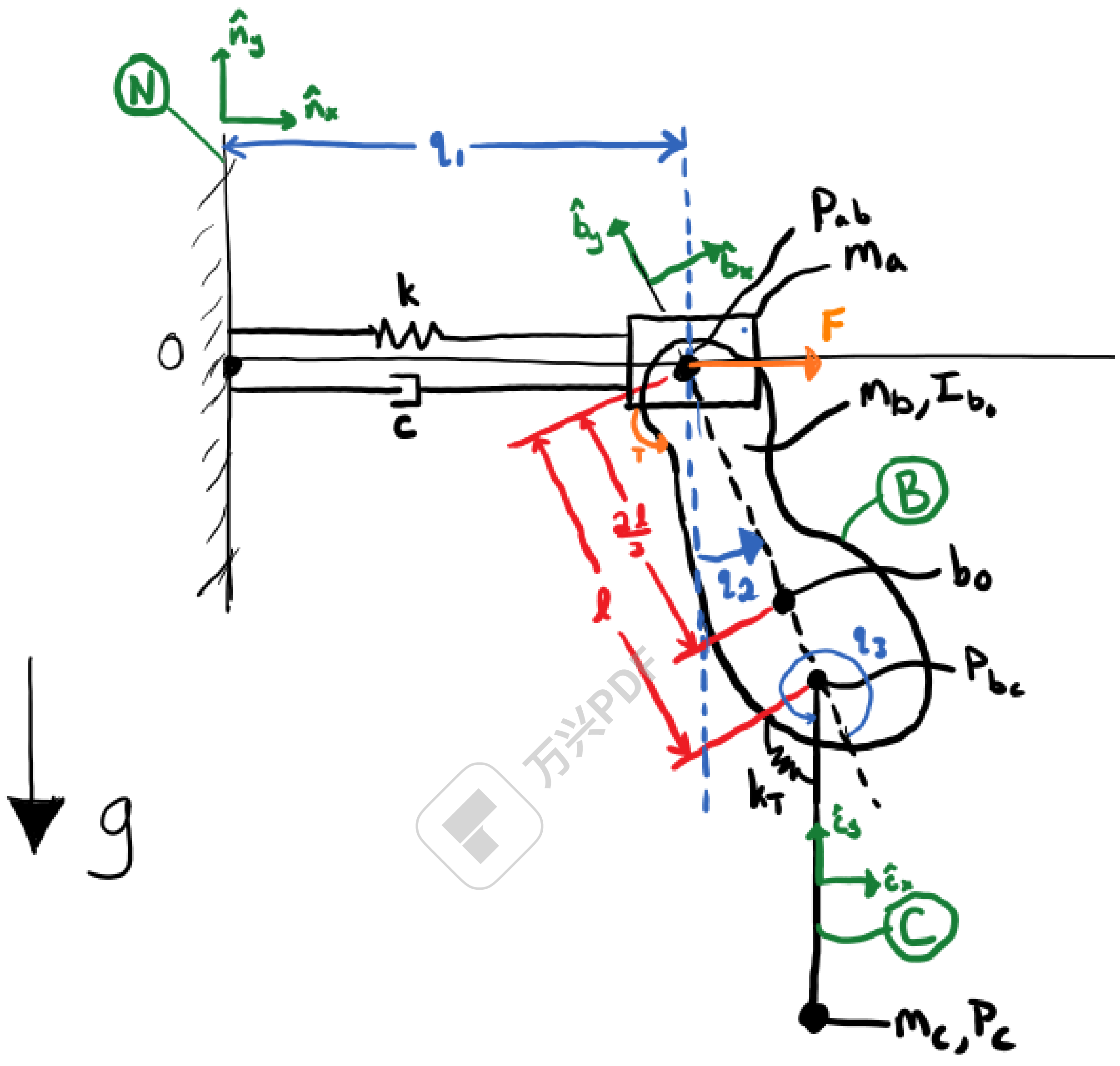


The system will be modeled using `JointsMethod`. First we need to create the `dynamicsymbols` needed to describe the system as shown in the above diagram. In this case, the generalized coordinates q_1 represent lateral distance of block from wall, q_2 represents angle of the compound pendulum from vertical, q_3 represents angle of the simple pendulum from the compound pendulum. The generalized speeds u_1 represents lateral speed of block, u_2 represents lateral speed of compound pendulum and u_3 represents angular speed of C relative to B.

We also create some `symbols` to represent the length and mass of the pendulum, as well as gravity and others.

```
>>> from sympy import zeros, symbols
>>> from sympy.physics.mechanics import Body, PinJoint, PrismaticJoint,␣
↪JointsMethod, inertia
>>> from sympy.physics.mechanics import dynamicsymbols
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1, q2, q3, u1, u2, u3')
```

(continues on next page)

(continued from previous page)

```
>>> l, k, c, g, kT = symbols('l, k, c, g, kT')
>>> ma, mb, mc, IBzz= symbols('ma, mb, mc, IBzz')
```

Next, we create the bodies and connect them using joints to establish the kinematics.

```
>>> wall = Body('N')
>>> block = Body('A', mass=ma)
>>> IB = inertia(block.frame, 0, 0, IBzz)
>>> compound_pend = Body('B', mass=mb, central_inertia=IB)
>>> simple_pend = Body('C', mass=mc)

>>> bodies = (wall, block, compound_pend, simple_pend)

>>> slider = PrismaticJoint('J1', wall, block, coordinates=q1, speeds=u1)
>>> rev1 = PinJoint('J2', block, compound_pend, coordinates=q2, speeds=u2,
...                    child_axis=compound_pend.z, child_joint_pos=l*2/
↪3*compound_pend.y,
...                    parent_axis=block.z)
>>> rev2 = PinJoint('J3', compound_pend, simple_pend, coordinates=q3,␣
↪speeds=u3,
...                    child_axis=simple_pend.z, parent_joint_pos=-l/3*compound_
↪pend.y,
...                    parent_axis=compound_pend.z, child_joint_pos=l*simple_
↪pend.y)

>>> joints = (slider, rev1, rev2)
```

Now we can apply loads (forces and torques) to the bodies, gravity acts on all bodies, a linear spring and damper act on block and wall, a rotational linear spring acts on C relative to B specified torque T acts on compound_pend and block, specified force F acts on block.

```
>>> F, T = dynamicsymbols('F, T')
>>> block.apply_force(F*block.x)
>>> block.apply_force(-k*q1*block.x, reaction_body=wall)
>>> block.apply_force(-c*u1*block.x, reaction_body=wall)
>>> compound_pend.apply_torque(T*compound_pend.z, reaction_body=block)
>>> simple_pend.apply_torque(-kT*q3*simple_pend.z, reaction_body=compound_
↪pend)
>>> block.apply_force(-wall.y*block.mass*g)
>>> compound_pend.apply_force(-wall.y*compound_pend.mass*g)
>>> simple_pend.apply_force(-wall.y*simple_pend.mass*g)
```

With the problem setup, the equations of motion can be generated using the JointsMethod class with KanesMethod in backend.

```
>>> method = JointsMethod(wall, slider, rev1, rev2)
>>> method.form_eoms()
Matrix([
[                                -c*u1(t) - k*q1(t) +␣
↪2*l*mb*u2(t)**2*sin(q2(t))/3 - l*mc*(-sin(q2(t))*sin(q3(t)) +␣
↪cos(q2(t))*cos(q3(t)))*Derivative(u3(t), t) - l*mc*(-sin(q2(t))*cos(q3(t)) -
↪ sin(q3(t))*cos(q2(t)))*(u2(t) + u3(t))**2 + l*mc*u2(t)**2*sin(q2(t)) -␣
```

(continues on next page)

