direction cosine matrix. This is usually not desirable though, so the user is free to overwrite the angular velocity vector between frames. Again, setting this in one frame will set an equal and opposite vector in the other frame, to ensure consistency. Also, the same tree search occurs for angular velocities, except it involves adding angular velocity vectors instead of multiplying direction cosine matrices.

One potential pitfall is the possibility of a user closing a loop of `ReferenceFrame`s. In doing so, it is possible for the user to create internally inconsistent rotations or angular velocities; in other terms, if they don't use the correct value when closing the loop, the code does not detect this and incorrect results might be generated. Users are instructed to avoid doing this, and we feel it is an unlikely scenario, as it requires extra, unneeded work on the user's part.

The `Point` class functions in a similar manner to `ReferenceFrame`, in that it has a tree which links the position of this point to other points and is searchable to find the position between two points which have not been directly assigned. However, the `Point` class does not extend this functionality to velocities; it only works for positions. This limitation is due to the complexities of identifying bound/unbound vectors within reference frames. It has been more reliable to ask the user to find these velocities themselves rather than attempting to do it automatically (and frequently incorrectly). In order to help with this task, the `Point` class has a number of methods using the one point and two point theorems [7] to make this step easier.

The `Dyadic` object has a similar construction approach as `Vector`, i.e. it is made up of a list of components. However, for a dyadic, there are three parts of each list entry: a measure number, the first basis vector, and the second basis vector. The lists which compose a `Dyadic` object look like:

```
[(J, A.x, A.x), (I, A.x, B.y)]
```

which in turn is printed as:

```
J*(A.x|A.x) + I*(A.x|B.y)
```

which again follows the SymPy convention that the displayed version of this object can be copied and pasted in order to remake the object. The vector math and calculus associated with `Dyadic` is similar to that in `Vector`. The first or second basis vector is operated upon, depending on whether pre or post dot and cross products are used.

For a user, once all kinematics and system parameters have been defined, the next step is generating the equations of motion. There are two included methods for generating EOM: Kane's and Lagrange's. Both of these objects require the system's definition to be passed in using the container classes `Particle` and `RigidBody`. These objects (as described above) store the minimum amount of information for a body that is needed to create the equations of motion.

KanesMethod Class

Upon initialization, it takes in a number of properties which describe the system: the inertial reference frame; independent and constrained generalized coordinates; independent, auxiliary, and constrained (dependent) generalized speeds; and configuration, velocity, and (optionally) acceleration constraint equations. At this point, the relationships between the independent and constrained speeds are found as in Kane's Method [7]; this is shown in equation (7), where $f_v(q,u,t)$ is the list of constraint equations and u is the list of independent and dependent speeds.

$$\begin{aligned} \mathbf{0} &= \mathbf{f_v}(q,u,t) & (2)\\ &= \nabla_u \mathbf{f_v}(q,u,t)\mathbf{u} & (3)\\ &= \mathbf{Bu} & (4)\\ &= \mathbf{B}_{ind}\mathbf{u}_{ind} + \mathbf{B}_{dep}\mathbf{u}_{dep} & (5)\\ \implies \mathbf{u}_{dep} &= -\mathbf{B}_{dep}^{-1}\mathbf{B}_{ind}\mathbf{u}_{ind} & (6)\\ &= \mathbf{A}\mathbf{u}_{ind} & (7) \end{aligned}$$

The configuration constraints are used directly when forming the equations of motion, but they do need to be supplied if the system is going to be linearized [11].

From here, the formulation of the term $\mathbf{F}_r$ follows what Kane presented, with the advantage that all of the vector addition and multiplication is carried out symbolically. If necessary, the unconstrained generalized active forces can be transformed into the constrained generalized active forces, where n is the total number of generalized speeds, p is the number of independent speeds, leaving m as the number of constrained speeds; this is shown in equation (8).

$$\tilde{\mathbf{F}}_{ind} = \mathbf{F}_{ind} + \mathbf{A}^T\mathbf{F}_{dep} \quad (8)$$

The formulation of the generalized inertia forces is more complicated. First, we defined $\mathbf{F}_r^*$ differently, as in equation (9).