(continued from previous page)

```
↪(2*l*mb*cos(q2(t))/3 + mc*(l*(-sin(q2(t))*sin(q3(t)) +␣
↪cos(q2(t))*cos(q3(t))) + l*cos(q2(t))))*Derivative(u2(t), t) - (ma + mb +␣
↪mc)*Derivative(u1(t), t) + F(t)],
[-2*g*l*mb*sin(q2(t))/3 - g*l*mc*(sin(q2(t))*cos(q3(t)) +␣
↪sin(q3(t))*cos(q2(t))) - g*l*mc*sin(q2(t)) + l**2*mc*(u2(t) +␣
↪u3(t))**2*sin(q3(t)) - l**2*mc*u2(t)**2*sin(q3(t)) - mc*(l**2*cos(q3(t)) +␣
↪l**2)*Derivative(u3(t), t) - (2*l*mb*cos(q2(t))/3 + mc*(l*(-
↪sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t))) +␣
↪l*cos(q2(t))))*Derivative(u1(t), t) - (IBzz + 4*l**2*mb/9 +␣
↪mc*(2*l**2*cos(q3(t)) + 2*l**2))*Derivative(u2(t), t) + T(t)],
[                                                                              ␣
↪                                                                              ␣
↪                -g*l*mc*(sin(q2(t))*cos(q3(t)) + sin(q3(t))*cos(q2(t))) -␣
↪kT*q3(t) - l**2*mc*u2(t)**2*sin(q3(t)) - l**2*mc*Derivative(u3(t), t) -␣
↪l*mc*(-sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t)))*Derivative(u1(t), t) -
↪ mc*(l**2*cos(q3(t)) + l**2)*Derivative(u2(t), t)]])

>>> method.mass_matrix_full
Matrix([
[1, 0, 0,                                                                      ␣
↪                      0,                                                      ␣
↪                                        0,                                    ␣
↪                   0],
[0, 1, 0,                                                                      ␣
↪                      0,                                                      ␣
↪                                        0,                                    ␣
↪                   0],
[0, 0, 1,                                                                      ␣
↪                      0,                                                      ␣
↪                                        0,                                    ␣
↪                   0],
[0, 0, 0,                                                                      ␣
↪             ma + mb + mc, 2*l*mb*cos(q2(t))/3 + mc*(l*(-
↪sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t))) + l*cos(q2(t))), l*mc*(-
↪sin(q2(t))*sin(q3(t)) + cos(q2(t))*cos(q3(t)))],
[0, 0, 0, 2*l*mb*cos(q2(t))/3 + mc*(l*(-sin(q2(t))*sin(q3(t)) +␣
↪cos(q2(t))*cos(q3(t))) + l*cos(q2(t))),                                       ␣
↪    IBzz + 4*l**2*mb/9 + mc*(2*l**2*cos(q3(t)) + 2*l**2),                     ␣
↪        mc*(l**2*cos(q3(t)) + l**2)],
[0, 0, 0,                                    l*mc*(-sin(q2(t))*sin(q3(t))␣
↪+ cos(q2(t))*cos(q3(t))),                                                     ␣
↪                mc*(l**2*cos(q3(t)) + l**2),                                  ␣
↪              l**2*mc]])

>>> method.forcing_full
Matrix([
[                                                                              ␣
↪                                                                              ␣
↪                   u1(t)],
[                                                                              ␣
↪                                                                              ␣
↪                   u2(t)],
```

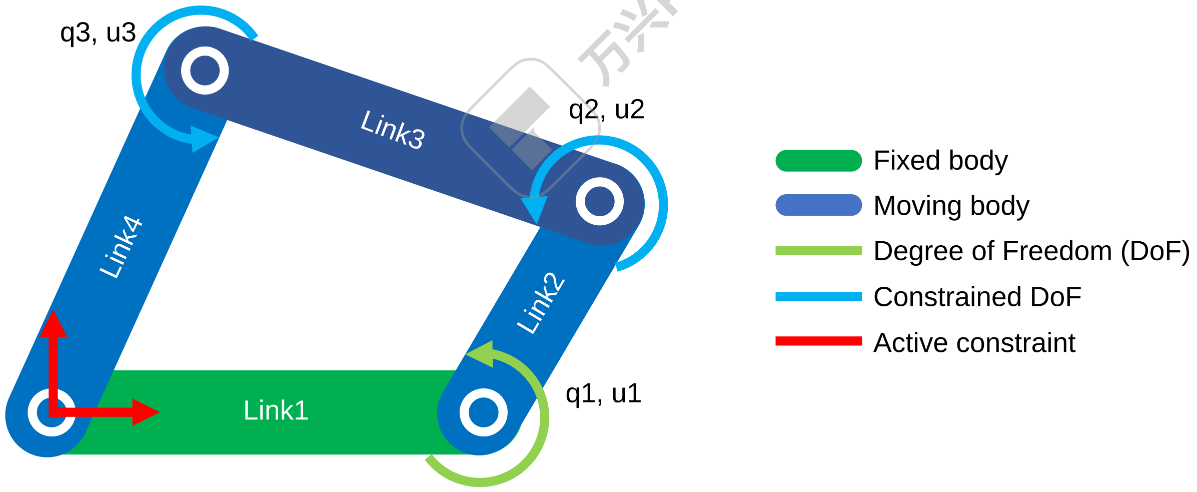
(continues on next page)

(continued from previous page)

```
[                                                                         ␣
↪                                                                         ␣
↪                   u3(t)],
[                  -c*u1(t) - k*q1(t) + 2*l*mb*u2(t)**2*sin(q2(t))/3 - l*mc*(-
↪sin(q2(t))*cos(q3(t)) - sin(q3(t))*cos(q2(t)))*(u2(t) + u3(t))**2 +␣
↪l*mc*u2(t)**2*sin(q2(t)) + F(t)],
[-2*g*l*mb*sin(q2(t))/3 - g*l*mc*(sin(q2(t))*cos(q3(t)) +␣
↪sin(q3(t))*cos(q2(t))) - g*l*mc*sin(q2(t)) + l**2*mc*(u2(t) +␣
↪u3(t))**2*sin(q3(t)) - l**2*mc*u2(t)**2*sin(q3(t)) + T(t)],
[                                                                         ␣
↪   -g*l*mc*(sin(q2(t))*cos(q3(t)) + sin(q3(t))*cos(q2(t))) - kT*q3(t) -␣
↪l**2*mc*u2(t)**2*sin(q3(t))]])
```

A four bar linkage

The four bar linkage is a common example used in mechanics, which can be formulated with only two holonomic constraints. This example will make use of joints functionality provided in *sympy.physics.mechanics* (page 1675). In summary we will use bodies and joints to define the open loop system. Next, we define the configuration constraints to close the loop. The JointsMethod will be used to do the "book-keeping" of the open-loop system. From this we will get the input used in combination with the constraints to manually setup the *KanesMethod* (page 1764) as the backend.



First we need to create the *dynamicsymbols()* (page 1666) needed to describe the system as shown in the above diagram. In this case, the generalized coordinates q_1, q_2 and q_3 represent the angles between the links. Likewise, the generalized speeds u_1, u_2 and u_3 represent the angular velocities between the links. We also create some *symbols()* (page 978) to represent the lengths and density of the links.

```
>>> from sympy import symbols, Matrix, solve, simplify
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1:4, u1:4')
>>> l1, l2, l3, l4, rho = symbols('l1:5, rho')
```

With all symbols defined, we can now define the bodies.

```
>>> N = ReferenceFrame('N')
>>> inertias = [inertia(N, 0, 0, rho * l ** 3 / 12) for l in (l1, l2, l3, l4)]
>>> link1 = Body('Link1', frame=N, mass=rho * l1, central_inertia=inertias[0])
>>> link2 = Body('Link2', mass=rho * l2, central_inertia=inertias[1])
>>> link3 = Body('Link3', mass=rho * l3, central_inertia=inertias[2])
>>> link4 = Body('Link4', mass=rho * l4, central_inertia=inertias[3])
```

Next, we also define the first three joints.

```
>>> joint1 = PinJoint('J1', link1, link2, coordinates=q1, speeds=u1,
...                   parent_axis=link1.z, parent_joint_pos=l1 / 2 * link1.x,
...                   child_axis=link2.z, child_joint_pos=-l2 / 2 * link2.x)
>>> joint2 = PinJoint('J2', link2, link3, coordinates=q2, speeds=u2,
...                   parent_axis=link2.z, parent_joint_pos=l2 / 2 * link2.x,
...                   child_axis=link3.z, child_joint_pos=-l3 / 2 * link3.x)
>>> joint3 = PinJoint('J3', link3, link4, coordinates=q3, speeds=u3,
...                   parent_axis=link3.z, parent_joint_pos=l3 / 2 * link3.x,
...                   child_axis=link4.z, child_joint_pos=-l4 / 2 * link4.x)
```

Now we can formulate the holonomic constraint that will close the kinematic loop.

```
>>> loop = link4.masscenter.pos_from(link1.masscenter) + l1 / 2 * link1.x +␣
↪l4 / 2 * link4.x
>>> fh = Matrix([loop.dot(link1.x), loop.dot(link1.y)])
```

In order to generate the equations of motions, we will use the JointsMethod as our fronted. Before setting up the *KanesMethod* (page 1764) as its backend we need to calculate the velocity constraints.

```
>>> method = JointsMethod(link1, joint1, joint2, joint3)
>>> t = dynamicsymbols._t
>>> qdots = solve(method.kdes, [q1.diff(t), q2.diff(t), q3.diff(t)])
>>> fhd = fh.diff(t).subs(qdots)
```

Now we can setup the *KanesMethod* (page 1764) as the backend and compute the equations of motion.

```
>>> method._method = KanesMethod(
...     method.frame, q_ind=[q1], u_ind=[u1], q_dependent=[q2, q3],
...     u_dependent=[u2, u3], kd_eqs=method.kdes,
...     configuration_constraints=fh, velocity_constraints=fhd,
...     forcelist=method.loads, bodies=method.bodies)
>>> simplify(method.method._form_eoms())
 Matrix([[l2*rho*(-2*l2**2*sin(q3)*u1' + 3*l2*l3*u1**2*sin(q2 + q3)*sin(q2) +␣
↪3*l2*l3*sin(q2)*cos(q2 + q3)*u1' - 3*l2*l3*sin(q3)*u1' +␣
↪3*l2*l4*u1**2*sin(q2 + q3)*sin(q2) + 3*l2*l4*sin(q2)*cos(q2 + q3)*u1' +␣
↪3*l3**2*u1**2*sin(q2)*sin(q3) + 6*l3**2*u1*u2*sin(q2)*sin(q3) +␣
↪3*l3**2*u2**2*sin(q2)*sin(q3) + 2*l3**2*sin(q2)*cos(q3)*u1' +␣
↪2*l3**2*sin(q2)*cos(q3)*u2' - l3**2*sin(q3)*cos(q2)*u1' -␣
↪l3**2*sin(q3)*cos(q2)*u2' + 3*l3*l4*u1**2*sin(q2)*sin(q3) +␣
↪6*l3*l4*u1*u2*sin(q2)*sin(q3) + 3*l3*l4*u2**2*sin(q2)*sin(q3) +␣
↪3*l3*l4*sin(q2)*cos(q3)*u1' + 3*l3*l4*sin(q2)*cos(q3)*u2' + l4**2*sin(q2)*u1
↪' + l4**2*sin(q2)*u2' + l4**2*sin(q2)*u3')/(6*sin(q3))]])
```

Potential Issues/Advanced Topics/Future Features in Physics/Mechanics

This document will describe some of the more advanced functionality that this module offers but which is not part of the "official" interface. Here, some of the features that will be implemented in the future will also be covered, along with unanswered questions about proper functionality. Also, common problems will be discussed, along with some solutions.

Common Issues

Here issues with numerically integrating code, choice of `dynamicsymbols` for coordinate and speed representation, printing, differentiating, and substitution will occur.

Numerically Integrating Code

See Future Features: Code Output

Differentiating

Differentiation of very large expressions can take some time in SymPy; it is possible for large expressions to take minutes for the derivative to be evaluated. This will most commonly come up in linearization.

Choice of Coordinates and Speeds

The Kane object is set up with the assumption that the generalized speeds are not the same symbol as the time derivatives of the generalized coordinates. This isn't to say that they can't be the same, just that they have to have a different symbol. If you did this:

```
>> KM.coords([q1, q2, q3])
>> KM.speeds([q1d, q2d, q3d])
```

Your code would not work. Currently, kinematic differential equations are required to be provided. It is at this point that we hope the user will discover they should not attempt the behavior shown in the code above.

This behavior might not be true for other methods of forming the equations of motion though.

Printing

The default printing options are to use sorting for `Vector` and `Dyad` measure numbers, and have unsorted output from the `mprint`, `mpprint`, and `mlatex` functions. If you are printing something large, please use one of those functions, as the sorting can increase printing time from seconds to minutes.

Substitution

There are two common issues with substitution in mechanics:

- When subbing in expressions for `dynamicsymbols`, sympy's normal `subs` will substitute in for derivatives of the dynamic symbol as well:

  ```
  >>> from sympy.physics.mechanics import dynamicsymbols
  >>> x = dynamicsymbols('x')
  >>> expr = x.diff() + x
  >>> sub_dict = {x: 1}
  >>> expr.subs(sub_dict)
  Derivative(1, t) + 1
  ```

  In this case, `x` was replaced with 1 inside the `Derivative` as well, which is undesired.

- Substitution into large expressions can be slow.

If your substitution is simple (direct replacement of expressions with other expressions, such as when evaluating at an operating point) it is recommended to use the provided `msubs` function, as it is significantly faster, and handles the derivative issue appropriately:

```
>>> from sympy.physics.mechanics import msubs
>>> msubs(expr, sub_dict)
Derivative(x(t), t) + 1
```

Linearization

Currently, the linearization methods don't support cases where there are non-coordinate, non-speed dynamic symbols outside of the "dynamic equations". It also does not support cases where time derivatives of these types of dynamic symbols show up. This means if you have kinematic differential equations which have a non-coordinate, non-speed dynamic symbol, it will not work. It also means if you have defined a system parameter (say a length or distance or mass) as a dynamic symbol, its time derivative is likely to show up in the dynamic equations, and this will prevent linearization.

Acceleration of Points

At a minimum, points need to have their velocities defined, as the acceleration can be calculated by taking the time derivative of the velocity in the same frame. If the 1 point or 2 point theorems were used to compute the velocity, the time derivative of the velocity expression will most likely be more complex than if you were to use the acceleration level 1 point and 2 point theorems. Using the acceleration level methods can result in shorted expressions at this point, which will result in shorter expressions later (such as when forming Kane's equations).

Advanced Interfaces

Advanced Functionality

Remember that the Kane object supports bodies which have time-varying masses and inertias, although this functionality isn't completely compatible with the linearization method.

Operators were discussed earlier as a potential way to do mathematical operations on Vector and Dyad objects. The majority of the code in this module is actually coded with them, as it can (subjectively) result in cleaner, shorter, more readable code. If using this interface in your code, remember to take care and use parentheses; the default order of operations in Python results in addition occurring before some of the vector products, so use parentheses liberally.

Future Features

This will cover the planned features to be added to this submodule.

Code Output

A function for generating code output for numerical integration is the highest priority feature to implement next. There are a number of considerations here.

Code output for C (using the GSL libraries), Fortran 90 (using LSODA), MATLAB, and SciPy is the goal. Things to be considered include: use of cse on large expressions for MATLAB and SciPy, which are interpretive. It is currently unclear whether compiled languages will benefit from common subexpression elimination, especially considering that it is a common part of compiler optimization, and there can be a significant time penalty when calling cse.

Care needs to be taken when constructing the strings for these expressions, as well as handling of input parameters, and other dynamic symbols. How to deal with output quantities when integrating also needs to be decided, with the potential for multiple options being considered.

References for Physics/Mechanics

Autolev Parser

Introduction

Autolev (now superseded by MotionGenesis) is a domain specific language used for symbolic multibody dynamics. The SymPy mechanics module now has enough power and functionality to be a fully featured symbolic dynamics module. This parser parses Autolev (version 4.1) code to SymPy code by making use of SymPy's math libraries and the mechanics module.

The parser has been built using the ANTLR framework and its main purpose is to help former users of Autolev to get familiarized with multibody dynamics in SymPy.

The sections below shall discuss details of the parser like usage, gotchas, issues and future improvements. For a detailed comparison of Autolev and SymPy Mechanics you might want to look at the *SymPy Mechanics for Autolev Users guide* (page 1734).

Usage

We first start with an Autolev code file.

Let us take this example (Comments % have been included to show the Autolev responses):

```
% double_pendulum.al
%-------------------
MOTIONVARIABLES' Q{2}', U{2}'
CONSTANTS L,M,G
NEWTONIAN N
FRAMES A,B
SIMPROT(N, A, 3, Q1)
% -> N_A = [COS(Q1), -SIN(Q1), 0; SIN(Q1), COS(Q1), 0; 0, 0, 1]
SIMPROT(N, B, 3, Q2)
% -> N_B = [COS(Q2), -SIN(Q2), 0; SIN(Q2), COS(Q2), 0; 0, 0, 1]
W_A_N>=U1*N3>
% -> W_A_N> = U1*N3>
W_B_N>=U2*N3>
% -> W_B_N> = U2*N3>
POINT O
PARTICLES P,R
P_O_P> = L*A1>
% -> P_O_P> = L*A1>
P_P_R> = L*B1>
% -> P_P_R> = L*B1>
V_O_N> = 0>
% -> V_O_N> = 0>
V2PTS(N, A, O, P)
% -> V_P_N> = L*U1*A2>
V2PTS(N, B, P, R)
% -> V_R_N> = L*U1*A2> + L*U2*B2>
MASS P=M, R=M
Q1' = U1
Q2' = U2
GRAVITY(G*N1>)
% -> FORCE_P> = G*M*N1>
% -> FORCE_R> = G*M*N1>
ZERO = FR() + FRSTAR()
% -> ZERO[1] = -L*M*(2*G*SIN(Q1)+L*(U2^2*SIN(Q1-Q2)+2*U1'+COS(Q1-Q2)*U2'))
% -> ZERO[2] = -L*M*(G*SIN(Q2)-L*(U1^2*SIN(Q1-Q2)-U2'-COS(Q1-Q2)*U1'))
KANE()
INPUT M=1,G=9.81,L=1
INPUT Q1=.1,Q2=.2,U1=0,U2=0
INPUT TFINAL=10, INTEGSTP=.01
CODE DYNAMICS() some_filename.c
```