$$\mathbf{F}_r^* = \mathbf{M}_m \dot{\mathbf{u}} + \mathbf{M}_n \tag{9}$$

$\mathbf{M}_m$ is the mass matrix of the system and $\mathbf{M}_n$ is all the terms in $\mathbf{F}_r^*$ which are not part of the mass matrix. The construction of these terms will be shown for a particle, but the procedure for rigid bodies is almost the same, albeit for the rotational components. If we start by defining the velocity of a particle P in the inertial frame A as:

$$^A\bar{v}^P = \sum_{i=1}^{n} {}^A\bar{v}_r^P \tag{10}$$

the inertial force R^* for that particle can be written as:

$$\bar{R}^* = -\frac{^A d}{dt}(m^A\bar{v}^P) \tag{11}$$

$$= -m\frac{^A d}{dt}{}^A\bar{v}^P - \dot{m}^A\bar{v}^P \tag{12}$$

$$= -\dot{m}^A\bar{v}^P - m\frac{^A d}{dt}\left(\sum_{i=1}^{n} {}^A\bar{v}_i^P u_i\right) \tag{13}$$

$$= -\dot{m}^A\bar{v}^P - m\sum_{i=1}^{n}\left({}^A\bar{v}_i^P\frac{d}{dt}u_i + u_i\frac{^A d}{dt}{}^A\bar{v}_i^P\right) \tag{14}$$

$$\tag{15}$$

We can then write:

$$\mathbf{F}^*{}_r = {}^A\bar{v}_r^P\bar{R}^* \tag{16}$$

$$\mathbf{F}^*{}_r = -\left[{}^A\bar{v}_r^P\dot{m}^A\bar{v}^P + {}^A\bar{v}_r^P m\sum_{i=1}^{n}\left(\dot{u}_i^A\bar{v}_i^P + u_i\frac{^A d}{dt}{}^A\bar{v}_i^P\right)\right] \tag{17}$$

From this equation, we can see that only the mass and partial velocity terms multiply $\dot{u}$ terms, or that the mass matrix coefficients are made up of the product of mass and the dot product between two partial velocities. In fact, the location of the element in the mass matrix corresponds to the index of each partial velocity (the order is not important as long as it is consistent). If we wanted to define this another way, we could write a vertical vector of the partial velocities of particle P, $[{}^A\bar{v}_r^P]$, and then the mass matrix would be:

$$\mathbf{M}_m = -m\left[{}^A\bar{v}_r^P\right]\left[{}^A\bar{v}_r^P\right]^T \tag{18}$$

In either case, the non-mass matrix terms (which make up M_n) are the collection of terms that do not involve $\dot{u}$'s. As in Kane's method, the rotational and translational inertia forces and torques are multiplied by the angular and translational partial velocities, for each body/particle, summed across all bodies and particles. To transform this for systems with nonholonomic constraints, we again use equation (8), substituting $\mathbf{F}^*$ for $\mathbf{F}$.

Using auxiliary speeds to bring noncontributing forces into evidence is important to fully utilize Kane's method. Upon initialization of a `KanesMethod` object the user needs to supply auxiliary speeds. Assuming the auxiliary speeds were used correctly when forming velocities, the user does not have to worry about these speeds again, and they will not show up in any output equations. When there are velocity constraints and auxiliary speeds, the relationship between auxiliary speeds and constrained speeds needs to be found and used in forming the equations. To accomplish this, a second, internal `KanesMethod` object is created which has the same constraints and constrained speeds, but auxiliary speeds are supplied as both independent and auxiliary speeds, leaving the rest of the math the same.

More plainly, for any instance of `KanesMethod`, auxiliary speeds are removed from (set to 0) the accelerations and velocities that form $\bar{R}, \bar{T}, \bar{R}^*$, and $\bar{T}^*$. The only difference between the user's and the internal instance is which generalized speeds are used to form the partial velocities: independent or auxiliary. The relationship between auxiliary speeds and constrained speeds is found the same way as the relationship between the independent speeds and constrained speeds, i.e. the matrix $\mathbf{A}$ is found to relate auxiliary and constrained speeds. The transformation from unconstrained auxiliary equations to the constrained auxiliary equations ($\mathbf{F}+\mathbf{F}^* = 0$ to $\tilde{\mathbf{F}}+\tilde{\mathbf{F}}^* = 0$ is performed exactly the same way, just using the matrix $\mathbf{A}$ that relates the auxiliary and constrained speeds. The entire process is transparent to the user.

As the constrained generalized speeds are not substituted out of the equations generated, we will have fewer equations than independent and dependent speeds. If we have n total generalized speeds and p independent speeds, we are left with $m = n - p$ constrained speeds. This means that we will have p equations containing time derivatives of n speeds, or not enough equations to solve for the $\dot{u}$'s. We instead use the m velocity constraint equations to generate the needed equations. When the velocity constraint equations are differentiated with respect to time, we get m equations (the additional number we needed) containing up to n $\dot{u}$'s, giving the required information to solve for the time derivatives of the generalized speeds.

We will only briefly discuss linearization here, as a more detailed description of the linearization procedure used within `KanesMethod` can be found elsewhere [11]. The only complications that arise are when the user has a constrained system, in which case relationships between dependent and independent coordinates needs to be considered, as well as the relationship between dependent speeds and independent speeds, independent coordinates, and dependent coordinates. The important message for the user is configuration constraints are not considered when forming the equations of motion, but if they exist and are not supplied, the linear equations generated will be incorrect.