The parser can be used as follows:

```
>>> from sympy.parsing.autolev import parse_autolev
>>> sympy_code = parse_autolev(open('double_pendulum.al'), include_
↪numeric=True)

# The include_pydy flag is False by default. Setting it to True will
```

(continues on next page)

(continued from previous page)

```
# enable PyDy simulation code to be outputted if applicable.

>>> print(sympy_code)
import sympy.physics.mechanics as me
import sympy as sm
import math as m
import numpy as np

q1, q2, u1, u2 = me.dynamicsymbols('q1 q2 u1 u2')
q1d, q2d, u1d, u2d = me.dynamicsymbols('q1 q2 u1 u2', 1)
l, m, g=sm.symbols('l m g', real=True)
frame_n=me.ReferenceFrame('n')
frame_a=me.ReferenceFrame('a')
frame_b=me.ReferenceFrame('b')
frame_a.orient(frame_n, 'Axis', [q1, frame_n.z])
# print(frame_n.dcm(frame_a))
frame_b.orient(frame_n, 'Axis', [q2, frame_n.z])
# print(frame_n.dcm(frame_b))
frame_a.set_ang_vel(frame_n, u1*frame_n.z)
# print(frame_a.ang_vel_in(frame_n))
frame_b.set_ang_vel(frame_n, u2*frame_n.z)
# print(frame_b.ang_vel_in(frame_n))
point_o=me.Point('o')
particle_p=me.Particle('p', me.Point('p_pt'), sm.Symbol('m'))
particle_r=me.Particle('r', me.Point('r_pt'), sm.Symbol('m'))
particle_p.point.set_pos(point_o, l*frame_a.x)
# print(particle_p.point.pos_from(point_o))
particle_r.point.set_pos(particle_p.point, l*frame_b.x)
# print(particle_p.point.pos_from(particle_r.point))
point_o.set_vel(frame_n, 0)
# print(point_o.vel(frame_n))
particle_p.point.v2pt_theory(point_o,frame_n,frame_a)
# print(particle_p.point.vel(frame_n))
particle_r.point.v2pt_theory(particle_p.point,frame_n,frame_b)
# print(particle_r.point.vel(frame_n))
particle_p.mass = m
particle_r.mass = m
force_p = particle_p.mass*(g*frame_n.x)
# print(force_p)
force_r = particle_r.mass*(g*frame_n.x)
# print(force_r)
kd_eqs = [q1d - u1, q2d - u2]
forceList = [(particle_p.point,particle_p.mass*(g*frame_n.x)), (particle_r.
↪point,particle_r.mass*(g*frame_n.x))]
kane = me.KanesMethod(frame_n, q_ind=[q1,q2], u_ind=[u1, u2], kd_eqs = kd_eqs)
fr, frstar = kane.kanes_equations([particle_p, particle_r], forceList)
zero = fr+frstar
# print(zero)
#---------PyDy code for integration----------
from pydy.system import System
sys = System(kane, constants = {l:1, m:1, g:9.81},
specifieds={},
```

(continues on next page)

(continued from previous page)

```
initial_conditions={q1:.1, q2:.2, u1:0, u2:0},
times = np.linspace(0.0, 10, 10/.01))

y=sys.integrate()
```

The commented code is not part of the output code. The print statements demonstrate how to get responses similar to the ones in the Autolev file. Note that we need to use SymPy functions like `.ang_vel_in()`, `.dcm()` etc in many cases unlike directly printing out the variables like `zero`. If you are completely new to SymPy mechanics, the *SymPy Mechanics for Autolev Users guide* (page 1734) guide should help. You might also have to use basic SymPy simplifications and manipulations like `trigsimp()`, `expand()`, `evalf()` etc for getting outputs similar to Autolev. Refer to the SymPy Tutorial to know more about these.

Gotchas

- Don't use variable names that conflict with Python's reserved words. This is one example where this is violated:

```
%Autolev Code
%------------
LAMBDA = EIG(M)
```

```
#SymPy Code
#----------
lambda = sm.Matrix([i.evalf() for i in (m).eigenvals().keys()])
```

- Make sure that the names of vectors and scalars are different. Autolev treats these differently but these will get overwritten in Python. The parser currently allows the names of bodies and scalars/vectors to coincide but doesn't do this between scalars and vectors. This should probably be changed in the future.

```
%Autolev Code
%------------
VARIABLES X,Y
FRAMES A
A> = X*A1> + Y*A2>
A = X+Y
```

```
#SymPy Code
#----------
x, y = me.dynamicsymbols('x y')
frame_a = me.ReferenceFrame('a')
a = x*frame_a.x + y*frame_a.y
a = x + y
# Note how frame_a is named differently so it doesn't cause a problem.
# On the other hand, 'a' gets rewritten from a scalar to a vector.
# This should be changed in the future.
```

- When dealing with Matrices returned by functions, one must check the order of the values as they may not be the same as in Autolev. This is especially the case for eigenvalues and eigenvectors.

```
%Autolev Code
%------------
EIG(M, E1, E2)
% -> [5; 14; 13]
E2ROW = ROWS(E2, 1)
EIGVEC> = VECTOR(A, E2ROW)
```

```
#SymPy Code
#-----------
e1 = sm.Matrix([i.evalf() for i in m.eigenvals().keys()])
# sm.Matrix([5;13;14]) different order
e2 = sm.Matrix([i[2][0].evalf() for i in m.eigenvects()]).reshape(m.
↪shape[0], m.shape[1])
e2row = e2.row(0)
# This result depends on the order of the vectors in the eigenvecs.
eigenvec = e2row[0]*a.x + e2row[1]*a.y + e2row[2]*a.y
```

- When using `EVALUATE`, use something like `90*UNITS(deg,rad)` for angle substitutions as radians are the default in SymPy. You could also add `np.deg2rad()` directly in the SymPy code.

  This need not be done for the output code (generated on parsing the `CODE` commands) as the parser takes care of this when deg units are given in the `INPUT` declarations.

  The `DEGREES` setting, on the other hand, works only in some cases like in `SIMPROT` where an angle is expected.

```
%Autolev Code
%------------
A> = Q1*A1> + Q2*A2>
B> = EVALUATE(A>, Q1:30*UNITS(DEG,RAD))
```

```
#SymPy Code
#-----------
a = q1*a.frame_a.x + q2*frame_a.y
b = a.subs({q1:30*0.0174533})
# b = a.subs({q1:np.deg2rad(30)}
```

- Most of the Autolev settings have not been parsed and have no effect on the parser. The only ones that work somewhat are `COMPLEX` and `DEGREES`. It is advised to look into alternatives to these in SymPy and Python.

- The `REPRESENT` command is not supported. Use the `MATRIX`, `VECTOR` or `DYADIC` commands instead. Autolev 4.1 suggests these over `REPRESENT` as well while still allowing it but the parser doesn't parse it.

- Do not use variables declarations of the type `WO{3}RD{2,4}`. The parser can only handle one variable name followed by one pair of curly braces and any number of `'` s. You would have to declare all the cases manually if you want to achieve something like `WO{3}RD{2,4}`.

- The parser can handle normal versions of most commands but it may not parse functions with Matrix arguments properly in most cases. Eg:

  `M=COEF([E1;E2],[U1,U2,U3])`

  This would compute the coefficients of `U1`, `U2` and `U3` in `E1` and `E2`. It is preferable to manually construct a Matrix using the regular versions of these commands.

```
%Autolev Code
%------------
% COEF([E1;E2],[U1,U2,U3])
M = [COEF(E1,U1),COEF(E1,U2),COEF(E1,U3) &
    ;COEF(E2,U1),COEF(E2,U2),COEF(E2,U3)]
```

- `MOTIONVARIABLE` declarations must be used for the generalized coordinates and speeds and all other variables must be declared in regular `VARIABLE` declarations. The parser requires this to distinguish between them to pass the correct parameters to the Kane's method object.

  It is also preferred to always declare the speeds corresponding to the coordinates and to pass in the kinematic differential equations. The parser is able to handle some cases where this isn't the case by introducing some dummy variables of its own but SymPy on its own does require them.

  Also note that older Autolev declarations like `VARIABLES U{3}'` are not supported either.

```
%Autolev Code
%------------
MOTIONVARIABLES' Q{2}', U{2}'
% ----- OTHER LINES ----
Q1' = U1
Q2' = U2
----- OTHER LINES ----
ZERO = FR() + FRSTAR()
```

```
#SymPy Code
#----------
q1, q2, u1, u2 = me.dynamicsymbols('q1 q2 u1 u2')
q1d, q2d, u1d, u2d = me.dynamicsymbols('q1 q2 u1 u2', 1)

# ------- other lines -------

kd_eqs = [q1d - u1, q2d - u2]
kane = me.KanesMethod(frame_n, q_ind=[q1,q2], u_ind=[u1, u2], kd_eqs =␣
↪kd_eqs)
fr, frstar = kane.kanes_equations([particle_p, particle_r], forceList)
zero = fr+frstar
```

- Need to change `me.dynamicsymbols._t` to `me.dynamicsymbols('t')` for all occurrences of it in the Kane's equations. For example have a look at line 10 of this spring damper example. This equation is used in forming the Kane's equations so we need to change `me.dynamicsymbols._t` to `me.dynamicsymbols('t')` in this case.

  The main reason that this needs to be done is because PyDy requires time dependent specifieds to be explicitly laid out while Autolev simply takes care of the stray time variables in the equations by itself.

  The problem is that PyDy's System class does not accept `dynamicsymbols._t` as a specified. Refer to issue #396. This change is not actually ideal so a better solution should be figured out in the future.

- The parser creates SymPy `symbols` and `dynamicsymbols` by parsing variable declarations in the Autolev Code.

  For intermediate expressions which are directly initialized the parser does not create SymPy symbols. It just assigns them to the expression.

  On the other hand, when a declared variable is assigned to an expression, the parser stores the expression against the variable in a dictionary so as to not reassign it to a completely different entity. This constraint is due to the inherent nature of Python and how it differs from a language like Autolev.

  Also, Autolev seems to be able to assume whether to use a variable or the rhs expression that variable has been assigned to in equations even without an explicit `RHS()` call in some cases. For the parser to work correctly however, it is better to use `RHS()` wherever a variable's rhs expression is meant to be used.

```
%Autolev Code
%------------
VARIABLES X, Y
E = X + Y
X = 2*Y

RHS_X = RHS(X)

I1 = X
I2 = Y
I3 = X + Y

INERTIA B,I1,I2,I3
% -> I_B_BO>> = I1*B1>*B1> + I2*B2>*B2> + I3*B3>*B3>
```

```
#SymPy Code
#----------
x,y = me.dynamicsymbols('x y')
e = x + y  # No symbol is made out of 'e'

# an entry like {x:2*y} is stored in an rhs dictionary

rhs_x = 2*y
```

(continues on next page)

(continued from previous page)

```
i1 = x  # again these are not made into SymPy symbols
i2 = y
i3 = x + y

body_b.inertia = (me.inertia(body_b_f, i1, i2, i3), b_cm)
# This prints as:
# x*b_f.x*b_f.x + y*b_f.y*b_f.y + (x+y)*b_f.z*b_f.z
# while Autolev's output has I1,I2 and I3 in it.
# Autolev however seems to know when to use the RHS of I1,I2 and I3
# based on the context.
```

- This is how the SOLVE command is parsed:

```
%Autolev Code
%------------
SOLVE(ZERO,X,Y)
A = RHS(X)*2 + RHS(Y)
```

```
#SymPy Code
#----------
print(sm.solve(zero,x,y))
# Behind the scenes the rhs of x
# is set to sm.solve(zero,x,y)[x].
a = sm.solve(zero,x,y)[x]*2 + sm.solve(zero,x,y)[y]
```

The indexing like [x] and [y] doesn't always work so you might want to look at the underlying dictionary that solve returns and index it correctly.

- Inertia declarations and Inertia functions work somewhat differently in the context of the parser. This might be hard to understand at first but this had to be done to bridge the gap due to the differences in SymPy and Autolev. Here are some points about them:

  1. Inertia declarations (INERTIA B,I1,I2,I3) set the inertias of rigid bodies.

  2. Inertia setters of the form I_C_D>> = expr however, set the inertias only when C is a body. If C is a particle then I_C_D>> = expr simply parses to i_c_d = expr and i_c_d acts like a regular variable.

  3. When it comes to inertia getters (I_C_D>> used in an expression or INERTIA commands), these MUST be used with the EXPRESS command to specify the frame as SymPy needs this information to compute the inertia dyadic.

```
%Autolev Code
%------------
INERTIA B,I1,I2,I3
I_B_BO>> = X*A1>*A1> + Y*A2>*A2>  % Parser will set the inertia of B
I_P_Q>> = X*A1>*A1> + Y^2*A2>*A2> % Parser just parses it as i_p_q = expr

E1 = 2*EXPRESS(I_B_O>>,A)
E2 =  I_P_Q>>
```

(continues on next page)

(continued from previous page)

```
E3 = EXPRESS(I_P_O>>,A)
E4 = EXPRESS(INERTIA(O),A)

% In E1 we are using the EXPRESS command with I_B_O>> which makes
% the parser and SymPy compute the inertia of Body B about point O.

% In E2 we are just using the dyadic object I_P_Q>> (as I_P_Q>> = expr
% doesn't act as a setter) defined above and not asking the parser
% or SymPy to compute anything.

% E3 asks the parser to compute the inertia of P about point O.

% E4 asks the parser to compute the inertias of all bodies wrt about O.
```

- In an inertia declaration of a body, if the inertia is being set about a point other than the center of mass, one needs to make sure that the position vector setter for that point and the center of mass appears before the inertia declaration as SymPy will throw an error otherwise.

```
%Autolev Code
%------------
P_SO_O> = X*A1>
INERTIA S_(O) I1,I2,I3
```

- Note that all Autolev commands have not been implemented. The parser now covers the important ones in their basic forms. If you are doubtful whether a command is included or not, please have a look at this file in the source code. Search for "<command>" to verify this. Looking at the code for the specific command will also give an idea about what form it is expected to work in.

Limitations and Issues

- A lot of the issues have already been discussed in the Gotchas section. Some of these are:
  - Vector names coinciding with scalar names are overwritten in Python.
  - Some convenient variable declarations aren't parsed.
  - Some convenient forms of functions to return matrices aren't parsed.
  - Settings aren't parsed.
  - symbols and rhs expressions work very differently in Python which might cause undesirable results.
  - Dictionary indexing for the parsed code of the `SOLVE` command is not proper in many cases.
  - Need to change `dynamicsymbols._t` to `dynamicsymbols('t')` for the PyDy simulation code to work properly.

Here are some other ones:

- Eigenvectors do not seem to work as expected. The values in Autolev and SymPy are not the same in many cases.
- Block matrices aren't parsed by the parser. It would actually be easier to make a change in SymPy to allow matrices to accept other matrices for arguments.
- The SymPy equivalent of the `TAYLOR` command `.series()` does not work with `dynamicsymbols()`.
- Only `DEPENDENT` constraints are currently parsed. Need to parse `AUXILIARY` constraints as well. This should be done soon as it isn't very difficult.
- None of the energy and momentum functions are parsed right now. It would be nice to get these working as well. Some changes should probably be made to SymPy. For instance, SymPy doesn't have a function equivalent to `NICHECK()`.
- The numerical integration parts work properly only in the case of the `KANE` command with no arguments. Things like `KANE(F1,F2)` do not currently work.
- Also, the PyDy numerical simulation code works only for cases where the matrix say `ZERO = FR() + FRSTAR()` is solved for. It doesn't work well when the matrix has some other equations plugged in as well. One hurdle faced in achieving this was that PyDy's System class automatically takes in the `forcing_full` and `mass_matrix_full` and solves them without giving the user the flexibility to specify the equations. It would be nice to add this functionality to the System class.

Future Improvements

1. Completing Dynamics Online

The parser has been built by referring to and parsing codes from the Autolev Tutorial and the book *Dynamics Online: Theory and Implementation Using Autolev*. Basically, the process involved going through each of these codes, validating the parser results and improving the rules if required to make sure the codes parsed well.

The parsed codes of these are available on GitLab here. The repo is private so access needs to be requested. As of now, most codes till Chapter 4 of *Dynamics Online* have been parsed.

Completing all the remaining codes of the book (namely, *2-10*, *2-11*, *rest of Ch4*, *Ch5* and *Ch6* (less important)) would make the parser more complete.

2. Fixing Issues

The second thing to do would be to go about fixing the problems described above in the *Gotchas* (page 1727) and *Limitations and Issues* (page 1732) sections in order of priority and ease. Many of these require changes in the parser code while some of these are better fixed by adding some functionality to SymPy.

3. Switching to an AST

The parser is currently built using a kind of Concrete Syntax Tree (CST) using the ANTLR framework. It would be ideal to switch from a CST to an Abstract Syntax Tree (AST). This way, the parser code will be independent of the ANTLR grammar which makes it a lot more flexible. It would also be easier to make changes to the grammar and the rules of the parser.

SymPy Mechanics for Autolev Users

Introduction

Autolev (now superseded by MotionGenesis) is a domain specific programming language which is used for symbolic multibody dynamics. The SymPy mechanics module now has enough power and functionality to be a fully featured symbolic dynamics module. The PyDy package extends the SymPy output to the numerical domain for simulation, analyses and visualization. Autolev and SymPy Mechanics have a lot in common but there are also many differences between them. This page shall expand upon their differences. It is meant to be a go-to reference for Autolev users who want to transition to SymPy Mechanics.