LangrangesMethod Class

In sympy.physics.mechanics, we assume that three basic sets of equations are necessary to completely describe a system and obtain the symbolic equations using `LagrangesMethod`:

$$\mathbf{0} = \mathbf{m_c}(q,t)\dot{q} + \mathbf{f_c}(q,t) \tag{19}$$

$$\mathbf{0} = \mathbf{m_{dc}}(\dot{q},q,t)\ddot{q} + \mathbf{f_{dc}}(\dot{q},q,t) \tag{20}$$

$$\mathbf{0} = \mathbf{m_d}(\dot{q},q,t)\ddot{q} + \Lambda_c(q,t)\lambda + \mathbf{f_d}(\dot{q},q,t) \tag{21}$$

where q represents a vector of generalized coordinates, $\dot{q}$ and $\ddot{q}$ are vectors of the first and second time derivatives of the generalized coordinates, λ is a vector of the Lagrange multipliers. $\mathbf{m_c}$, $\mathbf{m_{dc}}$, $\mathbf{m_d}$ and Λ_c are coefficient matrices. Equation (19) shows the constraint equations, (20) shows the first derivative of the constraint equations, and (21) shows the dynamical equations using Lagrange's equations. The `LagrangesMethod` class rearranges the equations of motion into the following form:

$$\mathbf{M}(q,t)x = \mathbf{f}(q,\dot{q},t) \tag{22}$$

where $\mathbf{M}$ is the 'mass matrix', and x is a vector of the states, namely the $\ddot{q}$'s and λ's, if any.

The `LagrangesMethod` class is initialized by suppliying properties that describe a system: the Lagrangian, a list of the generalized coordinates, holonomic and non-holonomic constraint equations (if any), the forces and/or moments acting on the system, and the inertial reference frame. The `LagrangesMethod` class requires that holonomic constraint equation be supplied after taking its first time derivative. The forces supplied upon initialization are primarily to account for the non-conservative forces acting on the system as the conservative forces are accounted for in the computation of the Lagrangian. Instead of the system's Lagrangian, the user may supply the system's kinetic energy, and account for any conservative and/or non-conservative forces in the force list.

Usage of the `LagrangesMethod` class does not allow for bringing non-contributing forces into evidence, or linearization functionality. Otherwise, it provides mass matrix/forcing function outputs as in the `KanesMethod` class.

CONCLUSIONS

We have presented a set of tools, PyDy, for forming the equations of motion of a constrained dynamic system in an automated fashion. The core of PyDy, the mechanics package within SymPy, is written in a high-level programming language on top of a fully function computer algebra system. We have demonstrated the use of our software, as well as how it fits into the larger scientific Python ecosystem. We also detailed how our software has been validated, why design decisions were made, and how its internal architecture functions. All of the source code, including extra code which generates a two dimensional animation of the N-link pendulum, are available online [13].

ACKNOWLEDGEMENTS

Three Google Summer of Code grants (2009, 2011, 2012) supported most of the software development. This material is also partially based upon work supported by the National Science Foundation under Grant No. 0928339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Team, S. D., 2012. *SymPy: Python library for symbolic mathematics*.

[2] SciPy stack. http://scipy.github.com/stackspec.html.

[3] Raymond, E., 1999. *The Cathedral and the Bazaar*.

[4] Meijaard, J. P., Papadopoulos, J. M., Ruina, A., and Schwab, A. L., 2007. "Linearized dynamics equations for the balance and steer of a bicycle: A benchmark and review". *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences,* ***463***(2084), August, pp. 1955–1982.

[5] Sympy documentation. https://docs.sympy.org.

[6] PyDy mailing list. https://groups.google.com/groups/pydy.

[7] Kane, T. R., and Levinson, D. A., 1985. *Dynamics: Theory and Applications*. McGraw Hill, New York, NY.

[8] Kane, T. R., and Levinson, D. A., 2000. *Dynamics Online: Theory and Implementation with AUTOLEV*. Online Dynamics, Inc., Sunnyvale, CA.

[9] Sayers, M. W., 1990. "Symbolic computer methods to automatically formulate vehicle simulation codes". PhD thesis, The University of Michigan.

[10] Engineering, E., 2013. OMD: Opensource multibody dynamics.

[11] Dale L. Peterson, Gilbert Gede, M. H., submitted Oct. 2012. "Linearization procedure for constrained multibody systems". *Multibody System Dynamics*.

[12] Crandall, S. H., Karnopp, D. C., Edward F. Kurtz, J., and Pridmore-Brown, D. C., 1968. *Dynamics of mechanical and electromechanical systems*. Krieger Publishing Company, Malabar, Florida.

[13] Source code for this paper. https://github.com/gilbertgede/idetc-2013-paper.