It would be nice to have a basic understanding of SymPy and SymPy Mechanics before going over this page. If you are completely new to Python, you can check out the official Python Tutorial. Check out the *SymPy Documentation* (page **??**), especially the tutorial to get a feel for SymPy. For an introduction to Multibody dynamics in Python, this lecture is very helpful.

You might also find the *Autolev Parser* (page 1724) which is a part of SymPy to be helpful.

Some Key Differences

Autolev	SymPy Mechanics
Autolev is a domain specific programming language designed to perform multibody dynamics. Since it is a language of its own, it has a very rigid language specification. It predefines, assumes and computes many things based on the input code. Its code is a lot cleaner and concise as a result of this.	SymPy is a library written in the general purpose language Python. Although Autolev's code is more compact, SymPy (by virtue of being an add on to Python) is more flexible. The users have more control over what they can do. For example, one can create a class in their code for let's say a type of rigibodies with common properties. The wide array of scientific Python libraries available is also a big plus.
Autolev generates Matlab, C, or Fortran code from a small set of symbolic mathematics.	SymPy generates numerical Python, C or Octave/Matlab code from a large set of symbolic mathematics created with SymPy. It also builds on the popular scientific Python stack such as NumPy, SciPy, IPython, matplotlib, Cython and Theano.
Autolev uses 1 (one) based indexing. The initial element of a sequence is found using a[1].	Python uses 0 (zero) based indexing. The initial element of a sequence is found using a[0].
Autolev is case insensitive.	SymPy code being Python code is case sensitive.
One can define their own commands in Autolev by making .R and .A files which can be used in their programs.	SymPy code is Python code, so one can define functions in their code. This is a lot more convenient.
Autolev is proprietary.	SymPy is open source.

Rough Autolev-SymPy Equivalents

The tables below give rough equivalents for some common Autolev expressions. **These are not exact equivalents**, but rather should be taken as hints to get you going in the right direction. For more detail read the built-in documentation on *SymPy vectors* (page 1592), *SymPy mechanics* (page 1675) and PyDy .

In the tables below, it is assumed that you have executed the following commands in Python:

```
import sympy.physics.mechanics as me
import sympy as sm
```

Mathematical Equivalents

Autolev	SymPy	Notes
`Constants A, B`	`a, b = sm.symbols('a b', real=True)`	Note that the names of the symbols can be different from the names of the variables they are assigned to. We can define `a, b = symbols('b a')` but its good practice to follow the convention.
`Constants C+`	`c = sm.symbols('c', real=True, nonnegative=True)`	Refer to SymPy *assumptions* (page 186) for more information.
`Constants D-`	`d = sm.symbols('d', real=True, nonpositive=True)`	
`Constants K{4}`	`k1, k2, k3, k4 = sm.symbols('k1 k2 k3 k4', real=True)`	
`Constants a{2:4}`	`a2, a3, a4 = sm.symbols('a2 a3 a4', real=True)`	
`Constants b{1:2, 1:2}`	`b11, b12, b21, b22 = sm.symbols('b11 b12 b21 b22', real=True)`	
`Specified Phi`	`phi = me.dynamicsymbols('phi ')`	
`Variables q, s`	`q, s = me.dynamicsymbols(q, s)`	
`Variables x''`	`x = me.dynamicsymbols('x' ) xd = me.dynamicsymbols('x' , 1) xd2 = me.dynamicsymbols('x' ,`	

Physical Equivalents

Autolev	SymPy	Notes

`Bodies A`
Declares A, its masscenter Ao, and orthonormal vectors A1>, A2> and A3> fixed in A.
	`m =sm.symbols('m')`
	`Ao = sm.symbols('Ao')`
	`Af = me.ReferenceFrame('Af' )`
	`I = me.outer(Af.x,Af.x)`
	`P = me.Point('P')`
	`A =me.RigidBody('A', Ao, Af, m, (I, P))`
	Af.x, Af.y and Af.z are equivalent to A1>, A2> and A3>.
		The 4th and 5th arguments are for the mass and inertia. These are specified after the declaration in Autolev.
		One can pass a dummy for the parameters and use setters `A.mass = \_` and `A.inertia = \_` to set them later.
		For more information refer to *mechanics/masses* . (page 1676)

`Frames A`
`V1> = X1*A1> + X2*A2>`
	`A = me.ReferenceFrame('A' )`
	`v1 = x1*A.x + x2*A.y`
		For more information refer to *physics/vectors.* (page 51)

`Newtonian N`	`N = me.ReferenceFrame('N' )`	SymPy doesn't specify that a frame is inertial during declaration. Many functions such as `set_ang_vel()` take the inertial reference frame as a parameter.

`Particles C`	`m = sm.symbols('m')`
	`Po = me.Point('Po')`
	`C = me.Particle('C', Po, m)`
		The 2nd and 3rd arguments are for the point and mass. In Autolev, these are specified after the declaration.
		One can pass a dummy and use setters (`A.point = \_` and `A.mass = \_`) to set them later.

`Points P, Q`	`P = me.Point('P')`
	`Q = me.Point('Q')`	

`Mass B=mB`	`mB = symbols('mB')`
	`B.mass = mB`	

`Inertia B, I1, I2, I3, I12, I23, I31`	`I = me.inertia(Bf, i1, i2, i3, i12, i23, i31)`
	`B.inertia = (I, P)` where B is a rigidbody, Bf is the related frame and P is the center of mass of B.
	Inertia dyadics can also be formed using vector outer products.
	`I = me.outer(N.x, N.x)`
		For more information refer to the *mechanics api.* (page 1743)

continues on next page

Table 1 – continued from previous page

Autolev	SymPy	Notes

`vec> = P_O_Q>/L`
`vec> = u1*N1> + u2*N2>`
`Cross(a>, b>)`
`Dot(a>, b>)`
`Mag(v>)`
`Unitvec(v>)`
`DYAD>> = 3*A1>*A1> + A2>*A2> + 2*A3>*A3>`
	`vec = (Qo.pos_from(O))/ L`
	`vec = u1*N.x + u2*N.y`
	`cross(a, b)`
	`dot(a, b)`
	`v.magnitude()`
	`v.normalize()`
	`dyad = 3*me.outer(a.x ,a.x) + me.outer(a.y, a.y) + 2*me.outer(a.z ,a.z)`
		For more information refer to *physics/vectors*. (page 1592)

`P_O_Q> = LA*A1>`
`P_P_Q> = LA*A1>`
	`Q.point = O. locatenew('Qo', LA*A.x)`
	where A is a reference frame.
	`Q.point = P.point. locatenew('Qo ', LA*A. x)`
		For more information refer to the *kinematics api*. (page 1652)
		All these vector and kinematic functions are to be used on `Point` objects and not `Particle` objects so `. point` must be used for particles.

`V_O_N> = u3*N.1> + u4*N.2>`
`Partials(V_O_N>, u3)`
	`O.set_vel(N, u1*N.x + u2*N.y)`
	`O.partial_velocity(N , u3)`
		The getter would be `O. vel(N)`.

`A_O_N> = 0>` Acceleration of point O in reference frame N.	`O.set_acc(N, 0)`	The getter would be `O. acc(N)`.
`W_B_N> = qB'*B3>` Angular velocity of body B in reference frame F.	`B.set_ang_vel(N, qBd*Bf.z)` where Bf is the frame associated with the body B.	The getter would be `B. ang_vel_in(N)`.
`ALF_B_N> =Dt(W_B_N>, N)` Angular acceleration of body B in reference frame N.	`B.set_ang_acc(N, diff(B.ang_vel_in(N) )`	The getter would be `B. ang_acc_in(N)`.

`Force_O> = F1*N1> + F2*N2>`
`Torque_A> = -c*qA'*A3>`
	In SymPy one should have a list which contains all the forces and torques.
	`fL.append((O, f1*N.x + f2*N.y))`
	where fL is the force list.
	`fl.append((A, -c*qAd*A. z))`	

`A_B = M` where M is a matrix and A, B are frames.
`D = A_B*2 + 1`
	`B.orient(A, 'DCM', M)`
	where M is a SymPy Matrix.
	`D = A.dcm(B)*2 + 1`	

`CM(B)`	`B.masscenter`	
`Mass(A,B,C)`	`A.mass + B.mass + C. mass`	

continues on next page

Table 1 – continued from previous page

Autolev	SymPy	Notes
`V1pt(A,B,P,Q)`	`Q.v1pt_theory(P, A, B)`	P and Q are assumed to be Point objects here. Remember to use `.point` for particles.
`V2pts(A,B,P,Q)`	`Q.v2pt_theory(P, A, B)`	
`A1pt(A,B,P,Q)`	`Q.a1pt_theory(P, A, B)`	
`A2pts(A,B,P,Q)`	`Q.a2pt_theory(P, A, B)`	
`Angvel(A,B)`	`B.ang_vel_in(A)`	
`Simprot(A, B, 1, qA)`	`B.orient(A, 'Axis', qA, A.x)`	
`Gravity(G*N1>)`	`fL.extend(gravity( g*N.x, P1, P2, ...))`	In SymPy we must use a forceList (here fL) which contains tuples of the form `(point, force_vector)`. This is passed to the `kanes_equations()` method of the KanesMethod object.
`CM(O,P1,R)`	`me.functions.center_of_mass(o, p1, r)`	
`Force(P/Q, v>)`	`fL.append((P, -1*v), (Q, v))`	
`Torque(A/B, v>)`	`fL.append((A, -1*v), (B, v))`	
`Kindiffs(A, B ...)`	`KM.kindiffdict()`	
`Momentum(option)`	`linear_momentum(N, B1, B2 ...)` reference frame followed by one or more bodies `angular_momentum(O, N, B1, B2 ...)` point, reference frame followed by one or more bodies	
`KE()`	`kinetic_energy(N, B1, B2 ...)` reference frame followed by one or more bodies	
`Constrain(...)`	`velocity_constraints = [...]` `u_dependent = [...]` `u_auxiliary = [...]` These lists are passed to the KanesMethod object.	For more details refer to *mechanics/kane* (page 1682) and the *kane api.* (page 1764)

continues on next page

Table 1 – continued from previous page

Autolev	SymPy	Notes
Fr() FrStar()	KM = KanesMethod(f, q_ind, u_ind, kd_eqs, q_dependent, configuration_constraints, u_dependent, velocity_constraints, acceleration_constraints, u_auxiliary) The KanesMethod object takes a reference frame followed by multiple lists as arguments. (fr, frstar) = KM.kanes_equations(fL, bL) where fL and bL are lists of forces and bodies respectively.	For more details refer to *mechanics/kane* (page 1682) and the *kane api.* (page 1764)

Numerical Evaluation and Visualization

Autolev's CODE Option() command allows one to generate Matlab, C, or Fortran code for numerical evaluation and visualization. Option can be Dynamics, ODE, Nonlinear or Algebraic.

Numerical evaluation for dynamics can be achieved using PyDy. One can pass in the KanesMethod object to the System class along with the values for the constants, specifieds, initial conditions and time steps. The equations of motion can then be integrated. The plotting is achieved using matlplotlib. Here is an example from the PyDy Documentation on how it is done:

```python
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))

y = sys.integrate()

import matplotlib.pyplot as plt
plt.plot(sys.times, y)
plt.legend((str(position), str(speed)))
plt.show()
```

For information on all the things PyDy can accomplish refer to the PyDy Documentation.

The tools in the PyDy workflow are :

- **SymPy: SymPy is a Python library for**
  symbolic computation. It provides computer algebra capabilities either as a stan-

dalone application, as a library to other applications, or live on the web as SymPy Live or SymPy Gamma.

- **NumPy: NumPy is a library for the**
  Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- **SciPy: SciPy is an open source**
  Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.
- **IPython: IPython is a command shell**
  for interactive computing in multiple programming languages, originally developed for the Python programming language, that offers introspection, rich media, shell syntax, tab completion, and history.
- **Aesara: Aesara is**
  a numerical computation library for Python. In Aesara, computations are expressed using a NumPy-esque syntax and compiled to run efficiently on either CPU or GPU architectures.
- **Cython: Cython is a superset of the**
  Python programming language, designed to give C-like performance with code that is mostly written in Python. Cython is a compiled language that generates CPython extension modules.
- **matplotlib: matplotlib is a**
  plotting library for the Python programming language and its numerical mathematics extension NumPy.

One will be able to write code equivalent to the Matlab, C or Fortran code generated by Autolev using these scientific computing tools. It is recommended to go over these modules to gain an understanding of scientific computing with Python.

Links

SymPy Introductory Tutorial (page 5)

SymPy Documentation (page **??**)

SymPy Physics Vector Documentation (page 1592)

SymPy Mechanics Documentation (page 1675)

PyDy Documentation

MultiBody Dynamics with Python

Mechanics API

Masses, Inertias & Particles, RigidBodys (Docstrings)

Particle

class sympy.physics.mechanics.particle.**Particle**(*name*, *point*, *mass*)

A particle.

Parameters

name : str

Name of particle

point : Point

A physics/mechanics Point which represents the position, velocity, and acceleration of this Particle

mass : sympifyable

A SymPy expression representing the Particle's mass

Explanation

Particles have a non-zero mass and lack spatial extension; they take up no space.

Values need to be supplied on initialization, but can be changed later.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import Symbol
>>> po = Point('po')
>>> m = Symbol('m')
>>> pa = Particle('pa', po, m)
>>> # Or you could change these later
>>> pa.mass = m
>>> pa.point = po
```

angular_momentum(*point*, *frame*)

Angular momentum of the particle about the point.

Parameters

point : Point

The point about which angular momentum of the particle is desired.

frame : ReferenceFrame

The frame in which angular momentum is desired.

Explanation

The angular momentum H, about some point O of a particle, P, is given by:

H = r x m * v

where r is the position vector from point O to the particle P, m is the mass of the particle, and v is the velocity of the particle in the inertial frame, N.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point,␣
↪ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, v, r = dynamicsymbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> A = O.locatenew('A', r * N.x)
>>> P = Particle('P', A, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.angular_momentum(O, N)
m*r*v*N.z
```

kinetic_energy(*frame*)

Kinetic energy of the particle.

Parameters

frame : ReferenceFrame

The Particle's velocity is typically defined with respect to an inertial frame but any relevant frame in which the velocity is known can be supplied.

Explanation

The kinetic energy, T, of a particle, P, is given by

'T = 1/2 m v^2'

where m is the mass of particle P, and v is the velocity of the particle in the supplied ReferenceFrame.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point,␣
↪ReferenceFrame
>>> from sympy import symbols
>>> m, v, r = symbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.kinetic_energy(N)
m*v**2/2
```

linear_momentum(*frame*)

Linear momentum of the particle.

Parameters

frame : ReferenceFrame

The frame in which linear momentum is desired.

Explanation

The linear momentum L, of a particle P, with respect to frame N is given by

L = m * v

where m is the mass of the particle, and v is the velocity of the particle in the frame N.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point,␣
↪ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, v = dynamicsymbols('m v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> A = Particle('A', P, m)
>>> P.set_vel(N, v * N.x)
>>> A.linear_momentum(N)
m*v*N.x
```

property mass

Mass of the particle.

parallel_axis(*point, frame*)

Returns an inertia dyadic of the particle with respect to another point and frame.

Parameters

point : sympy.physics.vector.Point

The point to express the inertia dyadic about.

frame : sympy.physics.vector.ReferenceFrame

The reference frame used to construct the dyadic.

Returns

inertia : sympy.physics.vector.Dyadic

The inertia dyadic of the particle expressed about the provided point and frame.

property point

Point of the particle.

property potential_energy

The potential energy of the Particle.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import symbols
>>> m, g, h = symbols('m g h')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.potential_energy = m * g * h
>>> P.potential_energy
g*h*m
```

RigidBody

class sympy.physics.mechanics.rigidbody.**RigidBody**(*name*, *masscenter*, *frame*, *mass*, *inertia*)

An idealized rigid body.

Explanation

This is essentially a container which holds the various components which describe a rigid body: a name, mass, center of mass, reference frame, and inertia.

All of these need to be supplied on creation, but can be changed afterwards.

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, RigidBody
>>> from sympy.physics.mechanics import outer
>>> m = Symbol('m')
>>> A = ReferenceFrame('A')
>>> P = Point('P')
```

(continues on next page)

(continued from previous page)

```
>>> I = outer (A.x, A.x)
>>> inertia_tuple = (I, P)
>>> B = RigidBody('B', P, A, m, inertia_tuple)
>>> # Or you could change them afterwards
>>> m2 = Symbol('m2')
>>> B.mass = m2
```

Attributes

name	(string) The body's name.
masscenter	(Point) The point which represents the center of mass of the rigid body.
frame	(ReferenceFrame) The ReferenceFrame which the rigid body is fixed in.
mass	(Sympifyable) The body's mass.
inertia	((Dyadic, Point)) The body's inertia about a point; stored in a tuple as shown above.

angular_momentum(*point, frame*)

Returns the angular momentum of the rigid body about a point in the given frame.

Parameters

point : Point

The point about which angular momentum is desired.

frame : ReferenceFrame

The frame in which angular momentum is desired.

Explanation

The angular momentum H of a rigid body B about some point O in a frame N is given by:

H = I . w + r x Mv

where I is the central inertia dyadic of B, w is the angular velocity of body B in the frame, N, r is the position vector from point O to the mass center of B, and v is the velocity of the mass center in the frame, N.

Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> M, v, r, omega = dynamicsymbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
```

(continues on next page)

(continued from previous page)

```
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, 1 * N.x)
>>> I = outer(b.x, b.x)
>>> B = RigidBody('B', P, b, M, (I, P))
>>> B.angular_momentum(P, N)
omega*b.x
```

property central_inertia

The body's central inertia dyadic.

kinetic_energy(*frame*)

Kinetic energy of the rigid body.

Parameters

frame : ReferenceFrame

The RigidBody's angular velocity and the velocity of it's mass center are typically defined with respect to an inertial frame but any relevant frame in which the velocities are known can be supplied.

Explanation

The kinetic energy, T, of a rigid body, B, is given by

'T = 1/2 (I omega^2 + m v^2)'

where I and m are the central inertia dyadic and mass of rigid body B, respectively, omega is the body's angular velocity and v is the velocity of the body's mass center in the supplied ReferenceFrame.

Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody
>>> from sympy import symbols
>>> M, v, r, omega = symbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer (b.x, b.x)
>>> inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, inertia_tuple)
>>> B.kinetic_energy(N)
M*v**2/2 + omega**2/2
```

linear_momentum(*frame*)

Linear momentum of the rigid body.

Parameters

frame : ReferenceFrame

The frame in which linear momentum is desired.

Explanation

The linear momentum L, of a rigid body B, with respect to frame N is given by

L = M * v*

where M is the mass of the rigid body and v* is the velocity of the mass center of B in the frame, N.

Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> M, v = dynamicsymbols('M v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer (N.x, N.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, N, M, Inertia_tuple)
>>> B.linear_momentum(N)
M*v*N.x
```

parallel_axis(*point*)

Returns the inertia dyadic of the body with respect to another point.

Parameters

point : sympy.physics.vector.Point

The point to express the inertia dyadic about.

Returns

inertia : sympy.physics.vector.Dyadic

The inertia dyadic of the rigid body expressed about the provided point.

property potential_energy

The potential energy of the RigidBody.

Examples

```
>>> from sympy.physics.mechanics import RigidBody, Point, outer,␣
↪ReferenceFrame
>>> from sympy import symbols
>>> M, g, h = symbols('M g h')
>>> b = ReferenceFrame('b')
>>> P = Point('P')
```

(continues on next page)

(continued from previous page)

```
>>> I = outer (b.x, b.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, Inertia_tuple)
>>> B.potential_energy = M * g * h
>>> B.potential_energy
M*g*h
```

inertia

`sympy.physics.mechanics.functions.`**`inertia`**(*frame, ixx, iyy, izz, ixy=0, iyz=0, izx=0*)

Simple way to create inertia Dyadic object.

Parameters

frame : ReferenceFrame

The frame the inertia is defined in

ixx : Sympifyable

the xx element in the inertia dyadic

iyy : Sympifyable

the yy element in the inertia dyadic

izz : Sympifyable

the zz element in the inertia dyadic

ixy : Sympifyable

the xy element in the inertia dyadic

iyz : Sympifyable

the yz element in the inertia dyadic

izx : Sympifyable

the zx element in the inertia dyadic

Explanation

If you do not know what a Dyadic is, just treat this like the inertia tensor. Then, do the easy thing and define it in a body-fixed frame.

Examples

```
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')
>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)
```

inertia_of_point_mass

sympy.physics.mechanics.functions.**inertia_of_point_mass**(*mass*, *pos_vec*, *frame*)

Inertia dyadic of a point mass relative to point O.

Parameters

mass : Sympifyable

Mass of the point mass

pos_vec : Vector

Position from point O to point mass

frame : ReferenceFrame

Reference frame to express the dyadic in

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import ReferenceFrame, inertia_of_point_
↪mass
>>> N = ReferenceFrame('N')
>>> r, m = symbols('r m')
>>> px = r * N.x
>>> inertia_of_point_mass(m, px, N)
m*r**2*(N.y|N.y) + m*r**2*(N.z|N.z)
```

linear_momentum

sympy.physics.mechanics.functions.**linear_momentum**(*frame*, **body*)

Linear momentum of the system.

Parameters

frame : ReferenceFrame

The frame in which linear momentum is desired.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose linear momentum is required.

Explanation

This function returns the linear momentum of a system of Particle's and/or RigidBody's. The linear momentum of a system is equal to the vector sum of the linear momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The linear momentum of the system, L, is equal to the vector sum of the linear momentum of the particle, L1, and the linear momentum of the rigid body, L2, i.e.

L = L1 + L2

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, linear_momentum
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = Point('Ac')
>>> Ac.set_vel(N, 25 * N.y)
>>> I = outer(N.x, N.x)
>>> A = RigidBody('A', Ac, N, 20, (I, Ac))
>>> linear_momentum(N, A, Pa)
10*N.x + 500*N.y
```

angular_momentum

`sympy.physics.mechanics.functions.`**`angular_momentum`**`(`*point, frame, *body*`)`

Angular momentum of a system.

Parameters

point : Point

The point about which angular momentum of the system is desired.

frame : ReferenceFrame

The frame in which angular momentum is desired.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose angular momentum is required.

Explanation

This function returns the angular momentum of a system of Particle's and/or RigidBody's. The angular momentum of such a system is equal to the vector sum of the angular momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The angular momentum of the system, H, is equal to the vector sum of the angular momentum of the particle, H1, and the angular momentum of the rigid body, H2, i.e.

H = H1 + H2

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, angular_
↪momentum
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> angular_momentum(O, N, Pa, A)
10*N.z
```

kinetic_energy

`sympy.physics.mechanics.functions.`**`kinetic_energy`**`(`*frame, *body*`)`

Kinetic energy of a multibody system.

Parameters

frame : ReferenceFrame

The frame in which the velocity or angular velocity of the body is defined.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose kinetic energy is required.

Explanation

This function returns the kinetic energy of a system of Particle's and/or RigidBody's. The kinetic energy of such a system is equal to the sum of the kinetic energies of its constituents. Consider a system, S, comprising a rigid body, A, and a particle, P. The kinetic energy of the system, T, is equal to the vector sum of the kinetic energy of the particle, T1, and the kinetic energy of the rigid body, T2, i.e.

T = T1 + T2

Kinetic energy is a scalar.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, kinetic_energy
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> kinetic_energy(N, Pa, A)
350
```

potential_energy

sympy.physics.mechanics.functions.**potential_energy**(**body*)

Potential energy of a multibody system.

Parameters

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose potential energy is required.

Explanation

This function returns the potential energy of a system of Particle's and/or RigidBody's. The potential energy of such a system is equal to the sum of the potential energy of its constituents. Consider a system, S, comprising a rigid body, A, and a particle, P. The potential energy of the system, V, is equal to the vector sum of the potential energy of the particle, V1, and the potential energy of the rigid body, V2, i.e.

V = V1 + V2

Potential energy is a scalar.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, potential_
↪energy
>>> from sympy import symbols
>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
```

(continues on next page)

(continued from previous page)

```
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> Pa = Particle('Pa', P, m)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> a = ReferenceFrame('a')
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * h
>>> potential_energy(Pa, A)
M*g*h + g*h*m
```

Lagrangian

sympy.physics.mechanics.functions.**Lagrangian**(*frame*, **body*)

Lagrangian of a multibody system.

Parameters

frame : ReferenceFrame

The frame in which the velocity or angular velocity of the body is defined to determine the kinetic energy.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose Lagrangian is required.

Explanation

This function returns the Lagrangian of a system of Particle's and/or RigidBody's. The Lagrangian of such a system is equal to the difference between the kinetic energies and potential energies of its constituents. If T and V are the kinetic and potential energies of a system then it's Lagrangian, L, is defined as

L = T - V

The Lagrangian is a scalar.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, Lagrangian
>>> from sympy import symbols
>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
```

(continues on next page)

(continued from previous page)

```
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * h
>>> Lagrangian(N, Pa, A)
-M*g*h - g*h*m + 350
```

Body (Docstrings)

Body

class sympy.physics.mechanics.body.**Body**(*name*, *masscenter=None*, *mass=None*, *frame=None*, *central_inertia=None*)

Body is a common representation of either a RigidBody or a Particle SymPy object depending on what is passed in during initialization. If a mass is passed in and central_inertia is left as None, the Particle object is created. Otherwise a RigidBody object will be created.

Parameters

name : String

Defines the name of the body. It is used as the base for defining body specific properties.

masscenter : Point, optional

A point that represents the center of mass of the body or particle. If no point is given, a point is generated.

mass : Sympifyable, optional

A Sympifyable object which represents the mass of the body. If no mass is passed, one is generated.

frame : ReferenceFrame, optional

The ReferenceFrame that represents the reference frame of the body. If no frame is given, a frame is generated.

central_inertia : Dyadic, optional

Central inertia dyadic of the body. If none is passed while creating RigidBody, a default inertia is generated.